4 The C programming language

The C programming language was developed at Bell Laboratories in the early 1970s as the system programming language for Unix, based on the earlier and even more primitive languages BCPL and B. When originally developed, it was targeted at machines that were extremely limited by modern standards: the first Unix implementation (and the B compiler that supported it) ran on a DEC PDP-7 with only 8192 18-bit words of memory (Dennis M. Ritchie, The development of the C language, in Thomas J. Bergin, Jr., and Richard G. Gibson, Jr., History of Programming Languages-II ed., ACM Press, 1996). So using as few resources as possible, both in the compiler and in the resulting code, was a priority.

This priority is reflected in the features (and lack of features) of C, and is partly responsible for its success. Programs written in C place almost no demands on the system they run on and give the programmer nearly complete control over their execution: this allows programs that were previously written in assembly language, like operating system kernels and device drivers, to be implemented in C. So C is often the first language ported to any new architecture, and many higher-level languages are either executed using interpreters written in C or use C as in intermediate language in the compilation process.

Since its initial development, C has gone through four major versions:

- The original **K&R C** defined in the 1978 first edition of Kernighan and Ritchie's book *The C Programming Language*;
- **ANSI C**, from 1988, which fixed some oddities in the syntax and which is documented in the 1988 second edition of *The C Programming Language*;
- **C99**, from 1999, the ISO/IEC 9899:1999 standard for C, which added some features from C++ and many new features for high-performance numerical computing;
- **C11**, from 2011, the ISO/IEC 9899:2011 standard for C, which relaxed some of the requirements of C99 that most compilers hadn't bothered implementing and which added a few extra features.

Unfortunately, C99 and C11 both exemplify the uselessness of standards committees in general and the ISO in particular. Because the ISO has no power to enforce standards on compiler writers, and because they will charge you CHF 198 just to look at the C11 standard, many compiler writers have ignored much of C99 and C11. In particular, Microsoft pretty much gave up on adding any features after ANSI C, and support for C99 and C11 is spotty in `gcc` and `clang`, the two dominant open source C compilers. So if you want to write portable C code, it is safest to limit yourself to features in ANSI C.

For this class, we will permit you to use any feature of C99 that `gcc` supports, which also includes all features of ANSI C. You can compile with C99 support by using `gcc --std=c99` or by calling `gcc` as `c99`, as in `c99 -o hello hello.c`. Compiling with straight `gcc` will give you GNU's own peculiar dialect of C, which is basically ANSI C with some extras. For maximum portability when using `gcc`, it is safest to use `gcc -ansi -pedantic`, which expects straight ANSI C and will complain about any extensions.

4.1 Structure of a C program

A C program consists of one or more files (which act a little bit like modules in more structured programming languages, each of which typically contains **definitions** of **functions**, each of which consists of **statements**, which are either **compound statements** like `if`, `while`, etc. or **expressions** that typically perform some sort of arithmetic or call other functions. Files may also include **declarations** of global variables (not recommended), and functions will often contain declarations of local variables that can only be used inside that function.

Here is a typical small C program that sums a range of integers. Since this is our first real program, it's a little heavy on the comments (shown between `/*` and `*/`).

```
#include <stdio.h>    /* This is needed to get the declarations of fprintf and printf */
#include <stdlib.h>   /* This is needed to get the declaration of atoi */

/* Return the sum of all integers i
 * such that start <= i and i < end. */
int
sumRange(int start, int end)
{
    int i;    /* loop variable */
    int sum;  /* sum of all values so far */

    /* a mathematician would use a formula for this,
     * but we are computer programmers! */
    sum = 0;

    /* The three parts of the header for this loop mean:
     * 1. Set i to start initially.
     * 2. Keep doing the loop as long as i is less than end.
     * 3. After each iteration, add 1 to i.
     */
    for(i = start; i < end; i++) {
        sum += i;  /* This adds i to sum */
    }

    /* This exits the function immediately,
     * sending the value of sum back to the caller. */
    return sum;
```

```
}

int
main(int argc, char **argv)
{
    int start;    /* initial value in range */
    int end;      /* one past the last value in the range */

    /* This tests for the wrong number of arguments.
     * The != operator returns true (1) if its arguments are not equal,
     * and false (0) otherwise.
     * Note that the program name itself counts as an argument
     * (which is why we want the argument count to be 3)
     * and appears in position 0 in the argument vector
     * (which means we can get it using argv[0]). */
    if(argc != 3) {
        fprintf(stderr, "Usage: %s\n start end", argv[0]);
        return 1;
    }

    /* Convert start and end positions from strings to ints */
    start = atoi(argv[1]);
    end = atoi(argv[2]);

    /* Call sumRange and print the result */
    printf("sumRange(%d, %d) = %d\n", start, end, sumRange(start, end));

    return 0;
}
```

examples/sumRange.c

This is what the program does if we compile and run it:

```
$ c99 -g -Wall -pedantic -o sumRange sumRange.c
$ ./sumRange 1 100
sumRange(1, 100) = 4950
```

The `sumRange.c` program contains two functions, `sumRange` and `main`. The `sumRange` function does the actual work, while `main` is the main routine of the program that gets called with the command-line arguments when the program is run. Every C program must have a routine named `main` with these particular arguments.

In addition, `main` may call three library functions, `fprintf` (which in this case is used to generate error messages), `printf` (which generates ordinary output), and `atoi` (which is used to translate the command-line arguments into numerical values). These functions must all be declared before they can be used. In the case of `sumRange`, putting the definition of `sumRange` before the definition of `main` is enough. For the library routines, the **include files** `stdio.h` and `stdlib.h` contain declarations of the functions that contain enough information about there return types and arguments that the compiler knows how to generate machine code to call them. These files are included in `sumRange.c` by the **C preprocessor**, which pastes in the contents of any file specified by the `#include` command, strips out any comments (delimited by `/*` and `*/`, or by `//` and the end of the line if you are using C99), and does some other tricks that allow you to muck with the source code before the actual compiler sees it (see Macros). You can see what the output of the preprocessor looks like by calling the C compiler with the `-E` option, as in `c99 -E sumRange.c`.

The **body** of each function consists of some **variable declarations** followed by a sequence of **statements** that tell the computer what to do. Unlike some languages, every variable used in a C program must be declared. A declaration specifies the **type** of a variable, which tells the compiler how much space to allocate for it and how to interpret some operations on its value. Statements may be **compound statements** like the `if` statement in `main` that executes its body only if the program is called with the wrong number of command-line arguments or the `for` loop in `sumRange` that executes its body as long as the test in its header remains true; or they may be **simple statements** that consist of a single **expression** followed by a semicolon.

An **expression** is usually either a bare function call whose value is discarded (for example, the calls to `fprintf` and `printf` in `main`), or an arithmetic expression (which may include function calls, like the calls to `atoi` or in `main`) whose value is assigned to some variable using the **assignment operator** `=` or sometimes variants like `+=` (which is shorthand for adding a value to an existing variable: `x += y` is equivalent to `x = x+y`).

When you compile a C program, after running the preprocessor, the compiler generates **assembly language** code that is a human-readable description of the ultimate machine code for your target CPU. Assembly language strips out all the human-friendly features of your program and reduces it to simple instructions usually involving moving things from one place to the other or performing a single arithmetic operation. For example, the C line

```
x = y + 1;  /* add 1 to y, store result in x */
```

gets translated into x86 assembly as

```
movl    -24(%rbp), %edi
addl    $1, %edi
    movl    %edi, -28(%rbp)
```

These three operations copy the value of `y` into the CPU register `%edi`, add 1 to the `%edi` register, and then copy the value back into `x`. This corresponds directly to what you would have to do to evaluate `x = y + 1` if you could only do one very basic operation at a time and couldn't do arithmetic operations on memory locations: fetch `y`, add 1, store `x`. Note that the CPU doesn't know about the names `y` and `x`; instead, it computes their addresses by adding -24 and -28 respectively to the base pointer register `%rbp`. This is why it can be hard to debug compiled code unless you tell the compiler to keep around extra information.

For an arbitrary C program, if you are using `gcc`, you can see what your code looks like in assembly language using the `-S` option. For example, `c99 -S sumRange.c` will create a file `sumRange.s` that looks like this:

```
        .file        "sumRange.c"
        .text
        .globl        sumRange
        .type        sumRange, @function
sumRange:
.LFB0:
        .cfi_startproc
        pushl        %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl        %esp, %ebp
        .cfi_def_cfa_register 5
        subl        $16, %esp
        movl        $0, -4(%ebp)
        movl        8(%ebp), %eax
        movl        %eax, -8(%ebp)
        jmp        .L2
.L3:
        movl        -8(%ebp), %eax
        addl        %eax, -4(%ebp)
        addl        $1, -8(%ebp)
.L2:
        movl        -8(%ebp), %eax
        cmpl        12(%ebp), %eax
        jl        .L3
        movl        -4(%ebp), %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size        sumRange, .-sumRange
        .section        .rodata
.LC0:
        .string        "Usage: %s\n start end"
.LC1:
```

```
        .string         "sumRange(%d, %d) = %d\n"
        .text
        .globl          main
        .type           main, @function
main:
.LFB1:
        .cfi_startproc
        pushl           %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl            %esp, %ebp
        .cfi_def_cfa_register 5
        andl            $-16, %esp
        subl            $32, %esp
        cmpl            $3, 8(%ebp)
        je              .L6
        movl            12(%ebp), %eax
        movl            (%eax), %edx
        movl            stderr, %eax
        movl            %edx, 8(%esp)
        movl            $.LC0, 4(%esp)
        movl            %eax, (%esp)
        call            fprintf
        movl            $1, %eax
        jmp             .L7
.L6:
        movl            12(%ebp), %eax
        addl            $4, %eax
        movl            (%eax), %eax
        movl            %eax, (%esp)
        call            atoi
        movl            %eax, 24(%esp)
        movl            12(%ebp), %eax
        addl            $8, %eax
        movl            (%eax), %eax
        movl            %eax, (%esp)
        call            atoi
        movl            %eax, 28(%esp)
        movl            28(%esp), %eax
        movl            %eax, 4(%esp)
        movl            24(%esp), %eax
        movl            %eax, (%esp)
        call            sumRange
        movl            %eax, 12(%esp)
        movl            28(%esp), %eax
        movl            %eax, 8(%esp)
```

```
        movl        24(%esp), %eax
        movl        %eax, 4(%esp)
        movl        $.LC1, (%esp)
        call        printf
        movl        $0, %eax
.L7:
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE1:
        .size        main, .-main
        .ident        "GCC: (Ubuntu 4.8.2-19ubuntu1) 4.8.2"
        .section        .note.GNU-stack,"",@progbits
```

examples/sumRange.s

You usually don't need to look at assembly language, but it can sometimes be enlightening to see what the compiler is doing with your code. One thing that I find interesting about this particular code (which is for the x86 architecture) is that most of the instructions are `movl`, the x86 instruction for copying a 32-bit quantity from one location to another: most of what this program is doing is copying data into the places expected by the library functions it is calling. Also noteworthy is that the beautiful compound statements like `if` and `for` that so eloquently express the intent of the programmer get turned into a pile of jump (`jmp`) and conditional jump (`jl`, `je`) instructions, the machine code versions of the often dangerous and confusing `goto` statement. This is because CPUs are dumb: they don't know how to carry out an `if` branch or a loop, and all they can do instead is be told to replace the value of their program counter register with some new value instead of just incrementing it as they usually do.

Assembly language is not the last stage in this process. The **assembler** (`as`) is a program that translates the assembly language in `sumRange.s` into machine code (which will be stored in `sumRange.o` if we aren't compiling a single program all at once). Machine code is not human-readable, and is close to the raw stream of bytes that gets stored in the computer's memory to represent a running program. The missing parts are that the addresses of each function and global variables are generally left unspecified, so that they can be moved around to make room for other functions and variables coming from other files and from system libraries. The job of stitching all of these pieces together, putting everything in the right place, filling in any placeholder addresses, and generating the **executable file** `sumRange` that we can actually run is given to the **linker** `ld`.

The whole process looks like this:

```
sumRange.c (source code)
   |
```

```
   v
[preprocessor (cpp)]
   |
   v
preprocessed version of sumRange.c
   |
   v
[compiler (gcc)]
   |
   v
sumRange.s (assembly code)
   |
   v
[assembler (as)]
   |
   v
sumRange.o (machine code)
   |
   v
[linker (ld)] <- system library (glibc.a)
   |
   v
sumRange (executable)
```

The good news is, you don't actually have to run all of these steps yourself; instead, `gcc` (which you may be calling as `c99`) will take care of everything for you, particularly for simple programs like `sumRange.c` that fit in a single file.

4.2 Numeric data types

All data stored inside a computer is ultimately represented as a sequence of **bits**, 0 or 1 values, typically organized into **words** consisting of several 8-bit **bytes**.[6]

A typical desktop computer might have enough RAM to store 2^{32} bytes (4 gigabytes); the Zoo machines store 2^{35} bytes (32 gigabytes). However, the **address space** of a process might be much larger: on a 64-bit machine, the address space is 2^{64} bytes. There's no way to store 2^{64} different addresses in 2^{35} bytes of RAM; instead, a **memory mapper**, typically built in to the CPU, translates the large addresses of the parts of the address space that are actually used into smaller addresses corresponding to actual RAM locations. In some cases, regions of memory that have not been used in a while will be **swapped out** to disk, leaving more RAM free for other parts of the process (or other processes). This technique is known as **virtual memory** and is usually invisible

[6] This convention was not always followed in the early days of computing. For example, the PDP-7 on which UNIX was first developed used 18-bit words, which conveniently translated into six octal digits back in the pre-hexadecimal era.

to the programmer. The use of virtual memory can increase the available space beyond the size of the RAM a little bit, but if you try to run a process that is actively using significantly more space that can be stored in RAM, it will slow down dramatically, because disk drives are roughly ten million times slower than memory.

The most basic kind of data represents integer values from some bounded range. C supports several **integer data types**, varying in their size (and thus range), and whether or not they are considered to be signed. These are described in more detail below.

For numerical computation, integer data types can be inconvenient. So C also supports **floating-point types** that consist of a fixed-size **mantissa**, which is essentially an integer, together with an **exponent** that is used to multiply the mantissa by 2^x for some x. These allow very small or very large values to be represented with small relative error, but do not allow exact computation because of the limited precision of the mantissa. Floating-point types are also described below.

All other data is represented by converting it to either integer or floating-point numbers. For example, text characters in C are represented as small integer values, so that the character constant `'z'` representation a lower-case "z" is exactly the same as the integer constant `122` (which is the ASCII code for "z"). A string like `"hi there"` is represented by a sequence of 8-bit ASCII characters, with a special 0 character to mark the end of the string. Strings that go beyond the English characters available in the ASCII encoding are typically represented using Unicode and encoded as sequences of bytes using a particular representation called UTF-8. The color of a pixel in an image might be represented as three 8-bit integers representing the intensity of red, green, and blue in the color, while an image itself might be a long sequence of such 3-byte RGB values. At the bottom, every operation applied to these more complex data types translates into a whole lot of copies and arithmetic operations on individual bytes and words.

From the CPU's point of view, even much of this manipulation consists of operating on integers that happen to represent addresses instead of data. So when a C program writes a zero to the 19th entry in a sequence of 4-byte integers, somewhere in the implementation of this operation the CPU will be adding $4 \cdot 19$ to a base address for the sequence to computer where to write this value. Unlike many higher-level languages, C allows the program direct access to address computations via **pointer types**, which are tricky enough to get their own chapter. Indeed, most of the structured types that C provides for representing more complicated data can best be understood as a thin layer of abstraction on top of pointers. We will see examples of these in later chapters as well.

For now, we concentrate on integer and floating-point types, and on the operations that can be applied to them.

4.2.1 Integer types in C

Most variables in C programs tend to hold integer values, and indeed most variables in C programs tend to be the default-width integer type `int`. Declaring a variable to have a particular integer type controls how much space is used to store the variable (any values too big to fit will be truncated) and specifies that the arithmetic on the variable is done using integer operations.

4.2.1.1 Basic integer types The standard C integer types are:

Name	Typical size	Signed by default?
`char`	8 bits	unspecified
`short`	16 bits	signed
`int`	32 bits	signed
`long`	32 bits	signed
`long long`	64 bits	signed

The typical size is for 32-bit architectures like the Intel i386. Some 64-bit machines might have 64-bit `int`s and `long`s, and some microcontrollers have 16-bit `int`s. Particularly bizarre architectures might have even wilder sizes, but you are not likely to see this unless you program vintage 1970s supercomputers. The general convention is that `int` is the most convenient size for whatever computer you are using and should be used by default.

Many compilers also support a `long long` type that is usually twice the length of a long (e.g. 64 bits on i386 machines). This type was not officially added to the C standard prior to C99, so it may or may not be available if you insist on following the ANSI specification strictly.

Each of these types comes in signed and unsigned variants.
This controls the interpretation of some operations (mostly comparisons and shifts) and determines the range of the type: for example, an `unsigned char` holds values in the range 0 through 255 while a `signed char` holds values in the range -128 through 127, and in general an unsigned n-bit type runs from 0 through $2^n - 1$ while the signed version runs from -2^{n-1} through $2^{n-1} - 1$. The representation of signed integers uses **two's-complement** notation, which means that a positive value x is represented as the unsigned value x while a negative value $-x$ is represented as the unsigned value $2^n - x$. For example, if we had a peculiar implementation of C that used 3-bit `int`s, the binary values and their interpretation as `int` or `unsigned int` would look like this:

bits	as `unsigned int`	as `int`
000	0	0
001	1	1

bits	as `unsigned int`	as `int`
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

The reason we get one extra negative value for an unsigned integer type is this allows us to interpret the first bit as the sign, which makes life a little easier for whoever is implementing our CPU. Two useful features of this representation are:

1. We can convert freely between signed and unsigned values as long as we are in the common range of both, and
2. Addition and subtraction work exactly the same we for both signed and unsigned values. For example, on our hypothetical 3-bit machine, $1+5$ represented as $001+101=110$ gives the same answer as $1+(-3)=001+101=110$. In the first case we interpret 110 as 6, while in the second we interpret it as -2, but both answers are right in their respective contexts.

Note that in order to make this work, we can't detect overflow: when the CPU adds two 3-bit integers, it doesn't know if we are adding $7+6=111+110=1101=13$ or $(-1)+(-2)=111+110=101=(-3)$. In both cases the result is truncated to 101, which gives the incorrect answer 5 when we are adding unsigned values.

This can often lead to surprising uncaught errors in C programs, although using more than 3 bits will make overflow less likely. It is usually a good idea to pick a size for a variable that is substantially larger than the largest value you expect the variable to hold (although most people just default to `int`), unless you are very short on space or time (larger values take longer to read and write to memory, and may make some arithmetic operations take longer).

Taking into account signed and unsigned versions, the full collection of integer types looks like this:

`char`	`signed char`	`unsigned char`
`short`		`unsigned short`
`int`		`unsigned int`
`long`		`unsigned long`
`long long`		`unsigned long long`

So these are all examples of declarations of integer variables:

```c
int i;
char c;
signed char temperature; /* degrees Celsius, only valid for Earth's surface */
long netWorthInPennies;
long long billGatesNetWorthInPennies;
unsigned short shaveAndAHaircutTwoBytes;
```

For `char`s, whether the character is signed ($-128 \ldots 127$) or unsigned ($0 \ldots 255$) is at the whim of the compiler. If it matters, declare your variables as `signed char` or `unsigned char`. For storing actual 8-bit characters that you aren't doing arithmetic on, it shouldn't matter.

There is a slight gotcha for character processing with input function like `getchar` and `getc`. These return the special value `EOF` (defined in `stdio.h` to be -1) to indicate end of file. But 255, which represents `'ÿ'` in the ISO Latin-1 alphabet and in Unicode and which may also appear quite often in binary files, will map to -1 if you put it in a character. So you should store the output of these functions in an `int` if you need to test for end of file. After you have done this test, it's OK to store a non-end-of-file character in a `char`.

```c
/* right */
int c;

while((c = getchar()) != EOF) {
    putchar(c);
}
```

```c
/* WRONG */
char c;

while((c = getchar()) != EOF) {  /* <- DON'T DO THIS! */
    putchar(c);
}
```

4.2.1.2 Overflow and the C standards So far we have been assuming that overflow implicitly applies a $(\bmod 2^b)$ operation, where b is the number of bits in our integer data type. This works on many machines, but as of the C11 standard, this is defined behavior only for *unsigned* integer types. For *signed* integer types, the effect of overflow is **undefined**. This means that the result of adding two very large signed `int`s could be arbitrary, and not only might depend on what CPU, compiler, and compiler options you are using, but might even vary from one execution of your program to another. In many cases this is not an issue, but undefined behavior is often exploited by compilers to speed up compiled code by omitting otherwise necessary instructions to force a particular outcome. This is especially true if you turn on the optimizer using the `-O` flag.

This means that you should not depend on reasonable behavior for overflow of signed types. Usually this is not a problem, because signed computations often represent real-world values where overflow will produce bad results anyway. For unsigned computations, the implicit modulo operation applied to overflow can be useful for some applications.

4.2.1.3 C99 fixed-width types C99 provides a `stdint.h` header file that defines integer types with known size independent of the machine architecture. So in C99, you can use `int8_t` instead of `signed char` to guarantee a signed type that holds exactly 8 bits, or `uint64_t` instead of `unsigned long long` to get a 64-bit unsigned integer type. The full set of types typically defined are `int8_t`, `int16_t`, `int32_t`, and `int64_t` for signed integers and `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t` for unsigned integers. There are also types for integers that contain the fewest number of bits greater than some minimum (e.g., `int_least16_t` is a signed type with at least 16 bits, chosen to minimize space) or that are the fastest type with at least the given number of bits (e.g., `int_fast16_t` is a signed type with at least 16 bits, chosen to minimize time). The `stdint.h` file also defines constants giving the minimum and maximum values of these and standard integer types; for example, `INT_MIN` and `INT_MAX` give the smallest and largest values that can be stored in an `int`.

All of these types are defined as aliases for standard integer types using `typedef`; the main advantage of using `stdint.h` over defining them yourself is that if somebody ports your code to a new architecture, `stdint.h` should take care of choosing the right types automatically. The main disadvantage is that, like many C99 features, `stdint.h` is not universally available on all C compilers. Also, because these fixed-width types are a late addition to the language, the built-in routines for printing and parsing integers, as well as the mechanisms for specifying the size of an integer constant, are not adapted to deal with them.

But if you do need to print or parse types defined in `stdint.h`, the larger `inttypes.h` header defines macros that give the corresponding format strings for `printf` and `scanf`. The `inttypes.h` file includes `stdint.h`, so you do not need to include both. Below is an example of a program that uses the various features provided by `inttypes.h` and `stdint.h`.

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

#include <inttypes.h>

/* example of using fixed-width types */

/* largest value we can apply 3x+1 to without overflow */
#define MAX_VALUE ((UINT64_MAX - 1) / 3)
```

```c
int
main(int argc, char **argv)
{
    uint64_t big;

    if(argc != 2) {
        fprintf(stderr, "Usage: %s number\n", argv[0]);
        return 1;
    }

    /* parse argv[1] as a uint64_t */
    /* SCNu64 expands to the format string for scanning uint64_t (without the %) */
    /* We then rely on C concatenating adjacent string constants. */
    sscanf(argv[1], "%" SCNu64, &big);

    /* do some arithmetic on big */
    while(big != 1) {
        /* PRIu64 expands to the format string for printing uint64_t */
        printf("%" PRIu64 "\n", big);

        if(big % 2 == 0) {
            big /= 2;
        } else if(big <= MAX_VALUE) {
            big = 3*big + 1;
        } else {
            /* overflow! */
            puts("overflow");
            return 1;
        }
    }

    puts("Reached 1");
    return 0;
}
```

examples/integerTypes/fixedWidth.c

4.2.2 size_t and ptrdiff_t

The type aliases `size_t` and `ptrdiff_t` are provided in `stddef.h` to represent the return types of the `sizeof` operator and pointer subtraction. On a 32-bit architecture, `size_t` will be equivalent to the unsigned 32-bit integer type `uint32_t` (or just `unsigned int`) and `ptrdiff_t` will be equivalent to the signed 32-bit integer type `int32_t` (`int`). On a 64-bit architecture, `size_t` will be equivalent to `uint64_t` and `ptrdiff_t` will be equivalent to `int64_t`.

The place where you will most often see `size_t` is as an argument to `malloc`, where it gives the number of bytes to allocate.

Because `stdlib.h` includes `stddef.h`, it is often not necessary to include `stddef.h` explicitly.

4.2.2.1 Integer constants

Constant integer values in C can be written in any of four different ways:

- In the usual decimal notation, e.g. `0`, `1`, `-127`, `9919291`, `97`.
- In **octal** or base 8, when the leading digit is `0`, e.g. `01` for 1, `010` for 8, `0777` for 511, `0141` for 97. Octal is not used much any more, but it is still conventional for representing Unix file permissions.
- In **hexadecimal** or base 16, when prefixed with `0x`. The letters `a` through `f` are used for the digits 10 through 15. For example, `0x61` is another way to write 97.
- Using a **character constant**, which is a single ASCII character or an **escape sequence** inside single quotes. The value is the ASCII value of the character: `'a'` is 97.[7] Unlike languages with separate character types, C characters are identical to integers; you can (but shouldn't) calculate 97^2 by writing `'a'*'a'`. You can also store a character anywhere.

Except for character constants, you can insist that an integer constant is unsigned or long by putting a `u` or `l` after it. So `1ul` is an `unsigned long` version of 1. By default integer constants are (signed) `int`s. For `long long` constants, use `ll`, e.g., the `unsigned long long` constant `0xdeadbeef01234567ull`. It is also permitted to write the `l` as `L`, which can be less confusing if the `l` looks too much like a `1`.

Some examples:

`'a'`	`int`
`97`	`int`
`97u`	`unsigned int`
`0xbea00d1ful`	`unsigned long`, written in hexadecimal
`0777s`	`short`, written in octal

A curious omission is that there is no way to write a binary integer directly in C. So if you want to write the bit pattern 00101101, you will need to encode it in hexadecimal as `0x2d` (or octal as `055`). Another potential trap is that leading zeros matter: `012` is an octal value corresponding to what normal people call 10.

[7] Certain ancient versions of C ran on machines with a different character set encoding, like EBCDIC. The C standard does not guarantee ASCII encoding.

4.2.2.1.1 Naming constants Having a lot of numeric constants in your program—particularly if the same constant shows up in more than one place—is usually a sign of bad programming. There are a few constants, like 0 and 1, that make sense on their own, but many constant values are either mostly arbitrary, or might change if the needs of the program change. It's helpful to assign these constants names that explain their meaning, instead of requiring the user to guess why there is a `37` here or an `0x1badd00d` there. This is particularly important if the constants might change in later versions of the program, since even though you could change every `37` in your program into a `38`, this might catch other `37` values that have different intended meanings.

For example, suppose that you have a function (call it `getchar`) that needs to signal that sometimes it didn't work. The usual way is to return a value that the function won't normally return. Now, you could just tell the user what value that is:

```c
/* get a character (as an `int` ASCII code) from `stdin` */
/* return -1 on end of file */
int getchar(void);
```

and now the user can write

```c
    while((c = getchar()) != -1) {
        ...
    }
```

But then somebody reading the code has to remember that `-1` means "end of file" and not "signed version of `0xff`" or "computer room on fire, evacuate immediately." It's much better to define a constant `EOF` that happens to equal `-1`, because among other things if you change the special return value from `getchar` later then this code will still work (assuming you fixed the definition of `EOF`):

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

So how do you declare a constant in C? The traditional approach is to use the C preprocessor, the same tool that gets run before the compiler to expand out `#include` directives. To define `EOF`, the file `/usr/include/stdio.h` includes the text

```c
#define EOF (-1)
```

What this means is that whenever the characters `EOF` appear in a C program as a separate word (e.g. in `1+EOF*3` but not in `appurtenancesTherEOF`), then the preprocessor will replace them with the characters `(-1)`. The parentheses around the `-1` are customary to ensure that the `-1` gets treated as a separate constant and not as part of some larger expression. So from the compiler's perspective, `EOF` really is `-1`, but from the programmer's perspective, it's end-of-file. This is a special case of the C preprocessor's macro mechanism.

In general, any time you have a non-trivial constant in a program, it should be `#define`d. Examples are things like array dimensions, special tags or return values from functions, maximum or minimum values for some quantity, or standard mathematical constants (e.g., `/usr/include/math.h` defines `M_PI` as pi to umpteen digits). This allows you to write

```
char buffer[MAX_FILENAME_LENGTH+1];

area = M_PI*r*r;

if(status == COMPUTER_ROOM_ON_FIRE) {
    evacuate();
}
```

instead of

```
char buffer[513];

area = 3.141592319*r*r;

if(status == 136) {
    evacuate();
}
```

which is just an invitation to errors (including the one in the area computation).

Like `typedef`s, `#define`s that are intended to be globally visible are best done in header files; in large programs you will want to `#include` them in many source files. The usual convention is to write `#define`d names in all-caps to remind the user that they are macros and not real variables.

4.2.2.2 Integer operators

4.2.2.2.1 Arithmetic operators The usual `+` (addition), `-` (negation or subtraction), and `*` (multiplication) operators work on integers pretty much the way you'd expect. The only caveat is that if the result lies outside of the range of whatever variable you are storing it in, it will be truncated instead of causing an error:

```
unsigned char c;

c = -1;             /* sets c = 255 */
c = 255 + 255;      /* sets c = 254 */
c = 256 * 1772717;  /* sets c = 0 */
```

This can be a source of subtle bugs if you aren't careful. The usual giveaway is that values you thought should be large positive integers come back as random-looking negative integers.

Division (`/`) of two integers also truncates: `2/3` is 0, `5/3` is 1, etc. For positive integers it will always round down.

Prior to C99, if either the numerator or denominator was negative, the behavior was unpredictable and depended on what your processor chose to do. In practice this meant you should never use `/` if one or both arguments might be negative. The C99 standard specified that integer division always removes the fractional part, effectively rounding toward 0; so `(-3)/2` is `-1`, `3/-2` is `-1`, and `(-3)/-2` is `1`.

There is also a remainder operator `%` with e.g. `2%3` = 2, `5%3` = 2, `27 % 2` = 1, etc. The sign of the modulus is ignored, so `2%-3` is also `2`. The sign of the dividend carries over to the remainder: `(-3)%2` and `(-3)%(-2)` are both `-1`. The reason for this rule is that it guarantees that `y == x*(y/x) + y%x` is always true.

4.2.2.2.2 Bitwise operators In addition to the arithmetic operators, integer types support **bitwise logical** operators that apply some Boolean operation to all the bits of their arguments in parallel. What this means is that the i-th bit of the output is equal to some operation applied to the i-th bit(s) of the input(s). The bitwise logical operators are `~` (bitwise negation: used with one argument as in `~0` for the all-1's binary value), `&` (bitwise AND), '|' (bitwise OR), and '^' (bitwise XOR, i.e. sum mod 2). These are mostly used for manipulating individual bits or small groups of bits inside larger words, as in the expression `x & 0x0f`, which strips off the bottom four bits stored in `x`.

Examples:

`x`	`y`	expression	value
0011	0101	`x&y`	0001
0011	0101	`x|y`	0111
0011	0101	`x^y`	0110
0011	0101	`~x`	1100

The shift operators `<<` and `>>` shift the bit sequence left or right: `x << y` produces the value $x \cdot 2^y$ (ignoring overflow); this is equivalent to shifting every bit in `x` `y` positions to the left and filling in `y` zeros for the missing positions. In the other direction, `x >> y` produces the value $\lfloor x \cdot 2^- y \rfloor$ by shifting `x` `y` positions to the right. The behavior of the right shift operator depends on whether `x` is unsigned or signed; for unsigned values, it shifts in zeros from the left end always; for signed values, it shifts in additional copies of the leftmost bit (the sign bit). This makes `x >> y` have the same sign as `x` if `x` is signed.

If `y` is negative, the behavior of the shift operators is undefined.

Examples (`unsigned char x`):

x	y	x << y	x >> y
00000001	1	00000010	00000000
11111111	3	11111000	00011111

Examples (`signed char x`):

x	y	x << y	x >> y
00000001	1	00000010	00000000
11111111	3	11111000	11111111

Shift operators are often used with bitwise logical operators to set or extract individual bits in an integer value. The trick is that `(1 << i)` contains a 1 in the `i`-th least significant bit and zeros everywhere else. So `x & (1<<i)` is nonzero if and only if `x` has a 1 in the `i`-th place. This can be used to print out an integer in binary format (which standard `printf` won't do).

The following program gives an example of this technique. For example, when called as `./testPrintBinary 123`, it will print `111010` followed by a newline.

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

/* print out all bits of n */
void
print_binary(unsigned int n)
{
    unsigned int mask = 0;

    /* this grotesque hack creates a bit pattern 1000... */
    /* regardless of the size of an unsigned int */
    mask = ~mask ^ (~mask >> 1);

    for(; mask != 0; mask >>= 1) {
        putchar((n & mask) ? '1' : '0');
    }
}

int
main(int argc, char **argv)
{
    if(argc != 2) {
        fprintf(stderr, "Usage: %s n\n", argv[0]);
        return 1;
    }
```

```
    print_binary(atoi(argv[1]));
    putchar('\n');

    return 0;
}
```

examples/integerTypes/testPrintBinary.c

In the other direction, we can set the `i`-th bit of `x` to 1 by doing `x | (1 << i)` or to 0 by doing `x & ~(1 << i)`. See the section on bit manipulation. for applications of this to build arbitrarily-large bit vectors.

4.2.2.2.3 Logical operators

To add to the confusion, there are also three **logical** operators that work on the **truth-values** of integers, where 0 is defined to be false and anything else is defined by be true. These are `&&` (logical AND), `||`, (logical OR), and `!` (logical NOT). The result of any of these operators is always 0 or 1 (so `!!x`, for example, is 0 if `x` is 0 and 1 if `x` is anything else). The `&&` and `||` operators evaluate their arguments left-to-right and ignore the second argument if the first determines the answer (this is the only place in C where argument evaluation order is specified); so

```
    0 && executeProgrammer();
    1 || executeProgrammer();
```

is in a very weak sense perfectly safe code to run.

Watch out for confusing `&` with `&&`. The expression `1 & 2` evaluates to 0, but `1 && 2` evaluates to 1. The statement `0 & executeProgrammer();` is also unlikely to do what you want.

Yet another logical operator is the **ternary operator** `?:`, where `x ? y : z` equals the value of `y` if `x` is nonzero and `z` if `x` is zero. Like `&&` and `||`, it only evaluates the arguments it needs to:

```
    fileExists(badFile) ? deleteFile(badFile) : createFile(badFile);
```

Most uses of `?:` are better done using an if-then-else statement.

4.2.2.2.4 Relational operators

Logical operators usually operate on the results of **relational operators** or comparisons: these are `==` (equality), `!=` (inequality), `<` (less than), `>` (greater than), `<=` (less than or equal to) and `>=` (greater than or equal to). So, for example,

```
    if(size >= MIN_SIZE && size <= MAX_SIZE) {
        puts("just right");
    }
```

tests if `size` is in the (inclusive) range [`MIN_SIZE`..`MAX_SIZE`].

Beware of confusing `==` with `=`. The code

```
/* DANGER! DANGER! DANGER! */
if(x = 5) {
    ...
```

is perfectly legal C, and will set `x` to 5 rather than testing if it's equal to 5. Because 5 happens to be nonzero, the body of the `if` statement will always be executed. This error is so common and so dangerous that `gcc` will warn you about any tests that look like this if you use the `-Wall` option. Some programmers will go so far as to write the test as `5 == x` just so that if their finger slips, they will get a syntax error on `5 = x` even without special compiler support.

4.2.2.3 Converting to and from strings

To input or output integer values, you will need to convert them from or to strings. Converting from a string is easy using the `atoi` or `atol` functions declared in `stdlib.h`; these take a string as an argument and return an `int` or `long`, respectively. C99 also provides `atoll` for converting to `long long`. These routines have no ability to signal an error other than returning 0, so if you do `atoi("Sweden")`, that's what you'll get.

Output is usually done using `printf` (or `sprintf` if you want to write to a string without producing output). Use the `%d` format specifier for `int`s, `short`s, and `char`s that you want the numeric value of, `%ld` for `long`s, and `%lld` for `long long`s.

A contrived program that uses all of these features is given below:

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

/* This program can be used to show how atoi etc. handle overflow. */
/* For example, try "overflow 1000000000000". */
int
main(int argc, char **argv)
{
    char c;
    int i;
    long l;
    long long ll;

    if(argc != 2) {
        fprintf(stderr, "Usage: %s n\n", argv[0]);
        return 1;
    }

    c = atoi(argv[1]);
```

```c
    i = atoi(argv[1]);
    l = atol(argv[1]);
    ll = atoll(argv[1]);

    printf("char: %d  int: %d  long: %ld  long long: %lld", c, i, l, ll);

    return 0;
}
```

examples/integerTypes/overflow.c

4.2.3 Floating-point types

Real numbers are represented in C by the **floating point** types `float`, `double`, and `long double`. Just as the integer types can't represent all integers because they fit in a bounded number of bytes, so also the floating-point types can't represent all real numbers. The difference is that the integer types can represent values within their range exactly, while floating-point types almost always give only an approximation to the correct value, albeit across a much larger range. The three floating point types differ in how much space they use (32, 64, or 80 bits on x86 CPUs; possibly different amounts on other machines), and thus how much precision they provide. Most math library routines expect and return `double`s (e.g., `sin` is declared as `double sin(double)`, but there are usually `float` versions as well (`float sinf(float)`).

4.2.3.1 Floating point basics The core idea of floating-point representations (as opposed to **fixed point representations** as used by, say, `int`s), is that a number x is written as $m \cdot b^e$ where m is a **mantissa** or fractional part, b is a **base**, and e is an **exponent**. On modern computers the base is almost always 2, and for most floating-point representations the mantissa will be scaled to be between 1 and b. This is done by adjusting the exponent, e.g.

$1 = 1 \cdot 2^0$
$2 = 1 \cdot 2^1$
$0.375 = 1.5 \cdot 2^{-2}$

etc.

The mantissa is usually represented in base 2, as a binary fraction. So (in a very low-precision format), \$1 would be $1.000 \cdot 2^0$, 2 would be $1.000 \cdot 2^1$, and $0.375 = 3/8$ would be $1.100 \cdot 2^{-2}$, where the first 1 after the decimal point counts as 1/2, the second as 1/4, etc. Note that for a properly-scaled (or **normalized**) floating-point number in base 2 the digit before the decimal point is always 1. For this reason it is usually dropped to save space (although this requires a special representation for 0).

Negative values are typically handled by adding a **sign bit** that is 0 for positive numbers and 1 for negative numbers.

4.2.3.2 Floating-point constants

Any number that has a decimal point in it will be interpreted by the compiler as a floating-point number. Note that you have to put at least one digit after the decimal point: `2.0`, `3.75`, `-12.6112`. You can specific a floating point number in scientific notation using `e` for the exponent: `6.022e23`.

4.2.3.3 Operators

Floating-point types in C support most of the same arithmetic and relational operators as integer types; `x > y`, `x / y`, `x + y` all make sense when `x` and `y` are `float`s. If you mix two different floating-point types together, the less-precise one will be extended to match the precision of the more-precise one; this also works if you mix integer and floating point types as in `2 / 3.0`. Unlike integer division, floating-point division does not discard the fractional part (although it may produce round-off error: `2.0/3.0` gives `0.66666666666666663`, which is not quite exact). Be careful about accidentally using integer division when you mean to use floating-point division: `2/3` is `0`. Casts can be used to force floating-point division (see below).

Some operators that work on integers will *not* work on floating-point types. These are `%` (use `modf` from the math library if you really need to get a floating-point remainder) and all of the bitwise operators `~`, `<<`, `>>`, `&`, `^`, and `|`.

4.2.3.4 Conversion to and from integer types

Mixed uses of floating-point and integer types will convert the integers to floating-point.

You can convert floating-point numbers to and from integer types explicitly using casts. A typical use might be:

```
/* return the average of a list */
double
average(int n, int a[])
{
    int sum = 0;
    int i;

    for(i = 0; i < n; i++) {
        sum += a[i];
    }

    return (double) sum / n;
}
```

If we didn't put in the `(double)` to convert `sum` to a `double`, we'd end up doing integer division, which would truncate the fractional part of our average. Note that casts bind tighter than arithmetic operations, so the `(double)` applies to just `sum`, and not the whole expression `sum / n`.

In the other direction, we can write:

```
i = (int) f;
```

to convert a `float f` to `int i`. This conversion loses information by throwing away the fractional part of `f`: if `f` was `3.2`, `i` will end up being just `3`.

The math library contains a pile of functions for converting values of type `double` to integer values of type `double` that give more control over the rounding: see for example the descriptions of `floor`, `ceil`, `round`, `trunc`, and `nearbyint` in the GNU libc reference manual.

4.2.3.5 The IEEE-754 floating-point standard

The IEEE-754 floating-point standard is a standard for representing and manipulating floating-point quantities that is followed by all modern computer systems. It defines several standard representations of floating-point numbers, all of which have the following basic pattern (the specific layout here is for 32-bit `float`s):

```
bit  31 30     23 22                    0
     S  EEEEEEEE MMMMMMMMMMMMMMMMMMMMMMM
```

The bit numbers are counting from the least-significant bit. The first bit is the sign (0 for positive, 1 for negative). The following 8 bits are the exponent in **excess-127** binary notation; this means that the binary pattern 01111111 = 127 represents an exponent of 0, 10000000 = 128 represents 1, 01111110 = 126 represents -1, and so forth. The mantissa fits in the remaining 24 bits, with its leading 1 stripped off as described above.

Certain numbers have a special representation. Because 0 cannot be represented in the standard form (there is no 1 before the decimal point), it is given the special representation `0 00000000 00000000000000000000000`. (There is also a -0 = `1 00000000 00000000000000000000000`, which looks equal to +0 but prints differently.) Numbers with exponents of 11111111 = 255 = 2^{128} represent non-numeric quantities such as "not a number" (`NaN`), returned by operations like (`0.0/0.0`) and positive or negative infinity. A table of some typical floating-point numbers (generated by the program float.c) is given below:

```
        0 =                        0 = 0 00000000 00000000000000000000000
       -0 =                       -0 = 1 00000000 00000000000000000000000
    0.125 =                    0.125 = 0 01111100 00000000000000000000000
     0.25 =                     0.25 = 0 01111101 00000000000000000000000
      0.5 =                      0.5 = 0 01111110 00000000000000000000000
        1 =                        1 = 0 01111111 00000000000000000000000
        2 =                        2 = 0 10000000 00000000000000000000000
```

```
       4 =                     4 = 0 10000001 00000000000000000000000
       8 =                     8 = 0 10000010 00000000000000000000000
   0.375 =                 0.375 = 0 01111101 10000000000000000000000
    0.75 =                  0.75 = 0 01111110 10000000000000000000000
     1.5 =                   1.5 = 0 01111111 10000000000000000000000
       3 =                     3 = 0 10000000 10000000000000000000000
       6 =                     6 = 0 10000001 10000000000000000000000
     0.1 =   0.10000000149011612 = 0 01111011 10011001100110011001101
     0.2 =   0.20000000298023224 = 0 01111100 10011001100110011001101
     0.4 =   0.40000000596046448 = 0 01111101 10011001100110011001101
     0.8 =   0.80000001192092896 = 0 01111110 10011001100110011001101
   1e+12 =          999999995904 = 0 10100110 11010001101010010100101
   1e+24 = 1.0000000138484279e+24 = 0 11001110 10100111100001000011100
   1e+36 = 9.9999996169031625e+35 = 0 11110110 10000001001011111001110
     inf =                   inf = 0 11111111 00000000000000000000000
    -inf =                  -inf = 1 11111111 00000000000000000000000
     nan =                   nan = 0 11111111 10000000000000000000000
```

What this means in practice is that a 32-bit floating-point value (e.g. a `float`) can represent any number between `1.17549435e-38` and `3.40282347e+38`, where the `e` separates the (base 10) exponent. Operations that would create a smaller value will underflow to 0 (slowly—IEEE 754 allows "denormalized" floating point numbers with reduced precision for very small values) and operations that would create a larger value will produce `inf` or `-inf` instead.

For a 64-bit `double`, the size of both the exponent and mantissa are larger; this gives a range from `1.7976931348623157e+308` to `2.2250738585072014e-308`, with similar behavior on underflow and overflow.

Intel processors internally use an even larger 80-bit floating-point format for all operations. Unless you declare your variables as `long double`, this should not be visible to you from C except that some operations that might otherwise produce overflow errors will not do so, provided all the variables involved sit in registers (typically the case only for local variables and function parameters).

4.2.3.6 Error In general, floating-point numbers are not exact: they are likely to contain **round-off error** because of the truncation of the mantissa to a fixed number of bits. This is particularly noticeable for large values (e.g. `1e+12` in the table above), but can also be seen in fractions with values that aren't powers of 2 in the denominator (e.g. `0.1`). Round-off error is often invisible with the default float output formats, since they produce fewer digits than are stored internally, but can accumulate over time, particularly if you subtract floating-point quantities with values that are close (this wipes out the mantissa without wiping out the error, making the error much larger relative to the number that remains).

The easiest way to avoid accumulating error is to use high-precision floating-point numbers (this means using `double` instead of `float`). On modern CPUs there is little or no time penalty for doing so, although storing `double`s instead of `float`s will take twice as much space in memory.

Note that a consequence of the internal structure of IEEE 754 floating-point numbers is that small integers and fractions with small numerators and power-of-2 denominators can be represented *exactly*—indeed, the IEEE 754 standard carefully defines floating-point operations so that arithmetic on such exact integers will give the same answers as integer arithmetic would (except, of course, for division that produces a remainder). This fact can sometimes be exploited to get higher precision on integer values than is available from the standard integer types; for example, a `double` can represent any integer between -2^{53} and 2^{53} exactly, which is a much wider range than the values from 2^{-31} to $2^{31}-1$ that fit in a 32-bit `{.backtick}int` or `{.backtick}long`. (A 64-bit `{.backtick}long long` does better.) So `{.backtick}double\` should be considered for applications where large precise integers are needed (such as calculating the net worth in pennies of a billionaire.)

One consequence of round-off error is that it is very difficult to test floating-point numbers for equality, unless you are sure you have an exact value as described above. It is generally not the case, for example, that `(0.1+0.1+0.1) == 0.3` in C. This can produce odd results if you try writing something like `for(f = 0.0; f <= 0.3; f += 0.1)`: it will be hard to predict in advance whether the loop body will be executed with `f = 0.3` or not. (Even more hilarity ensues if you write `for(f = 0.0; f != 0.3; f += 0.1)`, which after not quite hitting `0.3` exactly keeps looping for much longer than I am willing to wait to see it stop, but which I suspect will eventually converge to some constant value of `f` large enough that adding `0.1` to it has no effect.) Most of the time when you are tempted to test floats for equality, you are better off testing if one lies within a small distance from the other, e.g. by testing `fabs(x-y) <= fabs(EPSILON * y)`, where `EPSILON` is usually some application-dependent tolerance. This isn't quite the same as equality (for example, it isn't transitive), but it usually closer to what you want.

4.2.3.7 Reading and writing floating-point numbers Any numeric constant in a C program that contains a decimal point is treated as a `double` by default. You can also use `e` or `E` to add a base-10 exponent (see the table for some examples of this.) If you want to insist that a constant value is a `float` for some reason, you can append `F` on the end, as in `1.0F`.

For I/O, floating-point values are most easily read and written using `scanf` (and its relatives `fscanf` and `sscanf`) and `printf`. For `printf`, there is an elaborate variety of floating-point format codes; the easiest way to find out what these do is experiment with them. For `scanf`, pretty much the only two codes you need are `"%lf"`, which reads a `double` value into a `double *`, and `"%f"`, which reads a `float` value into a `float *`. Both these formats are exactly the same in `printf`, since a `float` is promoted to a `double` before being passed as an argument to `printf` (or any other function that doesn't declare the type of its arguments). But you have to be careful with the arguments to `scanf` or you will get odd results as only 4 bytes of your 8-byte `double` are filled in, or—even worse—8 bytes of your 4-byte `float` are.

4.2.3.8 Non-finite numbers in C

The values `nan`, `inf`, and `-inf` can't be written in this form as floating-point constants in a C program, but `printf` will generate them and `scanf` seems to recognize them. With some machines and compilers you may be able to use the macros `INFINITY` and `NAN` from `<math.h>` to generate infinite quantities. The macros `isinf` and `isnan` can be used to detect such quantities if they occur.

4.2.3.9 The math library

(See also K&R Appendix B4.)

Many mathematical functions on floating-point values are not linked into C programs by default, but can be obtained by linking in the math library. Examples would be the trigonometric functions `sin`, `cos`, and `tan` (plus more exotic ones), `sqrt` for taking square roots, `pow` for exponentiation, `log` and `exp` for base-e logs and exponents, and `fmod` for when you really want to write `x%y` but one or both variables is a `double`. The standard math library functions all take `double`s as arguments and return `double` values; most implementations also provide some extra functions with similar names (e.g., `sinf`) that use `float`s instead, for applications where space or speed is more important than accuracy.

There are two parts to using the math library. The first is to include the line

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

somewhere at the top of your source file. This tells the preprocessor to paste in the declarations of the math library functions found in `/usr/include/math.h`.

The second step is to link to the math library when you compile. This is done by passing the flag `-lm` to `gcc` *after* your C program source file(s). A typical command might be:

```
c99 -o program program.c -lm
```

If you don't do this, you will get errors from the compiler about missing functions. The reason is that the math library is not linked in by default, since for many system programs it's not needed.

4.3 Operator precedence

Operator precedence in C controls the interpretation of ambiguous expressions like `2+3*4`, which could in principle be parsed either as `2+(3*4)` (the right way) or as `(2+3)*4` (the cheap calculator way). For the most part, C parses unparenthesized expressions the right way, but if you are not sure what it will do with an expression, you can always put in parentheses to force it to do the right thing.

There is a table on page 53 of Kernighan and Ritchie that shows the precedence of all operators in C, which we reproduce below.

The interpretation of this table is that higher entries bind tighter than lower ones; so the fact that `*` has higher precedence that `+` and both have higher precedence that `>` means that `2+3*4 > 5` gets parsed as `(2+(3*4)) > 5`.

Associativity controls how an expression with multiple operators of the same precedence is interpreted. The fact that `+` and `-` associate left-to-right means that the expression `2+3-4-5` is interpreted as `(((2+3)-4)-5)`: the leftmost operation is done first. Unary operators, ternary `?:` and assignment operators are the only ones that associate right-to-left. For assignment operators, this is so `x = y = 0` is interpreted as `x = (y = 0)` (assigning `0` to both `x` and `y`) and not `(x = y) = 0` (which would give an error because `(x = y)` isn't something you can assign to). For unary operators, this mostly affects expressions like `*p++`, which is equivalent to `*(p++)` (increment the pointer first then dereference it) rather than `(*p)++` (increment the thing that `p` points to).

`() [] -> .`	function calls and indexing
`! ~ -` (unary) `*` (unary) `&`(unary) `++ --` `(`*type*`)` `sizeof`	unary operators (associate right-to-left)
`*` (binary) `/ %`	multiplication and division
`+` (binary) `-` (binary)	addition and subtraction
`<< >>`	shifts
`< <= >= >`	inequalities
`== !=`	equality
`&` (binary)	bitwise AND
`^`	bitwise XOR
`|`	bitwise OR
`&&`	logical AND
`||`	logical OR
`?:`	ternary if (associates right-to-left)
`= += -= *= /= %= &= ^= |= <<= >>=`	assignment (associate right-to-left)
`,`	comma

4.4 Programming style

The C programming language imposes very few constraints on how programs are formatted and organized. Both of the following are legitimate C programs, which compile to exactly the same machine code using `gcc` with a high enough optimization level:

```
/*
 * Count down from COUNTDOWN_START (defined below) to 0.
 * Prints all numbers in the range including both endpoints.
 */

#include <stdio.h>

#define COUNTDOWN_START (10)

int
main(int argc, char **argv)
{
    for(int i = COUNTDOWN_START; i >= 0; i--) {
        printf("%d\n", i);
    }

    return 0;
}
```

examples/style/countdown.c

```
#include <stdio.h>
int main(int _,char**xb){_=0xb;while(_--)printf("%d\n",_);return ++_;}
```

examples/style/badCountdown.c

The difference between these programs is that the first is designed to be easy to read and understand while the second is not. Though the compiler can't tell the difference between them, the second will be much harder to debug or modify to accomplish some new task.

Certain formatting and programming conventions have evolved over the years to make C code as comprehensible as possible, and as we introduce various features of C, we will talk about how best to use them to make your programs understood by both computers and humans.

Submitted assignments may be graded for style in addition to correctness. Below is a checklist that has been used in past versions of the course to identify some of the more egregious violations of reasonable coding practice. For more extreme examples of what not to do, see the International Obfuscated C Code Contest.

```
Style grading checklist
Score is 20 points minus 1 for each box checked (but never less than 0)

Comments

[ ] Undocumented module.
```

```
[ ] Undocumented function other than main.
[ ] Underdocumented function: return value or args not described.
[ ] Undocumented program input and output (when main is provided).
[ ] Undocumented struct or union components.
[ ] Undocumented #define.
[ ] Failure to cite code taken from other sources.
[ ] Insufficient comments.
[ ] Excessive comments.

Naming

[ ] Meaningless function name.
[ ] Confusing variable name.
[ ] Inconsistent variable naming style (UgLyName, ugly_name, NAME___UGLY_1).
[ ] Inconsistent use of capitalization to distinguish constants.

Whitespace

[ ] Inconsistent or misleading indentation.
[ ] Spaces not used or used misleadingly to break up complicated expressions.
[ ] Blank lines not used or used misleadingly to break up long function bodies.

Macros

[ ] Non-trivial constant with no symbolic name.
[ ] Failure to parenthesize expression in macro definition.
[ ] Dependent constant not written as expression of earlier constant.
[ ] Underdocumented parameterized macro.

Global variables

[ ] Inappropriate use of a global variable.

Functions

[ ] Kitchen-sink function that performs multiple unrelated tasks.
[ ] Non-void function that returns no useful value.
[ ] Function with too many arguments.

Code organization

[ ] Lack of modularity.
[ ] Function used in multiple source files but not declared in header file.
[ ] Internal-use-only function not declared static.
[ ] Full struct definition in header files when components should be hidden.
[ ] #include "file.c"
```

```
[ ] Substantial repetition of code.

Miscellaneous

[ ] Other obstacle to readability not mentioned above.
```

4.5 Variables

Variables in C are a direct abstraction of physical memory locations. To understand how variables work, it helps to start by understanding how computer memory works.

4.5.1 Memory

Memory consists of many bytes of storage, each of which has an address which is itself a sequence of bits. Though the actual memory architecture of a modern computer is complex, from the point of view of a C program we can think of as simply a large **address space** that the CPU can store things in (and load things from), provided it can supply an address to the memory. Because we don't want to have to type long strings of bits all the time, the C compiler lets us give names to particular regions of the address space, and will even find free space for us to use.

4.5.2 Variables as names

A **variable** is a name given in a program for some region of memory. Each variable has a **type**, which tells the compiler how big the region of memory corresponding to it is and how to treat the bits stored in that region when performing various kinds of operations (e.g. integer variables are added together by very different circuitry than floating-point variables, even though both represent numbers as bits). In modern programming languages, a variable also has a **scope** (a limit on where the name is meaningful, which allows the same name to be used for different variables in different parts of the program) and an **extent** (the duration of the variable's existence, controlling when the program allocates and deallocates space for it).

4.5.2.1 Variable declarations Before you can use a variable in C, you must **declare** it. Variable declarations show up in three places:

- Outside a function. These declarations declare **global variables** that are visible throughout the program (i.e. they have **global scope**). Use of global variables is almost always a mistake.

- In the argument list in the header of a function. These variables are **parameters** to the function. They are only visible inside the function body (**local scope**), exist only from when the function is called to when the function returns (**bounded extent**—note that this is different from what happens in some garbage-collected languages like Scheme), and get their initial values from the arguments to the function when it is called.
- Inside a function. (Before C99, only at the start of a block delimited by curly braces.) Such variables are visible only within the block in which they are declared (local scope again) and exist only when the containing function is active (bounded extent). The convention in C is has generally been to declare all such **local variables** at the top of a function; this is different from the convention in C++ or Java, which encourage variables to be declared when they are first used. This convention may be less strong in C99 code, since C99 adopts the C++ rule of allowing variables to be declared anywhere (which can be particularly useful for index variables in `for` loops).

Another feature of function parameters and local variables is that if a function is called more than once (even if the function calls itself), each copy of the function gets its own local variables.

Variable declarations consist of a type name followed by one or more variable names separated by commas and terminated by a semicolon (except in argument lists, where each declaration is terminated by a comma). I personally find it easiest to declare variables one per line, to simplify documenting them. It is also possible for global and local variables (but not function arguments) to assign an initial value to a variable by putting in something like `= 0` after the variable name. It is good practice to put a comment after each variable declaration that explains what the variable does (with a possible exception for conventionally-named loop variables like `i` or `j` in short functions). Below is an example of a program with some variable declarations in it:

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

/* This program counts the number of digits in its input. */

/*
 *This global variable is not used; it is here only to demonstrate
 * what a global variable declaration looks like.
 */
unsigned long SpuriousGlobalVariable = 127;

int
main(int argc, char **argv)
{
    int c;              /* character read */
```

```
    int count = 0;      /* number of digits found */

    while((c = getchar()) != EOF) {
        if(isdigit(c)) {
            count++;
        }
    }

    printf("%d\n", count);

    return 0;
}
```

examples/variables/countDigits.c

4.5.2.2 Variable names

The evolution of variable names in different programming languages:

11101001001001 Physical addresses represented as bits.

`#FC27` Typical assembly language address represented in hexadecimal to save typing (and because it's easier for humans to distinguish #A7 from #B6 than to distinguish 10100111 from 10110110.)

`A1$` A string variable in BASIC, back in the old days where BASIC variables were one uppercase letter, optionally followed by a number, optionally followed by $ for a string variable and % for an integer variable. These type tags were used because BASIC interpreters didn't have a mechanism for declaring variable types.

`IFNXG7` A typical FORTRAN variable name, back in the days of 6-character all-caps variable names. The `I` at the start means it's an integer variable. The rest of the letters probably abbreviate some much longer description of what the variable means. The default type based on the first letter was used because FORTRAN programmers were lazy, but it could be overridden by an explicit declaration.

`i, j, c, count, top_of_stack, accumulatedTimeInFlight` Typical names from modern C programs. There is no type information contained in the name; the type is specified in the declaration and remembered by the compiler elsewhere. Note that there are two different conventions for representing multi-word names: the first is to replace spaces with underscores, and the second is to capitalize the first letter of each word (possibly excluding the first letter), a style called camel case. You should pick **one** of these two conventions and stick to it.

`prgcGradeDatabase` An example of **Hungarian notation**, a style of variable naming in which the type of the variable is encoded in the first few character.

> The type is now back in the variable name again. This is *not* enforced by the compiler: even though `iNumberOfStudents` is supposed to be an `int`, there is nothing to prevent you from declaring `float iNumberOfStudents` if you are teaching a class on improper chainsaw handling and want to allow for the possibility of fractional students. See this MSDN page for a much more detailed explanation of the system.
>
> Not clearly an improvement on standard naming conventions, but it is popular in some programming shops.

In C, variable names are called **identifiers**. These are also used to identify things that are not variables, like functions and user-defined types.

An identifier in C must start with a lower or uppercase letter or the underscore character `_`. Typically variables starting with underscores are used internally by system libraries, so it's dangerous to name your own variables this way. Subsequent characters in an identifier can be letters, digits, or underscores. So for example `a`, `____a___a_a_11727_a`, `AlbertEinstein`, `aAaAaAaAaAAAAAa`, and `______` are all legal identifiers in C, but `$foo` and `01` are not.

The basic principle of variable naming is that a variable name is a substitute for the programmer's memory. It is generally best to give identifiers names that are easy to read and describe what the variable is used for. Such variables are called **self-documenting**. None of the variable names in the preceding list are any good by this standard. Better names would be `total_input_characters`, `dialedWrongNumber`, or `stepsRemaining`. Non-descriptive single-character names are acceptable for certain conventional uses, such as the use of `i` and `j` for loop iteration variables, or `c` for an input character. Such names should only be used when the scope of the variable is small, so that it's easy to see all the places where it is used at the same time.

C identifiers are case-sensitive, so `aardvark`, `AArDvARK`, and `AARDVARK` are all different variables. Because it is hard to remember how you capitalized something before, it is important to pick a standard convention and stick to it. The traditional convention in C goes like this:

- Ordinary variables and functions are lowercased or camel-cased, e.g. `count`, `countOfInputBits`.
- User-defined types (and in some conventions global variables) are capitalized, e.g. `Stack`, `TotalBytesAllocated`.
- Constants created with `#define` or `enum` are put in all-caps: `MAXIMUM_STACK_SIZE`, `BUFFER_LIMIT`.

4.5.3 Using variables

Ignoring pointers for the moment, there are essentially two things you can do to a variable. You can assign a value to it using the `=` operator, as in:

```
x = 2;        /* assign 2 to x */
y = 3;        /* assign 3 to y */
```

or you can use its value in an expression:

```
x = y+1;      /* assign y+1 to x */
```

The assignment operator is an ordinary operator, and assignment expressions can be used in larger expressions:

```
x = (y=2)*3; /* sets y to 2 and x to 6 */
```

This feature is usually only used in certain standard idioms, since it's confusing otherwise.

There are also shorthand operators for expressions of the form *variable* `=` *variable operator expression*. For example, writing `x += y` is equivalent to writing `x = x + y`, `x /= y` is the same as `x = x / y`, etc.

For the special case of adding or subtracting 1, you can abbreviate still further with the `++` and `--` operators. These come in two versions, depending on whether you want the result of the expression (if used in a larger expression) to be the value of the variable before or after the variable is incremented:

```
x = 0;
y = x++;      /* sets x to 1 and y to 0 (the old value) */
y = ++x;      /* sets x to 2 and y to 2 (the new value) */
y = x--;      /* sets x to 1 and y to 2 (the old value) */
y = --x;      /* sets x to 0 and y to 0 (the new value) */
```

The intuition is that if the `++` comes before the variable, the increment happens before the value of the variable is read (a **preincrement**; if it comes after, it happens after the value is read (a **postincrement**). This is confusing enough that it is best not to use the value of preincrement or postincrement operations except in certain standard idioms. But using `x++` or `++x` by itself as a substitute for `x = x+1` is perfectly acceptable style.[8]

4.5.4 Initialization

It is a serious error to use the value of a variable that has never been assigned to, because you will get whatever junk is sitting in memory at the address allocated to the variable, and this might be some arbitrary leftover value from a previous function call that doesn't even represent the same type.[9]

[8] C++ programmers will prefer `++x` if they are not otherwise using the return value, because if `x` is some very complicated type with overloaded `++`, using preincrement avoids having to save a copy of the old value.

[9] Exception: Global variables and static local variables are guaranteed to be initialized to an all-0 pattern, which will give the value 0 for most types.

Fortunately, C provides a way to guarantee that a variable is initialized as soon as it is declared. Many of the examples in the notes do not use this mechanism, because of bad habits learned by the instructor using early versions of C that imposed tighter constraints on initialization. But initializing variables is a good habit to get in the practice of doing.

For variables with simple types (that is, not arrays, structs, or unions), an initializer looks like an assignment:

```
int sum = 0;
int n = 100;
int nSquared = n*n;
double gradeSchoolPi = 3.14;
const char * const greeting = "Hi!";
const int greetingLength = strlen(greeting);
```

For ordinary local variables, the initializer value can be any expression, including expressions that call other functions. There is an exception for variables allocated when the program starts (which includes global variables outside functions and `static` variables inside functions), which can only be initialized to constant expressions.

The last two examples show how initializers can set the values of variables that are declared to be `const` (the variable `greeting` is both constant itself, because of `const greeting`, and points to data that is also constant, because it is of type `const char`). This is the only way to set the values of such variables without cheating, because the compiler will complain if you try to do an ordinary assignment to a variable declared to be constant.

For fixed-size arrays and structs, it is possible to supply an initializer for each component, by enclosing the initializer values in braces, separated by commas. For example:

```
int threeNumbers[3] = { 1, 2, 3 };

struct numericTitle {
    int number;
    const char *name;
};

struct numericTitle s = { 7, "Samurai" };
struct numericTitle n = { 3, "Ninjas" };
```

4.5.5 Storage class qualifiers

It is possible to specify additional information about how a variable can be used using **storage class qualifiers**, which usually go before the type of a variable in a declaration.

4.5.5.1 Scope and extent

Most variables that you will use in C are either parameters to functions or local variables inside functions. These have **local scope**, meaning the variable names can only be used in the function in which they are declared, and **automatic extent**, meaning the space for the variable is allocated, typically on the stack, when the function is called, and reclaimed when the function exits. (If the function calls itself, you get another copy of all the local variables; see recursion.)

On *very rare* occasions you might want to have a variable that survives the entire execution of a program (has **static extent**) or that is visible throughout the program (has **global scope**). C provides a mechanism for doing this *that you shold never use under normal circumstances.* Pretty much the only time you are going to want to have a variable with static extent is if you are keeping track of some piece of information that (a) you only need one instance of, (b) you need to survive between function calls, and (c) it would be annoying to pass around as an extra argument to any function that uses it. An example would be the internal data structures used by `malloc`, or the count variable in the function below:

```
/* returns the number of times this function has previously been called */
/* this can be used to generate unique numerical identifiers */
unsigned long long
ticketMachine(void)
{
    static unsigned long long count = 0;

    return count++;
}
```

To declare a local variable with static extent, use the `static` qualifier as in the above example. To declare a global variable with static extent, declare it outside a function. In both cases you should provide an initializer for the variable.

4.5.5.1.1 Additional qualifiers for global variables

It is possible to put some additional constraints on the visibility of global variables. By default, a global variable will be visible everywhere, but functions files other than the one in which it is defined won't necessarily know what type it has. This latter problem can be fixed using an `extern` declaration, which says that there is a variable somewhere else of a particular type that we are declaring (but not defining, so no space is allocated). In contrast, the `static` keyword (on a global variable) specifies that it will only be visible in the current file, even if some other file includes a declaration of a global variable of the same name.

Here are three variable declarations that illustrate how this works:

```
    unsigned short Global = 5;    /* global variable, can be used anywhere */
```

```
extern float GlobalFloat;     /* this global variable, defined somewhere else, has type

static char Character = 'c';  /* global variable, can only be used by functions in this
```

(Note the convention of putting capital letters on global variables to distinguish them from local variables.)

Typically, an `extern` definition would appear in a header file so that it can be included in any function that uses the variable, while an ordinary global variable definition would appear in a C file so it only occurs once.

4.5.6 Marking variables as constant

The **const** qualifier declares a variable to be constant:

```
const int three = 3;   /* this will always be 3 */
```

It is an error to apply any sort of assignment (`=`, `+=`, `++`, etc.) to a variable qualified as `const`.

4.5.6.1 Pointers to `const`

A pointer to a region that should not be modified should be declared with `const` type:

```
const char *string = "You cannot modify this string.";
```

The `const` in the declaration above applies to the characters that `string` points to: `string` is not `const` itself, but is instead a *pointer to* `const`. It is still possible to make `string` point somewhere else, say by doing an assignment:

```
string = "You cannot modify this string either."
```

If you want to make it so that you can't assign to `string`, put `const` right before the variable name:

```
/* prevent assigning to string as well */
const char * const string = "You cannot modify this string.";
```

Now `string` is a `const` pointer to `const`: you can neither modify `string` nor the values it points to.

Note that `const` only restricts what you can do using this particular variable name. If you can get at the memory that something points to by some other means, say through another pointer, you may be able to change the values in these memory locations anyway:

```
int x = 5;
const int *p = &x;
int *q;
```

```c
*p = 1; /* will cause an error at compile time */
x = 3;  /* also changes *p, but will not cause an error */
```

4.6 Input and output

Input and output from C programs is typically done through the **standard I/O library**, whose functions etc. are declared in `stdio.h`. A detailed descriptions of the functions in this library is given in Appendix B of Kernighan and Ritchie. We'll talk about some of the more useful functions and about how input-output (I/O) works on Unix-like operating systems in general.

4.6.1 Character streams

The standard I/O library works on **character streams**, objects that act like long sequences of incoming or outgoing characters. What a stream is connected to is often not apparent to a program that uses it; an output stream might go to a terminal, to a file, or even to another program (appearing there as an input stream).

Three standard streams are available to all programs: these are `stdin` (standard input), `stdout` (standard output), and `stderr` (standard error). Standard I/O functions that do not take a stream as an argument will generally either read from `stdin` or write to `stdout`. The `stderr` stream is used for error messages. It is kept separate from `stdout` so that you can see these messages even if you redirect output to a file:

```
$ ls no-such-file > /tmp/dummy-output
ls: no-such-file: No such file or directory
```

4.6.2 Reading and writing single characters

To read a single character from `stdin`, use `getchar`:

```c
int c;

c = getchar();
```

The `getchar` routine will return the special value `EOF` (usually -1; short for *end of file*) if there are no more characters to read, which can happen when you hit the end of a file or when the user types the end-of-file key control-D to the terminal. Note that the return value of `getchar` is declared to be an `int` since `EOF` lies outside the normal character range.

To write a single character to `stdout`, use `putchar`:

```c
putchar('!');
```

Even though putchar can only write single bytes, it takes an int as an argument. Any value outside the range 0..255 will be truncated to its last byte, as in the usual conversion from int to unsigned char.

Both getchar and putchar are wrappers for more general routines getc and putc that allow you to specify which stream you are using. To illustrate getc and putc, here's how we might define getchar and putchar if they didn't exist already:

```
int
getchar2(void)
{
    return getc(stdin);
}

int
putchar2(int c)
{
    return putc(c, stdout);
}
```

Note that putc, putchar2 as defined above, and the original putchar all return an int rather than void; this is so that they can signal whether the write succeeded. If the write succeeded, putchar or putc will return the value written. If the write failed (say because the disk was full), then putc or putchar will return EOF.

Here's another example of using putc to make a new function putcerr that writes a character to stderr:

```
int
putcerr(int c)
{
    return putc(c, stderr);
}
```

A rather odd feature of the C standard I/O library is that if you don't like the character you just got, you can put it back using the ungetc function. The limitations on ungetc are that (a) you can only push one character back, and (b) that character can't be EOF. The ungetc function is provided because it makes certain high-level input tasks easier; for example, if you want to parse a number written as a sequence of digits, you need to be able to read characters until you hit the first non-digit. But if the non-digit is going to be used elsewhere in your program, you don't want to eat it. The solution is to put it back using ungetc.

Here's a function that uses ungetc to peek at the next character on stdin without consuming it:

```
/* return the next character from stdin without consuming it */
int
```

```c
peekchar(void)
{
    int c;

    c = getchar();
    if(c != EOF) ungetc(c, stdin);      /* puts it back */

    return c;
}
```

4.6.3 Formatted I/O

Reading and writing data one character at a time can be painful. The C standard I/O library provides several convenient routines for reading and writing formatted data. The most commonly used one is `printf`, which takes as arguments a format string followed by zero or more values that are filled in to the format string according to patterns appearing in it.

Here are some typical `printf` statements:

```c
    printf("Hello\n");          /* print "Hello" followed by a newline */
    printf("%c", c);            /* equivalent to putchar(c) */
    printf("%d", n);            /* print n (an int) formatted in decimal */
    printf("%u", n);            /* print n (an unsigned int) formatted in decimal */
    printf("%o", n);            /* print n (an unsigned int) formatted in octal */
    printf("%x", n);            /* print n (an unsigned int) formatted in hexadecimal */
    printf("%f", x);            /* print x (a float or double) */

    /* print total (an int) and average (a double) on two lines with labels */
    printf("Total: %d\nAverage: %f\n", total, average);
```

For a full list of formatting codes see Table B-1 in Kernighan and Ritchie, or run `man 3 printf`.

The inverse of `printf` is `scanf`. The `scanf` function reads formatted data from `stdin` according to the format string passed as its first argument and stuffs the results into variables whose *addresses* are given by the later arguments. This requires prefixing each such argument with the `&` operator, which takes the address of a variable.

Format strings for `scanf` are close enough to format strings for `printf` that you can usually copy them over directly. However, because `scanf` arguments don't go through argument promotion (where all small integer types are converted to `int` and `float`s are converted to `double`), you have to be much more careful about specifying the type of the argument correctly. For example, while `printf("%f", x)` will work whether `x` is a `float` or a `double`, `scanf("%f", &x)` will work only if `x` is a `float`, which means that `scanf("%lf", &x)` is needed if `x` is in fact a `double`.

Some examples:

```
scanf("%c", &c);           /* like c = getchar(); c must be a char; will NOT put EOF in
scanf("%d", &n);           /* read an int formatted in decimal */
scanf("%u", &n);           /* read an unsigned int formatted in decimal */
scanf("%o", &n);           /* read an unsigned int formatted in octal */
scanf("%x", &n);           /* read an unsigned int formatted in hexadecimal */
scanf("%f", &x);           /* read a float */
scanf("%lf", &x);          /* read a double */

/* read total (an int) and average (a float) on two lines with labels */
/* (will also work if input is missing newlines or uses other whitespace, see below) */
scanf("Total: %d\nAverage: %f\n", &total, &average);
```

For a full list of formatting codes, run `man 3 scanf`.

The `scanf` routine usually eats whitespace (spaces, tabs, newlines, etc.) in its input whenever it sees a conversion specification or a whitespace character in its format string. The one exception is that a `%c` conversion specifier will not eat whitespace and will instead return the next character whether it is whitespace or not. Non-whitespace characters that are not part of conversion specifications must match exactly. To detect if `scanf` parsed everything successfully, look at its return value; it returns the number of values it filled in, or `EOF` if it hits end-of-file before filling in any values.

The `printf` and `scanf` routines are wrappers for `fprintf` and `fscanf`, which take a stream as their first argument, e.g.:

```
fprintf(stderr, "BUILDING ON FIRE, %d%% BURNT!!!\n", percentage);
```

This sends the output the the standard error output handle `stderr`. Note the use of "%%" to print a single percent in the output.

4.6.4 Rolling your own I/O routines

Since we can write our own functions in C, if we don't like what the standard routines do, we can build our own on top of them. For example, here's a function that reads in integer values without leading minus signs and returns the result. It uses the `peekchar` routine we defined above, as well as the `isdigit` routine declared in `ctype.h`.

```
/* read an integer written in decimal notation from stdin until the first
 * non-digit and return it.  Returns 0 if there are no digits. */
int
readNumber(void)
{
```

```
    int accumulator;    /* the number so far */
    int c;              /* next character */

    accumulator = 0;

    while((c = peekchar()) != EOF && isdigit(c)) {
        c = getchar();                  /* consume it */
        accumulator *= 10;              /* shift previous digits over */
        accumulator += (c - '0');       /* add decimal value of new digit */
    }

    return accumulator;
}
```

Here's another implementation that does *almost* the same thing:

```
int
readNumber2(void)
{
    int n;

    if(scanf("%u", &n) == 1) {
        return n;
    } else {
        return 0;
    }
}
```

The difference is that `readNumber2` will consume any whitespace before the first digit, which may or may not be what we want.

More complex routines can be used to parse more complex input. For example, here's a routine that uses `readNumber` to parse simple arithmetic expressions, where each expression is either a number or of the form `(`*expression*`+`*expression*`)` or `(`*expression*`*`*expression*`)`. The return value is the value of the expression after adding together or multiplying all of its subexpressions. (A complete program including this routine and the others defined earlier that it uses can be found examples/IO/calc.c.

```
#define EXPRESSION_ERROR (-1)

/* read an expression from stdin and return its value */
/* returns EXPRESSION_ERROR on error */
int
readExpression(void)
{
    int e1;             /* value of first sub-expression */
    int e2;             /* value of second sub-expression */
```

```
    int c;
    int op;             /* operation: '+' or '*' */

    c = peekchar();

    if(c == '(') {
        c = getchar();

        e1 = readExpression();
        op = getchar();
        e2 = readExpression();

        c = getchar();  /* this had better be ')' */
        if(c != ')') return EXPRESSION_ERROR;

        /* else */
        switch(op) {
        case '*':
            return e1*e2;
            break;
        case '+':
            return e1+e2;
            break;
        default:
            return EXPRESSION_ERROR;
            break;
        }
    } else if(isdigit(c)) {
        return readNumber();
    } else {
        return EXPRESSION_ERROR;
    }
}
```

Because this routine calls itself recursively as it works its way down through the input, it is an example of a recursive descent parser. Parsers for more complicated languages like C are usually not written by hand like this, but are instead constructed mechanically using a Parser generator.

4.6.5 File I/O

Reading and writing files is done by creating new streams attached to the files. The function that does this is `fopen`. It takes two arguments: a filename, and a flag that controls whether the file is opened for reading or writing. The return value of `fopen` has type `FILE *` and can be used in `putc`, `getc`, `fprintf`, etc.

just like `stdin`, `stdout`, or `stderr`. When you are done using a stream, you should close it using `fclose`.

Here's a program that reads a list of numbers from a file whose name is given as `argv[1]` and prints their sum:

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

/*
 * Sum integers in a file.
 *
 * 2018-01-24 Includes bug fixes contributed by Zhe Hua.
 */

int
main(int argc, char **argv)
{
    FILE *f;
    int x;
    int sum;

    if(argc != 2) {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(1);
    }

    f = fopen(argv[1], "r");
    if(f == 0) {
        /* perror is a standard C library routine */
        /* that prints a message about the last failed library routine */
        /* prepended by its argument */
        perror(argv[1]);
        exit(2);
    }

    /* else everything is ok */
    sum = 0;
    while(fscanf(f, "%d", &x) == 1) {
        sum += x;
    }

    printf("%d\n", sum);

    /* not strictly necessary but it's polite */
    fclose(f);
```

```c
    return 0;
}
```

examples/IO/sum.c

To write to a file, open it with `fopen(filename, "w")`. Note that as soon as you call `fopen` with the `"w"` flag, any previous contents of the file are erased. If you want to append to the end of an existing file, use `"a"` instead. You can also add `+` onto the flag if you want to read and write the same file (this will probably involve using `fseek`).

Some operating systems (Windows) make a distinction between text and binary files. For text files, use the same arguments as above. For binary files, add a `b`, e.g. `fopen(filename, "wb")` to write a binary file.

```c
/* leave a greeting in the current directory */

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

#define FILENAME "hello.txt"
#define MESSAGE "hello world"

int
main(int argc, char **argv)
{
    FILE *f;

    f = fopen(FILENAME, "w");
    if(f == 0) {
        perror(FILENAME);
        exit(1);
    }

    /* unlike puts, fputs doesn't add a newline */
    fputs(MESSAGE, f);
    putc('\n', f);

    fclose(f);

    return 0;
}
```

examples/IO/helloFile.c

4.7 Statements and control structures

The bodies of C functions (including the `main` function) are made up of **statements**. These can either be **simple statements** that do not contain other statements, or **compound statements** that have other statements inside them. **Control structures** are compound statements like if/then/else, while, for, and do..while that control how or whether their component statements are executed.

4.7.1 Simple statements

The simplest kind of statement in C is an expression (followed by a semicolon, the terminator for all simple statements). Its value is computed and discarded. Examples:

```
x = 2;              /* an assignment statement */
x = 2+3;            /* another assignment statement */
2+3;                /* has no effect---will be discarded by smart compilers */
puts("hi");         /* a statement containing a function call */
root2 = sqrt(2);    /* an assignment statement with a function call */
```

Most statements in a typical C program are simple statements of this form.

Other examples of simple statements are the jump statements `return`, `break`, `continue`, and `goto`. A `return` statement specifies the return value for a function (if there is one), and when executed it causes the function to exit immediately. The `break` and `continue` statements jump immediately to the end of a loop (or `switch`; see below) or the next iteration of a loop; we'll talk about these more when we talk about loops. The `goto` statement jumps to another location in the same function, and exists for the rare occasions when it is needed. Using it in most circumstances is a sin.

4.7.2 Compound statements

Compound statements come in two varieties: conditionals and loops.

4.7.2.1 Conditionals These are compound statements that test some condition and execute one or another block depending on the outcome of the condition. The simplest is the `if` statement:

```
if(houseIsOnFire) {
    /* ouch! */
    scream();
    runAway();
}
```

The **body** of the `if` statement is executed only if the expression in parentheses at the top evaluates to true (which in C means any value that is not 0).

The braces are not strictly required, and are used only to group one or more statements into a single statement. If there is only one statement in the body, the braces can be omitted:

```
if(programmerIsLazy) omitBraces();
```

This style is recommended only for very simple bodies. Omitting the braces makes it harder to add more statements later without errors.

```
if(underAttack)
    launchCounterAttack();   /* executed only when attacked */
    hideInBunker();          /* ### DO NOT INDENT LIKE THIS ### executed always */
```

In the example above, the lack of braces means that the `hideInBunker()` statement is *not* part of the `if` statement, despite the misleading indentation. This sort of thing is why I generally always put in braces in an `if`.

An `if` statement may have an `else` clause, whose body is executed if the test is false (i.e. equal to 0).

```
if(happy) {
    smile();
} else {
    frown();
}
```

A common idiom is to have a chain of `if` and `else if` branches that test several conditions:

```
if(temperature < 0) {
    puts("brrr");
} else if(temperature < 100) {
    puts("hooray");
} else {
    puts("ouch!");
}
```

This can be inefficient if there are a lot of cases, since the tests are applied sequentially. For tests of the form *<expression>* `==` *<small constant>*, the `switch` statement may provide a faster alternative. Here's a typical `switch` statement:

```
/* print plural of cow, maybe using the obsolete dual number construction */
switch(numberOfCows) {
case 1:
    puts("cow");
    break;
case 2:
```

```
        puts("cowen");
        break;
    default:
        puts("cows");
        break;
    }
```

This prints the string "cow" if there is one cow, "cowen" if there are two cowen, and "cows" if there are any other number of cows. The `switch` statement evaluates its argument and jumps to the matching `case` label, or to the `default` label if none of the cases match. Cases must be constant integer values.

The `break` statements inside the block jump to the end of the block. Without them, executing the `switch` with `numberOfCows` equal to 1 would print all three lines. This can be useful in some circumstances where the same code should be used for more than one case:

```
    switch(c) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        type = VOWEL;
        break;
    default:
        type = CONSONANT;
        break;
    }
```

or when a case "falls through" to the next:

```
    switch(countdownStart) {
    case 3:
        puts("3");
    case 2:
        puts("2");
    case 1:
        puts("1")
    case 0:
        puts("KABLOOIE!");
        break;
    default:
        puts("I can't count that high!");
        break;
    }
```

Note that it is customary to include a `break` on the last case even though it has no effect; this avoids problems later if a new case is added. It is also customary to include a `default` case even if the other cases supposedly exhaust all the possible values, as a check against bad or unanticipated inputs.

```
switch(oliveSize) {
case JUMBO:
    eatOlives(SLOWLY);
    break;
case COLLOSSAL:
    eatOlives(QUICKLY);
    break;
case SUPER_COLLOSSAL:
    eatOlives(ABSURDLY);
    break;
default:
    /* unknown size! */
    abort();
    break;
}
```

Though `switch` statements are better than deeply nested if/else-if constructions, it is often even better to organize the different cases as data rather than code. We'll see examples of this when we talk about function pointers.

Nothing in the C standards prevents the `case` labels from being buried inside other compound statements. One rather hideous application of this fact is Duff's device.

4.7.2.2 Loops

There are three kinds of loops in C.

4.7.2.2.1 The while loop

A `while` loop tests if a condition is true, and if so, executes its body. It then tests the condition is true again, and keeps executing the body as long as it is. Here's a program that deletes every occurrence of the letter `e` from its input.

```
#include <stdio.h>

int
main(int argc, char **argv)
{
    int c;

    while((c = getchar()) != EOF) {
        switch(c) {
        case 'e':
        case 'E':
            break;
```

```
        default:
            putchar(c);
            break;
        }
    }

    return 0;
}
```

Note that the expression inside the `while` argument both assigns the return value of `getchar` to `c` and tests to see if it is equal to `EOF` (which is returned when no more input characters are available). This is a very common idiom in C programs. Note also that even though `c` holds a single character, it is declared as an `int`. The reason is that `EOF` (a constant defined in `stdio.h`) is outside the normal character range, and if you assign it to a variable of type `char` it will be quietly truncated into something else. Because C doesn't provide any sort of exception mechanism for signalling unusual outcomes of function calls, designers of library functions often have to resort to extending the output of a function to include an extra value or two to signal failure; we'll see this a lot when the null pointer shows up in the chapter on pointers.

4.7.2.2.2 The do..while loop The `do`..`while` statement is like the `while` statement except the test is done at the end of the loop instead of the beginning. This means that the body of the loop is always executed at least once.

Here's a loop that does a random walk until it gets back to 0 (if ever). If we changed the `do`..`while` loop to a `while` loop, it would never take the first step, because `pos` starts at 0.

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

int
main(int argc, char **argv)
{
    int pos = 0;       /* position of random walk */

    srandom(time(0));  /* initialize random number generator */

    do {
        pos += random() & 0x1 ? +1 : -1;
        printf("%d\n", pos);
    } while(pos != 0);

    return 0;
```

```
}
```

examples/statements/randomWalk.c

The `do..while` loop is used much less often in practice than the `while` loop.

It is theoretically possible to convert a `do..while` loop to a `while` loop by making an extra copy of the body in front of the loop, but this is not recommended since it's almost always a bad idea to duplicate code.

4.7.2.2.3 The for loop The `for` loop is a form of syntactic sugar that is used when a loop iterates over a sequence of values stored in some variable (or variables). Its argument consists of three expressions: the first initializes the variable and is called once when the statement is first reached. The second is the test to see if the body of the loop should be executed; it has the same function as the test in a `while` loop. The third sets the variable to its next value. Some examples:

```
/* count from 0 to 9 */
for(i = 0; i < 10; i++) {
    printf("%d\n", i);
}

/* and back from 10 to 0 */
for(i = 10; i >= 0; i--) {
    printf("%d\n", i);
}

/* this loop uses some functions to move around */
for(c = firstCustomer(); c != END_OF_CUSTOMERS; c = customerAfter(c)) {
    helpCustomer(c);
}

/* this loop prints powers of 2 that are less than n*/
for(i = 1; i < n; i *= 2) {
    printf("%d\n", i);
}

/* this loop does the same thing with two variables by using the comma operator */
for(i = 0, power = 1; power < n; i++, power *= 2) {
    printf("2^%d = %d\n", i, power);
}

/* Here are some nested loops that print a times table */
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        printf("%d*%d=%d ", i, j, i*j);
```

```
        }
        putchar('\n');
    }
```

A `for` loop can always be rewritten as a `while` loop.

```
    for(i = 0; i < 10; i++) {
        printf("%d\n", i);
    }

    /* is exactly the same as */

    i = 0;
    while(i < 10) {
        printf("%d\n", i);
        i++;
    }
```

4.7.2.2.4 Loops with break, continue, and goto

The `break` statement immediately exits the innermmost enclosing loop or `switch` statement.

```
    for(i = 0; i < n; i++) {
        openDoorNumber(i);
        if(boobyTrapped()) {
            break;
        }
    }
```

The `continue` statement skips to the next iteration. Here is a program with a loop that iterates through all the integers from -10 through 10, skipping 0:

```
#include <stdio.h>

/* print a table of inverses */
#define MAXN (10)

int
main(int argc, char **argv)
{
    int n;

    for(n = -MAXN; n <= MAXN; n++) {
        if(n == 0) continue;
        printf("1.0/%3d = %+f\n", n, 1.0/n);
    }

    return 0;
```

```
}
```

examples/statements/inverses.c

Occasionally, one would like to break out of more than one nested loop. The way to do this is with a `goto` statement.

```
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            doSomethingTimeConsumingWith(i, j);
            if(checkWatch() == OUT_OF_TIME) {
                goto giveUp;
            }
        }
    }
giveUp:
    puts("done");
```

The target for the `goto` is a **label**, which is just an identifier followed by a colon and a statement (the empty statement `;` is ok).

The `goto` statement can be used to jump anywhere within the same function body, but breaking out of nested loops is widely considered to be its only genuinely acceptable use in normal code.

4.7.2.3 Choosing where to put a loop exit

Choosing where to put a loop exit is usually pretty obvious: you want it after any code that you want to execute at least once, and before any code that you want to execute only if the termination test fails.

If you know in advance what values you are going to be iterating over, you will most likely be using a `for` loop:

```
for(i = 0; i < n; i++) {
    a[i] = 0;
}
```

Most of the rest of the time, you will want a `while` loop:

```
while(!done()) {
    doSomething();
}
```

The `do..while` loop comes up mostly when you want to try something, then try again if it failed:

```
do {
    result = fetchWebPage(url);
} while(result == 0);
```

Finally, leaving a loop in the middle using `break` can be handy if you have something extra to do before trying again:

```
for(;;) {
    result = fetchWebPage(url);
    if(result != 0) {
        break;
    }
    /* else */
    fprintf(stderr, "fetchWebPage failed with error code %03d\n", result);
    sleep(retryDelay);  /* wait before trying again */
}
```

(Note the empty `for` loop header means to loop forever; `while(1)` also works.)

4.8 Functions

A **function**, **procedure**, or **subroutine** encapsulates some complex computation as a single operation. Typically, when we **call** a function, we pass as **arguments** all the information this function needs, and any effect it has will be reflected in either its **return value** or (in some cases) in changes to values pointed to by the arguments. Inside the function, the arguments are copied into local variables, which can be used just like any other local variable—they can even be assigned to without affecting the original argument.

4.8.1 Function definitions

A typical function definition looks like this:

```
/* Returns the square of the distance between two points separated by
   dx in the x direction and dy in the y direction. */
int
distSquared(int dx, int dy)
{
    return dx*dx + dy*dy;
}
```

examples/functions/distSquaredNoHeader.c

The part outside the braces is called the **function declaration**; the braces and their contents is the **function body**.

Like most complex declarations in C, once you delete the type names the declaration looks like how the function is used: the name of the function comes before the parentheses and the arguments inside. The `int`s scattered about specify the type of the return value of the function (before the function name) and of the parameters (inside the parentheses after the function name); these are used by the compiler to determine how to pass values in and out of the function and (usually for more complex types, since numerical types will often convert automatically) to detect type mismatches.

If you want to define a function that doesn't return anything, declare its return type as `void`. You should also declare a parameter list of `void` if the function takes no arguments.

```
/* Prints "hi" to stdout */
void
helloWorld(void)
{
    puts("hi");
}
```

examples/functions/helloWorld.c

It is not strictly speaking an error to omit the second `void` here. Putting `void` in for the parameters tells the compiler to enforce that no arguments are passed in. If we had instead declared `helloWorld` as

```
/* Prints "hi" to stdout */
void
helloWorld()    /* DANGER! */
{
    puts("hi");
}
```

it would be possible to call it as

```
    helloWorld("this is a bogus argument");
```

without causing an error. The reason is that a function declaration with no arguments means that the function can take an unspecified number of arguments, and it's up to the user to make sure they pass in the right ones. There are good historical reasons for what may seem like obvious lack of sense in the design of the language here, and fixing this bug would break most C code written before 1989. But you shouldn't ever write a function declaration with an empty argument list, since you want the compiler to know when something goes wrong.

4.8.2 When to write a function

As with any kind of abstraction, there are two goals to making a function:

- **Encapsulation:** If you have some task to carry out that is simple do describe from the outside but messy to understand from the inside, wrapping it in a function lets somebody carry out this task without having to know the details. This is also useful if you want to change the implementation later.

- **Code re-use:** If you find yourself writing the same lines of code in several places (or worse, are tempted to copy a block of code to several places), you should probably put this code in a function (or perhaps more than one function, if there is no succinct way to describe what this block of code is doing).

Both of these goals may be trumped by the goal of making your code understandable. If you can't describe what a function is doing in a single, simple sentence, this is a sign that maybe you need to restructure your code. Having a function that does more than one thing (or does different thing depending on its arguments) is likely to lead to confusion. So, for example, this is not a good function definition:

```c
/*** ### UGLY CODE AHEAD ### ***/

/*
 * If getMaximum is true, return maximum of x and y,
 * else return minimum.
 */
int
computeMaximumOrMinimum(int x, int y, int getMaximum)
{
    if(x > y) {
        if(getMaximum) {
            return x;
        } else {
            return y;
        }
    } else {
        if(getMaximum) {
            return y;
        } else {
            return x;
        }
    }
}
```

Better would be to write two functions:

```c
/* return the maximum of x and y */
int
maximum(int x, int y)
{
    if(x > y) {
        return x;
    } else {
        return y;
    }
```

```
}

/* return the minimum of x and y */
int
minimum(int x, int y)
{
    if(x < y) {
        return x;
    } else {
        return y;
    }
}
```

At the same time, it's possible for a function to be too simple. Suppose I write the function

```
/* print x to stdout followed by a newline */
void
printIntWithNewline(int x)
{
    printf("%d\n", x);
}
```

It's pretty clear from the name what this function does. But since anybody who has been using C for a while has seen `printf("%d\n", ...)` over and over again, it's usually more clear to expand out the definition:

```
    printIntWithNewline(2+5);   /* this could do anything */
    printf("%d\n", 2+7);        /* this does exactly what it says */
```

As with all caveats, this caveat comes with its own caveat: what might justify a function like this is if you want to be able to do some kind of specialized formatting that should be consistent for all values of a particular form. So you might write a `printDistance` function like the above as a stub for a fancier function that might use different units at different scales or something.

A similar issue will come up with non-syntactic macros, which also tend to fail the "does this make my code more or less understandable" test. Usually it is a bad idea to try to replace common C idioms.

4.8.3 Calling a function

A function call consists of the function followed by its arguments (if any) inside parentheses, separated by comments. For a function with no arguments, call it with nothing between the parentheses. A function call that returns a value can be used in an expression just like a variable. A call to a `void` function can only be used as an expression by itself:

```c
    totalDistance += distSquared(x1 - x2, y1 - y2);
    helloWorld();
    greetings += helloWorld();  /* ERROR */
```

4.8.4 The return statement

To return a value from a function, write a `return` statement, e.g.

```c
    return 172;
```

The argument to `return` can be any expression. Unlike the expression in, say, an `if` statement, you do not need to wrap it in parentheses. If a function is declared `void`, you can do a `return` with no expression, or just let control reach the end of the function.

Executing a `return` statement immediately terminates the function. This can be used like `break` to get out of loops early.

```c
/* returns 1 if n is prime, 0 otherwise */
int
isPrime(int n)
{
    int i;

    if (n < 2) return 0;   /* special case for 0, 1, negative n */

    for(i = 2; i < n; i++) {
        if (n % i == 0) {
            /* found a factor */
            return 0;
        }
    }

    /* no factors */
    return 1;
}
```

examples/functions/isPrime.c

4.8.5 Function declarations and modules

By default, functions have **global scope**: they can be used anywhere in your program, even in other files. If a file doesn't contain a declaration for a function `someFunc` before it is used, the compiler will assume that it is declared like `int someFunc()` (i.e., return type `int` and unknown arguments). This can produce infuriating complaints later when the compiler hits the real declaration and insists that your function `someFunc` should be returning an `int` and you are a bonehead for declaring it otherwise.

The solution to such insulting compiler behavior errors is to either (a) move the function declaration before any functions that use it; or (b) put in a declaration without a body before any functions that use it, in addition to the declaration that appears in the function definition. (Note that this violates the **no separate but equal** rule, but the compiler should tell you when you make a mistake.) Option (b) is generally preferred, and is the only option when the function is used in a different file.

To make sure that all declarations of a function are consistent, the usual practice is to put them in an include file. For example, if `distSquared` is used in a lot of places, we might put it in its own file `distSquared.c`:

```
#include "distSquared.h"

int
distSquared(int dx, int dy)
{
    return dx*dx + dy*dy;
}
```

examples/functions/distSquared.c

The file `distSquared.c` above uses `#include` to include a copy of the following header file `distSquared.h`:

```
/* Returns the square of the distance between two points separated by
   dx in the x direction and dy in the y direction. */
int distSquared(int dx, int dy);
```

examples/functions/distSquared.h

Note that the declaration in `distSquared.h` doesn't have a body. Instead, it's terminated by a semicolon, like a variable declaration. It's also worth noting that we moved the documenting comment to `distSquared.h`: the idea is that `distSquared.h` is the public face of this (very small one-function) module, and so the explanation of how to use the function should be there.

The reason `distSquared.c` includes `distSquared.h` is to get the compiler to verify that the declarations in the two files match. But to use the `distSquared` function, we also put `#include "distSquared.h"` at the top of the file that uses it:

```
#include "distSquared.h"

#define THRESHOLD (100)

int
tooClose(int x1, int y1, int x2, int y2)
```

```c
{
    return distSquared(x1 - x2, y1 - y2) < THRESHOLD;
}
```

examples/functions/tooClose.c

The `#include` on line 1 uses double quotes instead of angle brackets; this tells the compiler to look for `distSquared.h` in the current directory instead of the system include directory (typically `/usr/include`).

4.8.6 Static functions

By default, all functions are global; they can be used in any file of your program whether or not a declaration appears in a header file. To restrict access to the current file, declare a function `static`, like this:

```c
static void
helloHelper(void)
{
    puts("hi!");
}

void
hello(int repetitions)
{
    int i;

    for(i = 0; i < repetitions; i++) {
        helloHelper();
    }
}
```

examples/functions/staticHello.c

The function `hello` will be visible everywhere. The function `helloHelper` will only be visible in the current file.

It's generally good practice to declare a function static unless you intend to make it available, since not doing so can cause **namespace conflicts**, where the presence of two functions with the same name either prevent the program from linking or—even worse—cause the wrong function to be called. The latter can happen with library functions, since C allows the programmer to override library functions by defining a new function with the same name. Early on in my career as a C programmer, I once had a program fail in a spectacularly incomprehensible way because I'd written a `select` function without realizing that `select` is a core library function in Unix.

4.8.7 Local variables

A function may contain definitions of **local variables**, which are visible only inside the function and which survive only until the function returns. These may be declared at the start of any block (group of statements enclosed by braces), but it is conventional to declare all of them at the outermost block of the function.

```
/* Given n, compute n! = 1*2*...*n */
/* Warning: will overflow on 32-bit machines if n > 12 */
int
factorial(int n)
{
    int i;
    int product;

    if(n < 2) return n;
    /* else */

    product = 1;

    for(i = 2; i <= n; i++) {
        product *= i;
    }

    return product;
}
```

examples/functions/factorial.c

4.8.8 Mechanics of function calls

Several things happen under the hood when a function is called. Since a function can be called from several different places, the CPU needs to store its previous state to know where to go back. It also needs to allocate space for function arguments and local variables.

Some of this information will be stored in **registers**, memory locations built into the CPU itself, but most will go on the **stack**, a region of memory that on typical machines grows downward, even though the most recent additions to the stack are called the "top" of the stack. The location of the top of the stack is stored in the CPU in a special register called the **stack pointer**.

So a typical function call looks like this internally:

1. The current **instruction pointer** or **program counter** value, which gives the address of the next line of machine code to be executed, is pushed

onto the stack.

2. Any arguments to the function are copied either into specially designated registers or onto new locations on the stack. The exact rules for how to do this vary from one CPU architecture to the next, but a typical convention might be that the first few arguments are copied into registers and the rest (if any) go on the stack.
3. The instruction pointer is set to the first instruction in the code for the function.
4. The code for the function allocates additional space on the stack to hold its local variables (if any) and to save copies of the values of any registers it wants to use (so that it can restore their contents before returning to its caller).
5. The function body is executed until it hits a `return` statement.
6. Returning from the function is the reverse of invoking it: any saved registers are restored from the stack, the return value is copied to a standard register, and the values of the instruction pointer and stack pointer are restored to what they were before the function call.

From the programmer's perspective, the important point is that both the arguments and the local variables inside a function are stored in freshly-allocated locations that are thrown away after the function exits. So after a function call the state of the CPU is restored to its previous state, except for the return value. Any arguments that are passed to a function are passed as copies, so changing the values of the function arguments inside the function has no effect on the caller. Any information stored in local variables is lost.

Under very rare circumstances, it may be useful to have a variable local to a function that persists from one function call to the next. You can do so by declaring the variable `static`. For example, here is a function that counts how many times it has been called:

```
/* return the number of times the function has been called */
int
counter(void)
{
    static count = 0;

    return ++count;
}
```

examples/functions/staticCounter.c

Static local variables are stored outside the stack with global variables, and have unbounded extent. But they are only visible inside the function that declares them. This makes them slightly less dangerous than global variables—there is no fear that some foolish bit of code elsewhere will quietly change their value—but it is still the case that they usually aren't what you want. It is also likely that operations on static variables will be slightly slower than operations on ordinary ("automatic") variables, since making them persistent means that they have to be stored in (slow) main memory instead of (fast) registers.

4.9 Pointers

4.9.1 Memory and addresses

Memory in a typical modern computer is divided into two classes: a small number of **registers**, which live on the CPU chip and perform specialized functions like keeping track of the location of the next machine code instruction to execute or the current stack frame, and **main memory**, which (mostly) lives outside the CPU chip and which stores the code and data of a running program. When the CPU wants to fetch a value from a particular location in main memory, it must supply an address: a 32-bit or 64-bit unsigned integer on typical current architectures, referring to one of up to 2^{32} or 2^{64} distinct 8-bit locations in the memory. These integers can be manipulated like any other integer; in C, they appear as **pointers**, a family of types that can be passed as arguments, stored in variables, returned from functions, etc.

4.9.2 Pointer variables

A **pointer variable** is a variable that holds a pointer, just like an `int` variable is a variable that holds an `int`.

4.9.2.1 Declaring a pointer variable

The convention is C is that the declaration of a complex type looks like its use. To declare a pointer-valued variable, write a declaration for the thing that it points to, but include a `*` before the variable name:

```
int *pointerToInt;
double *pointerToDouble;
char *pointerToChar;
char **pointerToPointerToChar;
```

These declarations create four pointer variables, named `pointerToInt`, `pointerToDouble`, `pointerToChar`, and `pointerToPointerToChar`. On a typical 64-bit machine, each will be allocated 8 bytes, enough to represent an address in memory.

The contents of these variables are initially arbitrary: to use them, you will need to compute the address of something and assign it to the variable.

4.9.2.2 Assigning to pointer variables

Declaring a pointer-valued variable allocates space to hold the pointer but *not* to hold anything it points to.

Like any other variable in C, a pointer-valued variable will initially contain garbage—in this case, the address of a location that might or might not contain something important. To initialize a pointer variable, you have to assign to it the address of something that already exists. Typically this is done using the `&` (**address-of**) operator:

```
int n;              /* an int variable */
int *p;             /* a pointer to an int */

p = &n;             /* p now points to n */
```

4.9.2.3 Using a pointer Pointer variables can be used in two ways. The simplest way is to get their value as with any other variable. This value will be an address, which can be stored in another pointer variable of the same type.

```
int n;              /* an int variable */
int *p;             /* a pointer to an int */
int *q;             /* another pointer to an int */

p = &n;             /* p now points to n */
q = p;              /* q now points to n as well */
```

But more often you will want to work on the value stored at the location pointed to. You can do this by using the `*` (**dereference**) operator, which acts as an inverse of the address-of operator:

```
int n;              /* an int variable */
int *p;             /* a pointer to an int */

p = &n;             /* p now points to n */

*p = 2;             /* sets n to 2 */
*p = *p + *p;       /* sets n to 4 */
```

The `*` operator binds very tightly, so you can usually use `*p` anywhere you could use the variable it points to without worrying about parentheses. However, a few operators, such as the `--` and `++` operators and the `.` operator used to unpack structs, bind tighter. These require parentheses if you want the `*` to take precedence.

```
(*p)++;             /* increment the value pointed to by p */
*p++;               /* WARNING: increments p itself */
```

4.9.2.4 Printing pointers You can print a pointer value using `printf` with the `%p` format specifier. To do so, you should convert the pointer to the generic pointer type `void *` first using a cast, although on machines that don't have different representations for different pointer types, this may not be necessary.

Here is a short program that prints out some pointer values:

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

int G = 0;   /* a global variable, stored in BSS segment */

int
main(int argc, char **argv)
{
    static int s;  /* static local variable, stored in BSS segment */
    int a;         /* automatic variable, stored on stack */
    int *p;        /* pointer variable for malloc below */

    /* obtain a block big enough for one int from the heap */
    p = malloc(sizeof(int));

    printf("&G   = %p\n", (void *) &G);
    printf("&s   = %p\n", (void *) &s);
    printf("&a   = %p\n", (void *) &a);
    printf("&p   = %p\n", (void *) &p);
    printf("p    = %p\n", (void *) p);
    printf("main = %p\n", (void *) main);

    free(p);

    return 0;
}
```

examples/pointers/lookingAtPointers.c

When I run this on a Mac OS X 10.6 machine after compiling with `gcc`, the output is:

```
&G   = 0x100001078
&s   = 0x10000107c
&a   = 0x7fff5fbff2bc
&p   = 0x7fff5fbff2b0
p    = 0x100100080
main = 0x100000e18
```

The interesting thing here is that we can see how the compiler chooses to allocate space for variables based on their storage classes. The global variable `G` and the static local variable `s` both persist between function calls, so they get placed in the BSS segment (see .bss) that starts somewhere around `0x100000000`, typically after the code segment containing the actual code of the program. Local variables `a` and `p` are allocated on the stack, which grows down from somewhere near the top of the address space. The block returned from `malloc` that `p` points to is allocated off the heap, a region of memory that may also grow over time and starts after the BSS segment. Finally, `main` appears at 0x100000e18; this is in the code segment, which is a bit lower in memory than all the global variables.

4.9.3 The null pointer

The special value `0`, known as the **null pointer**, may be assigned to a pointer of any type. It may or may not be represented by the actual address `0`, but it will act like `0` in all contexts (e.g., it has the value false in an `if` or `while` statement). Null pointers are often used to indicate missing data or failed functions. Attempting to dereference a null pointer can have catastrophic effects, so it's important to be aware of when you might be supplied with one.

4.9.4 Pointers and functions

A simple application of pointers is to get around C's limit on having only one return value from a function. Because C arguments are copied, assigning a value to an argument inside a function has no effect on the outside. So the `doubler` function below doesn't do much:

```c
#include <stdio.h>

/* doesn't work */
void
doubler(int x)
{
    x *= 2;
}

int
main(int argc, char **argv)
{
    int y;

    y = 1;

    doubler(y);                 /* no effect on y */

    printf("%d\n", y);          /* prints 1 */

    return 0;
}
```

examples/pointers/badDoubler.c

However, if instead of passing the value of `y` into `doubler` we pass a pointer to `y`, then the `doubler` function can reach out of its own stack frame to manipulate `y` itself:

```c
#include <stdio.h>

void
doubler(int *x)
{
    *x *= 2;
}

int
main(int argc, char **argv)
{
    int y;

    y = 1;

    doubler(&y);                /* sets y to 2 */

    printf("%d\n", y);          /* prints 2 */

    return 0;
}
```

examples/pointers/goodDoubler.c

Generally, if you pass the value of a variable into a function (with no `&`), you can be assured that the function can't modify your original variable. When you pass a pointer, you should assume that the function can and will change the variable's value. If you want to write a function that takes a pointer argument but promises not to modify the target of the pointer, use `const`, like this:

```c
void
printPointerTarget(const int *p)
{
    printf("%d\n", *p);
}
```

The `const` qualifier tells the compiler that the target of the pointer shouldn't be modified. This will cause it to return an error if you try to assign to it anyway:

```c
void
printPointerTarget(const int *p)
{
    *p = 5;  /* produces compile-time error */
    printf("%d\n", *p);
}
```

Passing `const` pointers is mostly used when passing large structures to functions, where copying a 32-bit pointer is cheaper than copying the thing it points to.

If you really want to modify the target anyway, C lets you "cast away `const`":

```
void
printPointerTarget(const int *p)
{
    *((int *) p) = 5;  /* no compile-time error */
    printf("%d\n", *p);
}
```

There is usually no good reason to do this. The one exception might be if the target of the pointer represents an abstract data type, and you want to modify its representation during some operation to optimize things somehow in a way that will not be visible outside the abstraction barrier, making it appear to leave the target constant.

Note that while it is safe to pass pointers down into functions, it is very dangerous to pass pointers up. The reason is that the space used to hold any local variable of the function will be reclaimed when the function exits, but the pointer will still point to the same location, *even though something else may now be stored there*. So this function is very dangerous:

```
int *
dangerous(void)
{
    int n;

    return &n;          /* NO! */
}

...

    *dangerous() = 12;  /* writes 12 to some unknown location */
```

An exception is when you can guarantee that the location pointed to will survive even after the function exits, e.g. when the location is dynamically allocated using `malloc` (see below) or when the local variable is declared `static`:

```
int *
returnStatic(void)
{
    static int n;

    return &n;
}

...
```

```
*returnStatic() = 12;        /* writes 12 to the hidden static variable */
```

Usually returning a pointer to a `static` local variable is not good practice, since the point of making a variable local is to keep outsiders from getting at it. If you find yourself tempted to do this, a better approach is to allocate a new block using `malloc` (see below) and return a pointer to that. The downside of the `malloc` method is that the caller has to promise to call `free` on the block later, or you will get a storage leak.

4.9.5 Pointer arithmetic and arrays

Because pointers are just numerical values, one can do arithmetic on them. Specifically, it is permitted to

- Add an integer to a pointer or subtract an integer from a pointer. The effect of `p+n` where `p` is a pointer and `n` is an integer is to compute the address equal to `p` plus `n` times the size of whatever `p` points to (this is why `int *` pointers and `char *` pointers aren't the same).
- Subtract one pointer from another. The two pointers must have the same type (e.g. both `int *` or both `char *`). The result is a signed integer value of type `ptrdiff_t`, equal to the numerical difference between the addresses divided by the size of the objects pointed to.
- Compare two pointers using `==`, `!=`, `<`, `>`, `<=`, or `>=`.
- Increment or decrement a pointer using `++` or `--`.

4.9.5.1 Arrays

The main application of pointer arithmetic in C is in **arrays**. An array is a block of memory that holds one or more objects of a given type. It is declared by giving the type of object the array holds followed by the array name and the size in square brackets:

```
int a[50];           /* array of 50 ints */
char *cp[100];       /* array of 100 pointers to char */
```

Declaring an array allocates enough space to hold the specified number of objects (e.g. 200 bytes for `a` above and 400 for `cp`—note that a `char *` is an address, so it is much bigger than a `char`). The number inside the square brackets must be a constant whose value can be determined at compile time.

The array name acts like a constant pointer to the zeroth element of the array. It is thus possible to set or read the zeroth element using `*a`. But because the array name is constant, you can't assign to it:

```
*a = 12;              /* sets zeroth element to 12 */

a = &n;               /* #### DOESN'T WORK #### */
```

More common is to use square brackets to refer to a particular element of the array. The expression `a[n]` is defined to be equivalent to `*(a+n)`; the **index** `n` (an integer) is added to the base of the array (a pointer), to get to the location of the `n`-th element of `a`. The implicit `*` then dereferences this location so that you can read its value (in a normal expression) or assign to it (on the left-hand side of an assignment operator). The effect is to allow you to use `a[n]` just as you would any other variable of type `int` (or whatever type `a` was declared as).

Note that C doesn't do any sort of bounds checking. Given the declaration `int a[50];`, only indices from `a[0]` to `a[49]` can be used safely. However, the compiler will not blink at `a[-12]` or `a[10000]`. If you read from such a location you will get garbage data; if you write to it, you will overwrite god-knows-what, possibly trashing some other variable somewhere else in your program or some critical part of the stack (like the location to jump to when you return from a function). It is up to you as a programmer to avoid such **buffer overruns**, which can lead to very mysterious (and in the case of code that gets input from a network, security-damaging) bugs. The valgrind program can help detect such overruns in some cases.

Another curious feature of the definition of `a[n]` as identical to `*(a+n)` is that it doesn't actually matter which of the array name or the index goes inside the braces. So all of `a[0]`, `*a`, and `0[a]` refer to the zeroth entry in `a`. Unless you are deliberately trying to obfuscate your code, it's best to write what you mean.

4.9.5.2 Arrays and functions

Because array names act like pointers, they can be passed into functions that expect pointers as their arguments. For example, here is a function that computes the sum of all the values in an array `a` of size `n`:

```c
/* compute the sum of the first n elements of array a */
int
sumArray(int n, const int *a)
{
    int i;
    int sum;

    sum = 0;
    for(i = 0; i < n; i++) {
        sum += a[i];
    }

    return sum;
}
```

examples/pointers/sumArray.c

Note the use of `const` to promise that `sumArray` won't modify the contents of `a`.

Another way to write the function header is to declare `a` as an array of unknown size:

```
/* return the sum of the values in a, an array of size n */
int
sumArray(int n, const int a[])
{
    ...
}
```

This has *exactly* the same meaning to the compiler as the previous definition. Even though normally the declarations `int a[10]` and `int *a` mean very different things (the first one allocates space to hold 10 `int`s, and prevents assigning a new value to `a`), in a function argument `int a[]` is just syntactic sugar for `int *a`. You can even modify what `a` points to inside `sumArray` by assigning to it. This will allow you to do things that you usually don't want to do, like write this hideous routine:

```
/* return the sum of the first n values in a */
int
sumArray(int n, const int a[])
{
    const int *an;      /* pointer to first element not in a */
    int sum;

    sum = 0;
    an = a+n;

    while(a < an) {
        sum += *a++;
    }

    return sum;
}
```

4.9.5.3 Multidimensional arrays

Arrays can themselves be members of arrays. The result is a multidimensional array, where a value in row `i` and column `j` is accessed by `a[i][j]`.

Declaration is similar to one-dimensional arrays:

```
    int a[3][6];    /* declares an array of 3 rows of 6 ints each */
```

This declaration produces an array of 18 `int` values, packed contiguously in memory. The interpretation is that `a` is an array of 3 objects, each of which is an array of 6 `int`s.

If we imagine the array to contain increasing values like this:

```
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
```

the actual positions in memory will look like this:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
 ^                 ^                 ^
a[0]              a[1]              a[2]
```

To look up a value, we do the usual array-indexing magic. Suppose we want to find `a[1][4]`. The name `a` acts as a pointer to the base of the array.The name `a[1]` says to skip ahead 1 times the size of the things pointed to by `a`, which are arrays of 6 `int`s each, for a total size of 24 bytes assuming 4-byte `int`s. For `a[1][4]`, we start at `a[1]` and move forward 4 times the size of the thing pointed to by `a[1]`, which is an `int`; this puts us 24+16 bytes from `a`, the position of 10 in the picture above.

Like other array declarations, the size must be specified at compile time in pre-C99 C. If this is not desirable, a similar effect can be obtained by allocating each row separately using `malloc` and building a master list of pointers to rows, of type `int **`. The downside of this approach is that the array is no longer contiguous (which may affect cache performance) and it requires reading a pointer to find the location of a particular value, instead of just doing address arithmetic starting from the base address of the array. But elements can still be accessed using the `a[i][j]` syntax. An example of this approach is given below:

```c
/* Demo program for malloc'd two-dimensional arrays */

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

/* frees a 2d array created by malloc2d */
void
free2d(void **a)
{
    void **row;

    /* first free rows */
    for(row = a; *row != 0; row++) {
        free(*row);
    }

    /* then free array of rows */
    free(a);
}

/* returns a two-dimensional array with numRows rows and
```

```
 * rowSize bytes per row, or 0 on allocation failure.
 * The caller is responsible for freeing the result with free2d. */
void **
malloc2d(size_t numRows, size_t rowSize)
{
    void **a;
    size_t i;

    /* a is an array of void * pointers that point to the rows */
    /* The last element is 0, so free2d can detect the last row */
    a = malloc(sizeof(void *) * (numRows + 1));        /* one extra for sentinel */
    if(a == 0) {
        /* malloc failed */
        return 0;
    }

    /* now allocate the actual rows */
    for(i = 0; i < numRows; i++) {
        a[i] = malloc(rowSize);
        if(a[i] == 0) {
            /* note that 0 in a[i] will stop freed2d after it frees previous rows */
            free2d(a);
            return 0;
        }
    }

    /* initialize the sentinel value */
    a[numRows] = 0;

    return a;
}

int
main(int argc, char **argv)
{
    int rows;
    int cols;
    int **a;
    int i;
    int j;

    if(argc != 3) {
        fprintf(stderr, "Usage: %s rows cols\n", argv[0]);
        return 1;
    }
    /* else */
```

```
    rows = atoi(argv[1]);
    cols = atoi(argv[2]);

    /* note that void ** is not converted automatically,
     * so we need an explicit cast */
    a = (int **) malloc2d(rows, cols * sizeof(int));
    if(a == 0) {
        fprintf(stderr, "malloc2d failed, exiting\n");
        return 2;
    }

    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            a[i][j] = i - j;
        }
    }

    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            printf("%4d", a[i][j]);
        }
        putchar('\n');
    }

    free2d((void **) a);                          /* always clean up */

    return 0;
}
```

examples/pointers/malloc2d.c

4.9.5.4 Variable-length arrays C99 adds the feature of **variable-length arrays**, where the size of the array is determined at run-time. These can only appear as local variables in procedures (*automatic variables*) or in argument lists. In the case of variable-length arrays in argument lists, it is also necessary that the length of the array be computable from previous arguments.

For example, we could make the length of the array explicit in our `sumArray` function:

```
/* return the sum of the values in a, an array of size n */
int
sumArray(int n, const int a[n])
{
    int i;
```

```
    int sum;

    sum = 0;
    for(i = 0; i < n; i++) {
        sum += a[i];
    }

    return sum;
}
```

This doesn't accomplish much, because the length of the array is not used. However, it does become useful if we have a two-dimensional array, as otherwise there is no way to compute the length of each row:

```
int
sumMatrix(int rows, int cols, const int m[rows][cols])
{
    int i;
    int j;
    int sum;

    sum = 0;
    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            sum += a[i][j];
        }
    }

    return sum;
}
```

Here the fact that each row of `m` is known to be an array of `cols` many `int`s makes the implicit pointer computation in `a[i][j]` actually work. It is considerably more difficult to to this in ANSI C; the simplest approach is to pack `m` into a one-dimensional array and do the address computation explicitly:

```
int
sumMatrix(int rows, int cols, const int a[])
{
    int i;
    int j;
    int sum;

    sum = 0;
    for(i = 0; i < rows; i++) {
        for(j = 0; j < cols; j++) {
            sum += a[i*cols + j];
        }
```

```
    }

    return sum;
}
```

Variable-length arrays can sometimes be used for run-time storage allocation, as an alternative to `malloc` and `free` (see below). A variable-length array allocated as a local variable will be deallocated when the containing scope (usually a function body, but maybe just a compound statement marked off by braces) exits. One consequence of this is that you can't return a variable-length array from a function.

Here is an example of code using this feature:

```
/* reverse an array in place */
void
reverseArray(int n, int a[n])
{
    /* algorithm: copy to a new array in reverse order */
    /* then copy back */

    int i;
    int copy[n];

    for(i = 0; i < n; i++) {
        /* the -1 is needed to that a[0] goes to a[n-1] etc. */
        copy[n-i-1] = a[i];
    }

    for(i = 0; i < n; i++) {
        a[i] = copy[i];
    }
}
```

While using variable-length arrays for this purpose can simplify code in some cases, as a general programming practice it is **extremely dangerous**. The reason is that, unlike allocations through `malloc`, variable-length array allocations are typically allocated on the stack (which is often more constrainted than the heap) and have no way of reporting failure. So if there isn't enough room for your variable-length array, odds are you won't find out until a segmentation fault occurs somewhere later in your code when you try to use it.

(As an additional annoyance, `gdb` is confused by two-dimensional variable-length arrays.)

Here's a safer version of the above routine, using `malloc` and `free`.

```
/* reverse an array in place */
void
```

```
reverseArray(int n, int a[n])
{
    /* algorithm: copy to a new array in reverse order */
    /* then copy back */

    int i;
    int *copy;

    copy = (int *) malloc(n * sizeof(int));
    assert(copy);  /* or some other error check */

    for(i = 0; i < n; i++) {
        /* the -1 is needed to that a[0] goes to a[n-1] etc. */
        copy[n-i-1] = a[i];
    }

    for(i = 0; i < n; i++) {
        a[i] = copy[i];
    }

    free(copy);
}
```

4.9.6 Pointers to void

A special pointer type is `void *`, a "pointer to `void`". Such pointers are declared in the usual way:

```
    void *nothing;      /* pointer to nothing */
```

Unlike ordinary pointers, you can't dereference a `void *` pointer or do arithmetic on it, because the compiler doesn't know what type it points to. However, you are allowed to use a `void *` as a kind of "raw address" pointer value that you can store arbitrary pointers in. It is permitted to assign to a `void *` variable from an expression of any pointer type; conversely, a `void *` pointer value can be assigned to a pointer variable of any type. An example is the return value of `malloc` or the argument to `free`, both of which are declared as `void *`. (Note that K&R suggests using an explicit cast for the return value of `malloc`. This has since been acknowledged by the authors to be an error, which arose from the need for a cast prior to the standardization of `void *` in ANSI C.)

```
    int *block;

    block = malloc(sizeof(int) * 12);  /* void * converted to int * before assignment */
    free(block);                       /* int * converted to void * before passing to free *
```

If you need to use a `void *` pointer as a pointer of a particular type in an expression, you can **cast** it to the appropriate type by prefixing it with a type name in parentheses, like this:

```
int a[50];          /* typical array of ints */
void *p;            /* dangerous void pointer */

a[12] = 17;         /* save that valuable 17 */
p = a;              /* p now holds base address of a */

printf("%d\n", ((int *) p)[12]);  /* get 17 back */
```

Usually if you have to start writing casts, it's a sign that you are doing something wrong, and you run the danger of **violating the type system**—say, by tricking the compiler into treating a block of bits that are supposed to be an `int` as four `char`s. But violating the type system like this will be necessary for some applications, because even the weak type system in C turns out to be too restrictive for writing certain kinds of "generic" code that work on values of arbitrary types.

4.9.6.1 Alignment

One issue with casting pointers to and from `void *` is that you may violate the **alignment restrictions** for a particular kind of pointer on some architectures.

Back in the 8-bit era of the 1970s, a single load or store operation would access a single byte of memory, and because some data (`char`s) are still only one byte wide, C pointers retain the ability to address individual bytes. But present-day memory architectures typically have a wider data path, and the CPU may load or store as many as 8 bytes (64 bits) in a single operation. This makes it natural to organize memory into 4-byte or 8-byte words even though addresses still refer to individual bytes. The effect of the memory architecture is that the address of memory words must be **aligned** to a multiple of the word size: so with 4-byte words, the address `0x1037ef44` (a multiple of 4) could refer to a full word, but `0x1037ef45` (one more than a multiple of 4) could only be used to refer to a byte within a word.

What this means for a C program depends on your particular CPU and compiler. If you try to use something like `0x1037ef45` as an `int *`, one of three things might happen:

1. The CPU might load the 4 bytes starting at this address, using two accesses to memory to piece together the full `int` out of fragments of words. This is done on Intel architectures, but costs performance.
2. The CPU might quietly zero out the last two bits of the address, loading from `0x1037ef44` even though you asked for `0x1037ef45`. This happens on some other architectures, notably ARM.
3. The CPU might issue a run-time exception.

All of these outcomes are bad, and the C standard does not specify what happens if you try to dereference a pointer value that does not satisfy the alignment restrictions of its target type. Fortunately, unless you are doing very nasty things with casts, this is unlikely to come up, because any pointer value you will see in a typical program is likely to arise in one of three ways:

1. By taking the address of some variable. This pointer will be appropriately aligned, because the compiler allocates space for each variable (including fields within `struct`s) with appropriate alignment.
2. By computing an offset address using pointer arithmetic either explicitly (`p + n`) or implicitly (`p[n]`). In either case, as long as the base pointer is correctly aligned, the computed pointer will also be correctly aligned.
3. By obtaining a pointer to an allocated block of memory using `malloc` or a similar function. Here `malloc` is designed to always return blocks with the maximum possible required alignment, just to avoid problems when you use the results elsewhere.

On many compilers, you can use `__alignof(`*type*`)` to get the alignment restriction for a particular type. This was formalized in C11 without the underscores: `alignof`. Usually if your code needs to include `__alignof` or `alignof` something has already gone wrong.

The other place where alignment can create issues is that if you make a `struct` with components with different alignment restrictions, you may end up with some empty space. For example, on a machine that enforces 4-byte alignment for `int`s, building a `struct` that contains a `char` and an `int` will give you something bigger than you might expect:

```
#include <stdio.h>

struct ci {
    char c;  /* offset 0 */
             /* 3 unused bytes go here */
    int i;   /* offset 4 */
};

struct ic {
    int i;   /* offset 0 */
    char c;  /* offset 4 */
             /* 3 unused bytes go here */
};

int
main(int argc, char **argv)
{
    printf("sizeof(struct ci) == %lu\n", sizeof(struct ci));
    printf("sizeof(struct ic) == %lu\n", sizeof(struct ic));

    return 0;
```

```
}
```

examples/alignment/structPacking.c

```
$ c99 -Wall -o structPacking structPacking.c
$ ./structPacking
sizeof(struct ci) == 8
sizeof(struct ic) == 8
```

In both cases, the compiler packs in an extra 3 bytes to make the size of the struct a multiple of the worst alignment of any of its components. If it didn't do this, you would have trouble as soon as you tried to make an array of these things.

4.9.7 Run-time storage allocation using `malloc`

C does not generally permit arrays to be declared with variable sizes. C also doesn't let local variables outlive the function they are declared in. Both features can be awkward if you want to build data structures at run time that have unpredictable (perhaps even changing) sizes and that are intended to persist longer than the functions that create them. To build such structures, the standard C library provides the `malloc` routine, which asks the operating system for a block of space of a given size (in bytes). With a bit of pushing and shoving, this can be used to obtain a block of space that for all practical purposes acts just like an array.

To use `malloc`, you must include `stdlib.h` at the top of your program. The declaration for `malloc` is

```
void *malloc(size_t);
```

where `size_t` is an integer type (often `unsigned long`). Calling `malloc` with an argument of n allocates and returns a pointer to the start of a block of n bytes if possible. If the system can't give you the space you asked for (maybe you asked for more space than it has), `malloc` returns a null pointer. It is good practice to test the return value of `malloc` whenever you call it.

Because the return type of `malloc` is `void *`, its return value can be assigned to any variable with a pointer type. Computing the size of the block you need is your responsibility—and you will be punished for any mistakes with difficult-to-diagnose buffer overrun errors—but this task is made slightly easier by the built-in `sizeof` operator that allows you to compute the size in bytes of any particular data type. A typical call to `malloc` might thus look something like this:

```
#include <stdlib.h>

/* allocate and return a new integer array with n elements */
/* calls abort() if there isn't enough space */
```

```c
int *
makeIntArray(int n)
{
    int *a;

    a = malloc(sizeof(int) * n);

    if(a == 0) abort();                 /* die on failure */

    return a;
}
```

examples/pointers/makeIntArray.c

If you don't want to do the multiplication yourself, or if you want to guarantee that the allocated data is initialized to zero, you can use `calloc` instead of `malloc`. The `calloc` function is also declared in `stdlib.h` and takes two arguments: the number of things to allocated, and the size of each thing. Here's a version of `makeIntArray` that uses `calloc`. Aside from zeroing out the data, it is equivalent to the `malloc` version.

```c
#include <stdlib.h>

/* allocate and return a new integer array with n elements */
/* initializes array to zero */
/* calls abort() if there isn't enough space */
int *
makeIntArray(int n)
{
    int *a;

    a = calloc(n, sizeof(int));

    if(a == 0) abort();                 /* die on failure */

    return a;
}
```

examples/pointers/calloc.c

When you are done with a region allocated using `malloc`'d or `calloc`, you should return the space to the system using the `free` routine, also defined in `stdlib.h`. If you don't do this, your program will quickly run out of space. The `free` routine takes a `void *` as its argument and returns nothing. It is good practice to write a matching **destructor** that de-allocates an object for each **constructor** (like `makeIntArray`) that makes one.

```c
void
destroyIntArray(int *a)
```

```
{
    free(a);
}
```

It is a serious error to do anything at all with a block after it has been `free`d. This is not necessarily because `free` modifies the contents of the block (although it might), but because when you free a block you are granting the storage allocator permission to hand the same block out in response to a future call to `malloc`, and you don't want to step on whatever other part of your program is now trying to use that space.

It is also possible to grow or shrink a previously allocated block. This is done using the `realloc` function, which is declared as

```
void *realloc(void *oldBlock, size_t newSize);
```

The `realloc` function returns a pointer to the resized block. It may or may not allocate a new block. If there is room, it may leave the old block in place and return its argument. But it may allocate a new block and copy the contents of the old block, so you should assume that the old pointer has been `free`d.

Here's a typical use of `realloc` to build an array that grows as large as it needs to be:

```
/* read numbers from stdin until there aren't any more */
/* returns an array of all numbers read, or null on error */
/* returns the count of numbers read in *count */
int *
readNumbers(int *count /* RETVAL */)
{
    int mycount;        /* number of numbers read */
    int size;           /* size of block allocated so far */
    int *a;             /* block */
    int n;              /* number read */

    mycount = 0;
    size = 1;

    a = malloc(sizeof(int) * size);     /* allocating zero bytes is tricky */
    if(a == 0) return 0;

    while(scanf("%d", &n) == 1) {
        /* is there room? */
        while(mycount >= size) {
            /* double the size to avoid calling realloc for every number read */
            size *= 2;
            a = realloc(a, sizeof(int) * size);
            if(a == 0) return 0;
        }
```

```
        /* put the new number in */
        a[mycount++] = n;
    }

    /* now trim off any excess space */
    a = realloc(a, sizeof(int) * mycount);
    /* note: if a == 0 at this point we'll just return it anyway */

    /* save out mycount */
    *count = mycount;

    return a;
}
```

examples/pointers/readNumbers.c

Because errors involving `malloc` and its friends can be very difficult to spot, it is recommended to test any program that uses `malloc` using valgrind.

4.9.8 Function pointers

A **function pointer**, internally, is just the numerical address for the code for a function. When a function name is used by itself without parentheses, the value is a pointer to the function, just as the name of an array by itself is a pointer to its zeroth element. Function pointers can be stored in variables, `struct`s, `union`s, and arrays and passed to and from functions just like any other pointer type. They can also be called: a variable of type function pointer can be used in place of a function name.

Function pointers are not used as much in C as in functional languages, but there are many common uses even in C code.

4.9.8.1 Function pointer declarations

A function pointer declaration looks like a function declaration, except that the function name is wrapped in parentheses and preceded by an asterisk. For example:

```
/* a function taking two int arguments and returning an int */
int function(int x, int y);

/* a pointer to such a function */
int (*pointer)(int x, int y);
```

As with function declarations, the names of the arguments can be omitted.

Here's a short program that uses function pointers:

```c
/* Functional "hello world" program */

#include <stdio.h>

int
main(int argc, char **argv)
{
    /* function for emitting text */
    int (*say)(const char *);

    say = puts;

    say("hello world");

    return 0;
}
```

4.9.8.2 Callbacks A **callback** is when we pass a function pointer into a function so that that function can call our function when some event happens or it needs to compute something.

A classic example is the comparison argument to `qsort`, from the standard library:

```c
/* defined in stdlib.h */
void
qsort(
    void *base,
    size_t n,
    size_t size,
    int (*cmp)(const void *key1, const void *key2)
);
```

This is a generic sorting routine that will sort any array in place. It needs to know (a) the base address of the array; (b) how many elements there are; (c) how big each element is; and (d) how to compare two elements. The only tricky part is supplying the comparison, which could involve arbitrarily-complex code. So we supply this code as a function with an interface similar to `strcmp`.

```c
static int
compare_ints(void *key1, void *key2)
{
    return *((int *) key1) - *((int *) key2);
}

int
sort_int_array(int *a, int n)
```

```
{
    qsort(a, n, sizeof(*a), compare_ints);
}
```

Other examples might include things like registering an error handler for a library, instead of just having it call `abort()` or something equally catastrophic, or providing a cleanup function for freeing data passed into a data structure.

4.9.8.3 Dispatch tables

Alternative to gigantic `if/else if` or `switch` statements. The idea is to build an array of function pointers (or, more generally, some sort of dictionary data structure), and use the value we might otherwise be feeding to `switch` as an index into this array. Here is a simple example, which echoes most of the characters in its input intact, except for echoing every lowercase vowel twice:

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

/*
 * Demonstrate use of dispatch tables.
 */

/* print a character twice */
/* like putchar, returns character if successful or EOF on error */
int
putcharTwice(int c)
{
    if(putchar(c) == EOF || putchar(c) == EOF) {
        return EOF;
    } else {
        return c;
    }
}

#define NUM_CHARS (UCHAR_MAX + 1) /* UCHAR_MAX is in limits.h */

int
main(int argc, char **argv)
{
    /* this declares table as an array of function pointers */
    int (*table[UCHAR_MAX+1])(int);
    int i;
    int c;
```

```
    for(i = 0; i < UCHAR_MAX; i++) {
        /* default is to call putchar */
        table[i] = putchar;
    }

    /* but lower-case vowels show up twice */
    table['a'] = putcharTwice;
    table['e'] = putcharTwice;
    table['i'] = putcharTwice;
    table['o'] = putcharTwice;
    table['u'] = putcharTwice;

    while((c = getchar()) != EOF) {
        table[c](c);
    }

    return 0;
}
```

examples/pointers/dispatchTable.c

And here is the program translating Shakespeare into mock-Swedish:

```
$ c99 -Wall -pedantic -g3 -o dispatchTable dispatchTable.c
$ echo Now is the winter of our discontent made glorious summer by this sun of York. | ./dis
Noow iis thee wiinteer oof oouur diiscoonteent maadee glooriioouus suummeer by thiis suun oo
```

In this particular case, we did a lot of work to avoid just writing a `switch` statement. But being able to build a dispatch table dynamically can be very useful sometimes. An example might be a graphical user interface where each button has an associated function. If buttons can be added by different parts of the program, using a table mapping buttons to functions allows a single dispatch routine to figure out where to route button presses.

(For some applications, we might want to pass additional information in to the function to change its behavior. This can be done by replacing the function pointers with closures.)

4.9.9 The restrict keyword

In C99, it is possible to declare that a pointer variable is the only way to reach its target as long as it is in scope. This is not enforced by the compiler; instead, it is a promise from the programmer *to* the compiler that any data reached through this point will not be changed by other parts of the code, which allows the compiler to optimize code in ways that are not possible if pointers might point to the same place (a phenomenon called **pointer aliasing**). For example, consider the following short function:

```c
// write 1 + *src to *dst and return *src
int
copyPlusOne(int * restrict dst, int * restrict src)
{
    *dst = *src + 1;
    return *src;
}
```

For this function, the output of `c99 -O3 -S` includes one more instruction if the `restrict` qualifiers are removed. The reason is that if `dst` and `src` may point to the same location, `src` needs to be re-read for the `return` statement, in case it changed. But if they are guaranteed to point to different locations, the compiler can re-use the previous value it already has in one of the CPU registers.

For most code, this feature is useless, and potentially dangerous if someone calls your routine with aliased pointers. However, it may sometimes be possible to increase performance of time-critical code by adding a `restrict` keyword. The cost is that the code might no longer work if called with aliased pointers.

Curiously, C assumes that two pointers are never aliases if you have two arguments with different pointer types, neither of which is `char *` or `void *`.[10] This is known as the **strict aliasing rule** and cannot be overridden from within the program source code: there is no `unrestrict` keyword. You probably only need to worry about this if you are casting pointers to different types and then passing the cast pointers around in the same context as the original pointers.

4.10 Strings

Processing strings of characters is one of the oldest application of mechanical computers, arguably predating numerical computation by at least fifty years. Assuming you've already solved the problem of how to represent characters in memory (e.g. as the C `char` type encoded in ASCII), there are two standard ways to represent strings:

- As a **delimited string**, where the end of a string is marked by a special character. The advantages of this method are that only one extra byte is needed to indicate the length of an arbitrarily long string, that strings can be manipulated by simple pointer operations, and in some cases that common string operations that involve processing the entire string can be performed very quickly. The disadvantage is that the delimiter can't appear inside any string, which limits what kind of data you can store in a string.
- As a **counted string**, where the string data is prefixed or supplemented with an explicit count of the number of characters in the string. The

[10] The reason for excluding `char *` and `void *` is that these are often used to represent pointers to objects with arbitrary types.

advantage of this representation is that a string can hold arbitrary data (including delimiter characters) and that one can quickly jump to the end of the string without having to scan its entire length. The disadvantage is that maintaining a separate count typically requires more space than adding a one-byte delimiter (unless you limit your string length to 255 characters) and that more care needs to be taken to make sure that the count is correct.

4.10.1 C strings

Because delimited strings are simpler and take less space, C went for delimited strings. A string is a sequence of characters terminated by a null character `'\0'`. Looking back from almost half a century later, this choice may have been a mistake in the long run, but we are pretty much stuck with it.

Note that the null character is *not* the same as a null pointer, although both appear to have the value `0` when used in integer contexts. A string is represented by a variable of type `char *`, which points to the zeroth character of the string. The programmer is responsible for allocating and managing space to store strings, except for explicit **string constants**, which are stored in a special non-writable string space by the compiler.

If you want to use counted strings instead, you can build your own using a `struct`. Most scripting languages written in C (e.g. Perl, Python_programming_language, PHP, etc.) use this approach internally. (Tcl is an exception, which is one of many good reasons not to use Tcl).

4.10.2 String constants

A string constant in C is represented by a sequence of characters within double quotes. Standard C character escape sequences like `\n` (newline), `\r` (carriage return), `\a` (bell), `\0x17` (character with hexadecimal code `0x17`), `\\` (backslash), and `\"` (double quote) can all be used inside string constants. The value of a string constant has type `const char *`, and can be assigned to variables and passed as function arguments or return values of this type.

Two string constants separated only by whitespace will be concatenated by the compiler as a single constant: `"foo" "bar"` is the same as `"foobar"`. This feature is not much used in normal code, but shows up sometimes in macros.

4.10.2.1 String encodings Standard C strings are assumed to be in ASCII, a 7-bit code developed in the 1960s to represent English-language text. If you want to write text that includes any letters not in the usual 26-letter Latin alphabet, you will need to use a different encoding. C does not provide very good support for this, but for fixed strings, you can often get away with using Unicode as long as both your text editor and your terminal are set to use the UTF-8 encoding.

The reason this works is that UTF-8 encodes each Unicode character as one or more 8-bit characters, and does this in a way that guarantees that you never accidentally create a null. So a C string containing UTF-8 characters looks like an ordinary C string to all the C library routines. This also works if you include a Unicode string with a UTF-8 encoding in a comment, as illustrated in the file unicode.c. But this use of Unicode in C is very limited.

Some issues you will quickly run into if you are trying to do something more sophisticated:

```
1. You cannot use non-ASCII letters anywhere outside a string constant or comment without co
1. If you include a UTF-8 encoded string somewhere, even though both your text editor and te
1. You can't generally put a multibyte character into a `char` variable, or write it as a `c
1. You may find out that some other tools have their own ideas about what encodings to expec
```

There exists libraries for working with Unicode strings in C, but they are clunky. If you need to handle a lot of non-ASCII text, you may be better of working with a different language. However, even moving away from C is not always a panacea, and Unicode support in other tools may be hit-or-miss.

4.10.3 String buffers

The problem with string constants is that you can't modify them. If you want to build strings on the fly, you will need to allocate space for them. The traditional approach is to use a **buffer**, an array of `char`s. Here is a particularly painful hello-world program that builds a string by hand:

```c
#include <stdio.h>

int
main(int argc, char **argv)
{
    char hi[3];

    hi[0] = 'h';
    hi[1] = 'i';
    hi[2] = '\0';

    puts(hi);

    return 0;
}
```

examples/strings/hi.c

Note that the buffer needs to have size at least 3 in order to hold all three characters. A common error in programming with C strings is to forget to leave space for the null at the end (or to forget to add the null, which can have comical results depending on what you are using your surprisingly long string for).

4.10.3.1 String buffers and the perils of `gets`

Fixed-size buffers are a common source of errors in older C programs, particularly ones written with the library routine `gets`. The problem is that if you do something like

```
strcpy(smallBuffer, bigString);
```

the `strcpy` function will happily keep copying characters across memory long after it has passed the end of `smallBuffer`. While you can avoid this to a certain extent when you control where `bigString` is coming from, the situation becomes particularly fraught if the string you are trying to store comes from the input, where it might be supplied by anybody, including somebody who is trying to execute a **buffer overrun attack** to seize control of your program.

If you do need to read a string from the input, you should allocate the receiving buffer using `malloc` and expand it using `realloc` as needed. Below is a program that shows how to do this, with some bad alternatives commented out:

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

#define NAME_LENGTH (2)

#define INITIAL_LINE_LENGTH (2)

/* return a freshly-malloc'd line with next line of input from stdin */
char *
getLine(void)
{
    char *line;
    int size;   /* how much space do I have in line? */
    int length; /* how many characters have I used */
    int c;

    size = INITIAL_LINE_LENGTH;
    line = malloc(size);
    assert(line);

    length = 0;

    while((c = getchar()) != EOF && c != '\n') {
        if(length >= size-1) {
```

```
            /* need more space! */
            size *= 2;

            /* make length equal to new size */
            /* copy contents if necessary */
            line = realloc(line, size);
        }

        line[length++] = c;
    }

    line[length] = '\0';

    return line;
}

int
main(int argc, char **argv)
{
    int x = 12;
    /* char name[NAME_LENGTH]; */
    char *line;
    int y = 17;

    puts("What is your name?");

    /* gets(name); */                       /* may overrun buffer */
    /* scanf("%s\n", name); */              /* may overrun buffer */
    /* fgets(name, NAME_LENGTH, stdin); */  /* may truncate input */
    line = getLine();                      /* has none of these problems */

    printf("Hi %s!  Did you know that x == %d and y == %d?\n", line, x, y);

    free(line);  /* but we do have to free line when we are done with it */

    return 0;
}
```

examples/strings/getLine.c

4.10.4 Operations on strings

Unlike many programming languages, C provides only a rudimentary string-processing library. The reason is that many common string-processing tasks in C can be done very quickly by hand.

For example, suppose we want to copy a string from one buffer to another. The library function `strcpy` declared in `string.h` will do this for us (and is usually the right thing to use), but if it didn't exist we could write something very close to it using a famous C idiom.

```
void
strcpy2(char *dest, const char *src)
{
    /* This line copies characters one at a time from *src to *dest. */
    /* The postincrements increment the pointers (++ binds tighter than *) */
    /*  to get to the next locations on the next iteration through the loop. */
    /* The loop terminates when *src == '\0' == 0. */
    /* There is no loop body because there is nothing to do there. */
    while(*dest++ = *src++);
}
```

The externally visible difference between `strcpy2` and the original `strcpy` is that `strcpy` returns a `char *` equal to its first argument. It is also likely that any implementation of `strcpy` found in a recent C library takes advantage of the width of the memory data path to copy more than one character at a time.

Most C programmers will recognize the `while(*dest++ = *src++);` from having seen it before, although experienced C programmers will generally be able to figure out what such highly abbreviated constructions mean. Exposure to such constructions is arguably a form of hazing.

Because C pointers act exactly like array names, you can also write `strcpy2` using explicit array indices. The result is longer but may be more readable if you aren't a C fanatic.

```
char *
strcpy2a(char *dest, const char *src)
{
    int ;

    i = 0;
    for(i = 0; src[i] != '\0'; i++) {
        dest[i] = src[i];
    }

    /* note that the final null in src is not copied by the loop */
    dest[i] = '\0';

    return dest;
}
```

An advantage of using a separate index in `strcpy2a` is that we don't trash `dest`, so we can return it just like `strcpy` does. (In fairness, `strcpy2` could have saved a copy of the original location of `dest` and done the same thing.)

Note that nothing in strcpy2, strcpy2a, or the original strcpy will save you if dest points to a region of memory that isn't big enough to hold the string at src, or if somebody forget to tack a null on the end of src (in which case strcpy will just keep going until it finds a null character somewhere). As elsewhere, it's your job as a programmer to make sure there is enough room. Since the compiler has no idea what dest points to, this means that you have to remember how much room is available there yourself.

If you are worried about overrunning dest, you could use strncpy instead. The strncpy function takes a third argument that gives the maximum number of characters to copy; however, if src doesn't contain a null character in this range, the resulting string in dest won't either. Usually the only practical application to strncpy is to extract the first k characters of a string, as in

```
/* copy the substring of src consisting of characters at positions
    start..end-1 (inclusive) into dest */
/* If end-1 is past the end of src, copies only as many characters as
    available. */
/* If start is past the end of src, the results are unpredictable. */
/* Returns a pointer to dest */
char *
copySubstring(char *dest, const char *src, int start, int end)
{
    /* copy the substring */
    strncpy(dest, src + start, end - start);

    /* add null since strncpy probably didn't */
    dest[end - start] = '\0';

    return dest;
}
```

Another quick and dirty way to extract a substring of a string you don't care about (and can write to) is to just drop a null character in the middle of the sacrificial string. This is generally a bad idea unless you are certain you aren't going to need the original string again, but it's a surprisingly common practice among C programmers of a certain age.

A similar operation to strcpy is strcat. The difference is that strcat concatenates src on to the end of dest; so that if dest previous pointed to "abc" and src to "def", dest will now point to "abcdef". Like strcpy, strcat returns its first argument. A no-return-value version of strcat is given below.

```
void
strcat2(char *dest, const char *src)
{
    while(*dest) dest++;
    while(*dest++ = *src++);
```

```
}
```

Decoding this abomination is left as an exercise for the reader. There is also a function `strncat` which has the same relationship to `strcat` that `strncpy` has to `strcpy`.

As with `strcpy`, the actual implementation of `strcat` may be much more subtle, and is likely to be faster than rolling your own.

4.10.5 Finding the length of a string

Because the length of a string is of fundamental importance in C (e.g., when deciding if you can safely copy it somewhere else), the standard C library provides a function `strlen` that counts the number of non-null characters in a string. Note that if you are allocating space for a copy of a string, you will need to add one to the value returned by `strlen` to account for the null.

Here's a possible implementation:

```
int
strlen(const char *s)
{
    int i;

    for(i = 0; *s; i++, s++);

    return i;
}
```

Note the use of the comma operator in the increment step. The comma operator applied to two expressions evaluates both of them and discards the value of the first; it is usually used only in `for` loops where you want to initialize or advance more than one variable at once.

Like the other string routines, using `strlen` requires including `string.h`.

4.10.5.1 The strlen tarpit

A common mistake is to put a call to `strlen` in the header of a loop; for example:

```
/* like strcpy, but only copies characters at indices 0, 2, 4, ...
   from src to dest */
char *
copyEvenCharactersBadVersion(char *dest, const char *src)
{
    int i;
    int j;

    /* BAD: Calls strlen on every pass through the loop */
```

```
    for(i = 0, j = 0; i < strlen(src); i += 2, j++) {
        dest[j] = src[i];
    }

    dest[j] = '\0';

    return dest;
}
```

The problem is that `strlen` has to scan all of `src` every time the test is done, which adds time proportional to the length of `src` to each iteration of the loop. So `copyEvenCharactersBadVersion` takes time proportional to the *square* of the length of `src`.

Here's a faster version:

```
/* like strcpy, but only copies characters at indices 0, 2, 4, ...
   from src to dest */
char *
copyEvenCharacters(char *dest, const char *src)
{
    int i;
    int j;
    int len;    /* length of src */

    len = strlen(src);

    /* GOOD: uses cached value of strlen(src) */
    for(i = 0, j = 0; i < len; i += 2, j++) {
        dest[j] = src[i];
    }

    dest[j] = '\0';

    return dest;
}
```

Because it doesn't call `strlen` all the time, this version of `copyEvenCharacters` will run much faster than the original even on small strings, and several million times faster if `src` is megabytes long.

4.10.6 Comparing strings

If you want to test if strings `s1` and `s2` contain the same characters, writing `s1 == s2` won't work, since this tests instead whether `s1` and `s2` point to the same address. Instead, you should use `strcmp`, declared in `string.h`. The `strcmp` function walks along both of its arguments until it either hits a null on both and returns 0, or hits two different characters, and returns a positive integer if the first string's character is bigger and a negative integer if the second string's character is bigger (a typical implementation will just subtract the two characters). A straightforward implementation might look like this:

```
int
strcmp(const char *s1, const char *s2)
{
    while(*s1 && *s2 && *s1 == *s2) {
        s1++;
        s2++;
    }

    return *s1 - *s2;
}
```

To use `strcmp` to test equality, test if the return value is `0`:

```
    if(strcmp(s1, s2) == 0) {
        /* strings are equal */
        ...
    }
```

You may sometimes see this idiom instead:

```
    if(!strcmp(s1, s2)) {
        /* strings are equal */
        ...
    }
```

My own feeling is that the first version is more clear, since `!strcmp` always suggested to me that you were testing for the negation of some property (e.g. not equal). But if you think of `strcmp` as telling you when two strings are different rather than when they are equal, this may not be so confusing.

4.10.7 Formatted output to strings

You can write formatted output to a string buffer with `sprintf` just like you can write it to `stdout` with `printf` or to a file with `fprintf`. Make sure when you do so that there is enough room in the buffer you are writing to, or the usual bad things will happen.

4.10.8 Dynamic allocation of strings

When allocating space for a copy of a string `s` using `malloc`, the required space is `strlen(s)+1`. Don't forget the `+1`, or bad things may happen.[11]

Because allocating space for a copy of a string is such a common operation, many C libraries provide a `strdup` function that does exactly this. If you don't have one (it's not required by the C standard), you can write your own like this:

```c
/* return a freshly-malloc'd copy of s */
/* or 0 if malloc fails */
/* It is the caller's responsibility to free the returned string when done. */
char *
strdup(const char *s)
{
    char *s2;

    s2 = malloc(strlen(s)+1);

    if(s2 != 0) {
        strcpy(s2, s);
    }

    return s2;
}
```

Exercise: Write a function `strcatAlloc` that returns a freshly-malloc'd string that concatenates its two arguments. Exactly how many bytes do you need to allocate?

4.10.9 Command-line arguments

Now that we know about strings, we can finally do something with `argc` and `argv`.

Recall that `argv` in `main` is declared as `char **`; this means that it is a pointer to a pointer to a `char`, or in this case the base address of an array of pointers to `char`, where each such pointer references a string. These strings correspond to the command-line arguments to your program, with the program name itself appearing in `argv[0]`[12]

[11] In this case you will get lucky most of the time, since the odds are that malloc will give you a block that is slightly bigger than `strlen(s)` anyway. But bugs that only manifest themselves occasionally are even worse than bugs that kill your program every time, because they are much harder to track down.

[12] Some programs (e.g. `/c/cs223/bin/submit`) will use this to change their behavior depending on what name you call them with.

The count `argc` counts all arguments including `argv[0]`; it is `1` if your program is called with no arguments and larger otherwise.

Here is a program that prints its arguments. If you get confused about what `argc` and `argv` do, feel free to compile this and play with it:

```
#include <stdio.h>

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

    printf("argc = %d\n\n", argc);

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

    return 0;
}
```

examples/strings/printArgs.c

Like strings, C terminates `argv` with a null: the value of `argv[argc]` is always 0 (a null pointer to `char`). In principle this allows you to recover `argc` if you lose it.

4.11 Structured data types

C has two kinds of structured data types: `struct`s and `union`s. A `struct` holds multiple values in consecutive memory locations, called **fields**, and implements what in type theory is called a **product type**: the set of possible values is the Cartesian product of the sets of possible values for its fields. In contrast, a `union` has multiple fields but they are all stored in the same location: effectively, this means that only one field at a time can hold a value, making a `union` a **sum type** whose set of possible values is the union of the sets of possible values for each of its fields. Unlike what happens in more sensible programming languages, `union`s are not tagged: unless you keep track of this somewhere else, you can't tell which field in a union is being used, and you can store a value of one type in a `union` and try to read it back as a different type, and C won't complain.[13]

[13]There are various ways to work around this. The simplest is to put a `union` inside a larger `struct` that includes an explicit type tag.

4.11.1 Structs

A `struct` is a way to define a type that consists of one or more other types pasted together. Here's a typical `struct` definition:

```
struct string {
    int length;
    char *data;
};
```

This defines a new type `struct string` that can be used anywhere you would use a simple type like `int` or `float`. When you declare a variable with type `struct string`, the compiler allocates enough space to hold both an `int` and a `char *` (8 bytes on a typical 32-bit machine). You can get at the individual components using the `.` operator, like this:

```
struct string {
    int length;
    char *data;
};

int
main(int argc, char **argv)
{
    struct string s;

    s.length = 4;
    s.data = "this string is a lot longer than you think";

    puts(s.data);

    return 0;
}
```

examples/structs/structExample.c

Variables of type `struct` can be assigned to, passed into functions, returned from functions, just like any other type. Each such operation is applied componentwise; for example, `s1 = s2;` is equivalent to `s1.length = s2.length; s1.data = s2.data;`.

These operations are not used as often as you might think: typically, instead of copying around entire structures, C programs pass around pointers, as is done with arrays. Pointers to `struct`s are common enough in C that a special syntax is provided for dereferencing them.[14] Suppose we have:

[14]Arguably, this is a bug in the design of the language: if the compiler knows that `sp` has type `struct string *`, there is no particular reason why it can't interpret `sp.length` as `sp->length`. But it doesn't do this, so you will have to remember to write `sp->length` instead.

```c
struct string s;             /* a struct */
struct string *sp;           /* a pointer to a struct */

s.length = 4;
s.data = "another overly long string";

sp = &s;                     /* sp now points to s */
```

We can then refer to elements of the `struct string` that `sp` points to (i.e. `s`) in either of two ways:

```c
puts((*sp).data);
puts(sp->data);
```

The second is more common, since it involves typing fewer parentheses. It is an error to write `*sp.data` in this case; since `.` binds tighter than `*`, the compiler will attempt to evaluate `sp.data` first and generate an error, since `sp` doesn't have a `data` field.

Pointers to `struct`s are commonly used in defining abstract data data, since it is possible to declare that a function returns e.g. a `struct string *` without specifying the components of a `struct string`. (All pointers to `struct`s in C have the same size and structure, so the compiler doesn't need to know the components to pass around the address.) Hiding the components discourages code that shouldn't look at them from doing so, and can be used, for example, to enforce consistency between fields.

For example, suppose we wanted to define a `struct string *` type that held counted strings that could only be accessed through a restricted interface that prevented (for example) the user from changing the string or its length. We might create a file `myString.h` that contained the declarations:

```c
/* make a struct string * that holds a copy of s */
/* returns 0 if malloc fails */
struct string *makeString(const char *s);

/* destroy a struct string * */
void destroyString(struct string *);

/* return the length of a struct string * */
int stringLength(struct string *);

/* return the character at position index in the struct string * */
/* or returns -1 if index is out of bounds */
int stringCharAt(struct string *s, int index);
```

examples/myString/myString.h

and then the actual implementation in `myString.c` would be the only place where the components of a `struct string` were defined:

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

#include "myString.h"

struct string {
    int length;
    char *data;
};

struct string *
makeString(const char *s)
{
    struct string *s2;

    s2 = malloc(sizeof(struct string));
    if(s2 == 0) { return 0; }  /* let caller worry about malloc failures */

    s2->length = strlen(s);

    s2->data = malloc(s2->length);
    if(s2->data == 0) {
        free(s2);
        return 0;
    }

    strncpy(s2->data, s, s2->length);

    return s2;
}

void
destroyString(struct string *s)
{
    free(s->data);
    free(s);
}

int
stringLength(struct string *s)
{
    return s->length;
}

int
stringCharAt(struct string *s, int index)
```

```
{
    if(index < 0 || index >= s->length) {
        return -1;
    } else {
        return s->data[index];
    }
}
```

examples/myString/myString.c

In practice, we would probably go even further and replace all the `struct string *` types with a new name declared with `typedef`.

4.11.1.1 Operations on structs

What you can do to structs is pretty limited: you can look up or set individual components in a struct, you can pass structs to functions or as return values from functions (which makes a copy of the original struct), and you can assign the contents of one struct to another using `s1 = s2` (which is equivalent to copying each component separately).

One thing that you *can't* do is test two structs for equality using `==`; this is because structs may contain extra space holding junk data. If you want to test for equality, you will need to do it componenti by component.

4.11.1.2 Layout in memory

The C99 standard guarantees that the components of a `struct` are stored in memory in the same order that they are defined in: that is, later components are placed at higher address. This allows sneaky tricks like truncating a structure if you don't use all of its components. Because of alignment restrictions, the compiler may add padding between components to put each component on its prefered alignment boundary.

You can find the position of a component within a `struct` using the `offsetof` macro, which is defined in `stddef.h`. This returns the number of bytes from the base of the struct that the component starts at, and can be used to do various terrifying non-semantic things with pointers.

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

int
main(int argc, char **argv)
{
    struct foo {
        int i;
        char c;
        double d;
```

```
        float f;
        char *s;
    };

    printf("i is at %lu\n", offsetof(struct foo, i));
    printf("c is at %lu\n", offsetof(struct foo, c));
    printf("d is at %lu\n", offsetof(struct foo, d));
    printf("f is at %lu\n", offsetof(struct foo, f));
    printf("s is at %lu\n", offsetof(struct foo, s));

    return 0;
}
```

examples/structs/offsetof.c

4.11.1.3 Bit fields

It is possible to specify the exact number of bits taken up by a member of a `struct` of integer type. This is seldom useful, but may in principle let you pack more information in less space. Bit fields are sometimes used to unpack data from an external source that uses this trick, but this is dangerous, because there is no guarantee that the compiler will order the bit fields in your `struct` in any particular order (at the very least, you will need to worry about endianness.

Example:

```
struct color {
    unsigned int red   : 2;
    unsigned int green : 2;
    unsigned int blue  : 2;
    unsigned int alpha : 2;
};
```

This defines a `struct` that (probably) occupies only one byte, and supplies four 2-bit fields, each of which can hold values in the range 0-3.

4.11.2 Unions

A `union` is just like a `struct`, except that instead of allocating space to store all the components, the compiler only allocates space to store the largest one, and makes all the components refer to the same address. This can be used to save space if you know that only one of several components will be meaningful for a particular object. An example might be a type representing an object in a LISP-like language like Scheme:

```
struct lispObject {
    int type;           /* type code */
    union {
```

```
        int     intVal;
        double  floatVal;
        char *  stringVal;
        struct {
            struct lispObject *car;
            struct lispObject *cdr;
        } consVal;
    } u;
};
```

Now if you wanted to make a `struct lispObject` that held an integer value, you might write

```
    lispObject o;

    o.type = TYPE_INT;
    o.u.intVal = 27;
```

Here `TYPE_INT` has presumably been defined somewhere. Note that nothing then prevents you from writing

```
    x = 2.7 * o.u.floatVal;        /* BAD */
```

The effects of this will be strange, since it's likely that the bit pattern representing 27 as an `int` represents something very different as a `double`. Avoiding such mistakes is your responsibility, which is why most uses of `union` occur inside larger `struct`s that contain enough information to figure out which variant of the `union` applies.

4.11.3 Enums

C provides the `enum` construction for the special case where you want to have a sequence of named constants of type `int`, but you don't care what their actual values are, as in

```
enum color { RED, BLUE, GREEN, MAUVE, TURQUOISE };
```

This will assign the value `0` to `RED`, `1` to `BLUE`, and so on. These values are effectively of type `int`, although you can declare variables, arguments, and return values as type `enum color` to indicate their intended interpretation.

Despite declaring a variable `enum color c` (say), the compiler will still allow `c` to hold arbitrary values of type `int`.
So the following ridiculous code works just fine:

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

enum foo { FOO };
```

```c
enum apple { MACINTOSH, CORTLAND, RED_DELICIOUS };
enum orange { NAVEL, CLEMENTINE, TANGERINE };

int
main(int argc, char **argv)
{
    enum foo x;

    if(argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        return 1;
    }

    printf("FOO = %d\n", FOO);
    printf("sizeof(enum foo) = %d\n", sizeof(enum foo));

    x = 127;

    printf("x = %d\n", x);

    /* note we can add apples and oranges */
    printf("%d\n", RED_DELICIOUS + TANGERINE);

    return 0;
}
```

examples/definitions/enumsAreInts.c

4.11.3.1 Specifying particular values

It is also possible to specify particular values for particular enumerated constants, as in

```c
enum color { RED = 37, BLUE = 12, GREEN = 66, MAUVE = 5, TURQUOISE };
```

Anything that doesn't get a value starts with one plus the previous value; so the above definition would set `TURQUOISE` to `6`. This may result in two names mapping to the same value.

4.11.3.2 What most people do

In practice, `enum`s are seldom used, and you will more commonly see a stack of `#define`s:

```c
#define RED       (0)
#define BLUE      (1)
#define GREEN     (2)
#define MAUVE     (3)
#define TURQUOISE (4)
```

The reason for this is partly historical—`enum` arrived late in the evolution of C—but partly practical: a table of `#define`s makes it much easier to figure out which color is represented by 3, without having to count through a list. But if you never plan to use the numerical values, `enum` may be a better choice, because it guarantees that all the values will be distinct.

4.11.3.3 Using `enum` with `union`

A natural place to use an `enum` is to tag a `union` with the type being used. For example, a Lisp-like language might implement the following multi-purpose data type:

```
enum TypeCode { TYPE_INT, TYPE_DOUBLE, TYPE_STRING };

struct LispValue {
    enum TypeCode typeCode;
    union {
        int i;
        double d;
        char *s;
    } value;
};
```

Here we don't care what the numeric values of `TYPE_INT`, `TYPE_DOUBLE`, and `TYPE_STRING` are, as long as we can apply `switch` to `typeCode` to figure out what to do with one of these things.

4.12 Type aliases using `typedef`

Suppose that you want to represent character strings as

```
struct string {
    int length;
    char *data;         /* malloc'd block */
};

int stringLength(const struct string *s);
```

If you later change the representation to, say, traditional null-terminated `char *` strings or some even more complicated type (`union string **some_string[2];`), you will need to go back and replace ever occurrence of `struct string *` in every program that uses it with the new type. Even if you don't expect to change the type, you may still get tired of typing `struct string *` all the time, especially if your fingers slip and give you `struct string` sometimes.

The solution is to use a `typedef`, which defines a new type name:

```
typedef struct string *String;
```

```
int stringLength(const String s);
```

The syntax for `typedef` looks like a variable declaration preceded by `typedef`, except that the variable is replaced by the new type name that acts like whatever type the defined variable would have had. You can use a name defined with `typedef` anywhere you could use a normal type name, as long as it is later in the source file than the `typedef` definition. Typically `typedef`s are placed in a header file (`.h` file) that is then included anywhere that needs them.

You are not limited to using `typedef`s only for complex types. For example, if you were writing numerical code and wanted to declare overtly that a certain quantity was not just any `double` but actually a length in meters, you could write

```
typedef double LengthInMeters;
typedef double AreaInSquareMeters;

AreaInSquareMeters rectangleArea(LengthInMeters height, LengthInMeters width);
```

Unfortunately, C does not do type enforcement on `typedef`'d types: it is perfectly acceptable to the compiler if you pass a value of type `AreaInSquareMeters` as the first argument to `rectangleArea`, since by the time it checks it has replaced by `AreaInSquareMeters` and `LengthInMeters` by `double`. So this feature is not as useful as it might be, although it does mean that you can write `rectangleArea(2.0, 3.0)` without having to do anything to convert `2.0` and `3.0` to type `LengthInMeters`.

4.12.1 Opaque structs

There are certain cases where the compiler needs to know the definition of a `struct`:

1. When the program accesses its components.
2. When the compiler needs to know its size. This may be because you are building an array of these `struct`s, because they appear in a larger `struct`, when you are passing the `struct` as an argument or assigning it to a variable, or just because you applied `sizeof` to the `struct`.

But the compile does *not* need to know the definition of a `struct` to know how create a pointer to it. This is because all `struct` pointers have the same size and structure.

This allows a trick called an **opaque struct**, which can be used for **information hiding**, where one part of your program is allowed to see the definition of a `struct` but other parts are not.

The idea is to create a header file that defines all the functions that might be used to access the `struct`, but does not define the `struct` itself. For example, suppose we want to create a counter, where the user can call a function `increment` that acts like `++` in the sense that it increments the counter and returns the new value, but we don't want to allow the user to change the value of the counter in any other way. This header file defines the **interface** to the counter.

Here is the header file:

```c
/* Create a new counter, initialized to 0.  Call counterDestroy to get rid of it. */
struct counter * counterCreate(void);

/* Free space used by a counter. */
void counterDestroy(struct counter *);

/* Increment a counter and return new value. */
int counterIncrement(struct counter *);
```

examples/structs/opaqueStructs/counter.h

We can now write code that uses the `struct counter *` type without knowing what it is actually pointing to:

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

#include "counter.h"

int
main(int argc, char **argv)
{
    struct counter *c;
    int value;

    c = counterCreate();

    while((value = counterIncrement(c)) < 10) {
        printf("%d\n", value);
    }

    counterDestroy(c);

    return 0;
}
```

examples/structs/opaqueStructs/testCounter.c

To make this work, we do have to provide an **implementation**. The obvious way to do it is have a `struct counter` store the counter value in an `int`, but one could imagine other (probably bad) implementations that did other things, as long as from the outside they acted like we expect.

We only put the definition of a `struct counter` in this file. This means that only functions in this file can access a counter's components, compute the size of a counter, and so forth. While we can't absolutely prevent some other function from extracting or modifying the contents of a counter (C doesn't provide that kind of memory protection), we can at least hint very strongly that the programmer shouldn't be doing this.

```c
#include <stdlib.h>
#include <assert.h>

#include "counter.h"

struct counter {
    int value;
};

struct counter *
counterCreate(void)
{
    struct counter *c;

    c = malloc(sizeof(struct counter));
    assert(c);

    c->value = 0;

    return c;
}

void
counterDestroy(struct counter *c)
{
    free(c);
}

int
counterIncrement(struct counter *c)
{
    return ++(c->value);
}
```

examples/structs/opaqueStructs/counter.c

We will see this trick used over and over again when we build abstract data types.

4.13 Macros

See K&R Appendix A12.3 for full details on macro expansion in ANSI C and http://gcc.gnu.org/onlinedocs/cpp/Macros.html for documentation on what `gcc` supports.

The short version: the command

```
#define FOO (12)
```

causes any occurrence of the word `FOO` in your source file to be replaced by `(12)` by the preprocessor. To count as a word, `FOO` can't be adjacent to other alphanumeric characters, so for example `FOOD` will *not* expand to `(12)D`.

4.13.1 Macros with arguments

To create a macro with arguments, put them in parentheses separated by commas after the macro name, e.g.

```
#define Square(x) ((x)*(x))
```

Now if you write `Square(foo)` it will expand as `((foo)*(foo))`. Note the heavy use of parentheses inside the macro definition to avoid trouble with operator precedence; if instead we had written

```
#define BadSquare(x) x*x
```

then `BadSquare(3+4)` would give `3+4*3+4`, which evaluates to `19`, which is probably not what we intended. The general rule is that macro arguments should always be put in parentheses if you are using them in an expression where precedence might be an issue.

4.13.1.1 Multiple arguments

You can have multiple arguments to a macro, e.g.

```
#define Average(x,y) (((x)+(y))/2.0)
```

The usual caveats about using lots of parentheses apply.

4.13.1.2 Perils of repeating arguments

Macros can have odd effects if their arguments perform side-effects. For example, `Square(++x)` expands to `((++x)*(++x))`; if `x` starts out equal to `1`, this expression may evaluate to any of `2`, `6`, or `9` depending on when the `++` operators are evaluated, and will definitely leave `3` in `x` instead of the `2` the programmer probably expects. For this reason it is generally best to avoid side-effects in macro arguments, and to mark macro names (e.g. by capitalization) to clearly distinguish them from function names, where this issue doesn't come up.

4.13.1.3 Variable-length argument lists

C99 added **variadic macros** that may have a variable number of arguments; these are mostly useful for dealing with variadic functions (like `printf`) that also take a variable number of arguments.

To define a variadic macro, define a macro with arguments where the last argument is three periods: `...` . The macro `__VA_ARGS__` then expands to whatever arguments matched this ellipsis in the macro call.

For example:

```
#include <stdio.h>

#define Warning(...) fprintf(stderr, __VA_ARGS__)

int
main(int argc, char **argv)
{
    Warning("%s: this program contains no useful code\n", argv[0]);

    return 1;
}
```

It is possible to mix regular arguments with `...`, as long as `...` comes last:

```
#define Useless(format, ...) printf(format, __VA_ARGS__)
```

4.13.1.4 Macros vs. inline functions

It is sometimes tempting to use a macro to avoid having to retype some small piece of code that does not seem big enough to justify a full-blown function, especially if the cost of the body of the function is small relative to the cost of a function call. **Inline functions** are a mechanism that is standard in C99 (and found in some compilers for older variants of C) that give you the ability to write a function that will never pay this function call overhead; instead, any call to an inline function is effectively replaced by the body of the function. Unlike parameterized macros, inline functions do not suffer from issues with duplicated parameters or weird text-substitution oddities.

To take a simple example, the `distSquared` function that we used to illustrate function definitions doesn't do very much: just two multiplications and an addition. If we are doing a lot of `distSquared` computations, we could easily double the cost of each computation with function call overhead. One alternative might be to use a macro:

```
#define DistSquared(x,y) ((x)*(x)+(y)*(y))
```

but this suffers from the parameter-duplication problem, which could be particularly unfortunate if we compute `DistSquared(expensiveFunctionWithManySideEffects(), 12)`. A better alternative is to use an inline function.

Like macros, inline functions should be defined in header files. Ordinary functions always go in C files because (a) we only want to compile them once, and (b) the linker will find them in whatever `.o` file they end up in anyway. But inline functions generally don't get compiled independently, so this doesn't apply.

Here is a header file for an inline version of `distSquared`:

```
/* Returns the square of the distance between two points separated by
   dx in the x direction and dy in the y direction. */
static inline int
distSquared(int dx, int dy)
{
    return dx*dx + dy*dy;
}
```

examples/functions/distSquaredInline.h

This looks exactly like the original `distSquared`, except that we added `static inline`. We want this function to be declared `static` because otherwise some compilers will try to emit a non-inline definition for it in ever C file this header is included in, which could have bad results.[15]

The nice thing about this approach is that if we do decide to make `distSquared` an ordinary function (maybe it will make debugging easier, or we realize we want to be able to take its address), then we can just move the definition into a `.c` file and take the `static inline` off. Indeed, this is probably the safest thing to start with, since we can also do the reverse if we find that function call overhead on this particular function really does account for a non-trivial part of our running time (see profiling).

4.13.2 Macros that include other macros

One macro can expand to another; for example, after defining

```
#define FOO BAR
#define BAR (12)
```

it will be the case that `FOO` will expand to `BAR` which will then expand to `(12)`. For obvious reasons, it is a bad idea to have a macro expansion contain the original macro name.

4.13.3 More specialized macros

Some standard idioms have evolved over the years to deal with issues that come up in defining complex macros. Usually, having a complex macro is a sign of

[15] This is also the simplest way to deal with the inconsistencies between different compilers in how they handle inline functions. For an extensive discussion of the terrifying portability issues that arise in pre-C99 C compilers, see http://www.greenend.org.uk/rjk/tech/inline.html.

bad design, but these tools can be useful in some situations.

4.13.3.1 Multiple expressions in a macro

Use the comma operator, e.g.

```
#define NoisyInc(x) (puts("incrementing"), (x)++)
```

The comma operator evaluates both of its operands and returns the value of the one on the right-hand side.

You can also choose between alternatives using the ternary `?:` operator, as in

```
#define Max(a,b) ((a) > (b) ? (a) : (b))
```

(but see the warning about repeated parameters above).

4.13.3.2 Non-syntactic macros

Suppose you get tired of writing

```
    for(i = 0; i < n; i++) ...
```

all the time. In principle, you can write a macro

```
#define UpTo(i, n) for((i) = 0; (i) < (n); (i)++)
```

and then write

```
    UpTo(i, 10) ...
```

in place of your former `for` loop headers. This is generally a good way to make your code completely unreadable. Such macros are called **non-syntactic** because they allow code that doesn't look like syntactically correct C.

Sometimes, however, it makes sense to use non-syntactic macros when you want something that writes to a variable without having to pass it to a function as a pointer. An example might be something like this `malloc` wrapper:

```
#define TestMalloc(x) ((x) = malloc(sizeof(*x)), assert(x))
```

(Strictly speaking, this is probably more of a "non-semantic" macro.)

Whether the confusion of having a non-syntactic macro is worth the gain in safety or code-writing speed is a judgment call that can only be made after long and painful experience. If in doubt, it's probably best not to do it.

4.13.3.3 Multiple statements in one macro

If you want to write a macro that looks like a function call but contains multiple statements, the correct way to do it is like

```
#define HiHi() do { puts("hi"); puts("hi"); } while(0)
```

This can safely be used in place of single statements, like this:[16]

[16] To make the example work, we are violating our usual rule of always using braces in `if` statements.

```c
    if(friendly)
        HiHi();
    else
        snarl();
```

Note that no construct except `do..while` will work here. Just using braces will cause trouble with the semicolon before the `else`, and no other compound statement besides `do..while` expects to be followed by a semicolon in this way.

4.13.3.4 String expansion

Let's rewrite `NoisyInc` to include the variable name:

```c
#define BadNoisyInc2(x) (puts("Incrementing x"), x++)
```

Will this do what we want? No. The C preprocessor is smart enough not to expand macro parameters inside strings, so `BadNoisyInc2(y)` will expand to `(puts("Incrementing x"), y++)`. Instead, we have to write

```c
#define NoisyInc2(x) (puts("Incrementing " #x), x++)
```

Here `#x` expands to whatever the value of `x` is wrapped in double quotes. The resulting string constant is then concatenated with the adjacent string constant according to standard C string constant concatenation rules.

To concatenate things that aren't strings, use the `##` operator, as in

```c
#define FakeArray(n) fakeArrayVariableNumber ## n
```

This lets you write `FakeArray(12)` instead of `fakeArrayVariableNumber12`. Note that there is generally no good reason to ever do this.

Where this feature does become useful is if you want to be able to refer to part of the source code of your program. For example, here is short program that includes a macro that prints the source code and value of an expression:

```c
#include <stdio.h>

#define PrintExpr(x) (printf("%s = %d\n", #x, (x)))

int
main(int argc, char **argv)
{
    PrintExpr(2+2);
    return 0;
}
```

examples/macros/printExpr.c

When run, this program prints

```
2+2 = 4
```

Without using a macro, there is no way to capture the text string "2+2" so we can print it.

This sort of trickery is mostly used in debugging. The `assert` macro is a more sophisticated version, which uses the built-in macros `__FILE__` (which expands to the current source file as a quoted string) and `__LINE__` (which expands to the current source line number, not quoted) to not only print out an offending expression, but also the location of it in the source.

4.13.3.5 Big macros Nothing restricts a macro expansion to a single line, although you must put a backslash at the end of each line to keep it going. Here is a macro that declares a specialized sorting routine for any type that supports `<`:

```
#define DeclareSort(prefix, type) \
static int \
_DeclareSort_ ## prefix ## _Compare(const void *a, const void *b) \
{ \
    const type *aa; const type *bb; \
    aa = a; bb = b; \
    if(*aa < *bb) return -1; \
    else if(*bb < *aa) return 1; \
    else return 0; \
} \
\
void \
prefix ## _sort(type *a, int n)\
{ \
    qsort(a, n, sizeof(type), _DeclareSort_ ## prefix ## _Compare); \
}
```

examples/macros/declareSort.h

A typical use might be

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

#include "declareSort.h"

/* note: must appear outside of any function, and has no trailing semicolon */
DeclareSort(int, int)

#define N (50)

int
main(int argc, char **argv)
```

```c
{
    int a[N];
    int i;

    for(i=0; i < N; i++) {
        a[i] = N-i;
    }

    int_sort(a, N);

    for(i=0; i < N; i++) {
        printf("%d ", a[i]);
    }
    putchar('\n');

    return 0;
}
```

examples/macros/useDeclareSort.c

Do this too much and you will end up reinventing C++ templates, which are a more or less equivalent mechanism for generating polymorphic code that improve on C macros like the one above by letting you omit the backslashes.

4.13.4 Conditional compilation

In addition to generating code, macros can be used for **conditional compilation**, where a section of the source code is included only if a particular macro is defined. This is done using the `#ifdef` and `#ifndef` preprocessor directives. In its simplest form, writing `#ifdef NAME` includes all code up to the next `#endif` if and only if `NAME` is defined. Similarly, `#ifndef NAME` includes all code up to the next `#endif` if and only if `NAME` is *not* defined.

Like regular C `if` statements, `#ifdef` and `#ifndef` directives can be nested, and can include else cases, which are separated by an `#else` directive.

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

int
main(int argc, char **argv)
{
#ifdef SAY_HI
    puts("Hi.");
#else  /* matches #ifdef SAY_HI */
#ifndef BE_POLITE
    puts("Go away!");
```

```
#else  /* matches #ifndef BE_POLITE */
    puts("I'm sorry, I don't feel like talking today.");
#endif /* matches #ifndef BE_POLITE */
#endif /* matches #ifdfe SAY_HI */

#ifdef DEBUG_ARITHMETIC
    assert(2+2 == 5);
#endif

    return 0;
}
```

examples/macros/ifdef.c

4.13.5 Defining macros on the command line

You can turn these conditional compilation directives on and off at compile time by passing the `-D` flag to `gcc`. Here is the program above, running after compiling with different choices of options:

```
$ gcc -DSAY_HI -o ifdef ifdef.c
$ ./ifdef
Hi.
$ gcc -DBE_POLITE -DDEBUG_ARITHMETIC -o ifdef ifdef.c
$ ./ifdef
I'm sorry, I don't feel like talking today.
ifdef: ifdef.c:18: main: Assertion `2+2 == 5' failed.
Aborted
```

An example of how this mechanism can be useful is the `NDEBUG` macro: if you define this before including `assert.h`, it turns every `assert` in your code into a no-op. This can be handy if you are pretty sure your code works and you want to speed it up in its final shipped version, or if you are pretty sure your code doesn't work but you want to hide the evidence. (It also means you should not perform side-effects inside an `assert` unless you are happy with them not happening.)

Using the flag `-DNAME` defines `NAME` to be `1`. If you want something else, use `-DNAME=VALUE`. This can be used to bake useful information into your program at compile time, and is often used to specify filenames. Below is a simple example.

```
#include <stdio.h>

int
main(int argc, char **argv)
{
#ifdef MESSAGE
```

```
    puts(MESSAGE);
#endif

    return 0;
}
```

examples/macros/message.c

```
$ gcc -DMESSAGE='"Hi there!"' -o message message.c
$ ./message
Hi there!
```

Note that we had to put an extra layer of single quotes in the command line to keep the shell from stripping off the double quotes. This is unavoidable: had we written `puts("MESSAGE")` in the code, the preprocessor would have recognized that `MESSAGE` appeared inside a string and would not have replaced it.[17]

4.13.6 The `#if` directive

The preprocessor also includes a more general `#if` directive that evaluates simple arithmetic expressions. The limitations are that it can only do integer arithmetic (using the widest signed integer type available to the compiler) and can only do it to integer and character constants and the special operator `defined(NAME)`, which evaluates to 1 if `NAME` is defined and 0 otherwise. The most common use of this is to combine several `#ifdef`-like tests into one:

```
#include <stdio.h>

int
main(int argc, char **argv)
{
#if VERBOSITY >= 3 && defined(SAY_HI)
    puts("Hi!");
#endif

    return 0;
}
```

examples/macros/if.c

[17]The `#` operator looks like it ought to be useful here, but it only works for expanding arguments to macros and not for expanding macros themselves. Attempting to get around this by wrapping `MESSAGE` in a macro that applies the `#` operator to its first argument will end in tears if `MESSAGE` contains any special characters like commas or right parentheses. The C preprocessor has many unfortunate limitations.

4.13.7 Debugging macro expansions

One problem with using a lot of macros is that you can end up with no idea what input is actually fed to the compiler after the preprocessor is done with it. You can tell `gcc` to tell you how everything expands using `gcc -E source_file.c`. If your source file contains any `#include` statements it is probably a good idea to send the output of `gcc -E` to a file so you can scroll down past the thousands of lines of text they may generate.

4.13.8 Can a macro call a preprocessor command?

E.g., can you write something like

```
#define DefinePlus1(x, y)  #define x ((y)+1)
#define IncludeLib(x)      #include "lib/" #x
```

The answer is **no**. C preprocessor commands are only recognized in unexpanded text. If you want self-modifying macros you will need to use a fancier macro processor like m4.

5 Data structures and programming techniques

Up until this point we have mostly concentrated on the details of the C programming language. In this part of the notes, we will be looking more at how to construct data structures and how to organize a program. In principle, these techniques can be applied to any programming language that supports the appropriate low-level data types, but we will continue to emphasize issues involved with implementation in C.

5.1 Asymptotic notation

Asymptotic notation is a tool for measuring the growth rate of functions, which for program design usually means the way in which the time or space costs of a program scale with the size of the input. We'll start with an example of why this is important.

5.1.1 Two sorting algorithms

Suppose we want to sort in increasing order a deck of n cards, numbered 1 through n. Here are two algorithms for doing this.

In the **mergesort** algorithm, we start with n piles of one card each. We then take pairs of piles and merge them together, by repeatedly pulling the smaller of the two smallest cards off the top of the pile and putting it on the bottom of our output pile. After the first round of this, we have $n/2$ piles of two cards each. After another round, $n/4$ piles of four cards each, and so on until we get one pile with n cards after roughly $\log_2 n$ rounds of merging.

Here's a picture of this algorithm in action on 8 cards:

```
5 7 1 2 3 4 8 6

57 12 34 68

1257 3468

12345678
```

Suppose that we want to estimate the cost of this algorithm without actually coding it up. We might observe that each time a card is merged into a new pile, we need to do some small, fixed number of operations to decide that it's the smaller card, and then do an additional small, fixed number of operations to physically move it to a new place. If we are really clever, we might notice that since the size of the pile a card is in doubles with each round, there can be at most $\lceil \log_2 n \rceil$ rounds until all cards are in the same pile. So the cost of getting a single card in the right place will be at most $c \log n$ where c counts the "small, fixed" number of operations that we keep mentioning, and the cost of getting every card in the right place will be at most $cn \log n$.

In the "'selection sort"' algorithm, we look through all the cards to find the smallest one, swap it to the beginning of the list, then look through the remaining cards for the second smallest, swap it to the next position, and so on.

Here's a picture of this algorithm in action on 8 cards:

```
57123486

17523486

12573486

12375486

12345786

12345786

12345687

12345678
```

This is a simpler algorithm to implement that mergesort, but it is usually slower on large inputs. We can formalize this by arguing that each time we scan k cards to find the smallest, it's going to take some small, fixed number of operations to test each card against the best one we found so far, and an additional small, fixed number of operations to swap the smallest card to the right place. To compute the total cost we have to add these costs for all cards, which will give us a total cost that looks something like $(c_1 n + c_2) + (c_1(n-1) + c_2) + (c_1(n-2) + c_2) + \ldots + (c11 + c_2) = c_1 n(n+1)/2 + c_2 n$.

For large n, it looks like this is going to cost more than mergesort. But how can we make this claim cleanly, particularly if we don't know the exact values of c, c_1, and c_2?

5.1.2 Big-O to the rescue

The idea is to replace complex running time formulae like $cn \log n$ or $c_1 n(n+1)/2 + c_2 n$ with an asymptotic growth rate $O(n \log n)$ or $O(n^2)$. These asymptotic growth rates omit the specific details of exactly how fast our algorithms run (which we don't necessarily know without actually coding them up) and concentrate solely on how the cost scales as the size of the input n becomes large.

This avoids two issues:

1. Different computers run at different speeds, and we'd like to be able to say that one algorithm is better than another without having to measure its running time on specific hardware.
2. Performance on large inputs is more important than performance on small inputs, since programs running on small inputs are usually pretty fast.

The idea of '‘'asymptotic notation"' is to consider the shape of the worst-case cost $T(n)$ to process an input of size n. Here, worst-case means we consider the input that gives the greatest cost, where cost is usually time, but may be something else like space. To formalize the notion of shape, we define classes of functions that behave like particular interesting functions for large inputs. The definition looks much like a limit in calculus:

$O(n)$ A function $f(n)$ is in the class $O(g(n))$ if there exist constants N and c such that $f(n) < c \cdot g(n)$ when $n > N$.

If $f(n)$ is in $O(g(n))$ we say $f(n)$ is '‘'big-O"' of $g(n)$ or just $f(n) = O(g(n))$.[18]

Unpacked, this definition says that $f(n)$ is less than a constant times $g(n)$ when n is large enough.

Some examples:

[18] This is an abuse of notation, where the equals sign is really acting like set membership. The general rule is that an expression $O(f(n)) = O(g(n))$ is true if for any choice of a function in $O(f(n))$, that function is in $O(g(n))$. This relation is transitive and symmetric, but unlike real equality it's not symmetric.

- Let $f(n) = 3n + 12$, and let $g(n) = n$. To show that $f(n)$ is in $O(g(n)) = O(n)$, we can pick whatever constants we want for c and N (as long as they work). So let's make N be 100 and c be 4. Then we need to show that if $n > 100$, $3n + 12 < 4n$. But $3n + 12 < 4n$ holds precisely when $12 < n$, which is implied by our assumption that $n > 100$.
- Let $f(n) = 4n^2 + 23n + 15$, and let $g(n) = n^2$. Now let N be 100 again and c be 5. So we need $4n^2 + 23n + 15 < 5n^2$, or $23n + 15 < n^2$. But $n > 100$ means that $n^2 > 100n = 50n + 50n > 50n + 5000 > 23n + 15$, which proves that $f(n)$ is in $O(n^2)$.
- Let $f(n) < 146$ for all n, and let $g(n) = 1$. Then for $N = 0$ and $c = 146$, $f(n) < 146 = 146g(n)$, and $f(n)$ is in $O(1)$.

Writing proofs like this over and over again is a nuisance, so we can use some basic rules of thumb to reduce messy functions $f(n)$ to their asymptotic forms:

- If c is a constant (doesn't depend on n), then $c \cdot f(n) = O(f(n))$. This follows immediately from being able to pick c in the definition. So we can always get rid of constant factors: $137n^5 = O(n^5)$.
- If $f(n) = g(n) + h(n)$, then the bigger of $g(n)$ or $h(n)$ wins. This is because if $g(n) \leq h(n)$, then $g(n) + h(n) \leq 2g(n)$, and then big-O eats the 2. So $12n^2 + 52n + 3 = O(n^2)$ because n^2 dominates all the other terms.
- To figure out which of two terms dominates, the rule is
  - Bigger exponents win: If $a < b$, then $O(n^a) + O(n^b) = O(n^b)$.
  - Polynomials beat logarithms: For any a and any $b > 0$, $O(\log^a n) + O(n^b) = O(n^b)$.
  - Exponentials beat polynomials: For any a and any $b > 1$, $O(n^a) + O(b^n) = O(b^n)$.
  - The distributive law works: Because $O(\log n)$ dominates $O(1)$, $O(n \log n)$ dominates $O(n)$.

This means that almost any asymptotic bound can be reduced down to one of a very small list of common bounds. Ones that you will typically see in practical algorithms, listed in increasing order, are $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, or $O(n^2)$.

Applying these rules to mergesort and selection sort gives us asymptotic bounds of $cn \log n = O(n \log n)$ (the constant vanishes) and $c_1 n(n+1)/2 + c_2 n = c_1 n^2/2 + c_1 n/2 + c_2 n = O(n^2) + O(n) + O(n) = O(n^2)$ (the constants vanish and then $O(n^2)$ dominates). Here we see that no matter how fast our machine is at different low-level operations, for large enough inputs mergesort will beat selection sort.

5.1.3 Asymptotic cost of programs

To compute the asymptotic cost of a program, the rule of thumb is that any simple statement costs $O(1)$ time to evaluate, and larger costs are the result of loops or calls to expensive functions, where a loop multiplies the cost by the number of iterations in the loop. When adding costs together, the biggest cost wins:

So this function takes $O(1)$ time:

```
/* return the sum of the integers i with 0 <= i  and i < n */
int
sumTo(int n)
{
    return n*(n-1)/2;
}
```

But this function, which computes exactly the same value, takes $O(n)$ time:

```
/* return the sum of the integers i with 0 <= i  and i < n */
int
sumTo(int n)
{
    int i;
    int sum = 0;

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

    return sum;
}
```

The reason it takes so long is that each iteration of the loop takes only $O(1)$ time, but we execute the loop n times, and $n \cdot O(1) = O(n)$.

Here's an even worse version that takes $O(n^2)$ time:

```
/* return the sum of the integers i with 0 <= i  and i < n */
int
sumTo(int n)
{
    int i;
    int j;
    int sum = 0;

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

    return sum;
}
```

Here we have two nested loops. The outer loop iterates exactly n times, and for each iteration the inner loop iterates at most n times, and the innermost iteration costs $O(1)$ each time, so the total is at most $O(n^2)$. (In fact, it's no better than this, because at least $n/2$ times we execute the inner loop, we do at least $n/2$ iterations.)

So even if we knew that the constant on the first implementation was really large (maybe our CPU is bad at dividing by 2?), for big values of n it's still likely to be faster than the other two.

(This example is a little misleading, because n is not the size of the input but the actual input value. More typical might be a statement that the cost of `strlen` is $O(n)$ where n is the length of the string.)

5.1.4 Other variants of asymptotic notation

Big-O notation is good for upper bounds, but the inequality in the definition means that it can't be used for anything else: it is the case that $12 = O(n^{67})$ just because $12 < n^{67}$ when n is large enough. There is an alternative definition, called '‘'big-Omega"', that works in the other direction:

$\Omega(n)$ A function $f(n)$ is in the class $\Omega(g(n))$ if there exist constants N and c such that $f(n) > c \cdot g(n)$ when $n > N$.

This is exactly the same as the definition of $O(g(n))$ except that the inequality goes in the other direction. So if we want to express that some algorithm is very expensive, we might write that it's $\Omega(n^2)$, which says that once the size of the input is big enough, then the cost grows at least as fast as n^2.

If you want to claim that your bound is **tight**—both an upper and a lower bound—use **big-Theta**: $f(n)$ is $\Theta(g(n))$ if it is both $O(f(n))$ and $\Omega(g(n))$.

Mostly we will just use big-O, with the understanding that when we say that a particular algorithm is $O(n)$, that's the best bound we could come up with.

5.2 Linked lists

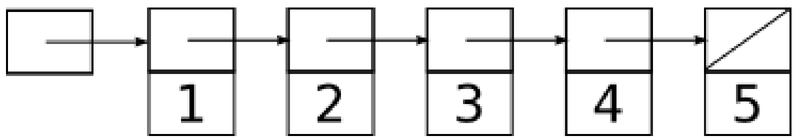
Linked lists are about the simplest data structure beyond arrays. They aren't very efficient for many purposes, but have very good performance for certain specialized applications.

The basic idea is that instead of storing n items in one big array, we store each item in its own `struct`, and each of these `structs` includes a pointer to the next `struct` in the list (with a null pointer to indicate that there are no more elements). If we follow the pointers we can eventually reach all of the elements.

For example, if we declare the struct holding each element like this:

```c
struct elt {
    struct elt *next;  /* pointer to next element in the list */
    int contents;      /* contents of this element */
};
```

We can build a structure like this:

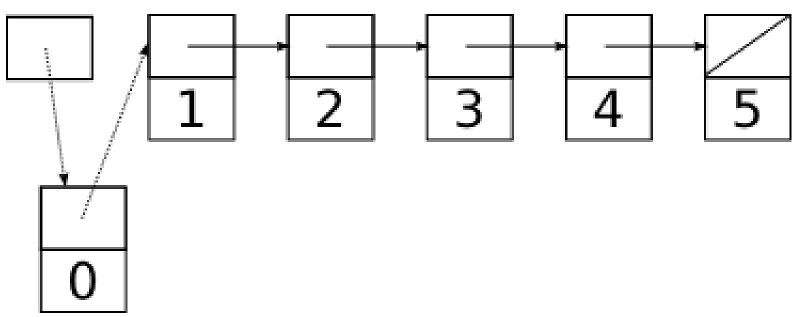


The box on the far left is not a `struct elt`, but a `struct elt *`; in order to keep track of the list we need a pointer to the first element. As usual in C, we will have to do all the work of allocating these elements and assigning the right pointer values in the right places ourselves.

5.2.1 Stacks

The selling point of linked lists in comparison to arrays is that inserting or removing elements can be cheap: at the front of the list, inserting a new element just requires allocating another `struct` and hooking up a few pointers, while removing an element just requires moving the pointer to the first element to point to the second element instead, and then freeing the first element.

For example here's what happens the linked list above looks like after we insert a new element at the front:



To make this work, we need to change two pointers: the head pointer and the `next` pointer in the new element holding 0. These operations aren't affected by the size of the rest of the list and so take $O(1)$ time.

Removal is the reverse of installation: We patch out the first element by shifting the head pointer to the second element, then deallocate it with `free`. (We do have to be careful to get any data we need out of it before calling free). This is also an $O(1)$ operation.

The fact that we can add and remove elements at the start of linked lists for cheap makes them particularly useful for implementing a **stack**, an abstract data type that supports operations **push** (insert a new element on the top of the stack) and **pop** (remove and return the element at the top of the stack. Here is an example of a simple linked-list implementation of a stack, together with some test code:

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

struct elt {
    struct elt *next;
    int value;
};

/*
 * We could make a struct for this,
 * but it would have only one component,
 * so this is quicker.
 */
typedef struct elt *Stack;

#define STACK_EMPTY (0)

/* push a new value onto top of stack */
void
stackPush(Stack *s, int value)
{
    struct elt *e;

    e = malloc(sizeof(struct elt));
    assert(e);

    e->value = value;
    e->next = *s;
    *s = e;
}

int
stackEmpty(const Stack *s)
{
    return (*s == 0);
}

int
```

```c
stackPop(Stack *s)
{
    int ret;
    struct elt *e;

    assert(!stackEmpty(s));

    ret = (*s)->value;

    /* patch out first element */
    e = *s;
    *s = e->next;

    free(e);

    return ret;
}

/* print contents of stack on a single line */
void
stackPrint(const Stack *s)
{
    struct elt *e;

    for(e = *s; e != 0; e = e->next) {
        printf("%d ", e->value);
    }

    putchar('\n');
}

int
main(int argc, char **argv)
{
    int i;
    Stack s;

    s = STACK_EMPTY;

    for(i = 0; i < 5; i++) {
        printf("push %d\n", i);
        stackPush(&s, i);
        stackPrint(&s);
    }

    while(!stackEmpty(&s)) {
```

```
        printf("pop gets %d\n", stackPop(&s));
        stackPrint(&s);
    }

    return 0;
}
```

examples/linkedLists/stack.c

Unlike most of our abstract data types, we do not include a `struct` representing the linked list itself. This is because the only thing we need to keep track of a linked list is the head pointer, and it feels a little silly to have a `struct` with just one component. But we might choose to do this if we wanted to make the linked list implementation opaque or allow for including more information later.

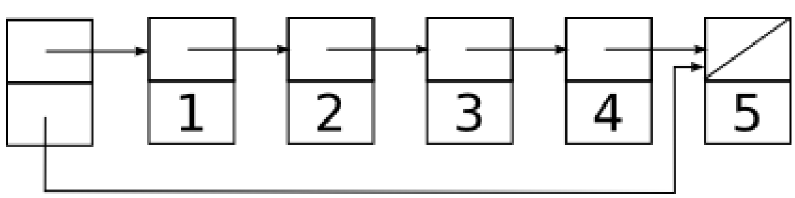
```
struct stack {
    struct elt *head;
};
```

5.2.1.1 Building a stack out of an array When the elements of a stack are small, or when a maximum number of elements is known in advance, it often makes sense to build a stack from an array (with a variable storing the index of the top element) instead of a linked list. The reason is that pushes and pops only require updating the stack pointer instead of calling `malloc` or `free` to allocate space, and pre-allocating is almost always faster than allocating as needed. This is the strategy used to store the function call stack in almost all programs (the exception is in languages like Scheme, where the call stack is allocated on the heap because stack frames may outlive the function call that creates them).

5.2.2 Queues

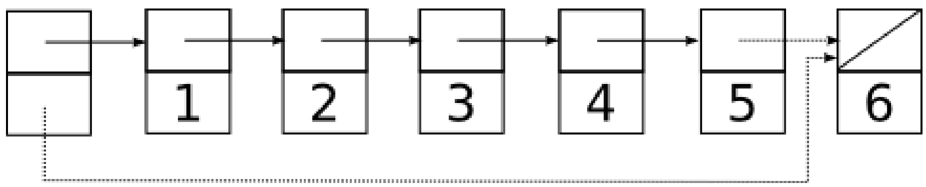
Stacks are last-in-first-out (LIFO) data structures: when we pop, we get the last item we pushed. What if we want a first-in-first-out (FIFO) data structure? Such a data structure is called a **queue** and can also be implemented by a linked list. The difference is that if we want $O(1)$ time for both the **enqueue** (push) and **dequeue** (pop) operations, we must keep around pointers to both ends of the linked list.

So now we get something that looks like this:



Enqueuing a new element typically requires (a) allocating a new `struct` to hold it; (b) making the old tail `struct` point at the new `struct`; and (c) updating the `tail` pointer to also point to the new `struct`. There is a minor complication when the stack is empty; in this case instead of updating `tail->next` we must put a pointer to the new `struct` in `head`. Dequeuing an element involves updating the head pointer and freeing the removed `struct`, exactly like a stack pop.

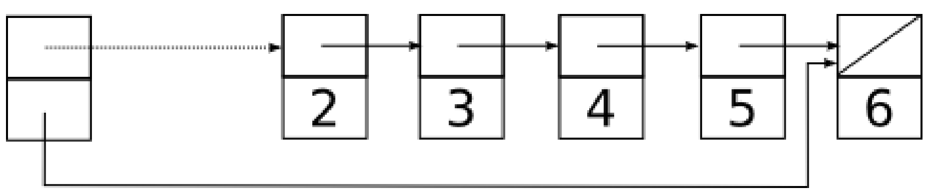
Here is the queue above after enqueuing a new element 6. The updated pointers are indicated by dotted lines:



Because we are only changing two pointers, each of which we can reach by following a constant number of pointers from the main `struct`, we can do this in $O(1)$ time.

There is a slight complication when we enqueue the very first element, because we need to update the head pointer instead of the pointer in the previous tail (which doesn't yet exist). This requires testing for an empty queue in the enqueue routine, which we'll do in the sample code below.

Dequeuing is easier because it requires updating only one pointer:



If we adopt the convention that a null in `head` means an empty queue, and use this property to check if the queue is empty when enqueuing, we don't even have to clear out `tail` when we dequeue the last element.

Here is a simple implementation of a queue holding `int`s, together with some test code showing how its behavior differs from a stack:

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

/* standard linked list element */
struct elt {
    struct elt *next;
```

```c
    int value;
};

struct queue {
    struct elt *head;  /* dequeue this next */
    struct elt *tail;  /* enqueue after this */
};

/* create a new empty queue */
struct queue *
queueCreate(void)
{
    struct queue *q;

    q = malloc(sizeof(struct queue));

    q->head = q->tail = 0;

    return q;
}

/* add a new value to back of queue */
void
enq(struct queue *q, int value)
{
    struct elt *e;

    e = malloc(sizeof(struct elt));
    assert(e);

    e->value = value;

    /* Because I will be the tail, nobody is behind me */
    e->next = 0;

    if(q->head == 0) {
        /* If the queue was empty, I become the head */
        q->head = e;
    } else {
        /* Otherwise I get in line after the old tail */
        q->tail->next = e;
    }

    /* I become the new tail */
    q->tail = e;
}
```

```
int
queueEmpty(const struct queue *q)
{
    return (q->head == 0);
}

/* remove and return value from front of queue */
int
deq(struct queue *q)
{
    int ret;
    struct elt *e;

    assert(!queueEmpty(q));

    ret = q->head->value;

    /* patch out first element */
    e = q->head;
    q->head = e->next;

    free(e);

    return ret;
}

/* print contents of queue on a single line, head first */
void
queuePrint(struct queue *q)
{
    struct elt *e;

    for(e = q->head; e != 0; e = e->next) {
        printf("%d ", e->value);
    }

    putchar('\n');
}

/* free a queue and all of its elements */
void
queueDestroy(struct queue *q)
{
    while(!queueEmpty(q)) {
        deq(q);
```

```
    }

    free(q);
}

int
main(int argc, char **argv)
{
    int i;
    struct queue *q;

    q = queueCreate();

    for(i = 0; i < 5; i++) {
        printf("enq %d\n", i);
        enq(q, i);
        queuePrint(q);
    }

    while(!queueEmpty(q)) {
        printf("deq gets %d\n", deq(q));
        queuePrint(q);
    }

    queueDestroy(q);

    return 0;
}
```

examples/linkedLists/queue.c

It is a bit trickier to build a queue out of an array than to build a stack. The difference is that while a stack pointer can move up and down, leaving the base of the stack in the same place, a naive implementation of a queue would have head and tail pointers both marching ever onward across the array leaving nothing but empty cells in their wake. While it is possible to have the pointers wrap around to the beginning of the array when they hit the end, if the queue size is unbounded the tail pointer will eventually catch up to the head pointer. At this point (as in a stack that overflows), it is necessary to allocate more space and copy the old elements over. See the section on ring buffers for an example of how to do this.

5.2.3 Looping over a linked list

Looping over a linked list is not hard if you have access to the `next` pointers. (For a more abstract way to do this see iterators.)

Let's imagine somebody gave us a pointer to the first `struct stack` in a list; call this pointer `first`. Then we can write a loop like this that prints the contents of the stack:

```
void
stackPrint(struct stack *first)
{
    struct stack *elt;

    for(elt = first; elt != 0; elt = elt->next) {
        puts(elt->book);
    }
}
```

There's not a whole lot to notice here except that `for` is perfectly happy to iterate over something that isn't a range of integers. The running time is linear in the length of the list ($O(n)$).

5.2.4 Looping over a linked list backwards

What if we want to loop over a linked list backwards? The `next` pointers all go the wrong way, so we have to save a trail of breadcrumbs to get back. The safest way to do this is to reverse the original list into an auxiliary list:

```
void
stackPrintReversed(struct stack *first)
{
    struct stack *elt;
    Stack s2;                   /* uses imperative implementation */

    s2 = stackCreate();

    for(elt = first; elt != 0; elt = elt->next) {
        s2 = stackPush(s2, elt->book);
    }

    stackPrint(s2);
    stackDestroy(s2);
}
```

Pushing all the elements from the first list onto `s2` puts the first element on the bottom, so when we print `s2` out, it's in the reverse order of the original stack.

We can also write a recursive function that prints the elements backwards. This function effectively uses the function call stack in place of the extra stack `s2` above.

```c
void
stackPrintReversedRecursive(struct stack *first)
{
    if(first != 0) {
        /* print the rest of the stack */
        stackPrintReversedRecursive(first->next);

        /* then print the first element */
        puts(first->book);
    }
}
```

The code in `stackPrintReversedRecursive` is shorter than the code in `stackPrintReversed`, and it is likely to be faster since it doesn't require allocating a second stack and copying all the elements. But it will only work for small stacks: because the function call stack is really a fixed-size array, if the input to `stackPrintReversedRecursive` is too big the recursion will go too deep cause a *stack overflow*.

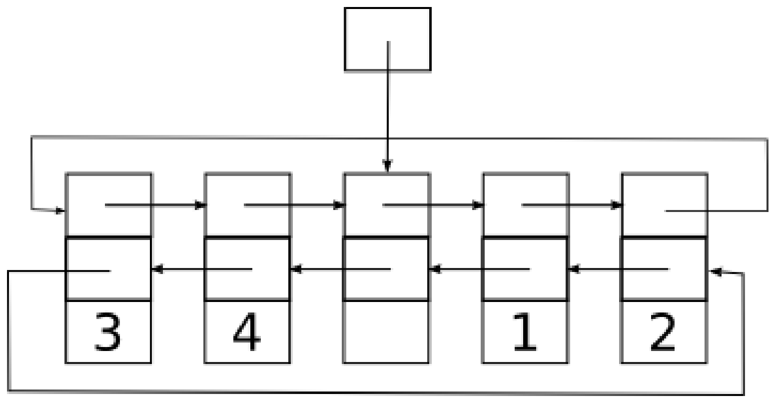
If we want to do this sort of thing a lot, we should build a **doubly-linked list**, with a pointer in each element both to the next element and the previous element instead of a singly-linked list (see below for more).

5.2.5 Deques and doubly-linked lists

Suppose we want a data structure that represents a line of elements where we can push or pop elements at either end. Such a data structure is known as a **deque** (pronounced like "deck"), and can be implemented with all operations taking $O(1)$ time by a **doubly-linked list**, where each element has a pointer to both its successor and its predecessor.

An ordinary singly-linked list is not good enough. The reason is that even if we keep a pointer to both ends as in a queue, when it comes time to pop an element off the tail, we have no pointer to its predecessor ready to hand; the best we can do is scan from the head until we get to an element whose successor is the tail, which takes $O(n)$ time.

So instead we need a doubly-linked list, where each node points to both its successor and predecessor. The most straightforward way to build this is to make it circular, and use a dummy node to represent the head of the list. The resulting data structure might look something like this:



Below is an implementation of this structure. We have separated the interface in `deque.h` from the implementation in `deque.c`. This will allow us to change the implementation if we decide we don't like it, without affecting any other code in the system.

A nice feature of this data structure is that we don't need to use null pointers to mark the ends of the deque. Instead, each end is marked by a pointer to the dummy head element. For an empty deque, this just means that the head points to itself. The cost of this is that to detect an empty deque we have to test for equality with the head (which might be slightly more expensive that just testing for null) and the head may contain some wasted space for its missing value if we allocate it like any other element.[19]

To keep things symmetric, we implement the pointers as an array, indexed by the directions `DEQUE_FRONT` and `DEQUE_BACK` (defined in `deque.h`). This means we can use the same code to push or pop on either end of the deque.

```c
typedef struct deque Deque;

#define DEQUE_FRONT (0)
#define DEQUE_BACK (1)

#define DEQUE_EMPTY (-1)  /* returned by dequePop if deque is empty */

/* return a new empty deque */
Deque *dequeCreate(void);

/* push new value onto direction side of deque d */
void dequePush(Deque *d, int direction, int value);
```

[19] The example below uses the `offsetof` macro, defined in `stddef.h`, to allocate a truncated head that doesn't include this extra space. This is probably more trouble than it is worth in this case, but might be useful if we were creating a lot of dummy heads and the contents were more than 4 bytes long.

```c
/* pop and return first value on direction side of deque d */
/* returns DEQUE_EMPTY if deque is empty */
int dequePop(Deque *d, int direction);

/* return 1 if deque contains no elements, 0 otherwise */
int dequeIsEmpty(const Deque *d);

/* free space used by a deque */
void dequeDestroy(Deque *d);
```

examples/linkedLists/deque/deque.h

```c
#include <stdlib.h>
#include <assert.h>
#include <stddef.h>  /* for offsetof */

#include "deque.h"

#define NUM_DIRECTIONS (2)

struct deque {
    struct deque *next[NUM_DIRECTIONS];
    int value;
};

Deque *
dequeCreate(void)
{
    Deque *d;

    /*
     * We don't allocate the full space for this object
     * because we don't use the value field in the dummy head.
     *
     * Saving these 4 bytes doesn't make a lot of sense here,
     * but it might be more significant if value where larger.
     */
    d = malloc(offsetof(struct deque, value));

    /* test is to deal with malloc failure */
    if(d) {
        d->next[DEQUE_FRONT] = d->next[DEQUE_BACK] = d;
    }

    return d;
}
```

```c
void
dequePush(Deque *d, int direction, int value)
{
    struct deque *e;  /* new element */

    assert(direction == DEQUE_FRONT || direction == DEQUE_BACK);

    e = malloc(sizeof(struct deque));
    assert(e);

    e->next[direction] = d->next[direction];
    e->next[!direction] = d;
    e->value = value;

    d->next[direction] = e;
    e->next[direction]->next[!direction] = e;  /* preserves invariant */
}

int
dequePop(Deque *d, int direction)
{
    struct deque *e;
    int retval;

    assert(direction == DEQUE_FRONT || direction == DEQUE_BACK);

    e = d->next[direction];

    if(e == d) {
        return DEQUE_EMPTY;
    }

    /* else remove it */
    d->next[direction] = e->next[direction];
    e->next[direction]->next[!direction] = d;

    retval = e->value;

    free(e);

    return retval;
}

int
dequeIsEmpty(const Deque *d)
```

```c
{
    return d->next[DEQUE_FRONT] == d;
}

void
dequeDestroy(Deque *d)
{
    while(!dequeIsEmpty(d)) {
        dequePop(d, DEQUE_FRONT);
    }

    free(d);
}
```

examples/linkedLists/deque/deque.c

And here is some test code:

testDeque.c.

5.2.5.1 Alternate implementation using a ring buffer The `deque.h` file carefully avoids revealing any details of the implementation of a deque. This allows us to replace the implementation with a different implementation that is more efficient in its use of both time and space, at the cost of additional code complexity. Below is a replacement for `deque.c` that uses a **ring buffer** in place of the circular linked list.

The idea of a ring buffer is to store the deque elements in an array, with a pointer to the first element and a length field that says how many elements are in the deque. The information needed to manage the array (which is allocated using `malloc`) is stored in a `struct`.

The sequence of elements wraps around the endpoints of the array, leaving a gap somewhere in the middle. Deque pushes extend the sequence into this gap from one side or another, while pops increase the size of the gap. If the user wants to do a push and the array is full, we build a new, larger deque, move all the elements there, and then transplant all the bits of the new `struct deque` into the old one. This transplant trick avoids changing the address of the `struct deque` that the user needs to access it.

```c
#include <stdlib.h>
#include <assert.h>

#include "deque.h"

/*
 * Alternative implementation of a deque using a ring buffer.
 *
```

```c
 * Conceptually, this is an array whose indices wrap around at
 * the endpoints.
 *
 * The region in use is specified by a base index pointing
 * to the first element, and a length count giving the number
 * of elements.  A size field specifies the number of slots
 * in the block.
 *
 * Picture:
 *
 *  ---------------------------------------------------
 *  |7|8|9| | | | | | | | | | | | | | | |1|2|3|4|5|6|
 *  ---------------------------------------------------
 *       ^                                 ^
 *       |                                 |
 *       base + length - 1                 base
 *
 */

struct deque {
    size_t base;     /* location of front element */
    size_t length;   /* length of region in use */
    size_t size;     /* total number of positions in contents */
    int *contents;
};

#define INITIAL_SIZE (8)

/* create a new deque of the given size */
static Deque *
dequeCreateInternal(size_t size)
{
    struct deque *d;

    d = malloc(sizeof(struct deque));
    assert(d);

    d->base = 0;
    d->length = 0;
    d->size = size;

    d->contents = malloc(sizeof(int) * d->size);
    assert(d->contents);

    return d;
}
```

```c
/* return a new empty deque */
Deque *
dequeCreate(void)
{
    return dequeCreateInternal(INITIAL_SIZE);
}

void
dequePush(Deque *d, int direction, int value)
{
    struct deque *d2;     /* replacement deque if we grow */
    int *oldContents;     /* old contents of d */

    /*
     * First make sure we have space.
     */

    if(d->length == d->size) {
        /* nope */
        d2 = dequeCreateInternal(d->size * 2);

        /* evacuate d */
        while(!dequeIsEmpty(d)) {
            dequePush(d2, DEQUE_BACK, dequePop(d, DEQUE_FRONT));
        }

        /* do a transplant from d2 to d */
        /* but save old contents so we can free them */
        oldContents = d->contents;
        *d = *d2;    /* this is equivalent to copying the components one by one */

        /* these are the pieces we don't need any more */
        free(oldContents);
        free(d2);
    }

    /*
     * This requires completely different code
     * depending on the direction, which is
     * annoying.
     */
    if(direction == DEQUE_FRONT) {
        /* d->base is unsigned, so we have to check for zero first */
        if(d->base == 0) {
            d->base = d->size - 1;
```

```c
        } else {
            d->base--;
        }

        d->length++;

        d->contents[d->base] = value;
    } else {
        d->contents[(d->base + d->length++) % d->size] = value;
    }
}

/* pop and return first value on direction side of deque d */
/* returns DEQUE_EMPTY if deque is empty */
int
dequePop(Deque *d, int direction)
{
    int retval;

    if(dequeIsEmpty(d)) {
        return DEQUE_EMPTY;
    }

    /* else */
    if(direction == DEQUE_FRONT) {
        /* base goes up by one, length goes down by one */
        retval = d->contents[d->base];

        d->base = (d->base+1) % d->size;
        d->length--;

        return retval;
    } else {
        /* length goes down by one */
        return d->contents[(d->base + --d->length) % d->size];
    }
}

int
dequeIsEmpty(const Deque *d)
{
    return d->length == 0;
}

void
dequeDestroy(Deque *d)
```

```
{
    free(d->contents);
    free(d);
}
```

examples/linkedLists/deque/ringBuffer.c

Here is a Makefile that compiles testDeque.c against both the linked list and the ring buffer implementations. You can do make time to race them against each other.

```
CC=gcc
CFLAGS=-std=c99 -Wall -pedantic -O3 -g3

# how many iterations for test
ITERATIONS=10000000
VALGRIND_ITERATIONS=100

all: testDeque testRingBuffer

test: all
	./testDeque $(ITERATIONS)
	valgrind -q --leak-check=yes ./testDeque $(VALGRIND_ITERATIONS)
	./testRingBuffer $(ITERATIONS)
	valgrind -q --leak-check=yes ./testRingBuffer $(VALGRIND_ITERATIONS)

time: all
	time ./testDeque $(ITERATIONS)
	time ./testRingBuffer $(ITERATIONS)

testDeque: testDeque.o deque.o
	$(CC) $(CFLAGS) -o $@ $^

testRingBuffer: testDeque.o ringBuffer.o
	$(CC) $(CFLAGS) -o $@ $^


clean:
	$(RM) testDeque testRingBuffer *.o
```

examples/linkedLists/deque/Makefile

5.2.6 Circular linked lists

For some applications, there is no obvious starting or ending point to a list, and a circular list (where the last element points back to the first) may be appropriate. Circular doubly-linked lists can also be used to build deques; a single pointer into the list tracks the head of the deque, with some convention adopted for whether the head is an actual element of the list (at the front, say, with its left neighbor at the back) or a dummy element that is not considered to be part of the list.

The selling point of circular doubly-linked lists as a concrete data structure is that insertions and deletions can be done anywhere in the list with only local information. For example, here are some routines for manipulating a doubly-linked list directly. We'll make our lives easy and assume (for the moment) that the list has no actual contents to keep track of.

```
#include <stdlib.h>

/* directions for doubly-linked list next pointers */
#define RIGHT (0)
#define LEFT (1)

struct elt {
    struct elt *next[2];
};

typedef struct elt *Elt;

/* create a new circular doubly-linked list with 1 element */
/* returns 0 on allocation error */
Elt
listCreate(void)
{
    Elt e;

    e = malloc(sizeof(*e));
    if(e) {
        e->next[LEFT] = e->next[RIGHT] = e;
    }

    return e;
}

/* remove an element from a list */
/* Make sure you keep a pointer to some other element! */
/* does not free the removed element */
void
listRemove(Elt e)
{
    /* splice e out */
    e->next[RIGHT]->next[LEFT] = e->next[LEFT];
    e->next[LEFT]->next[RIGHT] = e->next[RIGHT];
```

```
}

/* insert an element e into list in direction dir from head */
void
listInsert(Elt head, int dir, Elt e)
{
    /* fill in e's new neighbors */
    e->next[dir] = head->next[dir];
    e->next[!dir] = head;

    /* make neigbhors point back at e */
    e->next[dir]->next[!dir] = e;
    e->next[!dir]->next[dir] = e;
}

/* split a list, removing all elements between e1 and e2 */
/* e1 is the leftmost node of the removed subsequence, e2 rightmost */
/* the removed elements are formed into their own linked list */
/* comment: listRemove could be implemented as listSplit(e,e) */
void
listSplit(Elt e1, Elt e2)
{
    /* splice out the new list */
    e2->next[RIGHT]->next[LEFT] = e1->next[LEFT];
    e1->next[LEFT]->next[RIGHT] = e2->next[RIGHT];

    /* fix up the ends */
    e2->next[RIGHT] = e1;
    e1->next[LEFT] = e2;
}

/* splice a list starting at e2 after e1 */
/* e2 becomes e1's right neighbor */
/* e2's left neighbor becomes left neighbor of e1's old right neighbor */
void
listSplice(Elt e1, Elt e2)
{
    /* fix up tail end */
    e2->next[LEFT]->next[RIGHT] = e1->next[RIGHT];
    e1->next[RIGHT]->next[LEFT] = e2->next[LEFT];

    /* fix up e1 and e2 */
    e1->next[RIGHT] = e2;
    e2->next[LEFT] = e1;
}
```

```c
/* free all elements of the list containing e */
void
listDestroy(Elt e)
{
    Elt target;
    Elt next;

    /* we'll free elements until we get back to e, then free e */
    /* note use of pointer address comparison to detect end of loop */
    for(target = e->next[RIGHT]; target != e; target = next) {
        next = target->next[RIGHT];
        free(target);
    }

    free(e);
}
```

examples/linkedLists/circular.c

The above code might or might not actually work. What if it doesn't? It may make sense to include some sanity-checking code that we can run to see if our pointers are all going to the right place:

```c
/* assert many things about correctness of the list */
/* Amazingly, this is guaranteed to abort or return no matter
   how badly screwed up the list is. */
void
listSanityCheck(Elt e)
{
    Elt check;

    assert(e != 0);

    check = e;

    do {

        /* are our pointers consistent with our neighbors? */
        assert(check->next[RIGHT]->next[LEFT] == check);
        assert(check->next[LEFT]->next[RIGHT] == check);

        /* on to the next */
        check = check->next[RIGHT];

    } while(check != e);
}
```

What if we want to store something in this list? The simplest approach is to extend the definition of `struct elt`:

```
struct elt {
    struct elt *next[2];
    char *name;
    int socialSecurityNumber;
    int gullibility;
};
```

But then we can only use the code for one particular type of data. An alternative approach is to define a new `Elt`-plus struct:

```
struct fancyElt {
    struct elt *next[2];
    char *name;
    int socialSecurityNumber;
    int gullibility;
};
```

and then use pointer casts to convert the fancy structs into `Elt`s:

```
    struct fancyElt *e;

    e = malloc(sizeof(*e));

    /* fill in fields on e */

    listInsert(someList, (Elt) e);
```

The trick here is that as long as the initial part of the `struct fancyElt` looks like a `struct elt`, any code that expects a `struct elt` will happily work with it and ignore the fields that happen to be sitting later in memory. (This trick is how C++ inheritance works.)

The downside is that if something needs to be done with the other fields (e.g freeing `e->name` if `e` is freed), then the `Elt` functions won't know to do this. So if you use this trick you should be careful.

A similar technique using `void *` pointers can be used to implement generic containers.

5.2.7 What linked lists are and are not good for

Linked lists are good for any task that involves inserting or deleting elements next to an element you already have a pointer to; such operations can usually be done in $O(1)$ time. They generally beat arrays (even resizeable arrays) if you need to insert or delete in the middle of a list, since an array has to copy any elements above the insertion point to make room; if inserts or deletes always happen at the end, an array may be better.

Linked lists are not good for any operation that requires random access, since reaching an arbitrary element of a linked list takes as much as $O(n)$ time. For such applications, arrays are better if you don't need to insert in the middle; if you do, you should use some sort of tree.

5.2.8 Further reading

A description of many different kinds of linked lists with pictures can be found in the WikiPedia article on the subject.

Animated versions can be found at http://www.cs.usfca.edu/~galles/visualization/Algorithms.html.

5.3 Abstract data types

One of the hard parts about computer programming is that, in general, *programs are bigger than brains*. Unless you have an unusally capacious brain, it is unlikely that you will be able to understand even a modestly large program in its entirety. So in order to be able to write and debug large programs, it is important to be able to break it up into pieces, where each piece can be treated as a tool whose use and description is simpler (and therefor fits in your brain better) than its actual code. Then you can forget about what is happening inside that piece, and just treat it as an easily-understood black box from the outside.

This process of wrapping functionality up in a box and forgetting about its internals is called **abstraction**, and it is the single most important concept in computer science. In these notes we will describe a particular kind of abstraction, the construction of **abstract data types** or ADTs. Abstract data types are data types whose implementation is not visible to their user; from the outside, all the user knows about an ADT is what operations can be performed on it and what those operations are supposed to do.

ADTs have an outside and an inside. The outside is called the **interface**; it consists of the minimal set of type and function declarations needed to use the ADT. The inside is called the **implementation**; it consists of type and function definitions, and sometime auxiliary data or helper functions, that are *not* visible to users of the ADT. This separation between interface and implementation is called the **abstraction barrier**, and allows the implementation to change without affecting the rest of the program.

What joins the implementation to the interface is an **abstraction function**. This is a function (in the mathematical sense) that takes any state of the implementation and trims off any irrelevant details to leave behind an idealized pictures of what the data type is doing. For example, a linked list implementation translates to a sequence abstract data type by forgetting about the pointers used to hook up the elements and just keeping the sequence of elements themselves.

To exclude bad states of the implementation (for example, a singly-linked list that loops back on itself instead of having a terminating null pointer), we may have a **representation invariant**, which is just some property of the implementation that is always true. Representation invariants are also useful for detecting when we've bungled our implementation, and a good debugging strategy for misbehaving abstract data type implementations is often to look for the first point at which they violated some property that we thought was an invariant.

Some programming language include very strong mechanisms for enforcing abstraction barriers. C relies somewhat more on politeness, and as a programmer you violate an abstraction barrier (by using details of an implementation that are supposed to be hidden) at your peril. In C, the interface will typically consist of function and type declarations contained in a header file, with implementation made up of the corresponding function definitions (and possibly a few extra `static` functions) in one or more `.c` files. The opaque struct technique can be used to hide implementation details of the type.

5.3.1 A sequence type

Too much abstraction at once can be hard to take, so let's look at a concrete example of an abstract data type. This ADT will represent an infinite sequence of `int`s. Each **instance** of the `Sequence` type supports a single operation `seq_next` that returns the next `int` in the sequence. We will also need to provide one or more **constructor** functions to generate new `Sequence`s, and a **destructor** function to tear them down.

Here is an example of a typical use of a `Sequence`:

```
void
seq_print(Sequence s, int limit)
{
    int i;

    for(i = seq_next(s); i < limit; i = seq_next(s)) {
        printf("%d\n", i);
    }
}
```

Note that `seq_print` doesn't need to know anything at all about what a `Sequence` is or how `seq_next` works in order to print out all the values in the sequence until it hits one greater than or equal to limit. This is a good thing— it means that we can use with any implementation of `Sequence` we like, and we don't have to change it if `Sequence` or `seq_next` changes.

5.3.1.1 Interface

In C, the interface of an abstract data type will usually be declared in a header file, which is included both in the file that implements the ADT (so that the compiler can check that the declarations match up with the actual definitions in the implementation. Here's a header file for sequences:

```c
/* opaque struct: hides actual components of struct sequence,
 * which are defined in sequence.c */
typedef struct sequence *Sequence;

/* constructors */
/* all our constructors return a null pointer on allocation failure */

/* returns a Sequence representing init, init+1, init+2, ... */
Sequence seq_create(int init);

/* returns a Sequence representing init, init+step, init+2*step, ... */
Sequence seq_create_step(int init, int step);

/* destructor */
/* destroys a Sequence, recovering all interally-allocated data */
void seq_destroy(Sequence);

/* accessor */
/* returns the first element in a sequence not previously returned */
int seq_next(Sequence);
```

examples/ADT/sequence/sequence.h

Here we have defined two different constructors for `Sequence`s, one of which gives slightly more control over the sequence than the other. If we were willing to put more work into the implementation, we could imagine building a very complicated `Sequence` type that supported a much wider variety of sequences (for example, sequences generated by functions or sequences read from files); but we'll try to keep things simple for now. We can always add more functionality later, since the users won't notice if the `Sequence` type changes internally.

5.3.1.2 Implementation The implementation of an ADT in C is typically contained in one (or sometimes more than one) `.c` file. This file can be compiled and linked into any program that needs to use the ADT. Here is our implementation of `Sequence`:

```c
#include <stdlib.h>

#include "sequence.h"

struct sequence {
    int next;   /* next value to return */
    int step;   /* how much to increment next by */
};
```

```c
Sequence
seq_create(int init)
{
    return seq_create_step(init, 1);
}

Sequence
seq_create_step(int init, int step)
{
    Sequence s;

    s = malloc(sizeof(*s));
    if(s == 0) return 0;
    s->next = init;
    s->step = step;
    return s;
}

void
seq_destroy(Sequence s)
{
    free(s);
}

int
seq_next(Sequence s)
{
    int ret;            /* saves the old value before we increment it */

    ret = s->next;
    s->next += s->step;

    return ret;
}
```

examples/ADT/sequence/sequence.c

Things to note here: the definition of `struct sequence` appears only in this file; this means that only the functions defined here can (easily) access the `next` and `step` components. This protects `Sequence`s to a limited extent from outside interference, and defends against users who might try to "violate the abstraction boundary" by examining the components of a `Sequence` directly. It also means that if we change the components or meaning of the components in `struct sequence`, we only have to fix the functions defined in `sequence.c`.

5.3.1.3 Compiling and linking Now that we have `sequence.h` and `sequence.c`, how do we use them? Let's suppose we have a simple main program:

```c
#include <stdio.h>

#include "sequence.h"


void
seq_print(Sequence s, int limit)
{
    int i;

    for(i = seq_next(s); i < limit; i = seq_next(s)) {
        printf("%d\n", i);
    }
}


int
main(int argc, char **argv)
{
    Sequence s;
    Sequence s2;

    puts("Stepping by 1:");

    s = seq_create(0);
    seq_print(s, 5);
    seq_destroy(s);

    puts("Now stepping by 3:");

    s2 = seq_create_step(1, 3);
    seq_print(s2, 20);
    seq_destroy(s2);

    return 0;
}
```

examples/ADT/sequence/main.c

We can compile `main.c` and `sequence.c` together into a single binary with the command `c99 main.c sequence.c`. Or we can build a `Makefile` which will compile the two files separately and then link them. Using `make` may be more efficient, especially for large programs consisting of many components, since if we make any changes `make` will only recompile those files we have changed. So here is our `Makefile`:

```
CC=c99
CFLAGS=-g3 -pedantic -Wall

all: seqprinter

seqprinter: main.o sequence.o
	$(CC) $(CFLAGS) -o $@ $^

test: seqprinter
	./seqprinter

# these rules say to rebuild main.o and sequence.o if sequence.h changes
main.o: main.c sequence.h
sequence.o: sequence.c sequence.h

clean:
	$(RM) -f seqprinter *.o
```

examples/ADT/sequence/Makefile

And now running `make test` produces this output. Notice how the built-in `make` variables `$@` and `$^` expand out to the left-hand side and right-hand side of the dependency line for building `seqprinter`.

```
$ make test
c99 -g3 -pedantic -Wall   -c -o main.o main.c
c99 -g3 -pedantic -Wall   -c -o sequence.o sequence.c
c99 -g3 -pedantic -Wall -o seqprinter main.o sequence.o
./seqprinter
Stepping by 1:
0
1
2
3
4
Now stepping by 3:
1
4
7
10
13
16
19
```

5.3.2 Designing abstract data types

Now we've seen how to implement an abstract data type. How do we choose when to use when, and what operations to give it? Let's try answering the second question first.

5.3.2.1 Parnas's Principle

Parnas's Principle is a statement of the fundamental idea of *information hiding*, which says that abstraction boundaries should be as narrow as possible:

- The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

(David Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, 15(12): 1059–1062, 1972.)

For ADTs, this means we should provide as few functions for accessing and modifying the ADT as we can get away with. The `Sequence` type we defined early has a particularly narrow interface; the developer of `Sequence` (whoever is writing `sequence.c`) needs to know nothing about what its user wants except for the arguments passed in to `seq_create` or `seq_create_step`, and the user only needs to be able to call `seq_next`. More complicated ADTs might provide larger sets of operations, but in general we know that an ADT provides a successful abstraction when the operations are all "natural" ones given our high-level description. If we find ourselves writing a lot of extra operations to let users tinker with the guts of our implementation, that may be a sign that either we aren't taking our abstraction barrier seriously enough, or that we need to put the abstraction barrier in a different place.

5.3.2.2 When to build an abstract data type

The short answer: Whenever you can.

A better answer: The best heuristic I know for deciding what ADTs to include in a program is to write down a description of how your program is going to work. For each noun or noun phrase in the description, either identify a built-in data type to implement it or design an abstract data type.

For example: a grade database maintains a list of students, and for each student it keeps a list of grades. So here we might want data types to represent:

- A list of students,
- A student,
- A list of grades,

- A grade.

If grades are simple, we might be able to make them just be `int`s (or maybe `double`s); to be on the safe side, we should probably create a `Grade` type with a `typedef`. The other types are likely to be more complicated. Each student might have in addition to his or her grades a long list of other attributes, such as a name, an email address, etc. By wrapping students up as abstract data types we can extend these attributes if we need to, or allow for very general implementations (say, by allowing a student to have an arbitrary list of keyword-attribute pairs). The two kinds of lists are likely to be examples of *sequence* types; we'll be seeing a lot of ways to implement these as the course progresses. If we want to perform the same kinds of operations on both lists, we might want to try to implement them as a single list data type, which then is specialized to hold either students or grades; this is not always easy to do in C, but we'll see examples of how to do this, too.

Whether or not this set of four types is the set we will finally use, writing it down gives us a place to start writing our program. We can start writing interface files for each of the data types, and then evolve their implementations and the main program in parallel, adjusting the interfaces as we find that we have provided too little (or too much) data for each component to do what it must.

5.4 Hash tables

A **hash table** is a randomized data structure that supports the INSERT, DELETE, and FIND operations in expected $O(1)$ time. The core idea behind hash tables is to use a *hash function* that maps a large keyspace to a smaller domain of array indices, and then use constant-time array operations to store and retrieve the data.

5.4.1 Dictionary data types

A hash table is typically used to implement a **dictionary data type**, where keys are mapped to values, but unlike an array, the keys are not conveniently arranged as integers $0, 1, 2, \ldots$. Dictionary data types are a fundamental data structure often found in scripting languages like AWK, Perl, Python, PHP, Lua, or Ruby. For example, here is some Python code that demonstrates use of a dictionary accessed using an array-like syntax:

```python
title = {}   # empty dictionary
title["Barack"] = "President"
user = "Barack"
print("Welcome" + title[user] + " " + user)
```

In C, we don't have the convenience of reusing `[]` for dictionary lookups (we'd need C++ for that), but we can still get the same effect with more typing using functions. For example, using an abstract dictionary in C might look like this:

```
Dict *title;
const char *user;

title = dictCreate();
dictSet(title, "Barack", "President");
user = "Barack";
printf("Welcome %s %s\n", dictGet(title, user), user);
```

As with other abstract data types, the idea is that the user of the dictionary type doesn't need to know how it is implemented. For example, we could implement the dictionary as an array of `struct`s that we search through, but that would be expensive: $O(n)$ time to find a key in the worst case.

Closely related to a dictionary is a **set**, which has keys but no values. It's usually pretty straightforward to turn an implementation of a dictionary into a set (leave out the values) or vice versa (add values to the end of keys but don't use them in searching).

5.4.2 Basics of hashing

If our keys were conveniently named $0, 1, 2, \ldots, n-1$, we could simply use an array, and be able to find a record given a key in constant time. Unfortunately, naming conventions for most objects are not so convenient, and even enumerations like Social Security numbers are likely to span a larger range than we want to allocate. But we would like to get the constant-time performance of an array anyway.

The solution is to feed the keys through some hash function H, which maps them down to array indices. So in a database of people, to find "Smith, Wayland", we would first compute H("Smith, Wayland")\$ = 137\$ (say), and then look in position 137 in the array. Because we are always using the same function H, we will always be directed to the same position 137.

5.4.3 Resolving collisions

But what if H("Smith, Wayland") and H("Hephaestos") both equal 137? Now we have a **collision**, and we have to resolve it by finding some way to either (a) effectively store both records in a single array location, or (b) move one of the records to a new location that we can still find later. Let's consider these two approaches separately.

5.4.3.1 Chaining We can't really store more than one record in an array location, but we can fake it by making each array location be a pointer to a linked list. Every time we insert a new element in a particular location, we simply add it to this list.

Since the cost of scanning a linked list is linear in its size, this means that the worst-case cost of searching for a particular key will be linear in the number of keys in the table that hash to the same location. Under the assumption that the hash function is a random function (which does not mean that it returns random values every time you call it but instead means that we picked one of the many possible hash functions uniformly at random), on average we get n/m elements in each list.
So on average a failed search takes $O(n/m)$ time.

This quantity n/m is called the **load factor** of the hash table and is often written as α. If we want our hash table to be efficient, we will need to keep this load factor down. If we can guarantee that it's a constant, then we get constant-time searches.

5.4.3.2 Open addressing With *open addressing*, we store only one element per location, and handle collisions by storing the extra elements in other unused locations in the array. To find these other locations, we fix some *probe sequence* that tells us where to look if $A[H(x)]$ contains an element that is not x. A typical probe sequence (called *linear probing*) is just $H(x), H(x)+1, H(x)+2, \ldots,$ wrapping around at the end of the array. The idea is that if we can't put an element in a particular place, we just keep walking up through the array until we find an empty slot. As long as we follow the same probe sequence when looking for an element, we will be able to find the element again. If we are looking for an element and reach an empty location, then we know that the element is not present in the table.

For open addressing, we always have that $\alpha = n/m$ is less than or equal to 1, since we can't store more elements in the table than we have locations. In fact, we must ensure that the load factor is strictly less than 1, or some searches will never terminate because they never reach an empty location. Assuming $\alpha < 1$ and that the hash function is uniform, it is possible to calculate the worst-case expected cost of a FIND operation, which as before will occur when we have an unsuccessful FIND. Though we won't do this calculation here, the result is bounded by $1/(1 - n/m)$, which gets pretty bad if n/m is very close to 1, but is a constant as long as n/m is bounded by a constant (say 3/4, which makes the expected number of probes at most 4).

5.4.4 Choosing a hash function

Here we will describe three methods for generating hash functions. The first two are typical methods used in practice. The last has additional desirable theoretical properties.

5.4.4.1 Division method We want our hash function to look as close as it can to a random function, but random functions are (provably) expensive to store. So in practice we do something simpler and hope for the best. If the keys are large integers, a typical approach is to just compute the remainder mod m. This can cause problems if m is, say, a power of 2, since it may be that the low-order bits of all the keys are similar, which will produce lots of collisions. So in practice with this method m is typically chosen to be a large prime.

What if we want to hash strings instead of integers? The trick is to treat the strings as integers. Given a string $a_1 a_2 a_3 \ldots a_k$, we represent it as $\sum_i a_i b^i$, where b is a base chosen to be larger than the number of characters. We can then feed this resulting huge integer to our hash function. Typically we do not actually compute the huge integer directly, but instead compute its remainder mod m, as in this short C function:

```
/* treat strings as base-256 integers */
/* with digits in the range 1 to 255 */
#define BASE (256)

size_t
hash(const char *s, size_t m)
{
    size_t h;
    unsigned const char *us;

    /* cast s to unsigned const char * */
    /* this ensures that elements of s will be treated as having values >= 0 */
    us = (unsigned const char *) s;

    h = 0;
    while(*us != '\0') {
        h = (h * BASE + *us) % m;
        us++;
    }

    return h;
}
```

The division method works best when m is a prime, as otherwise regularities in the keys can produce clustering in the hash values. (Consider, for example, what happens if m is 256). But this can be awkward for computing hash functions quickly, because computing remainders is a relatively slow operation.

5.4.4.2 Multiplication method For this reason, the most commonly-used hash functions replace the modulus m with something like 2^{32} and replace the base with some small prime, relying on the multiplier to break up patterns in the input. This yields the **multiplication method**. Typical code might look something like this:

```c
#define MULTIPLIER (37)

size_t
hash(const char *s)
{
    size_t h;
    unsigned const char *us;

    /* cast s to unsigned const char * */
    /* this ensures that elements of s will be treated as having values >= 0 */
    us = (unsigned const char *) s;

    h = 0;
    while(*us != '\0') {
        h = h * MULTIPLIER + *us;
        us++;
    }

    return h;
}
```

The only difference between this code and the division method code is that we've renamed `BASE` to `MULTIPLIER` and dropped m. There is still some remainder-taking happening: since C truncates the result of any operation that exceeds the size of the (unsigned) integer type that holds it, the `h = h * MULTIPLIER + *us;` line effectively has a hidden mod 2^{32} or 2^{64} at the end of it (depending on how big your `size_t` is). This is why we can't use, say, 256, as the multiplier, because then the hash value `h` would be determined by just the last four characters of `s`.

The choice of 37 is based on folklore. I like 97 myself, and 31 also has supporters. Almost any medium-sized prime should work.

5.4.4.3 Universal hashing The preceding hash functions offer no guarantees that the adversary can't find a set of n keys that all hash to the same location; indeed, since they're deterministic, as long as the keyspace contains at least nm keys the adversary can always do so. Universal families of hash functions avoid this problem by choosing the hash function randomly, from some set of possible functions that is small enough that we can write our random choice down.

The property that makes a family of hash functions $\{H_r\}$ universal is that, for any distinct keys x and y, the probability that r is chosen so that $H_r(x) = H_r(y)$ is exactly $1/m$.

Why is this important? Recall that for chaining, the expected number of collisions between an element x and other elements was just the sum over all particular elements y of the probability that x collides with that particular element. If H_r is drawn from a universal family, this probability is $1/m$ for each y, and we get the same n/m expected collisions as if H_r were completely random.

Several universal families of hash functions are known. Here is a simple one that works when the size of the keyspace and the size of the output space are both powers of 2. Let the keyspace consist of n-bit strings and let $m = 2^k$. Then the random index r consists of nk independent random bits organized as n m-bit strings $a_1 a_2 \ldots a_n$. To compute the hash function of a particular input x, compute the bitwise exclusive or of a_i for each position i where the i-th bit of x is 1.

We can implement this in C as

```
/* implements universal hashing using random bit-vectors in x */
/* assumes number of elements in x is at least BITS_PER_CHAR * MAX_STRING_SIZE */

#define BITS_PER_CHAR (8)       /* not true on all machines! */
#define MAX_STRING_SIZE (128)   /* we'll stop hashing after this many */
#define MAX_BITS (BITS_PER_CHAR * MAX_STRING_SIZE)

size_t
hash(const char *s, size_t x[])
{
    size_t h;
    unsigned const char *us;
    int i;
    unsigned char c;
    int shift;

    /* cast s to unsigned const char * */
    /* this ensures that elements of s will be treated as having values >= 0 */
    us = (unsigned const char *) s;

    h = 0;
    for(i = 0; *us != 0 && i < MAX_BITS; us++) {
        c = *us;
        for(shift = 0; shift < BITS_PER_CHAR; shift++, i++) {
            /* is low bit of c set? */
            if(c & 0x1) {
                h ^= x[i];
            }

            /* shift c to get new bit in lowest position */
            c >>= 1;
        }
    }
```

```
    return h;
}
```

As you can see, this requires a lot of bit-fiddling. It also fails if we get a lot of strings that are identical for the first `MAX_STRING_SIZE` characters. Conceivably, the latter problem could be dealt with by growing `x` dynamically as needed. But we also haven't addressed the question of where we get these random values from—see the chapter on randomization for some possibilities.

In practice, universal families of hash functions are seldom used, since a reasonable fixed hash function is unlikely to be correlated with any patterns in the actual input. But they are useful for demonstrating provably good performance.

5.4.5 Maintaining a constant load factor

All of the running time results for hash tables depend on keeping the load factor α small. But as more elements are inserted into a fixed-size table, the load factor grows without bound. The usual solution to this problem is rehashing: when the load factor crosses some threshold, we create a new hash table of size $2n$ or thereabouts and migrate all the elements to it.

This approach raises the worst-case cost of an insertion to $O(n)$. However, we can bring the *expected* cost down to $O(1)$ by rehashing only with probability $O(1/n)$ for each insert after the threshold is crossed. Or we can apply **amortized analysis** to argue that the amortized cost (total cost divided by number of operations) is $O(1)$ assuming we double the table size on each rehash. Neither the expected-cost nor the amortized-cost approaches actually change the worst-case cost, but they make it look better by demonstrating that we at least don't incur that cost every time.

With enough machinery, it may be possible to **deamortize** the cost of rehashing by doing a little bit of it with every insertion. The idea is to build the new hash table incrementally, and start moving elements to it once it is fully initialized. This requires keeping around two copies of the hash table and searching both, and for most purposes is more trouble than it's worth. But a mechanism like this is often used for real-time garbage collection, where it's important not to have the garbage collector lock up the entire system while it does its work.

5.4.6 Examples

5.4.6.1 A low-overhead hash table using open addressing Here is a very low-overhead hash table based on open addressing. The application is rapidly verifying ID numbers in the range 000000000 to 999999999 by checking them against a list of known good IDs. Since the quantity of valid ID numbers may be very large, a goal of the mechanism is to keep the amount of extra storage used as small as possible. This implementation uses a tunable overhead parameter. Setting the parameter to a high value makes lookups fast but requires more space per ID number in the list. Setting it to a low value can reduce the storage cost arbitrarily close to 4 bytes per ID, at the cost of increasing search times.

Here is the header file giving the interface:

```
typedef struct idList *IDList;

#define MIN_ID (0)
#define MAX_ID (999999999)

/* build an IDList out of an unsorted array of n good ids */
/* returns 0 on allocation failure */
IDList IDListCreate(int n, int unsortedIdList[]);

/* destroy an IDList */
void IDListDestroy(IDList list);

/* check an id against the list */
/* returns nonzero if id is in the list */
int IDListContains(IDList list, int id);
```

examples/hashTables/idList/idList.h

And here is the implementation:

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

#include "idList.h"

/* overhead parameter that determines both space and search costs */
/* must be strictly greater than 1 */
#define OVERHEAD (1.1)
#define NULL_ID (-1)


struct idList {
    int size;
    int ids[1];         /* we'll actually malloc more space than this */
};

IDList
IDListCreate(int n, int unsortedIdList[])
{
    IDList list;
```

```c
    int size;
    int i;
    int probe;

    size = (int) (n * OVERHEAD + 1);

    list = malloc(sizeof(*list) + sizeof(int) * (size-1));
    if(list == 0) return 0;

    /* else */
    list->size = size;

    /* clear the hash table */
    for(i = 0; i < size; i++) {
        list->ids[i] = NULL_ID;
    }

    /* load it up */
    for(i = 0; i < n; i++) {

        assert(unsortedIdList[i] >= MIN_ID);
        assert(unsortedIdList[i] <= MAX_ID);

        /* hashing with open addressing by division */
        /* this MUST be the same pattern as in IDListContains */
        for(probe = unsortedIdList[i] % list->size;
            list->ids[probe] != NULL_ID;
            probe = (probe + 1) % list->size);

        assert(list->ids[probe] == NULL_ID);

        list->ids[probe] = unsortedIdList[i];
    }

    return list;
}

void
IDListDestroy(IDList list)
{
    free(list);
}

int
IDListContains(IDList list, int id)
{
```

```
    int probe;

    /* this MUST be the same pattern as in IDListCreate */
    for(probe = id % size;
        list->ids[probe] != NULL_ID;
        probe = (probe + 1) % size) {
        if(list->ids[probe] == id) {
            return 1;
        }
    }

    return 0;
}
```

examples/hashTables/idList/idList.c

5.4.6.2 A string to string dictionary using chaining Here is a more complicated string to string dictionary based on chaining.

```
typedef struct dict *Dict;

/* create a new empty dictionary */
Dict DictCreate(void);

/* destroy a dictionary */
void DictDestroy(Dict);

/* insert a new key-value pair into an existing dictionary */
void DictInsert(Dict, const char *key, const char *value);

/* return the most recently inserted value associated with a key */
/* or 0 if no matching key is present */
const char *DictSearch(Dict, const char *key);

/* delete the most recently inserted record with the given key */
/* if there is no such record, has no effect */
void DictDelete(Dict, const char *key);
```

examples/hashTables/dict/dict.h

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

#include "dict.h"

struct elt {
```

```
    struct elt *next;
    char *key;
    char *value;
};

struct dict {
    int size;           /* size of the pointer table */
    int n;              /* number of elements stored */
    struct elt **table;
};

#define INITIAL_SIZE (1024)
#define GROWTH_FACTOR (2)
#define MAX_LOAD_FACTOR (1)

/* dictionary initialization code used in both DictCreate and grow */
Dict
internalDictCreate(int size)
{
    Dict d;
    int i;

    d = malloc(sizeof(*d));

    assert(d != 0);

    d->size = size;
    d->n = 0;
    d->table = malloc(sizeof(struct elt *) * d->size);

    assert(d->table != 0);

    for(i = 0; i < d->size; i++) d->table[i] = 0;

    return d;
}

Dict
DictCreate(void)
{
    return internalDictCreate(INITIAL_SIZE);
}

void
DictDestroy(Dict d)
{
```

```
    int i;
    struct elt *e;
    struct elt *next;

    for(i = 0; i < d->size; i++) {
        for(e = d->table[i]; e != 0; e = next) {
            next = e->next;

            free(e->key);
            free(e->value);
            free(e);
        }
    }

    free(d->table);
    free(d);
}

#define MULTIPLIER (97)

static unsigned long
hash_function(const char *s)
{
    unsigned const char *us;
    unsigned long h;

    h = 0;

    for(us = (unsigned const char *) s; *us; us++) {
        h = h * MULTIPLIER + *us;
    }

    return h;
}

static void
grow(Dict d)
{
    Dict d2;            /* new dictionary we'll create */
    struct dict swap;   /* temporary structure for brain transplant */
    int i;
    struct elt *e;

    d2 = internalDictCreate(d->size * GROWTH_FACTOR);

    for(i = 0; i < d->size; i++) {
```

```
        for(e = d->table[i]; e != 0; e = e->next) {
            /* note: this recopies everything */
            /* a more efficient implementation would
             * patch out the strdups inside DictInsert
             * to avoid this problem */
            DictInsert(d2, e->key, e->value);
        }
    }

    /* the hideous part */
    /* We'll swap the guts of d and d2 */
    /* then call DictDestroy on d2 */
    swap = *d;
    *d = *d2;
    *d2 = swap;

    DictDestroy(d2);
}

/* insert a new key-value pair into an existing dictionary */
void
DictInsert(Dict d, const char *key, const char *value)
{
    struct elt *e;
    unsigned long h;

    assert(key);
    assert(value);

    e = malloc(sizeof(*e));

    assert(e);

    e->key = strdup(key);
    e->value = strdup(value);

    h = hash_function(key) % d->size;

    e->next = d->table[h];
    d->table[h] = e;

    d->n++;

    /* grow table if there is not enough room */
    if(d->n >= d->size * MAX_LOAD_FACTOR) {
        grow(d);
```

```
    }
}

/* return the most recently inserted value associated with a key */
/* or 0 if no matching key is present */
const char *
DictSearch(Dict d, const char *key)
{
    struct elt *e;

    for(e = d->table[hash_function(key) % d->size]; e != 0; e = e->next) {
        if(!strcmp(e->key, key)) {
            /* got it */
            return e->value;
        }
    }

    return 0;
}

/* delete the most recently inserted record with the given key */
/* if there is no such record, has no effect */
void
DictDelete(Dict d, const char *key)
{
    struct elt **prev;          /* what to change when elt is deleted */
    struct elt *e;              /* what to delete */

    for(prev = &(d->table[hash_function(key) % d->size]);
        *prev != 0;
        prev = &((*prev)->next)) {
        if(!strcmp((*prev)->key, key)) {
            /* got it */
            e = *prev;
            *prev = e->next;

            free(e->key);
            free(e->value);
            free(e);

            return;
        }
    }
}
```

examples/hashTables/dict/dict.c

And here is some (very minimal) test code.

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

#include "dict.h"

int
main()
{
    Dict d;
    char buf[512];
    int i;

    d = DictCreate();

    DictInsert(d, "foo", "hello world");
    puts(DictSearch(d, "foo"));
    DictInsert(d, "foo", "hello world2");
    puts(DictSearch(d, "foo"));
    DictDelete(d, "foo");
    puts(DictSearch(d, "foo"));
    DictDelete(d, "foo");
    assert(DictSearch(d, "foo") == 0);
    DictDelete(d, "foo");

    for(i = 0; i < 10000; i++) {
        sprintf(buf, "%d", i);
        DictInsert(d, buf, buf);
    }

    DictDestroy(d);

    return 0;
}
```

examples/hashTables/dict/test_dict.c

5.5 Generic containers

The first rule of programming is that you should never write the same code twice. Suppose that you happen to have lying around a dictionary type whose keys are ints and whose values are strings. Tomorrow you realize that what you really want is a dictionary type whose keys are strings and whose values are `int`s, or one whose keys are `int`s but whose values are stacks. If you have n different types that may appear as keys or values, can you avoid writing n^2 different dictionary implementations to get every possible combination?

Many languages provide special mechanisms to support **generic types**, where part of the type is not specified. It's as if you could declare an array in C to be an array of some type to be determined later, and then write functions that operate on any such array without knowing what the missing type is going to be (**templates** in C++ are an example of such a mechanism). Unfortunately, C does not provide generic types. But by aggressive use of function pointers and `void *`, it is possible to fake them.

5.5.1 Generic dictionary: interface

Below is an example of an interface to a generic dictionary type for storing maps from constant values to constant values. The `void *` pointers are used to avoid having to declare exactly what kinds of keys and values the dictionary will contain.

```
/* Set dict[key] = value. */
/* Both key and value are copied internally. */
/* If data is the null pointer, remove dict[key]. */
void dictSet(Dict d, const void *key, const void *value);

/* Return dict[key], or null if dict[key] has not been set. */
const void *dictGet(Dict d, const void *key);
```

We'll also need a constructor and destructor, but we'll get to those in a moment. First we need to think about what `dictSet` and `dictGet` are supposed to do, and how we might possibly be able to implement them. Suppose we want to build a dictionary with strings as both keys and values. Internally, this might be represented as some sort of hash table or tree. Suppose it's a hash table. Now, given some `void *key`, we'd like to be able to compute its hash value. But we don't know what type `key` points to, and if we guess wrong we are likely to end up with segmentation faults or worse. So we need some way to register a hash function for our keys, whatever type they might really be behind that `void *`.

Similarly, we will want to be able to compare keys for equality (since not all keys that hash together will necessarily be the same), and we may want to be able to copy keys and values so that the data inside the dictionary is not modified if somebody changes a value passed in from the outside. So we need a fair bit of information about keys and values. We'll organize all of this information in a struct made up of function pointers. (This includes a few extra components that came up while writing the implementation.)

```
/* Provides operations for working with keys or values */
struct dictContentsOperations {
```

```
    /* hash function */
    unsigned long (*hash)(const void *datum, void *arg);

    /* returns nonzero if *datum1 == *datum2 */
    int (*equal)(const void *datum1, const void *datum2, void *arg);

    /* make a copy of datum that will survive changes to original */
    void *(*copy)(const void *datum, void *arg);

    /* free a copy */
    void (*delete)(void *datum, void *arg);

    /* extra argument, to allow further specialization */
    void *arg;
};
```

We could write a similar but smaller struct for values, but to save a little bit of effort in the short run we'll use the same `struct` for both keys and values. We can now write a constructor for our generic dictionary that consumes two such structs that provide operations for working on keys and values, respectively:

```
/* create a new dictionary with given key and value operations */
/* Note: valueOps.hash and valueOps.equal are not used. */
Dict dictCreate(struct dictContentsOperations keyOps,
                struct dictContentsOperations valueOps);
```

So now to create a dict, we just need to fill in two `dictContentsOperations` structures. For convenience, it might be nice if `dict.c` provided some preloaded structures for common types like `int`s and strings. We can also use the `arg` field in `struct dictContentsOperations` to make the keys and values themselves be parameterized types, for example a type of byte-vectors of given length.

We can declare these various convenience structures in `dict.h` as

```
/* Some predefined dictContentsOperations structures */

/*
 * DictIntOps supports int's that have been cast to (void *), e.g.:
 *     d = dictCreate(DictIntOps, DictIntOps);
 *     dictSet(d, (void *) 1, (void * 2));
 *     x = (int) dictGet(d, (void * 1));
 */
struct dictContentsOperations DictIntOps;

/*
 * Supports null-terminated strings, e.g.:
 *     d = dictCreate(DictStringOps, DictStringOps);
 *     dictSet(d, "foo", "bar");
```

```c
 *     s = dictGet(d, "foo");
 * Note: no casts are needed since C automatically converts
 * between (void *) and other pointer types.
 */
struct dictContentsOperations DictStringOps;

/*
 * Supports fixed-size blocks of memory, e.g.:
 *     int x = 1;
 *     int y = 2;
 *     d = dictCreate(dictMemOps(sizeof(int)), dictMemOps(sizeof(int));
 *     dictSet(d, &x, &y);
 *     printf("%d", *dictGet(d, &x);
 */
struct dictContentsOperations dictMemOps(int size);
```

We'll define the operations in `DictIntOps` to expect `int`s cast directly to `void *`, the operations in `DictStringOps` to expect `char *` cast to `void *`, and the operations in `dictMemOps(size)` will expect `void *` arguments pointing to blocks of the given size. There is a subtle difference between a dictionary using `DictIntOps` and `dictMemOps(sizeof(int))`; in the former case, keys and values are the `int`s themselves (after being case), which in the latter, keys and values are pointers to `int`s.

Implementations of these structures can be found below.

To make a dictionary that maps strings to ints, we just call:

```c
    d = dictCreate(DictStringOps, DictIntOps);
```

and then we can do things like:

```c
    dictSet(d, "foo", (void *) 2);
    v = (int) dictGet(d, "foo');
```

If we find ourselves working with an integer-valued dictionary a lot, we might want to define a few macros or inline functions to avoid having to type casts all the time.

5.5.2 Generic dictionary: implementation

To implement our generic dictionary, we just take our favorite non-generic hash table, and replace any calls to fixed hash functions, copier, `free`, etc. with calls to elements of the appropriate structure. The result is shown below.

```c
typedef struct dict *Dict;

/* Provides operations for working with keys or values */
struct dictContentsOperations {
```

```c
    /* hash function */
    unsigned long (*hash)(const void *datum, void *arg);

    /* returns nonzero if *datum1 == *datum2 */
    int (*equal)(const void *datum1, const void *datum2, void *arg);

    /* make a copy of datum that will survive changes to original */
    void *(*copy)(const void *datum, void *arg);

    /* free a copy */
    void (*delete)(void *datum, void *arg);

    /* extra argument, to allow further specialization */
    void *arg;
};

/* create a new dictionary with given key and value operations */
/* Note: valueOps.hash and valueOps.equal are not used. */
Dict dictCreate(struct dictContentsOperations keyOps,
                 struct dictContentsOperations valueOps);

/* free a dictionary and all the space it contains */
/* This will call the appropriate delete function for all keys and */
/* values. */
void dictDestroy(Dict d);

/* Set dict[key] = value. */
/* Both key and value are copied internally. */
/* If data is the null pointer, remove dict[key]. */
void dictSet(Dict d, const void *key, const void *value);

/* Return dict[key], or null if dict[key] has not been set. */
const void *dictGet(Dict d, const void *key);

/* Some predefined dictContentsOperations structures */

/*
 * DictIntOps supports int's that have been cast to (void *), e.g.:
 *     d = dictCreate(DictIntOps, DictIntOps);
 *     dictSet(d, (void *) 1, (void * 2));
 *     x = (int) dictGet(d, (void * 1));
 */
struct dictContentsOperations DictIntOps;

/*
 * Supports null-terminated strings, e.g.:
```

```
 *     d = dictCreate(DictStringOps, DictStringOps);
 *     dictSet(d, "foo", "bar");
 *     s = dictGet(d, "foo");
 * Note: no casts are needed since C automatically converts
 * between (void *) and other pointer types.
 */
struct dictContentsOperations DictStringOps;

/*
 * Supports fixed-size blocks of memory, e.g.:
 *     int x = 1;
 *     int y = 2;
 *     d = dictCreate(dictMemOps(sizeof(int)), dictMemOps(sizeof(int));
 *     dictSet(d, &x, &y);
 *     printf("%d", *dictGet(d, &x);
 */
struct dictContentsOperations dictMemOps(int size);
```

examples/generic/dict.h

```
#include <stdlib.h>
#include <string.h>
#include "dict.h"

struct dictElt {
    unsigned long hash;           /* full hash of key */
    void *key;
    void *value;
    struct dictElt *next;
};

struct dict {
    int tableSize;          /* number of slots in table */
    int numElements;        /* number of elements */
    struct dictElt **table; /* linked list heads */
    /* these save arguments passed at creation */
    struct dictContentsOperations keyOps;
    struct dictContentsOperations valueOps;
};

#define INITIAL_TABLESIZE (16)
#define TABLESIZE_MULTIPLIER (2)
#define TABLE_GROW_DENSITY (1)

Dict
dictCreate(struct dictContentsOperations keyOps,
            struct dictContentsOperations valueOps)
```

```
{
    Dict d;
    int i;

    d = malloc(sizeof(*d));
    if(d == 0) return 0;

    d->tableSize = INITIAL_TABLESIZE;
    d->numElements = 0;
    d->keyOps = keyOps;
    d->valueOps = valueOps;
    d->table = malloc(sizeof(*(d->table)) * d->tableSize);
    if(d->table == 0) {
        free(d);
        return 0;
    }

    for(i = 0; i < d->tableSize; i++) d->table[i] = 0;

    return d;
}

void
dictDestroy(Dict d)
{
    int i;
    struct dictElt *e;
    struct dictElt *next;

    for(i = 0; i < d->tableSize; i++) {
        for(e = d->table[i]; e != 0; e = next) {
            next = e->next;
            d->keyOps.delete(e->key, d->keyOps.arg);
            d->valueOps.delete(e->value, d->valueOps.arg);
            free(e);
        }
    }
    free(d->table);
    free(d);
}

/* return pointer to element with given key, if any */
static struct dictElt *
dictFetch(Dict d, const void *key)
{
    unsigned long h;
```

```
    int i;
    struct dictElt *e;

    h = d->keyOps.hash(key, d->keyOps.arg);
    i = h % d->tableSize;
    for(e = d->table[i]; e != 0; e = e->next) {
        if(e->hash == h && d->keyOps.equal(key, e->key, d->keyOps.arg)) {
            /* found it */
            return e;
        }
    }
    /* didn't find it */
    return 0;
}

/* increase the size of the dictionary, rehashing all table elements */
static void
dictGrow(Dict d)
{
    struct dictElt **old_table;
    int old_size;
    int i;
    struct dictElt *e;
    struct dictElt *next;
    int new_pos;

    /* save old table */
    old_table = d->table;
    old_size = d->tableSize;

    /* make new table */
    d->tableSize *= TABLESIZE_MULTIPLIER;
    d->table = malloc(sizeof(*(d->table)) * d->tableSize);
    if(d->table == 0) {
        /* put the old one back */
        d->table = old_table;
        d->tableSize = old_size;
        return;
    }
    /* else */
    /* clear new table */
    for(i = 0; i < d->tableSize; i++) d->table[i] = 0;

    /* move all elements of old table to new table */
    for(i = 0; i < old_size; i++) {
        for(e = old_table[i]; e != 0; e = next) {
```

```
            next = e->next;
            /* find the position in the new table */
            new_pos = e->hash % d->tableSize;
            e->next = d->table[new_pos];
            d->table[new_pos] = e;
        }
    }

    /* don't need this any more */
    free(old_table);
}

void
dictSet(Dict d, const void *key, const void *value)
{
    int tablePosition;
    struct dictElt *e;

    e = dictFetch(d, key);
    if(e != 0) {
        /* change existing setting */
        d->valueOps.delete(e->value, d->valueOps.arg);
        e->value = value ? d->valueOps.copy(value, d->valueOps.arg) : 0;
    } else {
        /* create new element */
        e = malloc(sizeof(*e));
        if(e == 0) abort();

        e->hash = d->keyOps.hash(key, d->keyOps.arg);
        e->key = d->keyOps.copy(key, d->keyOps.arg);
        e->value = value ? d->valueOps.copy(value, d->valueOps.arg) : 0;

        /* link it in */
        tablePosition = e->hash % d->tableSize;
        e->next = d->table[tablePosition];
        d->table[tablePosition] = e;

        d->numElements++;

        if(d->numElements > d->tableSize * TABLE_GROW_DENSITY) {
            /* grow and rehash */
            dictGrow(d);
        }
    }
}
```

```c
const void *
dictGet(Dict d, const void *key)
{
    struct dictElt *e;

    e = dictFetch(d, key);
    if(e != 0) {
        return e->value;
    } else {
        return 0;
    }
}

/* int functions */
/* We assume that int can be cast to void * and back without damage */
static unsigned long dictIntHash(const void *x, void *arg) { return (int) x; }
static int dictIntEqual(const void *x, const void *y, void *arg)
{
    return ((int) x) == ((int) y);
}
static void *dictIntCopy(const void *x, void *arg) { return (void *) x; }
static void dictIntDelete(void *x, void *arg) { ; }

struct dictContentsOperations DictIntOps = {
    dictIntHash,
    dictIntEqual,
    dictIntCopy,
    dictIntDelete,
    0
};

/* common utilities for string and mem */
static unsigned long hashMem(const unsigned char *s, int len)
{
    unsigned long h;
    int i;

    h = 0;
    for(i = 0; i < len; i++) {
        h = (h << 13) + (h >> 7) + h + s[i];
    }
    return h;
}

static void dictDeleteFree(void *x, void *arg) { free(x); }
```

```c
/* string functions */
static unsigned long dictStringHash(const void *x, void *arg)
{
    return hashMem(x, strlen(x));
}

static int dictStringEqual(const void *x, const void *y, void *arg)
{
    return !strcmp((const char *) x, (const char *) y);
}

static void *dictStringCopy(const void *x, void *arg)
{
    const char *s;
    char *s2;

    s = x;
    s2 = malloc(sizeof(*s2) * (strlen(s)+1));
    strcpy(s2, s);
    return s2;
}

struct dictContentsOperations DictStringOps = {
    dictStringHash,
    dictStringEqual,
    dictStringCopy,
    dictDeleteFree,
    0
};

/* mem functions */
static unsigned long dictMemHash(const void *x, void *arg)
{
    return hashMem(x, (int) arg);
}

static int dictMemEqual(const void *x, const void *y, void *arg)
{
    return !memcmp(x, y, (size_t) arg);
}

static void *dictMemCopy(const void *x, void *arg)
{
    void *x2;

    x2 = malloc((size_t) arg);
```

```c
    memcpy(x2, x, (size_t) arg);
    return x2;
}

struct dictContentsOperations
dictMemOps(int len)
{
    struct dictContentsOperations memOps;

    memOps.hash = dictMemHash;
    memOps.equal = dictMemEqual;
    memOps.copy = dictMemCopy;
    memOps.delete = dictDeleteFree;
    memOps.arg = (void *) len;

    return memOps;
}
```

examples/generic/dict.c

And here is some test code and a Makefile: test-dict.c, tester.h, tester.c, Makefile.

5.6 Object-oriented programming

Generic containers as described in the previous section improve on containers for fixed types, but they are still limited to storing a single type. In some circumstances it makes sense to design data structures or functions to work on any type that supplies the right operations. To make this work, we will attach the functions for manipulating an object to the object itself. This is the central idea behind **object-oriented programming**.

As with most sophisticated programming techniques, C doesn't provide any direct support for object-oriented programming, but it is possible to make it work anyway by taking advantage of C's flexibility. We will use two basic ideas: first, we'll give each object a pointer to a **method table** in the form of a struct full of function pointers, and second we'll take advantage of the fact that C lays out struct fields in the order we declare them to allow **subtyping**, where some objects are represented by structs that extend the base struct used for all objects. (We could apply a similar technique to the method tables to allow subtypes to add more methods, but for the example in this section we will keep things simple and provide the same methods for all objects.)

Here is the file that declares the base object type. Note that we expose the details of both `struct object` and `struct methods`, since subtypes will need to implement these.

```c
#ifndef _OBJECT_H
```

```
#define _OBJECT_H

// truncated version of an object
// real object will have more fields after methods
// we expose this for implementers
struct object {
    const struct methods *methods;
};

typedef struct object Object;

struct methods {
    Object *(*clone)(const Object *self);
    void (*print)(const Object *self);
    void (*destroy)(Object *self);
};

#endif
```

examples/objectOriented/object.h

Objects in this system have three methods: `clone`, which makes a copy of the object, `print`, which sends a representation of the object to `stdout`, and `destroy`, which frees the object. Each of these methods takes the object itself as its first argument `self`, since C provides no other mechanism to tell these functions which object to work on. If we needed to pass additional arguments to a method, we would add these after `self`, but for these simple methods this is not needed.

To implement a subtype of object, we extend `struct object` by defining a new struct type that has `methods` as its first field, but may have additional fields to store the state of the object. We also write functions implementing the new type's methods, and build a constant global struct containing pointers to these functions, which all objects of the new type will point to. Finally, we build a constructor function that allocates and initializes an instance of the new object. This will be the only function that is exported from the module for our subtype, because anything else we want to do to an object, we can do by calling methods. This gives a very narrow interface to each subtype module, which is good, because the narrower the interface, the less likely we run into collisions with other modules.

Here is the very short header file for a subtype of `Object` that holds `int`s. Most of this is just the usual header file boilerplate.

```
#ifndef _INTOBJECT_H
#define _INTOBJECT_H

#include "object.h"
```

```c
Object *
intObjectCreate(int value);

#endif
```

examples/objectOriented/intObject.h

And here is the actual implementation. Note that it is only in this file that functions can access the representation `struct intObject` of these objects. Everywhere else, they just look like `Object`s. This does come with a cost: each of the method implementations has to cast in and out of `Object` pointers to work with the underlying `struct intObject`s, and even though this corresponds to precisely zero instructions at run time, if we fail to be disciplined enough to only apply `intObject` methods to `intObject`s, the compiler will not catch the error for us.

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

#include "intObject.h"

// wrap ints up as objects
// this extends Object with extra field
struct intObject {
    struct methods *methods;
    int value;
};

static void printInt(const Object *s);
static Object *cloneInt(const Object *s);
static void destroyInt(Object *s);

static struct methods intObjectMethods = {
    cloneInt,
    printInt,
    destroyInt
};

static void
printInt(const Object *self)
{
    printf("%d", ((struct intObject *) self)->value);
}

static Object *
```

```
cloneInt(const Object *self)
{
    return intObjectCreate(((struct intObject *) self)->value);
}

static void
destroyInt(Object *self)
{
    // we don't have any pointers, so we can just free the block
    free(self);
}

Object *
intObjectCreate(int value)
{
    struct intObject *self = malloc(sizeof(struct intObject));
    assert(self);

    self->methods = &intObjectMethods;
    self->value = value;

    return (Object *) self;
}
```

examples/objectOriented/intObject.c

Having implemented these objects, we can use them in any context where the three provided methods are enough to work with them. For example, here is the interface to a stack that works on arbitrarily `Object`s.

```
#ifndef _STACK_H
#define _STACK_H

#include "object.h"

// basic stack implementation
// stack is a pointer to its first element
// caller will keep a pointer to this
typedef struct elt *Stack;

// create and destroy stacks
Stack *stackCreate(void);
void stackDestroy(Stack *);

// usual functions
void stackPush(Stack *s, Object *);
```

```c
// don't call this on an empty stack
Object *stackPop(Stack *s);

// returns true if not empty
int stackNotEmpty(const Stack *s);

// print the elements of a stack to stdout
// using function print
void stackPrint(const Stack *s);

#endif
```

examples/objectOriented/stack.h

Internally, this stack will use the `clone` method to ensure that it gets its own copy of anything pushed onto the stack in `stackPush`, to protect against the caller later destroying or modifying the object being pushed; the `print` method to print objects in `stackPrint`; and the `destroy` method to clean up in `stackDestroy`. The implementation looks like this:

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

#include "stack.h"

struct elt {
    struct elt *next;
    Object *value;
};

// create and destroy stacks
Stack *
stackCreate(void) {
    Stack *s;

    s = malloc(sizeof(Stack));
    assert(s);
    *s = 0;  // empty stack
    return s;
}

void
stackDestroy(Stack *s) {
    Object *o;
    while(stackNotEmpty(s)) {
        o = stackPop(s);
```

```
        o->methods->destroy(o);
    }
    free(s);
}

// usual functions
void
stackPush(Stack *s, Object *value) {
    struct elt *e = malloc(sizeof(struct elt));
    e->next = *s;
    e->value = value->methods->clone(value);
    *s = e;
}

// don't call this on an empty stack
Object *
stackPop(Stack *s) {
    assert(stackNotEmpty(s));

    struct elt *e = *s;
    Object *ret = e->value;

    *s = e->next;
    free(e);

    return ret;
}

// returns true if not empty
int
stackNotEmpty(const Stack *s) {
    return *s != 0;
}

// print the elements of a stack to stdout
void
stackPrint(const Stack *s) {
    for(struct elt *e = *s; e; e = e->next) {
        e->value->methods->print(e->value);
        putchar(' ');
    }
    putchar('\n');
}
```

examples/objectOriented/stack.c

Because we are working in C, method calls are a little verbose, since we have to follow the method table pointer and supply the self argument ourself. Object-oriented programming languages generally provide syntactic sugar to simplify this task (and avoid possible errors). So a messy line in C like

```
e->value->methods->print(e->value);
```

would look in C++ like

```
e->value.print();
```

Something similar would happen in other object-oriented languages like Python or Java.

Because stack.c accesses objects only through their methods, it will work on any objects, even objects of different types mixed together. Below is a program that mixes `int` objects as defined above with string objects defined in string.h and string.c:

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

#include "object.h"
#include "intObject.h"
#include "stringObject.h"

#include "stack.h"

#define N (3)

// do some stack stuff
int
main(int argc, char **argv)
{
    char str[] = "hi";
    Object *o;

    int n = N;
    if(argc >= 2) {
        n = atoi(argv[1]);
    }

    Stack *s = stackCreate();

    for(int i = 0; i < n; i++) {
        // push a string onto the stack
        str[0] = 'a' + i;
        o = stringObjectCreate(str);
```

```
        stackPush(s, o);
        o->methods->destroy(o);
        stackPrint(s);

        // push an int onto the stack
        o = intObjectCreate(i);
        stackPush(s, o);
        o->methods->destroy(o);
        stackPrint(s);
    }

    while(stackNotEmpty(s)) {
        o = stackPop(s);
        putchar('[');
        o->methods->print(o);
        o->methods->destroy(o);
        fputs("] ", stdout);
        stackPrint(s);
    }

    stackDestroy(s);

    return 0;
}
```

examples/objectOriented/testStack.c

This pushes an alternating pile of ints and strings onto the stack, printing the stack after each push, then pops and prints these objects, again printing the stack after each push. Except for having to choose between `intObjectCreate` and `stringObjectCreate` at creation time, nothing in testStack.c depends on which of these two subtypes each object is.

Of course, to build `testStack` we need to link together a lot of files, which we can do with this Makefile. Running `make test` gives the following output, demonstrating that we are in fact successfully mixing ints with strings:

```
gcc -std=c99 -Wall -pedantic -g3   -c -o testStack.o testStack.c
gcc -std=c99 -Wall -pedantic -g3   -c -o stack.o stack.c
gcc -std=c99 -Wall -pedantic -g3   -c -o intObject.o intObject.c
gcc -std=c99 -Wall -pedantic -g3   -c -o stringObject.o stringObject.c
gcc -std=c99 -Wall -pedantic -g3 -o testStack testStack.o stack.o intObject.o stringObject.o
for i in ; do ./$i; done
for i in testStack; do valgrind -q --leak-check=full ./$i; done
ai
0 ai
bi 0 ai
1 bi 0 ai
```

```
ci 1 bi 0 ai
2 ci 1 bi 0 ai
[2] ci 1 bi 0 ai
[ci] 1 bi 0 ai
[1] bi 0 ai
[bi] 0 ai
[0] ai
[ai]
```

As with generic containers, the nice thing about this approach is that if we want to add more subtypes of object, we can do so the same way we did with `intObject` and `stringObject`, without having to ask anybody's permission to change any of the code in object.h, stack.h, stack.c, or testStack.c. This is very different from what would happen, for example, if an `Object` was implemented as a tagged union, where adding a new type would require rewriting the code for `Object`. The cost is that we have to follow function pointers and be disciplined in how we use them.

5.7 Recursion

Recursion is when a function calls itself. Some programming languages (particularly functional programming languages like Scheme, ML, or Haskell use recursion as a basic tool for implementing algorithms that in other languages would typically be expressed using **iteration** (loops). Procedural languages like C tend to emphasize iteration over recursion, but can support recursion as well.

5.7.1 Example of recursion in C

Here are a bunch of routines that print the numbers from 0 to 9:

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

/* all of these routines print numbers i where start <= i < stop */

void
printRangeIterative(int start, int stop)
{
    int i;

    for(i = start; i < stop; i++) {
        printf("%d\n", i);
    }
}
```

```c
void
printRangeRecursive(int start, int stop)
{
    if(start < stop) {
        printf("%d\n", start);
        printRangeRecursive(start+1, stop);
    }
}

void
printRangeRecursiveReversed(int start, int stop)
{
    if(start < stop) {
        printRangeRecursiveReversed(start+1, stop);
        printf("%d\n", start);
    }
}

void
printRangeRecursiveSplit(int start, int stop)
{
    int mid;

    if(start < stop) {
        mid = (start + stop) / 2;

        printRangeRecursiveSplit(start, mid);
        printf("%d\n", mid);
        printRangeRecursiveSplit(mid+1, stop);
    }
}

#define Noisy(x) (puts(#x), x)

int
main(int argc, char **argv)
{

    if(argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        return 1;
    }

    Noisy(printRangeIterative(0, 10));
    Noisy(printRangeRecursive(0, 10));
```

```
    Noisy(printRangeRecursiveReversed(0, 10));
    Noisy(printRangeRecursiveSplit(0, 10));

    return 0;
}
```

examples/recursion/recursion.c

And here is the output:

```
printRangeIterative(0, 10)
0
1
2
3
4
5
6
7
8
9
printRangeRecursive(0, 10)
0
1
2
3
4
5
6
7
8
9
printRangeRecursiveReversed(0, 10)
9
8
7
6
5
4
3
2
1
0
printRangeRecursiveSplit(0, 10)
0
1
2
```

```
3
4
5
6
7
8
9
```

The first function `printRangeIterative` is simple and direct: it's what we've been doing to get loops forever. The others are a bit more mysterious.

The function `printRangeRecursive` is an example of solving a problem using a divide and conquer approach. If we don't know how to print a range of numbers 0 through 9, maybe we can start by solving a simpler problem of printing the first number 0. Having done that, we have a new, smaller problem: print the numbers 1 through 9. But then we notice we already have a function `printRangeRecursive` that will do that for us. So we'll call it.

If you aren't used to this, it has the feeling of trying to make yourself fly by pulling very hard on your shoelaces.[20] But in fact the computer will happily generate the eleven nested instances of `printRangeRecursive` to make this happen. When we hit the bottom, the call stack will look something like this:

```
printRangeRecursive(0, 10)
 printRangeRecursive(1, 10)
  printRangeRecursive(2, 10)
   printRangeRecursive(3, 10)
    printRangeRecursive(4, 10)
     printRangeRecursive(5, 10)
      printRangeRecursive(6, 10)
       printRangeRecursive(7, 10)
        printRangeRecursive(8, 10)
         printRangeRecursive(9, 10)
          printRangeRecursive(10, 10)
```

This works because each call to `printRangeRecursive` gets its own parameters and its own variables separate from the others, even the ones that are still in progress. So each will print out `start` and then call another copy in to print `start+1` etc. In the last call, we finally fail the test `start < stop`, so the function exits, then its parent exits, and so on until we unwind all the calls on the stack back to the first one.

In `printRangeRecursiveReversed`, the calling pattern is exactly the same, but now instead of printing `start` on the way down, we print `start` on the way back up, after making the recursive call. This means that in `printRangeRecursiveReversed(0, 10)`, 0 is printed only after the results of

[20] A small child of my acquaintance once explained that this wouldn't work, because you would hit your head on the ceiling.

`printRangeRecursiveReversed(1, 10)`, which gives us the countdown effect.

So far these procedures all behave very much like ordinary loops, with increasing values on the stack standing in for the loop variable. More exciting is `printRangeRecursiveSplit`. This function takes a much more aggressive approach to dividing up the problem: it splits a range $[0, 10)$ as two ranges $[0, 5)$ and $[6, 10)$ separated by a midpoint 5.^[The notation $[x, y)$ means all numbers z such that $x \leq z < y$.] We want to print the midpoint in the middle, of course, and we can use `printRangeRecursiveSplit` recursively to print the two ranges. Following the execution of this procedure is more complicated, with the start of the sequence of calls looking something like this:

```
printRangeRecursiveSplit(0, 10)
 printRangeRecursiveSplit(0, 5)
  printRangeRecursiveSplit(0, 2)
   printRangeRecursiveSplit(0, 1)
    printRangeRecursiveSplit(0, 0)
    printRangeRecursiveSplit(1, 1)
   printRangeRecursiveSplit(2, 2)
  printRangeRecursiveSplit(3, 5)
   printRangeRecursiveSplit(3, 4)
    printRangeRecursiveSplit(3, 3)
    printRangeRecursiveSplit(4, 4)
   printRangeRecursiveSplit(5, 5)
 printRangeRecursiveSplit(6, 10)
  ... etc.
```

Here the computation has the structure of a tree instead of a list, so it is not so obvious how one might rewrite this procedure as a loop.

5.7.2 Common problems with recursion

Like iteration, recursion is a powerful tool that can cause your program to do much more than expected. While it may seem that errors in recursive functions would be harder to track down than errors in loops, most of the time there are a few basic causes.

5.7.2.1 Omitting the base case

Suppose we leave out the `if` statement in `printRangeRecursive`:

```c
void
printRangeRecursiveBad(int start, int stop)
{
    printf("%d\n", start);
    printRangeRecursiveBad(start+1, stop);
}
```

This will still work, in a sense. When called as `printRangeRecursiveBad(0, 10)`, it will print 0, call itself with `printRangeRecursiveBad(1, 10)`, print 1, 2, 3, etc., but there is nothing to stop it at 10 (or anywhere else). So our output will be a long string of numbers followed by a segmentation fault, when we blow out the stack.

This is the recursive version of an infinite loop: the same thing happens if we forget a loop test and write

```
void
printRangeIterativeBad(int start, int stop)
{
    for(i = 0; ; i++) {
        printf("%d\n", i);
    }
}
```

except that now the program just runs forever, since it never runs out of resources. This is an example of how iteration is more efficient than recursion, at least in C.

5.7.2.2 Blowing out the stack

Blowing out the stack is what happens when a recursion is too deep. Typically, the operating system puts a hard limit on how big the stack can grow, on the assumption that any program that grows the stack too much has gone insane and needs to be killed before it does more damage. One of the ways this can happen is if we forget the base case as above, but it can also happen if we just try to use a recursive function to do too much. For example, if we call `printRangeRecursive(0, 1000000)`, we will probably get a segmentation fault after the first 100,000 numbers or so.

For this reason, it's best to try to avoid linear recursions like the one in `printRangeRecursive`, where the depth of the recursion is proportional to the number of things we are doing. Much safer are even splits like `printRangeRecursiveSplit`, since the depth of the stack will now be only logarithmic in the number of things we are doing. Fortunately, linear recursions are often **tail-recursive**, where the recursive call is the last thing the recursive function does; in this case, we can use a standard transformation to convert the tail-recursive function into an iterative function.

5.7.2.3 Failure to make progress

Sometimes we end up blowing out the stack because we thought we were recursing on a smaller instance of the problem, but in fact we weren't. Consider this broken version of `printRangeRecursiveSplit`:

```
void
printRangeRecursiveSplitBad(int start, int stop)
{
    int mid;
```

```
    if(start == stop) {
        printf("%d\n", start);
    } else {
        mid = (start + stop) / 2;

        printRangeRecursiveSplitBad(start, mid);
        printRangeRecursiveSplitBad(mid, stop);
    }
}
```

This will get stuck on as simple a call as `printRangeRecursiveSplitBad(0, 1)`; it will set `mid` to 0, and while the recursive call to `printRangeRecursiveSplitBad(0, 0)` will work just fine, the recursive call to `printRangeRecursiveSplitBad(0, 1)` will put us back where we started, giving an infinite recursion.

Detecting these errors is usually not too hard (segmentation faults that produce huge piles of stack frames when you type `where` in gdb are a dead give-away). Figuring out how to make sure that you do in fact always make progress can be trickier.

5.7.3 Tail-recursion and iteration

Tail recursion is when a recursive function calls itself only once, and as the last thing it does. The `printRangeRecursive` function is an example of a tail-recursive function:

```
void
printRangeRecursive(int start, int stop)
{
    if(start < stop) {
        printf("%d\n", start);
        printRangeRecursive(start+1, stop);
    }
}
```

The nice thing about tail-recursive functions is that we can always translate them directly into iterative functions. The reason is that when we do the tail call, we are effectively replacing the current copy of the function with a new copy with new arguments. So rather than keeping around the old zombie parent copy—which has no purpose other than to wait for the child to return and then return itself—we can reuse it by assigning new values to its arguments and jumping back to the top of the function.

Done literally, this produces this `goto`-considered-harmful monstrosity:

```
void
printRangeRecursiveGoto(int start, int stop)
```

```
{
    topOfFunction:

    if(start < stop) {
        printf("%d\n", start);

        start = start+1;
        goto topOfFunction;
    }
}
```

But we can almost always remove `goto` statements using less dangerous control structures. In this particular case, the pattern of jumping back to just before an `if` matches up exactly with what we get from a `while` loop:

```c
void
printRangeRecursiveNoMore(int start, int stop)
{
    while(start < stop) {
        printf("%d\n", start);

        start = start+1;
    }
}
```

In functional programming languages, this transformation is usually done in the other direction, to unroll loops into recursive functions. Since C doesn't like recursive functions so much (they blow out the stack!), we usually do this transformation got get rid of recursion instead of adding it.

5.7.3.1 Binary search: recursive and iterative versions

Binary search is an algorithm for searching a sorted array for a particular target element, similar to playing Twenty Questions when the answer is a number (hopefully in a range that includes at most 2^{20} numbers). The algorithm starts by picking an value in the middle of the array. If the target is less than this value, we recurse on the bottom half of the array; else we recurse on the top half.

Here is an interface for binary search on an array of `int`s:

```c
/* returns 1 if target is present in sorted array */
int binarySearch(int target, const int *a, size_t length);
```

examples/binarySearch/binarySearch.h

Written recursively, we might implement the algorithm like this:

```c
#include <stddef.h>

#include "binarySearch.h"
```

```c
int
binarySearch(int target, const int *a, size_t length)
{
    size_t index;

    index = length/2;

    if(length == 0) {
        /* nothing left */
        return 0;
    } else if(target == a[index]) {
        /* got it */
        return 1;
    } else if(target < a[index]) {
        /* recurse on bottom half */
        return binarySearch(target, a, index);
    } else {
        /* recurse on top half */
        /* we throw away index+1 elements (including a[index]) */
        return binarySearch(target, a+index+1, length-(index+1));
    }
}
```

examples/binarySearch/binarySearchRecursive.c

This will work just fine, and indeed it finds the target element (or not) in $O(\log n)$ time, because we can only recurse $O(\log n)$ times before running out of elements and we only pay $O(1)$ cost per recursive call to `binarySearch`. But we do have to pay function call overhead for each recursive call, and there is a potential to run into stack overflow if our stack is very constrained.

Fortunately, we don't do anything with the return value from `binarySearch` but pass it on up the stack: the function is tail-recursive. This means that we can get rid of the recursion by reusing the stack from from the initial call. The mechanical way to do this is wrap the body of the routine in a `for(;;)` loop (so that we jump back to the top whenever we hit the bottom), and replace each recursive call with one or more assignments to update any parameters that change in the recursive call. The result looks like this:

```c
#include <stddef.h>

#include "binarySearch.h"

int
binarySearch(int target, const int *a, size_t length)
{
    size_t index;
```

```
    /* direct translation of recursive version */
    /* hence the weird organization of the loop */
    for(;;) {
        index = length/2;

        if(length == 0) {
            /* nothing left */
            return 0;
        } else if(target == a[index]) {
            /* got it */
            return 1;
        } else if(target < a[index]) {
            /* recurse on bottom half */
            length = index;
        } else {
            /* recurse on top half */
            /* we throw away index+1 elements (including a[index]) */
            a = a + index + 1;
            length = length - (index + 1);
        }
    }
}
```

examples/binarySearch/binarySearchIterative.c

Here's some simple test code to demonstrate that these two implementations in fact do the same thing: Makefile, testBinarySearch.c.

5.7.4 Mergesort: a recursive sorting algorithm

So far the examples we have given have not been very useful, or have involved recursion that we can easily replace with iteration. Here is an example of a recursive procedure that cannot be turned into an iterative version so easily.

We are going to implement the mergesort algorithm on arrays. This is a classic divide and conquer sorting algorithm that splits an array into two pieces, sorts each piece (recursively!), then merges the results back together. Here is the code, together with a simple test program.

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

/* merge sorted arrays a1 and a2, putting result in out */
void
merge(int n1, const int a1[], int n2, const int a2[], int out[])
```

```
{
    int i1;
    int i2;
    int iout;

    i1 = i2 = iout = 0;

    while(i1 < n1 || i2 < n2) {
        if(i2 >= n2 || ((i1 < n1) && (a1[i1] < a2[i2]))) {
            /* a1[i1] exists and is smaller */
            out[iout++] = a1[i1++];
        }  else {
            /* a1[i1] doesn't exist, or is bigger than a2[i2] */
            out[iout++] = a2[i2++];
        }
    }
}

/* sort a, putting result in out */
/* we call this mergeSort to avoid conflict with mergesort in libc */
void
mergeSort(int n, const int a[], int out[])
{
    int *a1;
    int *a2;

    if(n < 2) {
        /* 0 or 1 elements is already sorted */
        memcpy(out, a, sizeof(int) * n);
    } else {
        /* sort into temp arrays */
        a1 = malloc(sizeof(int) * (n/2));
        a2 = malloc(sizeof(int) * (n - n/2));

        mergeSort(n/2, a, a1);
        mergeSort(n - n/2, a + n/2, a2);

        /* merge results */
        merge(n/2, a1, n - n/2, a2, out);

        /* free the temp arrays */
        free(a1);
        free(a2);
    }
}
```

```c
#define N (20)

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

    for(i = 0; i < N; i++) {
        a[i] = N-i-1;
    }

    for(i = 0; i < N; i++) {
        printf("%d ", a[i]);
    }
    putchar('\n');

    mergeSort(N, a, b);

    for(i = 0; i < N; i++) {
        printf("%d ", b[i]);
    }
    putchar('\n');

    return 0;
}
```

examples/sorting/mergesort.c

The cost of this is pretty cheap: $O(n \log n)$, since each element of `a` is processed through `merge` once for each array it gets put in, and the recursion only goes $O(\log n)$ layers deep before we hit 1-element arrays.

The reason that we can't easily transform this into an iterative version is that the `mergeSort` function is not tail-recursive: not only does it call itself twice, but it also needs to free the temporary arrays at the end. Because the algorithm has to do these tasks on the way back up the stack, we need to keep the stack around to track them.

5.7.5 Asymptotic complexity of recursive functions

One issue with a recursive functions is that it becomes harder to estimate its asymptotic complexity. Unlike loops, where we can estimate the cost by simply multiplying the number of iterations by the cost of each iteration, the cost of a recursive function depends on the cost of its recursive calls. This would make it seem that we would need to be able to compute the cost of the function before we could compute the cost of the function.

Fortunately, for most recursive functions, the size of the input drops whenever we recurse. So the cost can be expressed in terms of a **recurrence**, a formula for the cost $T(n)$ on an input of size n in terms of the cost on smaller inputs. Some examples:

$T(n) = O(1) + T(n/2)$ This is the cost of binary search. To search an array of n elements, look up the middle element ($O(1)$ time) and, in the worst case, recurse on an array of $n/2$ elements.

$T(n) = 2T(n/2) + O(n)$ This is the cost of mergesort. Sort two half-size arrays recursively, then merge them in $O(n)$ time.

$T(n) = O(1) + T(n-1)$ This is the cost of most simple loops, if we think of them as a recursive process. Do $O(1)$ work on the first element, then do $T(n-1)$ work on the rest.

There are standard tools for solving many of the recurrences that arise in common algorithms, but these are overkill for our purposes, since there are only a handful of recurrences that are likely to come up in practice and we already solved most of them. Here is a table of some of the more common possibilities:

Recurrence	Solution	Example
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$	Finding a maximum
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$	Selection sort
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$	Binary search
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$	Mergesort

5.8 Binary trees

Divide and conquer yields algorithms whose execution has a tree structure. Sometimes we build data structures that are also trees. It is probably not surprising that divide and conquer is the natural way to build algorithms that use such trees as inputs.

5.8.1 Tree basics

Here is a typical binary tree. It is binary because every node has at most two children. This particular tree is also **complete** because the nodes consist only of **internal nodes** with exactly two children and **leaves** with no children. Not all binary trees will be complete.

```
  0
 / \
1   2
```

```
    / \
   3   4
  / \
 5   6
/ \
7   8
```

Structurally, a complete binary tree consists of either a single node (a leaf) or a root node with a left and right **subtree**, each of which is itself either a leaf or a root node with two subtrees. The set of all nodes underneath a particular node x is called the subtree rooted at x.

The **size** of a tree is the number of nodes; a leaf by itself has size 1. The **height** of a tree is the length of the longest path; 0 for a leaf, at least one in any larger tree. The **depth** of a node is the length of the path from the root to that node. The **height** of a node is the height of the subtree of which it is the root, i.e. the length of the longest path from that node to some leaf below it. A node u is an **ancestor** of a node v if v is contained in the subtree rooted at u; we may write equivalently that v is a **descendant** of u. Note that every node is both an ancestor and a descendant of itself. If we wish to exclude the node itself, we refer to a **proper ancestor** or **proper descendant**.

5.8.2 Binary tree implementations

In a low-level programming language like C, a binary tree typically looks a lot like a linked list with an extra outgoing pointer from each element, e.g.

```c
struct node {
    int key;
    struct node *left;  /* left child */
    struct node *right; /* right child */
};
```

An alternative is to put the pointers to the children in an array. This lets us loop over both children, pass in which child we are interested in to a function as an argument, or even change the number of children:

```c
#define NUM_CHILDREN (2)

struct node {
    int key;
    struct node *child[NUM_CHILDREN];
};
```

Which approach we take is going to be a function of how much we like writing `left` and `right` vs `child[0]` and `child[1]`. A possible advantage of `left` and `right` is that it is harder to make mistakes that are not caught by the compiler (`child[2]`). Using the preprocessor, it is in principle possible to have your cake and eat it too (`#define left child[0]`), but I would not recommend this unless you are deliberately trying to confuse people.

Missing children (and the empty tree) are represented by null pointers. Typically, individual tree nodes are allocated separately using `malloc`; however, for high-performance use it is not unusual for tree libraries to do their own storage allocation out of large blocks obtained from `malloc`.

Optionally, the `struct` may be extended to include additional information such as a pointer to the node's parent, hints for balancing, or aggregate information about the subtree rooted at the node such as its size or the sum/max/average of the keys of its nodes.

When it is not important to be able to move large subtrees around simply by adjusting pointers, a tree may be represented implicitly by packing it into an array. This is a standard approach for implementing heaps, which we will see soon.

5.8.3 The canonical binary tree algorithm

Pretty much every divide and conquer algorithm for binary trees looks like this:

```
void
doSomethingToAllNodes(struct node *root)
{
    if(root) {
        doSomethingTo(root);
        doSomethingToAllNodes(root->left);
        doSomethingToAllNodes(root->right);
    }
}
```

The function processes all nodes in what is called a **preorder traversal**, where the "preorder" part means that the root of any tree is processed first. Moving the call to `doSomethingTo` in between or after the two recursive calls yields an **inorder** or **postorder** traversal, respectively.

In practice we usually want to extract some information from the tree. For example, this function computes the size of a tree:

```
int
treeSize(struct node *root)
{
    if(root == 0) {
        return 0;
    } else {
        return 1 + treeSize(root->left) + treeSize(root->right);
```

```
    }
}
```

and this function computes the height:

```
int
treeHeight(struct node *root)
{
    int lh;     /* height of left subtree */
    int rh;     /* height of right subtree */

    if(root == 0) {
        return -1;
    } else {
        lh = treeHeight(root->left);
        rh = treeHeight(root->right);
        return 1 + (lh > rh ? lh : rh);
    }
}
```

Since both of these algorithms have the same structure, they both have the same asymptotic running time. We can compute this running time by observing that each recursive call to `treeSize` or `treeHeight` that does not get a null pointer passed to it gets a different node (so there are n such calls), and each call that does get a null pointer passed to it is called by a routine that doesn't, and that there are at most two such calls per node. Since the body of each call itself costs $O(1)$ (no loops), this gives a total cost of $\Theta(n)$.

So these are all $\Theta(n)$ algorithms.

5.8.4 Nodes vs leaves

For some binary trees we don't store anything interesting in the internal nodes, using them only to provide paths to the leaves. We might reasonably ask if an algorithm that runs in $O(n)$ time where n is the total number of nodes still runs in $O(m)$ time, where m counts only the leaves. For *complete* binary trees, we can show that we get the same asymptotic performance whether we count leaves only, internal nodes only, or both leaves and internal nodes.

Let $T(n)$ be the number of internal nodes in a complete binary tree with n leaves. It is easy to see that $T(1) = 0$ and $T(2) = 1$, but for larger trees there are multiple structures and so it makes sense to write a recurrence: $T(n) = 1+T(k)+T(n-k)$.

We can show by induction that the solution to this recurrence is exactly $T(n) = n-1$. We already have the base case $T(1) = 0$. For larger n, we have $T(n) = 1 + T(k) + T(n-k) = 1 + (k-1) + (n-k-1) = n-1$.

So a complete binary tree with $\Theta(n)$ nodes has $\Theta(n)$ internal nodes and $\Theta(n)$

leaves; if we don't care about constant factors, we won't care which number we use.

5.8.5 Special classes of binary trees

So far we haven't specified where particular nodes are placed in the binary tree. Most applications of binary trees put some constraints on how nodes relate to one another. Some possibilities:

- Heaps: Each node has a key that is less than the keys of both of its children. These allow for a very simple implementation using arrays, so we will look at these first.
- BinarySearchTrees: Each node has a key, and a node's key must be greater than all keys in the subtree of its left-hand child and less than all keys in the subtree of its right-hand child.

5.9 Heaps

A **heap** is a binary tree in which each element has a key (or sometimes **priority**) that is less than the keys of its children. Heaps are used to implement the **priority queue** abstract data type, which we'll talk about first.

5.9.1 Priority queues

In a standard queue, elements leave the queue in the same order as they arrive. In a priority queue, elements leave the queue in order of decreasing priority: the DEQUEUE operation becomes a DELETE-MIN operation (or DELETE-MAX, if higher numbers mean higher priority), which removes and returns the highest-priority element of the priority queue, regardless of when it was inserted. Priority queues are often used in operating system schedulers to determine which job to run next: a high-priority job (e.g., turn on the fire suppression system) runs before a low-priority job (floss the cat) even if the low-priority job has been waiting longer.

5.9.2 Expensive implementations of priority queues

Implementing a priority queue using an array or linked list is likely to be expensive. If the array or list is unsorted, it takes $O(n)$ time to find the minimum element; if it is sorted, it takes $O(n)$ time (in the worst case) to add a new element. So such implementations are only useful when the numbers of INSERT and DELETE-MIN operations are very different. For example, if DELETE-MIN is called only rarely but INSERT is called often, it may actually be cheapest to implement a priority queue as an unsorted linked list with $O(1)$

INSERTs and $O(n)$ DELETE-MINs. But if we expect that every element that is inserted is eventually removed, we want something for which both INSERT and DELETE-MIN are cheap operations.

5.9.3 Structure of a heap

A heap is a binary tree in which each node has a smaller key than its children; this property is called the **heap property** or **heap invariant**.

To insert a node in the heap, we add it as a new leaf, which may violate the heap property if the new node has a lower key than its parent. But we can restore the heap property (at least between this node and its parent) by swapping either the new node or its sibling with the parent, where in either case we move up the node with the smaller key. This may still leave a violation of the heap property one level up in the tree, but by continuing to swap small nodes with their parents we eventually reach the top and have a heap again. The time to complete this operation is proportional to the depth of the heap, which will typically be $O(\log n)$ (we will see how to enforce this in a moment).

To implement DELETE-MIN, we can easily find the value to return at the top of the heap. Unfortunately, removing it leaves a vacuum that must be filled in by some other element. The easiest way to do this is to grab a leaf (which probably has a very high key), and then float it down to where it belongs by swapping it with its smaller child at each iteration. After time proportional to the depth (again $O(\log n)$ if we are doing things right), the heap invariant is restored.

Similar local swapping can be used to restore the heap invariant if the priority of some element in the middle changes; we will not discuss this in detail.

5.9.4 Packed heaps

It is possible to build a heap using `struct`s and pointers, where each element points to its parent and children. In practice, heaps are instead stored in arrays, with an implicit pointer structure determined by array indices. For zero-based arrays as in C, the rule is that a node at position `i` has children at positions `2*i+1` and `2*i+2`; in the other direction, a node at position `i` has a parent at position `(i-1)/2` (which rounds down in C). This is equivalent to storing a heap in an array by reading through the tree in breadth-first search order:

```
   0
  / \
 1   2
/ \ / \
3 4 5 6
```

becomes

```
0 1 2 3 4 5 6
```

This approach works best if there are no gaps in the array. So to maximize efficiency we make this "no gaps" policy part of the invariant. We can do so because we don't care which leaf gets added when we do an INSERT, so we can make it be whichever leaf is at the end of the array. Similarly, in a DELETE-MIN operation, we can promote the element at the end of the array to the root before floating it down. Both these operations change the number of elements in the array, and INSERTs in particular might force us to reallocate eventually. So in the worst case INSERT can be an expensive operation, although as with growing hash tables, the amortized cost may still be small.

5.9.5 Bottom-up heapification

If we are presented with an unsorted array, we can turn it into a heap more quickly than the $O(n \log n)$ time required to do n INSERTs. The trick is to build the heap from the bottom up. This means starting with position $n-1$ and working back to position 0, so that when it comes time to fix the heap invariant at position i, we have already fixed it at all later positions (this is a form of dynamic programming). Unfortunately, it is not quite enough simply to swap `a[i]` with its smaller child when we get there, because we could find that `a[0]` (say) was the largest element in the heap. But the cost of floating `a[i]` down to its proper place will be proportional to its own height rather than the height of the entire heap. Since most of the elements of the heap are close to the bottom, the total cost will turn out to be $O(n)$.

5.9.6 Heapsort

Bottom-up heapification is used in the Heapsort algorithm, which first does bottom-up heapification in $O(n)$ time and then repeatedly calls DELETE-MAX to extract the largest remaining element. This is no faster than the $O(n \log n)$ cost of mergesort or quicksort in typical use, but requires very little auxiliary storage since we can maintain the heap in the bottom part of the same array whose top part stores the max elements extracted so far.

Here is a simple implementation of heapsort, that demonstrates how both bottom-up heapification and the DELETE-MAX procedure work by floating elements down to their proper places:

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

/* max heap implementation */

/* compute child 0 or 1 */
```

```c
#define Child(x, dir) (2*(x)+1+(dir))

/* float value at position pos down */
static void
floatDown(int n, int *a, int pos)
{
    int x;

    /* save original value once */
    x = a[pos];

    for(;;) {
        if(Child(pos, 1) < n && a[Child(pos, 1)] > a[Child(pos, 0)]) {
            /* maybe swap with Child(pos, 1) */
            if(a[Child(pos, 1)] > x) {
                a[pos] = a[Child(pos, 1)];
                pos = Child(pos, 1);
            } else {
                /* x is bigger than both kids */
                break;
            }
        } else if(Child(pos, 0) < n && a[Child(pos, 0)] > x) {
            /* swap with Child(pos, 0) */
            a[pos] = a[Child(pos, 0)];
            pos = Child(pos, 0);
        } else {
            /* done */
            break;
        }
    }

    a[pos] = x;
}

/* construct a heap bottom-up */
static void
heapify(int n, int *a)
{
    int i;

    for(i = n-1; i >= 0; i--) {
        floatDown(n, a, i);
    }
}

/* sort an array */
```

```c
void
heapSort(int n, int *a)
{
    int i;
    int tmp;

    heapify(n, a);

    for(i = n-1; i > 0; i--) {
        /* swap max to a[i] */
        tmp = a[0];
        a[0] = a[i];
        a[i] = tmp;

        /* float new a[0] down */
        floatDown(i, a, 0);
    }
}

#define N (100)
#define MULTIPLIER (17)

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

    if(argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        return 1;
    }

    for(i = 0; i < N; i++) { a[i] = (i*MULTIPLIER) % N; }

    for(i = 0; i < N; i++) { printf("%d ", a[i]); }
    putchar('\n');

    heapSort(N, a);

    for(i = 0; i < N; i++) { printf("%d ", a[i]); }
    putchar('\n');

    return 0;
}
```

examples/sorting/heapsort.c

5.9.7 More information

- Priority_queue
- Binary_heap
- http://mathworld.wolfram.com/Heap.html

5.10 Binary search trees

A **binary search tree** is a binary tree in which each node has a **key**, and a node's key must be greater than all keys in the subtree of its left-hand child and less than all keys in the subtree of its right-hand child. This allows a node to be searched for using essentially the same binary search algorithm used on sorted arrays.

5.10.1 Searching for a node

```c
/* returns pointer to node with given target key */
/* or 0 if no such node exists */
struct node *
treeSearch(struct node *root, int target)
{
    if(root == 0 || root->key == target) {
        return root;
    } else if(root->key > target) {
        return treeSearch(root->left, target);
    } else {
        return treeSearch(root->right, target);
    }
}
```

This procedure can be rewritten iteratively, which avoids stack overflow and is likely to be faster:

```c
struct node *
treeSearch(struct node *root, int target)
{
    while(root != 0 && root->key != target) {
        if(root->key > target) {
            root = root->left;
        } else {
            root = root->right;
        }
```

```
    }

    return root;
}
```

These procedures can be modified in the obvious way to deal with keys that aren't `int`s, as long as they can be compared (e.g., by using `strcmp` on strings).

5.10.2 Inserting a new node

As in a hash table, the insertion procedure mirrors the search procedure. We must be a little careful to avoid actually walking all the way down to a null pointer, since a null pointer now indicates a missing space for a leaf that we can fill with our new node. So the code is a little more complex.

```
// Insert a new key into a tree whose root is pointed to
// by *parent, which should be 0 if tree is empty.  May modify *parent.
void
treeInsert(struct tree **parent, int key)
{
    struct tree *newNode;

    for(;;) {
        if(*parent == 0) {
            // put it here
            *parent = malloc(sizeof(*newNode));
            assert(*parent);

            (*parent)->key = key;
            (*parent)->left = 0;
            (*parent)->right = 0;

            return;
        } else if(key == (*parent)->key) {
            // already present
            return;
        } else if(key < (*parent)->key) {
            // insert in left subtree
            parent = &(*parent)->left;
        } else {
            // insert in right subtree
            parent = &(*parent)->right;
        }
    }
}
```

Note that this function makes not attempt to keep the tree balanced. This may lead to very long paths if new keys are inserted in strictly increasing or strictly decreasing order.

5.10.3 Deleting a node

Deletion is more complicated. If a node has no children, we can just remove it, and the rest of the tree stays the same. A node with one child can be spliced out, connecting its parent directly to its child. But with two children, we can't do this.

The trick is to find the leftmost node in our target's right subtree (or vice versa). This node exists assuming the target has two children. As in a hash table, we can then swap our target node with this more convenient node. Because it is the leftmost node, it has no left child, so we can delete it using the no-children or one-child case.

5.10.4 Costs

Searching for or inserting a node in a binary search tree with n nodes takes time proportional to the depth of the node. In balanced trees, where the nodes in each subtree are divided roughly evenly between the two child subtrees, this will be $O(\log n)$, but for a badly unbalanced tree, this might be as much as $O(n)$. So making a binary search tree work efficiently requires keeping it balanced.

5.11 Augmented trees

An **augmented** data structure stores additional information in each of its nodes that caches values that might otherwise be expensive to compute. For trees, this might include information like the size of a subtree (which can be useful for **ranking** values, where we want to determine how many elements of the tree are smaller), the height of a subtree, or other summary information like the sum of all the keys in a subtree.

Augmented data structures, in a sense, violate the no-separate-but-equal rule that says we shouldn't store the same information in different places. The reason we try to avoid this is that it's trouble if the two copies diverge, and by not having two copies in the first place there is no possibility that they contradict each other. But in this case the reduced cost justifies breaking this rule.

The idea is that when we insert a new element into an augmented tree, it only changes the height/size/sum/etc. values for nodes on the path from the root to the new value. Since each of these aggregate values can be computed for a node in $O(1)$ time from the values in its children, we can update all the aggregate values on our way back up the stack after doing the insertion at a cost of $O(1)$

per node. This will give a total cost of $O(\log n)$ assuming our tree is reasonably balanced.

5.11.1 Applications

Storing the height field can be useful for balancing, as in AVL trees.

Storing the size allows ranking (computing the number of elements less than a given target value) and unraking (find an element with a particular rank). Sample code for doing this is given in the AVL tree sample implementation below.

Storing other aggregates like the sum of keys or values allows **range queries**, where we ask, for example, for some aggregate statistic (like the sum or average) of all the elements between some given minimum and maximum.

Assuming we keep the tree balanced and correctly maintain the aggregate data or each subtree, all of these operations can be done in $O(\log n)$ time.

5.12 Balanced trees

Binary search trees are a fine idea, but they only work if they are **balanced**—if moving from a tree to its left or right subtree reduces the size by a constant fraction. Balanced binary trees add some extra mechanism to the basic binary search tree to ensure balance. Finding efficient ways to balance a tree has been studied for decades, and several good mechanisms are known. We'll try to hit the high points of all of them.

5.12.1 Tree rotations

The problem is that as we insert new nodes, some paths through the tree may become very long. So we need to be able to shrink the long paths by moving nodes elsewhere in the tree.

But how do we do this? The idea is to notice that there may be many binary search trees that contain the same data, and that we can transform one into another by a local modification called a *rotation*:

```
    y             x
   / \   <==>    / \
  x   C         A   y
 / \               / \
A   B             B   C

Single rotation on x-y edge
```

If $A < x < B < y < C$, then both versions of this tree have the binary search tree property. By doing the rotation in one direction, we move A up and C down; in the other direction, we move A down and C up. So rotations can be used to transfer depth from the leftmost grandchild of a node to the rightmost and vice versa.

But what if it's the middle grandchild B that's the problem? A single rotation as above doesn't move B up or down. To move B, we have to reposition it so that it's on the end of something. We do this by splitting B into two subtrees B_1 and B_2, and doing two rotations that split the two subtrees while moving both up. For this we need to do two rotations:

```
    z               z                 y
   / \   ===>      / \     ===>      / \
  x   C           y   C             x   z
 / \             / \               /|   |\
A   y           x  B2             A B1 B2 C
   / \         / \
  B1 B2       A   B1

Double rotation: rotate xy then zy
```

Rotations in principle let us rebalance a tree, but we still need to decide when to do them. If we try to keep the tree in perfect balance (all paths nearly the same length), we'll spend so much time rotating that we won't be able to do anything else. So we need a scheme that keeps a tree balanced enough that we get paths of length $O(\log n)$ while still doing $O(\log n)$ rotations per operations.

5.12.2 Treaps

One of the simplest balanced tree data structures is the **treap**, a randomized tree that combines the invariants of a binary search tree and a heap. The idea is to give each node two keys: a tree key used for searching, and a heap key, which is chosen at random and used to decide which nodes rise to the top. If we ignore the heap keys, a treap is a binary search tree: the tree keys store the contents of the tree, and the tree key in each node is greater than all keys in its left subtree and less than all keys in its right subtree. If we ignore the tree keys: a treap is a heap: each node has a heap key that is larger than all heap keys in is descendants.

It is not immediately obvious that we can preserve both invariants at once, but if we are presented with an unorganized pile of nodes already assigned tree and heap keys, we can build a treap by making the node with the largest *heap* key the root, putting all the nodes with smaller *tree* heap keys in the left subtree and all the nodes with larger *tree* keys in the right subtree, then organizing both subtrees recursively in the same way. This tells use that for any assignment of tree and heap keys, a treap exists (and is in fact unique!), so the only trick is to make sure that it doesn't cost too much to preserve the treap property when we insert a new element.

To insert a new node, we assign it a random heap key and insert it at a leaf using the usual method for a binary search tree. Because the new node's random key may be large, this may violate the heap property. But we can rotate it up until the heap property is restored.

For deletions, we first have to search for a node with the key we want to delete, then remove it from the tree. If the node has has most one child, we can just patch it out, by changing its parent's pointer to point to the child (or to null, if there is no child). If the node has two children, we pick the bigger one and rotate it up. Repeating this process will eventually rotate the node we are deleting down until it either has one child or is a leaf.

It's not hard to see that the cost of any of these operations is proportional to length of some path in the treap. If all nodes have random heap keys, then the root node will be equally likely to be any of the nodes. This doesn't guarantee that we get a good split, but a very bad split requires picking a root node close to one side or the other, which is unlikely. People smarter than I am have analyzed the expected height of a tree constructed in this way and show that the length of the longest path converges to $(4.311\ldots)\log n$ in the limit (Devroye, A note on the height of binary search trees, JACM 1986; the upper bound is due to Robson, The height of binary search trees, Australian Computer Journal, 1979). This gives an $O(\log n)$ bound for the expected height in practice. However, we do have to be careful to make sure that whoever is supplying our inputs can't see what heap keys we pick, or they will be able to generate an unbalanced tree by repeatedly inserting and deleting nodes until they draw bad heap keys.

Below is an example of a binary search tree implemented as a treap. You can also find much more about treaps on the web page of Cecilia Aragon, one of the inventors of treaps.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>
#include <limits.h>

#define NUM_CHILDREN (2)

#define LEFT (0)
#define RIGHT (1)

// Invariants:
// - Every key below child[LEFT] < key < every key below child[RIGHT]
// - Every heapKey in both subtreaps < heapKey.
// heapKeys are chosen randomly to ensure balance with high probability.
```

```
struct treap {
    int key;
    int heapKey;
    struct treap *child[NUM_CHILDREN];
};

void
treapDestroy(struct treap **t)
{
    if(*t) {
        for(int dir = LEFT; dir <= RIGHT; dir++) {
            treapDestroy(&(*t)->child[dir]);
        }
    }

    free(*t);
    *t = 0;
}

void
treapPrintHelper(const struct treap *t, int depth)
{
    if(t == 0) {
        return;
    }

    treapPrintHelper(t->child[LEFT], depth+1);

    // print indented root
    for(int i = 0; i < depth; i++) {
        putchar(' ');
    }

    printf("%d [%d]\n", t->key, t->heapKey);

    treapPrintHelper(t->child[RIGHT], depth+1);
}

void
treapPrint(const struct treap *t)
{
    treapPrintHelper(t, 0);
}

// return 1 if it finds key, 0 otherwise
int
```

```c
treapSearch(const struct treap *t, int key)
{
    if(t == 0) {
        // no key!
        return 0;
    } else if(key == t->key) {
        // found it
        return 1;
    } else if(key < t->key) {
        // look in left
        return treapSearch(t->child[LEFT], key);
    } else {
        // look in right
        return treapSearch(t->child[RIGHT], key);
    }
}

// return largest element <= key
// or INT_MIN if there is no such element.
int
treapSearchMaxLE(const struct treap *t, int key)
{
    if(t == 0) {
        // no key!
        return INT_MIN;
    } else if(key == t->key) {
        // found it
        return key;
    } else if(key < t->key) {
        // look in left
        return treapSearchMaxLE(t->child[LEFT], key);
    } else {
        // look in right
        int result = treapSearchMaxLE(t->child[RIGHT], key);

        if(result == INT_MIN) {
            // didn't find it
            return t->key;
        } else {
            return result;
        }
    }
}

// rotate the treap pointed to by parent
// so that child in direction moves up
```

```
void
treapRotateUp(struct treap **parent, int dir)
{
    // get pointers to anything that might move
    assert(parent);
    struct treap *child = *parent;
    assert(child);
    struct treap *grandchild = child->child[dir];
    assert(grandchild);
    struct treap *middleSubtreap = grandchild->child[!dir];

    // do the move
    *parent = grandchild;
    grandchild->child[!dir] = child;
    child->child[dir] = middleSubtreap;
}


// insert key into treap pointed to by parent
// if not already present
void
treapInsert(struct treap **parent, int key)
{
    if(*parent == 0) {
        // no key!
        *parent = malloc(sizeof(struct treap));
        (*parent)->key = key;
        (*parent)->heapKey = rand();
        (*parent)->child[LEFT] = (*parent)->child[RIGHT] = 0;
    } else if(key == (*parent)->key) {
        // found it
        return;
    } else if(key < (*parent)->key) {
        // look in left
        treapInsert(&(*parent)->child[LEFT], key);
    } else {
        // look in right
        treapInsert(&(*parent)->child[RIGHT], key);
    }

    // check heap property
    for(int dir = LEFT; dir <= RIGHT; dir++) {
        if((*parent)->child[dir] != 0 && (*parent)->child[dir]->heapKey > (*parent)->heapKey
            treapRotateUp(parent, dir);
        }
    }
```

```c
}

// delete a node from a treap (if present)
void
treapDelete(struct treap **parent, int key)
{
    // first we look for it
    if(*parent == 0) {
        // not there
        return;
    } else if(key == (*parent)->key) {
        // got it; rotate down until we have a missing kid
        for(;;) {
            // do we have a missing child?
            for(int dir = LEFT; dir <= RIGHT; dir++) {
                if((*parent)->child[dir] == 0) {
                    // yes; free this node and promote other kid
                    struct treap *toDelete = *parent;
                    *parent = toDelete->child[!dir];
                    free(toDelete);
                    return;
                }
            }

            // no missing child, have to rotate down
            int biggerKid;
            if((*parent)->child[LEFT]->heapKey > (*parent)->child[RIGHT]->heapKey) {
                biggerKid = LEFT;
            } else {
                biggerKid = RIGHT;
            }

            // rotate up the bigger kid
            treapRotateUp(parent, biggerKid);

            // node to delete is now on opposite side
            parent = &(*parent)->child[!biggerKid];
        }
    } else {
        treapDelete(&(*parent)->child[key < (*parent)->key ? LEFT : RIGHT], key);
    }
}

#define TEST_THRESHOLD (10)

int
```

```
main(int argc, char **argv)
{
    if(argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        return 1;
    }

    struct treap *t = 0;
    int key;

    while(scanf("%d", &key) == 1) {
        if(key >= 0) {
            treapInsert(&t, key);
        } else {
            treapDelete(&t, -key);
        }

        treapPrint(t);
        printf("--- largest <= %d is %d\n", TEST_THRESHOLD, treapSearchMaxLE(t, TEST_THRESHO
    }

    treapDestroy(&t);

    return 0;
}
```

examples/trees/treap/treap.c

5.12.3 AVL trees

AVL trees solve the balancing problem by enforcing the invariant that the heights of the two subtrees sitting under each node differ by at most one. This does not guarantee perfect balance, but it does get close. Let $S(k)$ be the size of the smallest AVL tree with height k. This tree will have at least one subtree of height $k-1$, but its other subtree can be of height $k-2$ (and should be, to keep it as small as possible). We thus have the recurrence $S(k) = 1 + S(k-1) + S(k-2)$, which is very close to the Fibonacci recurrence.

It is possible to solve this exactly using generating functions. But we can get close by guessing that $S(k) \geq a^k$ for some constant a. This clearly works for $S(0) = a^0 = 1$. For larger k, compute

- $S(k) = 1 + a^{k-1} + a^{k-2} = 1 + a^k(1/a + 1/a^2) > a^k(1/a + 1/a^2)$.

This last quantity is at least a^k provided $(1/a + 1/a^2)$ is at least 1. We can solve exactly for the largest a that makes this work, but a very quick calculation shows that $a = 3/2$ works: $2/3 + 4/9 = 10/9 > 1$. It follows that any AVL tree with height k has at least $(3/2)^k$ nodes, or conversely that any AVL tree with $(3/2)^k$ nodes has height at most k. So the height of an arbitrary AVL tree with n nodes is no greater than $\log_{3/2} n = O(\log n)$.

How do we maintain this invariant? The first thing to do is add extra information to the tree, so that we can tell when the invariant has been violated. AVL trees store in each node the difference between the heights of its left and right subtrees, which will be one of -1, 0, or 1. In an ideal world this would require $\log_2 3 \approx 1.58$ bits per node, but since fractional bits are difficult to represent on modern computers a typical implementation uses two bits. Inserting a new node into an AVL tree involves

1. Doing a standard binary search tree insertion.
2. Updating the balance fields for every node on the insertion path.
3. Performing a single or double rotation to restore balance if needed.

Implementing this correctly is tricky. Intuitively, we can imagine a version of an AVL tree in which we stored the height of each node (using $O(\log \log n)$ bits). When we insert a new node, only the heights of its ancestors change—so step 2 requires updating $O(\log n)$ height fields. Similarly, it is only these ancestors that can be too tall. It turns out that fixing the closest ancestor fixes all the ones above it (because it shortens their longest paths by one as well). So just one single or double rotation restores balance.

Deletions are also possible, but are uglier: a deletion in an AVL tree may require as many as $O(\log n)$ rotations. The basic idea is to use the standard binary search tree deletion trick of either splicing out a node if it has no right child, or replacing it with the minimum value in its right subtree (the node for which is spliced out); we then have to check to see if we need to rebalance at every node above whatever node we removed.

Which rotations we need to do to rebalance depends on how some pair of siblings are unbalanced. Below, we show the possible cases.

Zig-zig case: This can occur after inserting in A or deleting in C. Here we rotate A up:

```
    y             x
   / \   ===>    / \
  x   C         A   y
 / \            |  / \
A   B           # B   C
|
#
```

Zig-zag case: This can occur after inserting in B or deleting in C. This requires a double rotation.

```
    z              z                y
   / \   ===>     / \     ===>     / \
```

```
  x   C          y   C              x   z
 / \            / \                /|   |\
A   y          x  B2              A B1 B2 C
   / \        / \
  B1 B2      A   B1
```

Zig-zag case, again: This last case comes up after deletion if both nephews of the short node are too tall. The same double rotation we used in the previous case works here, too. Note that one of the subtrees is still one taller than the others, but that's OK.

```
    z               z                  y
   / \   ===>      / \     ===>       / \
  x   C          y   C              x   z
 / \            / \                /|   |\
A   y          x  B2              A B1 B2 C
|  / \        / \                 |
# B1 B2      A   B1               #
             |
             #
```

5.12.3.1 Sample implementation If we are not fanatical about space optimization, we can just keep track of the heights of all nodes explicitly, instead of managing the $-1, 0, 1$ balance values. Below, we give a not-very-optimized example implementation that uses this approach to store a set of `int`s. This is pretty much our standard unbalanced BST (although we have to make sure that the insert and delete routines are recursive, so that we can fix things up on the way back out), with a layer on top, implemented in the `treeFix` function, that tracks the height and size of each subtree (although we don't use size), and another layer on top of that, implemented in the `treeBalance` function, that fixes any violations of the AVL balance rule.

```c
/*
 * Basic binary search tree data structure without balancing info.
 *
 * Convention:
 *
 * Operations that update a tree are passed a struct tree **,
 * so they can replace the argument with the return value.
 *
 * Operations that do not update the tree get a const struct tree *.
 */

#define LEFT (0)
#define RIGHT (1)
#define TREE_NUM_CHILDREN (2)
```

```c
struct tree {
    /* we'll make this an array so that we can make some operations symmetric */
    struct tree *child[TREE_NUM_CHILDREN];
    int key;
    int height;    /* height of this node */
    size_t size;   /* size of subtree rooted at this node */
};

#define TREE_EMPTY (0)
#define TREE_EMPTY_HEIGHT (-1)

/* free all elements of a tree, replacing it with TREE_EMPTY */
void treeDestroy(struct tree **root);

/* insert an element into a tree pointed to by root */
void treeInsert(struct tree **root, int newElement);

/* return 1 if target is in tree, 0 otherwise */
/* we allow root to be modified to allow for self-balancing trees */
int treeContains(const struct tree *root, int target);

/* delete minimum element from the tree and return its key */
/* do not call this on an empty tree */
int treeDeleteMin(struct tree **root);

/* delete target from the tree */
/* has no effect if target is not in tree */
void treeDelete(struct tree **root, int target);

/* return height of tree */
int treeHeight(const struct tree *root);

/* return size of tree */
size_t treeSize(const struct tree *root);

/* pretty-print the contents of a tree */
void treePrint(const struct tree *root);

/* return the number of elements in tree less than target */
size_t treeRank(const struct tree *root, int target);

/* return an element with the given rank */
/* rank must be less than treeSize(root) */
int treeUnrank(const struct tree *root, size_t rank);
```

```
/* check that aggregate data is correct throughout the tree */
void treeSanityCheck(const struct tree *root);
```

examples/trees/AVL/tree.h

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

#include "tree.h"

int
treeHeight(const struct tree *root)
{
    if(root == 0) {
        return TREE_EMPTY_HEIGHT;
    } else {
        return root->height;
    }
}

/* recompute height from height of kids */
static int
treeComputeHeight(const struct tree *root)
{
    int childHeight;
    int maxChildHeight;
    int i;

    if(root == 0) {
        return TREE_EMPTY_HEIGHT;
    } else {
        maxChildHeight = TREE_EMPTY_HEIGHT;

        for(i = 0; i < TREE_NUM_CHILDREN; i++) {
            childHeight = treeHeight(root->child[i]);
            if(childHeight > maxChildHeight) {
                maxChildHeight = childHeight;
            }
        }

        return maxChildHeight + 1;
    }
}
```

```c
size_t
treeSize(const struct tree *root)
{
    if(root == 0) {
        return 0;
    } else {
        return root->size;
    }
}

/* recompute size from size of kids */
static int
treeComputeSize(const struct tree *root)
{
    int size;
    int i;

    if(root == 0) {
        return 0;
    } else {
        size = 1;

        for(i = 0; i < TREE_NUM_CHILDREN; i++) {
            size += treeSize(root->child[i]);
        }

        return size;
    }
}

/* fix aggregate data in root */
/* assumes children are correct */
static void
treeAggregateFix(struct tree *root)
{
    if(root) {
        root->height = treeComputeHeight(root);
        root->size = treeComputeSize(root);
    }
}

/* rotate child in given direction to root */
static void
treeRotate(struct tree **root, int direction)
{
    struct tree *x;
```

```
    struct tree *y;
    struct tree *b;

    /*
     *      y           x
     *     / \         / \
     *    x   C  <=>  A   y
     *   / \             / \
     *  A   B           B   C
     */

    y = *root;                  assert(y);
    x = y->child[direction];    assert(x);
    b = x->child[!direction];

    /* do the rotation */
    *root = x;
    x->child[!direction] = y;
    y->child[direction] = b;

    /* fix aggregate data for y then x */
    treeAggregateFix(y);
    treeAggregateFix(x);
}

/* restore AVL property at *root after an insertion or deletion */
/* assumes subtrees already have AVL property */
static void
treeRebalance(struct tree **root)
{
    int tallerChild;

    if(*root) {
        for(tallerChild = 0; tallerChild < TREE_NUM_CHILDREN; tallerChild++) {
            if(treeHeight((*root)->child[tallerChild]) >= treeHeight((*root)->child[!tallerC

                /* which grandchild is the problem? */
                if(treeHeight((*root)->child[tallerChild]->child[!tallerChild])
                            > treeHeight((*root)->child[tallerChild]->child[tallerChild])) {
                    /* opposite-direction grandchild is too tall */
                    /* rotation at root will just change its parent but not change height */
                    /* so we rotate it up first */
                    treeRotate(&(*root)->child[tallerChild], !tallerChild);
                }

                /* now rotate up the taller child */
```

```
                treeRotate(root, tallerChild);

                /* don't bother with other child */
                break;
            }
        }

        /* test that we actually fixed it */
        assert(abs(treeHeight((*root)->child[LEFT]) - treeHeight((*root)->child[RIGHT])) <=

#ifdef PARANOID_REBALANCE
        treeSanityCheck(*root);
#endif
    }
}


/* free all elements of a tree, replacing it with TREE_EMPTY */
void
treeDestroy(struct tree **root)
{
    int i;

    if(*root) {
        for(i = 0; i < TREE_NUM_CHILDREN; i++) {
            treeDestroy(&(*root)->child[i]);
        }
        free(*root);
        *root = TREE_EMPTY;
    }
}



/* insert an element into a tree pointed to by root */
void
treeInsert(struct tree **root, int newElement)
{
    struct tree *e;

    if(*root == 0) {
        /* not already there, put it in */

        e = malloc(sizeof(*e));
        assert(e);
```

```c
        e->key = newElement;
        e->child[LEFT] = e->child[RIGHT] = 0;

        *root = e;
    } else if((*root)->key == newElement) {
        /* already there, do nothing */
        return;
    } else {
        /* do this recursively so we can fix data on the way back out */
        treeInsert(&(*root)->child[(*root)->key < newElement], newElement);
    }

    /* fix the aggregate data */
    treeAggregateFix(*root);
    treeRebalance(root);
}

/* return 1 if target is in tree, 0 otherwise */
int
treeContains(const struct tree *t, int target)
{
    while(t && t->key != target) {
        t = t->child[t->key < target];
    }

    return t != 0;
}

/* delete minimum element from the tree and return its key */
/* do not call this on an empty tree */
int
treeDeleteMin(struct tree **root)
{
    struct tree *toFree;
    int retval;

    assert(*root);  /* can't delete min from empty tree */

    if((*root)->child[LEFT]) {
        /* recurse on left subtree */
        retval = treeDeleteMin(&(*root)->child[LEFT]);
    } else {
        /* delete the root */
        toFree = *root;
        retval = toFree->key;
        *root = toFree->child[RIGHT];
```

```
        free(toFree);
    }

    /* fix the aggregate data */
    treeAggregateFix(*root);
    treeRebalance(root);

    return retval;
}

/* delete target from the tree */
/* has no effect if target is not in tree */
void
treeDelete(struct tree **root, int target)
{
    struct tree *toFree;

    /* do nothing if target not in tree */
    if(*root) {
        if((*root)->key == target) {
            if((*root)->child[RIGHT]) {
                /* replace root with min value in right subtree */
                (*root)->key = treeDeleteMin(&(*root)->child[RIGHT]);
            } else {
                /* patch out root */
                toFree = *root;
                *root = toFree->child[LEFT];
                free(toFree);
            }
        } else {
            treeDelete(&(*root)->child[(*root)->key < target], target);
        }

        /* fix the aggregate data */
        treeAggregateFix(*root);
        treeRebalance(root);
    }
}


/* how far to indent each level of the tree */
#define INDENTATION_LEVEL (2)

/* print contents of a tree, indented by depth */
static void
treePrintIndented(const struct tree *root, int depth)
```

```
{
    int i;

    if(root != 0) {
        treePrintIndented(root->child[LEFT], depth+1);

        for(i = 0; i < INDENTATION_LEVEL*depth; i++) {
            putchar(' ');
        }
        printf("%d Height: %d Size: %zu (%p)\n", root->key, root->height, root->size, (void

        treePrintIndented(root->child[RIGHT], depth+1);
    }
}

/* print the contents of a tree */
void
treePrint(const struct tree *root)
{
    treePrintIndented(root, 0);
}

size_t
treeRank(const struct tree *t, int target)
{
    size_t rank = 0;

    while(t && t->key != target) {
        if(t->key < target) {
            /* go right */
            /* root and left subtree are all less than target */
            rank += (1 + treeSize(t->child[LEFT]));
            t = t->child[RIGHT];
        } else {
            /* go left */
            t = t->child[LEFT];
        }
    }

    /* we must also count left subtree */
    return rank + treeSize(t->child[LEFT]);
}

int
treeUnrank(const struct tree *t, size_t rank)
{
```

```
    size_t leftSize;

    /* basic idea: if rank < treeSize(child[LEFT]), recurse in left child */
    /* if it's equal, return the root */
    /* else recurse in right child with rank = rank - treeSize(child[LEFT]) - 1 */
    while(rank != (leftSize = treeSize(t->child[LEFT]))) {
        if(rank < leftSize) {
            t = t->child[LEFT];
        } else {
            t = t->child[RIGHT];
            rank -= (leftSize + 1);
        }
    }

    return t->key;
}

/* check that aggregate data is correct throughout the tree */
void
treeSanityCheck(const struct tree *root)
{
    int i;

    if(root) {
        assert(root->height == treeComputeHeight(root));
        assert(root->size == treeComputeSize(root));

        assert(abs(treeHeight(root->child[LEFT]) - treeHeight(root->child[RIGHT])) < 2);

        for(i = 0; i < TREE_NUM_CHILDREN; i++) {
            treeSanityCheck(root->child[i]);
        }
    }
}

#ifdef TEST_MAIN
int
main(int argc, char **argv)
{
    int key;
    int i;
    const int n = 10;
    const int randRange = 1000;
    const int randTrials = 10000;
    struct tree *root = TREE_EMPTY;
```

```
    if(argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        return 1;
    }

    /* original test */
    for(i = 0; i < n; i++) {
        assert(!treeContains(root, i));
        treeInsert(&root, i);
        assert(treeContains(root, i));
        treeSanityCheck(root);
#ifdef PRINT_AFTER_OPERATIONS
        treePrint(root);
        puts("---");
#endif
    }

    /* check ranks */
    for(i = 0; i < n; i++) {
        assert(treeRank(root, i) == i);
        assert(treeUnrank(root, i) == i);
    }

    treeSanityCheck(root);

    /* now delete everything */
    for(i = 0; i < n; i++) {
        assert(treeContains(root, i));
        treeDelete(&root, i);
        assert(!treeContains(root, i));
        treeSanityCheck(root);
#ifdef PRINT_AFTER_OPERATIONS
        treePrint(root);
        puts("---");
#endif
    }

    treeSanityCheck(root);
    treeDestroy(&root);

    /* random test */
    srand(1);

    for(i = 0; i < randTrials; i++) {
        treeInsert(&root, rand() % randRange);
        treeDelete(&root, rand() % randRange);
```

```
    }

    treeSanityCheck(root);
    treeDestroy(&root);

#ifdef TEST_USE_STDIN
    while(scanf("%d", &key) == 1) {
        /* insert if positive, delete if negative */
        if(key > 0) {
            treeInsert(&root, key);
            assert(treeContains(root, key));
        } else if(key < 0) {
            treeDelete(&root, -key);
            assert(!treeContains(root, key));
        }
        /* else ignore 0 */

#ifdef PRINT_AFTER_OPERATIONS
        treePrint(root);
        puts("---");
#endif
    }

    treeSanityCheck(root);

    treeDestroy(&root);
#endif /* TEST_USE_STDIN */
    return 0;
}
#endif /* TEST_MAIN */
```

examples/trees/AVL/tree.c

This Makefile will compile and run some demo code in `tree.c` if run with `make test`.

(An older implementation can be found in the directory examples/trees/oldAvlTree.

5.12.4 2–3 trees

An early branch in the evolution of balanced trees was the 2–3 tree. Here all paths have the same length, but internal nodes have either 2 or 3 children. So a 2–3 tree with height k has between 2^k and 3^k leaves and a comparable number of internal nodes. The maximum path length in a tree with n nodes is at most $\lceil \lg n \rceil$, as in a perfectly balanced binary tree.

An internal node in a 2–3 tree holds one key if it has two children (including two nil pointers) and two if it has three children. A search that reaches a three-child node must compare the target with both keys to decide which of the three subtrees to recurse into. As in binary trees, these comparisons take constant time, so we can search a 2–3 tree in $O(\log n)$ time.

Insertion is done by expanding leaf nodes. This may cause a leaf to split when it acquires a third key. When a leaf splits, it becomes two one-key nodes and the middle key moves up into its parent. This may cause further splits up the ancestor chain; the tree grows in height by adding a new root when the old root splits. In practice only a small number of splits are needed for most insertions, but even in the worst case this entire process takes $O(\log n)$ time.

It follows that 2–3 trees have the same performance as AVL trees. Conceptually, they are simpler, but having to write separate cases for 2-child and 3-child nodes doubles the size of most code that works on 2–3 trees. The real significance of 2–3 trees is as a precursor to two other kinds of trees, the *red-black tree* and the *B-tree*.

5.12.5 Red-black trees

A red-black tree is a 2–3–4 tree (i.e. all nodes have 2, 3, or 4 children and 1, 2, or 3 internal keys) where each node is represented by a little binary tree with a black root and zero, one, or two red extender nodes as follows:

The invariant for a red-black tree is that

1. No two red nodes are adjacent.
2. Every path contains the same number of black nodes.

For technical reasons, we include the null pointers at the bottom of the tree as black nodes; this has no effect on the invariant, but simplifies the description of the rebalancing procedure.

From the invariant it follows that every path has between k and $2k$ nodes, where k is the *black-height*, the common number of black nodes on each path. From this we can prove that the height of the tree is $O(\log n)$.

Searching in a red-black tree is identical to searching in any other binary search tree; we simply ignore the color bit on each node. So search takes $O(\log n)$ time. For insertions, we use the standard binary search tree insertion algorithm, and insert the new node as a red node. This may violate the first part of the invariant (it doesn't violate the second because it doesn't change the number of black nodes on any path). In this case we need to fix up the constraint by recoloring nodes and possibly performing a single or double rotation.

Which operations we need to do depend on the color of the new node's uncle. If the uncle is red, we can recolor the node's parent, uncle, and grandparent and get rid of the double-red edge between the new node and its parent without

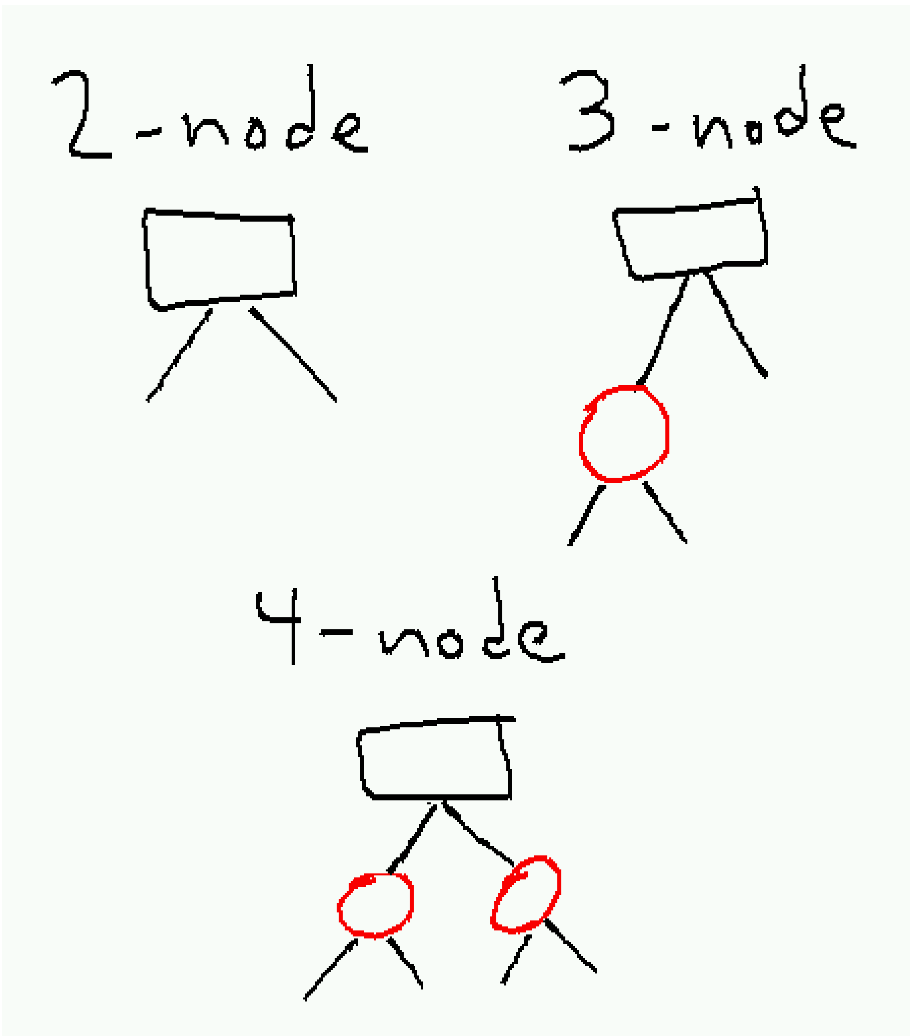


Figure 1: redblacknodes.png

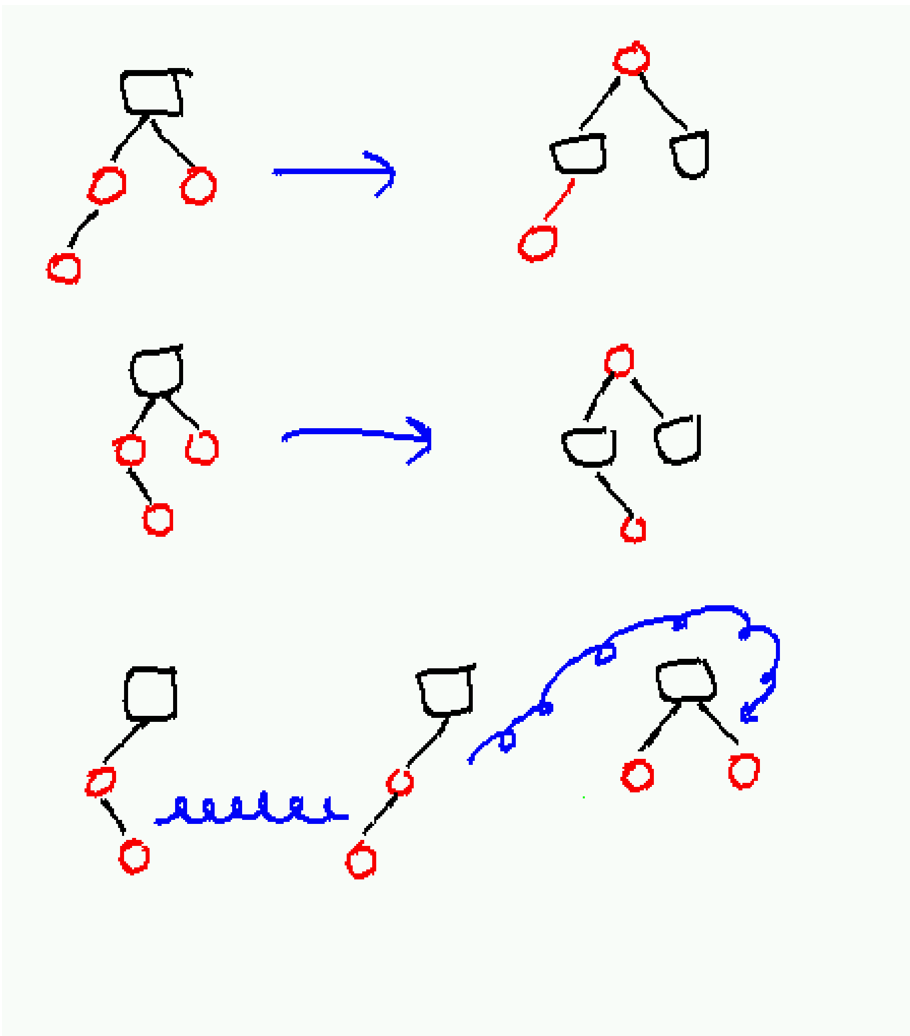


Figure 2: redblackrebalance.png

changing the number of black nodes on any path. In this case, the grandparent becomes red, which may create a new double-red edge which must be fixed recursively. Thus up to $O(\log n)$ such recolorings may occur at a total cost of $O(\log n)$.

If the uncle is black (which includes the case where the uncle is a null pointer), a rotation (possibly a double rotation) and recoloring is necessary. In this case (depicted at the bottom of the picture above), the new grandparent is always black, so there are no more double-red edges. So at most two rotations occur after any insertion.

Deletion is more complicated but can also be done in $O(\log n)$ recolorings and $O(1)$ (in this case up to 3) rotations. Because deletion is simpler in red-black trees than in AVL trees, and because operations on red-black trees tend to have slightly smaller constants than corresponding operation on AVL trees, red-black trees are more often used that AVL trees in practice.

5.12.6 B-trees

Neither is used as much as a B-tree, a specialized data structure optimized for storage systems where the cost of reading or writing a large block (of typically 4096 or 8192 bytes) is no greater than the cost of reading or writing a single bit. Such systems include typical disk drives, where the disk drive has to spend so long finding data on disk that it tries to amortize the huge (tens of millions of CPU clock cycles) seek cost over many returned bytes.

A B-tree is a generalization of a 2–3 tree where each node has between $M/2$ and $M-1$ children, where M is some large constant chosen so that a node (including up to $M-1$ pointers and up to $M-2$ keys) will just fit inside a single block. When a node would otherwise end up with M children, it splits into two nodes with $M/2$ children each, and moves its middle key up into its parent. As in 2–3 trees this may eventually require the root to split and a new root to be created; in practice, M is often large enough that a small fixed height is enough to span as much data as the storage system is capable of holding.

Searches in B-trees require looking through $\log_M n$ nodes, at a cost of $O(M)$ time per node. If M is a constant the total time is asymptotically $O(\log n)$. But the reason for using B-trees is that the $O(M)$ cost of reading a block is trivial compare to the much larger constant time to find the block on the disk; and so it is better to minimize the number of disk accesses (by making M large) than reduce the CPU time.

5.12.7 Splay trees

Yet another approach to balancing is to do it dynamically. Splay trees, described by Sleator and Tarjan in the paper "Self-adjusting binary search trees" (JACM

32(3):652–686, July 1985) are binary search trees in which every search operation rotates the target to the root. If this is done correctly, the **amortized cost** of each tree operation is $O(\log n)$, although particular rare operations might take as much as $O(n)$ time. Splay trees require no extra space because they store no balancing information; however, the constant factors on searches can be larger because every search requires restructuring the tree. For some applications this additional cost is balanced by the splay tree's ability to adapt to data access patterns; if some elements of the tree are hit more often than others, these elements will tend to migrate to the top, and the cost of a typical search will drop to $O(\log m)$, where m is the size of the "working set" of frequently-accessed elements.

5.12.7.1 How splaying works The basic idea of a splay operation is that we move some particular node to the root of the tree, using a sequence of rotations that tends to fix the balance of the tree if the node starts out very deep. So while we might occasionally drive the tree into a state that is highly unbalanced, as soon as we try to exploit this by searching for a deep node, we'll start balancing the tree so that we can't collect too much additional cost. In fact, in order to set up the bad state in the first place we will have to do a lot of cheap splaying operations: the missing cost of these cheap splays ends up paying for the cost of the later expensive search.

Splaying a node to the root involves performing rotations two layers at a time. There are two main cases, depending on whether the node's parent and grandparent are in the same direction (zig-zig) or in opposite directions (zig-zag), plus a third case when the node is only one step away from the root. At each step, we pick one of these cases and apply it, until the target node reaches the root of the tree.

This is probably best understood by looking at a figure from the original paper:

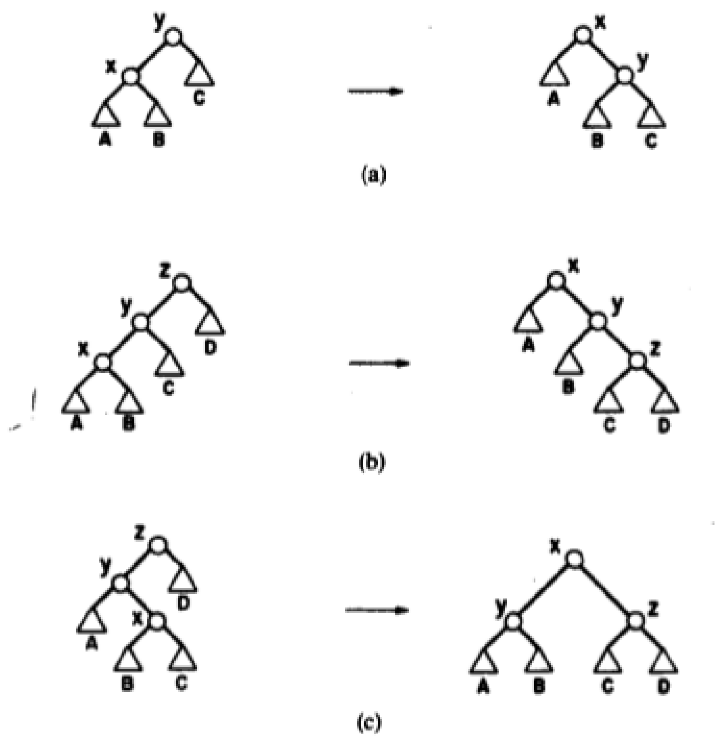


FIG. 3. A splaying step. The node accessed is x. Each case has a symmetric variant (not shown). (a) Zig: terminating single rotation. (b) Zig–zig: two single rotations. (c) Zig–zag: double rotation.

The bottom two cases are the ones we will do most of the time.

Just looking at the picture, it doesn't seem like zig-zig will improve balance much. But if we have a long path made up of zig-zig cases, each operation will push at least one node off of this path, cutting the length of the path in half. So the rebalancing happens as much because we are pushing nodes off of the long path as because the specific rotation operations improve things locally.

5.12.7.2 Analysis Sleator and Tarjan show that any sequence of m splay operations on an n-node splay tree has total cost at most $O((m+n)\log n + m)$. For large m (at least linear in n), the $O(m \log n)$ term dominates, giving an amortized cost per operation of $O(\log n)$, the same as we get from any balanced binary tree. This immediately gives a bound on search costs, because the cost of plunging down the tree to find the node we are looking for is proportional to the cost of splaying it up to the root.

Splay trees have a useful "caching" property in that they pull frequently-accessed nodes to the to the top and push less-frequently-accessed nodes down. The authors show that if only k of the n nodes are accessed, the long-run amortized cost per search drops to $O(\log k)$. For more general access sequences, it is conjectured that the cost to perform a sufficiently long sequence of searches using a splay tree is in fact optimal up to a constant factor (the "dynamic optimality conjecture"), but no one has yet been able to prove this conjecture (or provide a counterexample).[21]

5.12.7.3 Other operations A search operation consists of a standard binary tree search followed by splaying the target node to the root (if present) or the last non-null node we reached to the root instead (if not).

Insertion and deletion are built on top of procedures to split and join trees.

A split divides a single splay tree into two splay trees, consisting of all elements less than or equal to some value x and all elements greater than x. This is done by searching for x, which brings either x or the first element less than or greater than x to the root, then breaking the link between the root and its left or right child depending on whether the root should go in the right or left tree.

A join merges two splay trees L and R, where every element in L is less than every element in R. This involves splaying the largest element in L to the root, and then making the root of R the right child of this element.

To do an insert of x, we do a split around x, then make the roots of the two trees the children of a new element holding x (unless x is already present in the tree, in which case we stop before breaking the trees apart).

To do a delete of an element x, we splay x to the root, remove it, then join the two orphaned subtrees.

For each operation, we are doing a constant number of splays (amortized cost $O(\log n)$ each), plus $O(1)$ additional work. A bit of work is needed to ensure that the joins and splits don't break the amortized cost analysis, but this is done in the paper, so we will sweep it under the carpet with the rest of the analysis.

5.12.7.4 Top-down splaying There are a few remaining details that we need to deal with before trying to implement a splay trees. Because the splay tree could become very deep, we probably don't want to implement a splay recursively in a language like C, because we'll blow out our stack. We also have a problem if we are trying to rotate our target up from the bottom of figuring out what its ancestors are. We could solve both of these problems by including parent pointers in our tree, but this would add a lot of complexity and negate the space improvement over AVL trees of not having to store heights.

[21] A summary of the state of this problem as of 2013 can be found in http://arxiv.org/pdf/1306.0207v1.pdf.

The solution given in the Sleator-Tarjan paper is to replace the bottom-up splay procedure with a top-down splay procedure that accomplishes the same task. The idea is that rotating a node up from the bottom effectively splits the tree above it into two new left and right subtrees by pushing ancestors sideways according to the zig-zig and zig-zag patters. But we can recognize these zig-zig and zig-zag patterns from the top as well, and so we can construct these same left and right subtrees from the top down instead of the bottom up. When we do this, instead of adding new nodes to the tops of the trees, we will be adding new nodes to the bottoms, as the right child of the rightmost node in the left tree or the left child of the rightmost node in the left tree.

Here's the picture, from the original paper:

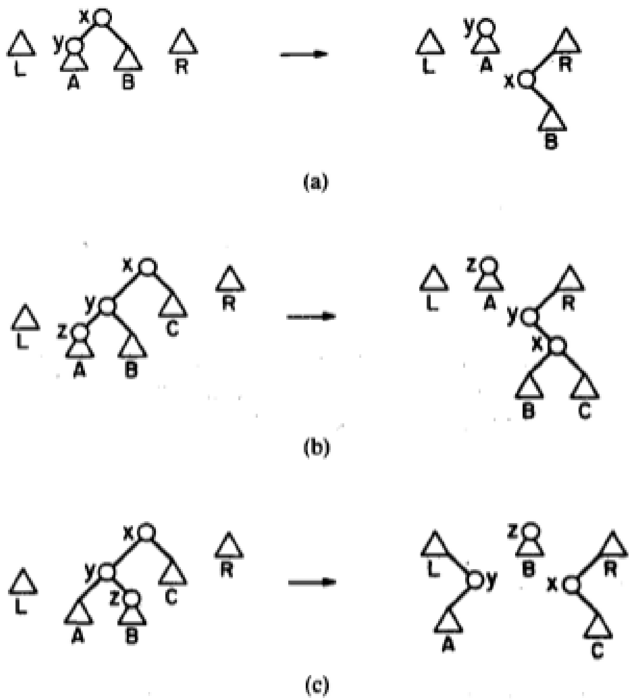


FIG. 11. Top-down splaying step. Symmetric variants of cases are omitted. Initial left tree is L, right tree is R. Labeled nodes are on the access path. (a) Zig: Node y contains the accessed item. (b) Zig–zig. (c) Zig–zag.

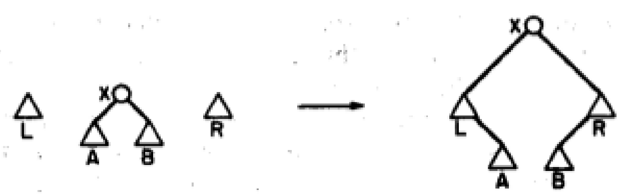


FIG. 12. Completion of top-down splaying. Node x contains the accessed item.

To implement this, we need to keep track of the roots of the three trees, as well as the locations in the left and right trees where we will be adding new vertices. The roots we can just keep pointers to. For the lower corners of the trees, it makes sense to store instead a pointer to the pointer location, so that we can modify the pointer in the tree (and then move the pointer to point to the pointer in the new corner). Initially, these corner pointers will just point to the left and right tree roots, which will start out empty.

The last step (shown as Figure 12 from the paper) pastes the tree back together by inserting the left and right trees between the new root and its children.

5.12.7.5 An implementation

Here is an implementation of a splay tree, with an interface similar to the previous AVL tree implementation.

```
/*
 * Basic binary search tree data structure without balancing info.
 *
 * Convention:
 *
 * Operations that update a tree are passed a struct tree **,
 * so they can replace the argument with the return value.
 *
 * Operations that do not update the tree get a const struct tree *.
 */

#define LEFT (0)
#define RIGHT (1)
#define TREE_NUM_CHILDREN (2)

struct tree {
    /* we'll make this an array so that we can make some operations symmetric */
    struct tree *child[TREE_NUM_CHILDREN];
    int key;
};

#define TREE_EMPTY (0)

/* free all elements of a tree, replacing it with TREE_EMPTY */
void treeDestroy(struct tree **root);

/* insert an element into a tree pointed to by root */
void treeInsert(struct tree **root, int newElement);

/* return 1 if target is in tree, 0 otherwise */
/* we allow root to be modified to allow for self-balancing trees */
int treeContains(struct tree **root, int target);

/* delete target from the tree */
/* has no effect if target is not in tree */
void treeDelete(struct tree **root, int target);

/* pretty-print the contents of a tree */
void treePrint(const struct tree *root);
```

examples/trees/splay/tree.h

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

#include "tree.h"

/* free all elements of a tree, replacing it with TREE_EMPTY */
void
treeDestroy(struct tree **root)
{
    /* we want to avoid doing this recursively, because the tree might be deep */
    /* so we will repeatedly delete the root until the tree is empty */
    while(*root) {
        treeDelete(root, (*root)->key);
    }
}

/* rotate child in given direction to root */
void
treeRotate(struct tree **root, int direction)
{
    struct tree *x;
    struct tree *y;
    struct tree *b;

    /*
     *      y           x
     *     / \         / \
     *    x   C  <=>  A   y
     *   / \             / \
     *  A   B           B   C
     */

    y = *root;                  assert(y);
    x = y->child[direction];    assert(x);
    b = x->child[!direction];

    /* do the rotation */
    *root = x;
    x->child[!direction] = y;
    y->child[direction] = b;
}
```

```
/* link operations for top-down splay */
/* this pastes a node in as !d-most node in subtree on side d */
static inline void
treeLink(struct tree ***hook, int d, struct tree *node)
{
    *hook[d] = node;
    /* strictly speaking we don't need to do this, but it allows printing the partial trees
    node->child[!d] = 0;
    hook[d] = &node->child[!d];
}

/* splay last element on path to target to root */
static void
treeSplay(struct tree **root, int target)
{
    struct tree *t;
    struct tree *child;
    struct tree *grandchild;
    struct tree *top[TREE_NUM_CHILDREN];   /* accumulator trees that will become subtrees o
    struct tree **hook[TREE_NUM_CHILDREN]; /* where to link new elements into accumulator t
    int d;
    int dChild;        /* direction of child */
    int dGrandchild;   /* direction of grandchild */

    /* we don't need to keep following this pointer, we'll just fix it at the end */
    t = *root;

    /* don't do anything to an empty tree */
    if(t == 0) { return; }

    /* ok, tree is not empty, start chopping it up */
    for(d = 0; d < TREE_NUM_CHILDREN; d++) {
        top[d] = 0;
        hook[d] = &top[d];
    }

    /* keep going until we hit the key or we would hit a null pointer in the child */
    while(t->key != target && (child = t->child[dChild = t->key < target]) != 0) {
        /* child is not null */
        grandchild = child->child[dGrandchild = child->key < target];

#ifdef DEBUG_SPLAY
        treePrint(top[0]);
        puts("---");
        treePrint(t);
        puts("---");
```

```
        treePrint(top[1]);
        puts("===");
#endif

        if(grandchild == 0 || child->key == target) {
            /* zig case; paste root into opposite-side hook */
            treeLink(hook, !dChild, t);
            t = child;
            /* we can break because we know we will hit child == 0 next */
            break;
        } else if(dChild == dGrandchild) {
            /* zig-zig case */
            /* rotate and then hook up child */
            /* grandChild becomes new root */
            treeRotate(&t, dChild);
            treeLink(hook, !dChild, child);
            t = grandchild;
        } else {
            /* zig-zag case */
            /* root goes to !dChild, child goes to dChild, grandchild goes to root */
            treeLink(hook, !dChild, t);
            treeLink(hook, dChild, child);
            t = grandchild;
        }
    }

    /* now reassemble the tree */
    /* t's children go in hooks, top nodes become t's new children */
    for(d = 0; d < TREE_NUM_CHILDREN; d++) {
        *hook[d] = t->child[d];
        t->child[d] = top[d];
    }

    /* and put t back in *root */
    *root = t;
}

/* return 1 if target is in tree, 0 otherwise */
int
treeContains(struct tree **root, int target)
{
    treeSplay(root, target);
    return *root != 0 && (*root)->key == target;
}
```

```c
/* insert an element into a tree pointed to by root */
void
treeInsert(struct tree **root, int newElement)
{
    struct tree *e;
    struct tree *t;
    int d;             /* which side of e to put old root on */

    treeSplay(root, newElement);

    t = *root;

    /* skip if already present */
    if(t && t->key == newElement) { return; }

    /* otherwise split the tree */
    e = malloc(sizeof(*e));
    assert(e);

    e->key = newElement;

    if(t == 0) {
        e->child[LEFT] = e->child[RIGHT] = 0;
    } else {
        /* split tree and put e on top */
        /* we know t is closest to e, so we don't have to move anything else */
        d = (*root)->key > newElement;
        e->child[d] = t;
        e->child[!d] = t->child[!d];
        t->child[!d] = 0;
    }

    /* either way we stuff e in *root */
    *root = e;
}

/* delete target from the tree */
/* has no effect if target is not in tree */
void
treeDelete(struct tree **root, int target)
{
    struct tree *left;
    struct tree *right;

    treeSplay(root, target);
```

```
    if(*root && (*root)->key == target) {
        /* save pointers to kids */
        left = (*root)->child[LEFT];
        right = (*root)->child[RIGHT];

        /* free the old root */
        free(*root);

        /* if left is empty, just return right */
        if(left == 0) {
            *root = right;
        } else {
            /* first splay max element in left to top */
            treeSplay(&left, INT_MAX);

            /* now paste in right subtree */
            left->child[RIGHT] = right;

            /* return left */
            *root = left;
        }
    }
}

/* how far to indent each level of the tree */
#define INDENTATION_LEVEL (2)

/* print contents of a tree, indented by depth */
static void
treePrintIndented(const struct tree *root, int depth)
{
    int i;

    if(root != 0) {
        treePrintIndented(root->child[LEFT], depth+1);

        for(i = 0; i < INDENTATION_LEVEL*depth; i++) {
            putchar(' ');
        }
        printf("%d (%p)\n", root->key, (void *) root);

        treePrintIndented(root->child[RIGHT], depth+1);
    }
}

/* print the contents of a tree */
```

```c
void
treePrint(const struct tree *root)
{
    treePrintIndented(root, 0);
}


#ifdef TEST_MAIN
int
main(int argc, char **argv)
{
    int i;
    const int n = 10;
    struct tree *root = TREE_EMPTY;

    if(argc != 1) {
        fprintf(stderr, "Usage: %s\n", argv[0]);
        return 1;
    }

    for(i = 0; i < n; i++) {
        assert(!treeContains(&root, i));
        treeInsert(&root, i);
        assert(treeContains(&root, i));
        treePrint(root);
        puts("===");
    }

    /* now delete everything */
    for(i = 0; i < n; i++) {
        assert(treeContains(&root, i));
        treeDelete(&root, i);
        assert(!treeContains(&root, i));
        treePrint(root);
        puts("===");
    }

    treeDestroy(&root);

    return 0;
}
#endif
```

examples/trees/splay/tree.c

Makefile. The file speedTest.c can be used to do a simple test of the efficiency of inserting many random elements. On my machine, the splay tree version is about 10% slower than the AVL tree for this test on a million elements. This probably indicates a bigger slowdown for `treeInsert` itself, because some of the time will be spent in `rand` and `treeDestroy`, but I was too lazy to actually test this further.

5.12.7.6 More information

For more details on splay trees, see the original paper, or any number of demos, animations, and other descriptions that can be found via Google.

5.12.8 Scapegoat trees

Scapegoat trees are another amortized balanced tree data structure. The idea of a scapegoat tree is that if we ever find ourselves doing an insert at the end of a path that is too long, we can find some subtree rooted at a node along this path that is particularly imbalanced and rebalance it all at once at a cost of $O(k)$ where k is the size of the subtree. These were shown by Galperin and Rivest (SODA 1993) to give $O(\log n)$ amortized cost for inserts, while guaranteeing $O(\log n)$ depth, so that inserts also run in $O(\log n)$ worst-case time; they also came up with the name "scapegoat tree", although it turns out the same data structure had previously been published by Andersson in 1989. Unlike splay trees, scapegoat trees do not require modifying the tree during a search, and unlike AVL trees, scapegoat trees do not require tracking any information in nodes (although they do require tracking the total size of the tree and, to allow for rebalancing after many deletes, the maximum size of the tree since the last time the entire tree was rebalanced).

Unfortunately, scapegoat trees are not very fast, so one is probably better off with an AVL tree.

5.12.9 Skip lists

Skip lists are yet another balanced tree data structure, where the tree is disguised as a tower of linked lists. Since they use randomization for balance, we describe them with other randomized data structures.

5.12.10 Implementations

AVL trees and red-black trees have been implemented for every reasonable programming language you've ever heard of. For C implementations, a good place to start is at http://adtinfo.org/.

5.13 Graphs

These are notes on implementing **graphs** and graph algorithms in C.

5.13.1 Basic definitions

A **graph** consists of a set of **nodes** or **vertices** together with a set of **edges** or **arcs** where each edge joins two vertices. Unless otherwise specified, a graph is **undirected**: each edge is an unordered pair $\{u, v\}$ of vertices, and we don't regard either of the two vertices as having a distinct role from the other. However, it is more common in computing to consider **directed graphs** or **digraphs** in which edges are *ordered* pairs (u, v); here the vertex u is the **source** of the edge and vertex v is the **sink** or **target** of the edge. Directed edges are usually drawn as arrows and undirected edges as curves or line segments. It is always possible to represent an undirected graph as a directed graph where each undirected edge $\{u, v\}$ becomes two oppositely directed edges (u, v) and (v, u).

Here is an example of a small graph, drawn using this file using the `circo` program from the GraphViz library:

Here is a similar directed graph, drawn using this file:

Given an edge (u, v), the vertices u and v are said to be **incident** to the edge and **adjacent** to each other. The number of vertices adjacent to a given vertex u is the **degree** of u; this can be divided into the **out-degree** (number of vertices v such that (u, v) is an edge) and the **in-degree** (number of vertices v such that (v, u) is an edge). A vertex v adjacent to u is called a **neighbor** of u, and (in a directed graph) is a **predecessor** of u if (v, u) is an edge and a **successor** of u if (u, v) is an edge. We will allow a node to be its own predecessor and successor.

5.13.2 Why graphs are useful

Graphs can be used to model any situation where we have things that are related to each other in pairs; for example, all of the following can be represented by graphs:

Family trees Nodes are members, with an edge from each parent to each of their children.

Transportation networks Nodes are airports, intersections, ports, etc. Edges are airline flights, one-way roads, shipping routes, etc.

Assignments Suppose we are assigning classes to classrooms. Let each node be either a class or a classroom, and put an edge from a class to a classroom if the class is assigned to that room. This is an example of a **bipartite graph**, where the nodes can be divided into two sets S and T and all edges go from S to T.

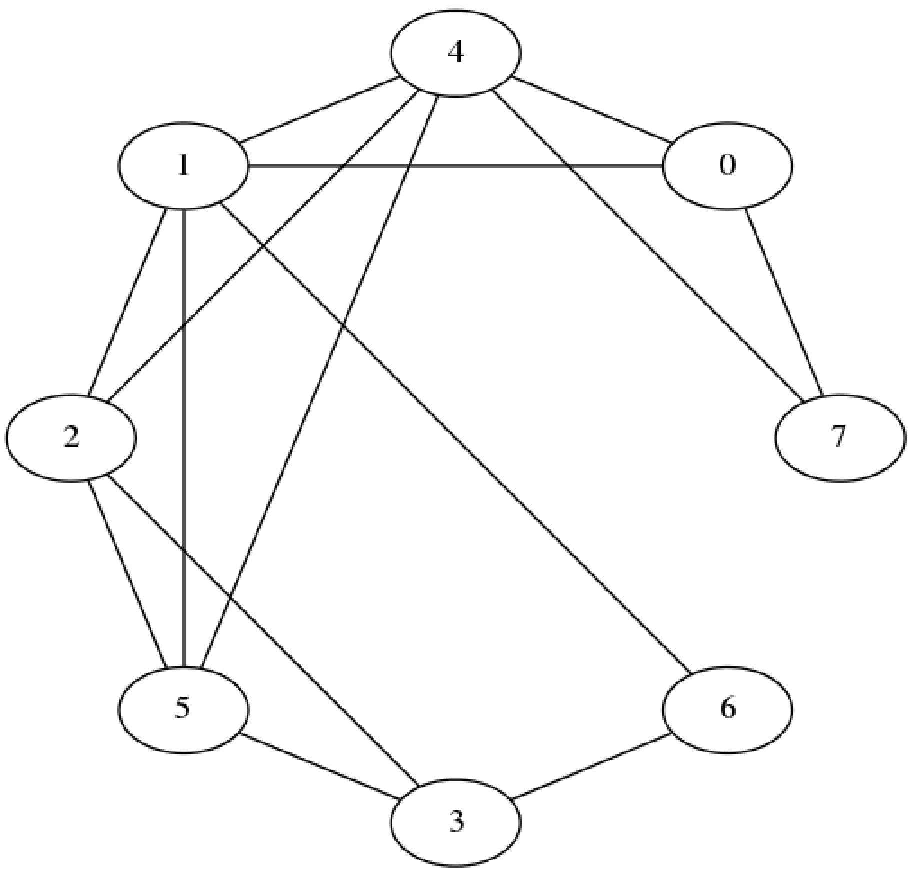


Figure 3: A graph

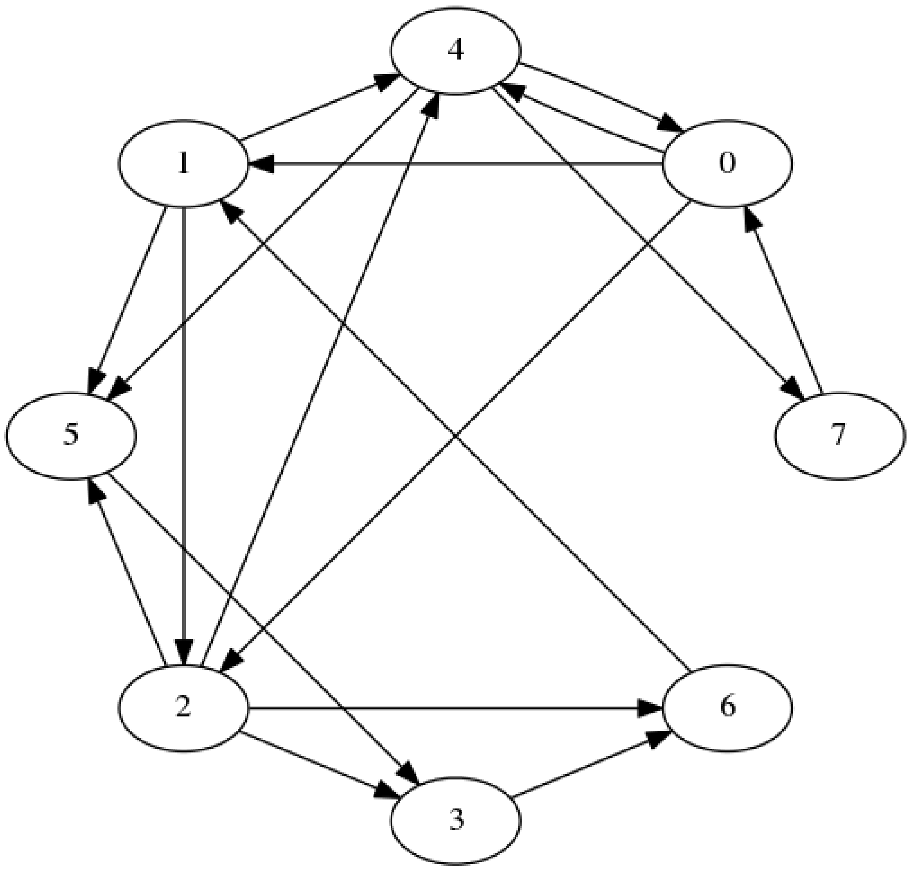


Figure 4: A directed graph

5.13.3 Operations on graphs

What would we like to do to graphs? Generally, we first have to build a graph by starting with a set of nodes and adding in any edges we need, and then we want to extract information from it, such as "Is this graph connected?", "What is the shortest path in this graph from s to t?", or "How many edges can I remove from this graph before some nodes become unreachable from other nodes?" There are standard algorithms for answering all of these questions; the information these algorithms need is typically (a) given a vertex u, what successors does it have; and sometimes (b) given vertices u and v, does the edge (u, v) exist in the graph?

5.13.4 Representations of graphs

A good graph representation will allow us to answer one or both of these questions quickly. There are generally two standard representations of graphs that are used in graph algorithms, depending on which question is more important.

For both representations, we simplify the representation task by insisting that vertices be labeled $0, 1, 2, \ldots, n-1$, where n is the number of vertices in the graph. If we have a graph with different vertex labels (say, airport codes), we can enforce an integer labeling by a preprocessing step where we assign integer labels, and then translate the integer labels back into more useful user labels afterwards. The preprocessing step can usually be done using a hash table in $O(n)$ time, which is likely to be smaller than the cost of whatever algorithm we are running on our graph, and the savings in code complexity and running time from working with just integer labels will pay this cost back many times over.

5.13.4.1 Adjacency matrices

An **adjacency matrix** is just a matrix `a` where `a[i][j]` is `1` if (i,j) is an edge in the graph and `0` otherwise. It's easy to build an adjacency matrix, and adding or testing for the existence of an edges takes $O(1)$ time. The downsides of adjacency matrices are that finding all the outgoing edges from a vertex takes $O(n)$ time even if there aren't very many, and the $O(n^2)$ space cost is high for "sparse graphs," those with much fewer than n^2 edges.

5.13.4.2 Adjacency lists

An **adjacency list** representation of a graph creates a list of successors for each node u. These lists may be represented as linked lists (the typical assumption in algorithms textbooks), or in languages like C may be represented by variable-length arrays. The cost for adding an edge is still $O(1)$, but testing for the existence of an edge (u, v) rises to $O(d^+(u))$, where $d^+(u)$ is the out-degree of u (i.e., the length of the list of u's successors). The cost of enumerating the successors of u is also $O(d^+(u))$, which is clearly the best possible since it takes that long just to write them all down. Finding predecessors of a node u is extremely expensive, requiring looking through every list of every node in time $O(n + m)$, where m is the total number of edges, although if this is something we actually need to do often we can store a second copy of the graph with the edges reversed.

Adjacency lists are thus most useful when we mostly want to enumerate outgoing edges of each node. This is common in search tasks, where we want to find a path from one node to another or compute the distances between pairs of nodes. If other operations are important, we can optimize them by augmenting the adjacency list representation; for example, using sorted arrays for the adjacency lists reduces the cost of edge existence testing to $O(\log(d^+(u)))$, and adding a second copy of the graph with reversed edges lets us find all predecessors of u in $O(d^-(u))$ time, where $d^-(u)$ is u's in-degree.

Adjacency lists also require much less space than adjacency matrices for sparse graphs: $O(n + m)$ vs $O(n^2)$ for adjacency matrices. For this reason adjacency lists are more commonly used than adjacency matrices.

5.13.4.2.1 An implementation

Here is an implementation of a basic graph type using adjacency lists.

```
/* basic directed graph type */

typedef struct graph *Graph;

/* create a new graph with n vertices labeled 0..n-1 and no edges */
Graph graphCreate(int n);

/* free all space used by graph */
void graphDestroy(Graph);

/* add an edge to an existing graph */
/* doing this more than once may have unpredictable results */
void graphAddEdge(Graph, int source, int sink);

/* return the number of vertices/edges in the graph */
int graphVertexCount(Graph);
int graphEdgeCount(Graph);

/* return the out-degree of a vertex */
int graphOutDegree(Graph, int source);

/* return 1 if edge (source, sink) exists), 0 otherwise */
int graphHasEdge(Graph, int source, int sink);

/* invoke f on all edges (u,v) with source u */
```

```
/* supplying data as final parameter to f */
/* no particular order is guaranteed */
void graphForeach(Graph g, int source,
        void (*f)(Graph g, int source, int sink, void *data),
        void *data);
```

examples/graphs/graph.h

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

#include "graph.h"

/* basic directed graph type */
/* the implementation uses adjacency lists
 * represented as variable-length arrays */

/* these arrays may or may not be sorted: if one gets long enough
 * and you call graphHasEdge on its source, it will be */

struct graph {
    int n;              /* number of vertices */
    int m;              /* number of edges */
    struct successors {
        int d;          /* number of successors */
        int len;        /* number of slots in array */
        int isSorted; /* true if list is already sorted */
        int list[];    /* actual list of successors starts here */
    } *alist[];
};

/* create a new graph with n vertices labeled 0..n-1 and no edges */
Graph
graphCreate(int n)
{
    Graph g;
    int i;

    g = malloc(sizeof(struct graph) + sizeof(struct successors *) * n);
    assert(g);

    g->n = n;
    g->m = 0;

    for(i = 0; i < n; i++) {
        g->alist[i] = malloc(sizeof(struct successors));
        assert(g->alist[i]);
```

```
        g->alist[i]->d = 0;
        g->alist[i]->len = 0;
        g->alist[i]->isSorted= 1;
    }

    return g;
}

/* free all space used by graph */
void
graphDestroy(Graph g)
{
    int i;

    for(i = 0; i < g->n; i++) free(g->alist[i]);
    free(g);
}

/* add an edge to an existing graph */
void
graphAddEdge(Graph g, int u, int v)
{
    assert(u >= 0);
    assert(u < g->n);
    assert(v >= 0);
    assert(v < g->n);

    /* do we need to grow the list? */
    while(g->alist[u]->d >= g->alist[u]->len) {
        g->alist[u]->len = g->alist[u]->len * 2 + 1;   /* +1 because it might have been 0 */
        g->alist[u] =
            realloc(g->alist[u],
                sizeof(struct successors) + sizeof(int) * g->alist[u]->len);
    }

    /* now add the new sink */
    g->alist[u]->list[g->alist[u]->d++] = v;
    g->alist[u]->isSorted = 0;

    /* bump edge count */
    g->m++;
}

/* return the number of vertices in the graph */
int
```

```c
graphVertexCount(Graph g)
{
    return g->n;
}

/* return the number of vertices in the graph */
int
graphEdgeCount(Graph g)
{
    return g->m;
}

/* return the out-degree of a vertex */
int
graphOutDegree(Graph g, int source)
{
    assert(source >= 0);
    assert(source < g->n);

    return g->alist[source]->d;
}

/* when we are willing to call bsearch */
#define BSEARCH_THRESHOLD (10)

static int
intcmp(const void *a, const void *b)
{
    return *((const int *) a) - *((const int *) b);
}

/* return 1 if edge (source, sink) exists), 0 otherwise */
int
graphHasEdge(Graph g, int source, int sink)
{
    int i;

    assert(source >= 0);
    assert(source < g->n);
    assert(sink >= 0);
    assert(sink < g->n);

    if(graphOutDegree(g, source) >= BSEARCH_THRESHOLD) {
        /* make sure it is sorted */
        if(! g->alist[source]->isSorted) {
            qsort(g->alist[source]->list,
```

```
                    g->alist[source]->d,
                    sizeof(int),
                    intcmp);
        }

        /* call bsearch to do binary search for us */
        return
            bsearch(&sink,
                    g->alist[source]->list,
                    g->alist[source]->d,
                    sizeof(int),
                    intcmp)
            != 0;
    } else {
        /* just do a simple linear search */
        /* we could call lfind for this, but why bother? */
        for(i = 0; i < g->alist[source]->d; i++) {
            if(g->alist[source]->list[i] == sink) return 1;
        }
        /* else */
        return 0;
    }
}

/* invoke f on all edges (u,v) with source u */
/* supplying data as final parameter to f */
void
graphForeach(Graph g, int source,
    void (*f)(Graph g, int source, int sink, void *data),
    void *data)
{
    int i;

    assert(source >= 0);
    assert(source < g->n);

    for(i = 0; i < g->alist[source]->d; i++) {
        f(g, source, g->alist[source]->list[i], data);
    }
}
```

examples/graphs/graph.c

And here is some test code: graphTest.c.

5.13.4.3 Implicit representations For some graphs, it may not make sense to represent them explicitly. An example might be the word-search graph from CS223/2005/Assignments/HW10, which consists of all words in a dictionary with an edge between any two words that differ only by one letter. In such a case, rather than building an explicit data structure containing all the edges, we might generate edges as needed when computing the neighbors of a particular vertex. This gives us an implicit or procedural representation of a graph.

Implicit representations require the ability to return a vector or list of values from the neighborhood-computing function. There are various way to do this, of which the most sophisticated might be to use an iterator.

5.13.5 Searching for paths in a graph

A **path** is a sequence of vertices $v_1, v_2, \dots v_k$ where each pair (v_i, v_{i+1}) is an edge. Often we want to find a path from a source vertex s to a target vertex t, or more generally to detect which vertices are reachable from a given source vertex s. We can solve these problems by using any of several standard graph search algorithms, of which the simplest and most commonly used are **depth-first search** and **breadth-first search**.

Both of these search algorithms are a special case of a more general algorithm for growing a directed tree in a graph rooted at a given node s. Here we are using *tree* as a graph theorist would, to mean any set of k nodes joined by $k-1$ edges. This is similar to trees used in data structures except that there are no limits on the number of children a node can have and no ordering constraints within the tree.

The general tree-growing algorithm might be described as follows:

1. Start with a tree consisting of just s.
2. If there is at least one edge that leaves the tree (i.e. goes from a node in the current tree to a node outside the current tree), pick the "best" such edge and add it and its sink to the tree.
3. Repeat step 2 until no edges leave the tree.

Practically, steps 2 and 3 are implemented by having some sort of data structure that acts as a bucket for unprocessed edges. When a new node is added to the tree, all of its outgoing edges are thrown into the bucket. The "best" outgoing edge is obtained by applying some sort of pop, dequeue, or delete-min operation to the bucket, depending on which it provides; if this edge turns out to be an internal edge of the tree (maybe we added its sink after putting it in the bucket), we throw it away. Otherwise we mark the edge and its sink as belonging to the tree and repeat.

The output of the generic tree-growing algorithm typically consists of (a) marks on all the nodes that are reachable from s, and (b) for each such node v, a parent pointer back to the source of the edge that brought v into the tree. Often these two values can be combined by using a null parent pointer to represent the absence of a mark (this usually requires making the root point to itself so that we know it's in the tree). Other values that may be useful are a table showing the order in which nodes were added to the tree.

What kind of tree we get depends on what we use for the bucket—specifically, on what edge is returned when we ask for the "best" edge. Two easy cases are:

1. The bucket is a stack. When we ask for an outgoing edge, we get the last edge inserted. This has the effect of running along as far as possible through the graph before backtracking, since we always keep going from the last node if possible. The resulting algorithm is called **depth-first search** and yields a **depth-first search tree**. If we don't care about the lengths of the paths we consider, depth-first search is a perfectly good algorithm for testing connectivity. It can also be implemented without any auxiliary data structures as a recursive procedure, as long as we don't go so deep as to blow out the system stack.
2. The bucket is a queue. Now when we ask for an outgoing edge, we get the first edge inserted. This favors edges that are close to the root: we don't start consider edges from nodes adjacent to the root until we have already added all the root's successors to the tree, and similarly we don't start considering edges at distance k until we have already added all the closer nodes to the tree. This gives **breadth-first search**, which constructs a **shortest-path tree** in which every path from the root to a node in the tree has the minimum length.

Structurally, these algorithms are almost completely identical; indeed, if we organize the stack/queue so that it can pop from both ends, we can switch between depth-first search and breadth-first search just by choosing which end to pop from.

Below, we give a [combined implementation](#combinedDFSBFS} of both depth-first search and breadth-first search that does precisely this, although this is mostly for show. Typical implementations of breadth-first search include a further optimization, where we test an edge to see if we should add it to the tree (and possibly add it) before inserting into the queue. This gives the same result as the DFS-like implementation but only requires $O(n)$ space for the queue instead of $O(m)$, with a smaller constant as well since don't need to bother storing source edges in the queue. An example of this approach is given below.

The running time of any of these algorithms is *very* fast: we pay $O(1)$ per vertex in setup costs and $O(1)$ per edge during the search (assuming the input is in adjacency-list form), giving a linear $O(n + m)$ total cost. Often it is more expensive to set up the graph in the first place than to run a search on it.

5.13.5.1 Implementation of depth-first and breadth-first search

Here is a simple implementation of depth-first search, using a recursive algorithm, and breadth-first search, using an iterative algorithm that maintains a queue of vertices. In both cases the algorithm is applied to a sample graph whose vertices are the integers 0 through $n-1$ for some n, and in which vertex x has edges to vertices $x/2$, $3 \cdot x$, and $x+1$, whenever these values are also integers in the range 0 through $n-1$. For large graphs it may be safer to run an iterative version of DFS that uses an explicit stack instead of a possibly very deep recursion.

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

typedef int Vertex;

#define VERTEX_NULL (-1)

struct node {
    Vertex *neighbors;   /* array of outgoing edges, terminated by VERTEX_NULL */
    Vertex parent;       /* for search */
};

struct graph {
    size_t n;            /* number of vertices */
    struct node *v;      /* list of vertices */
};

void
graphDestroy(struct graph *g)
{
    Vertex v;

    for(v = 0; v < g->n; v++) {
        free(g->v[v].neighbors);
    }

    free(g);
}

/* this graph has edges from x to x+1, x to 3*x, and x to x/2 (when x is even) */
struct graph *
makeSampleGraph(size_t n)
{
    struct graph *g;
    Vertex v;
    const int allocNeighbors = 4;
```

```
    int i;

    g = malloc(sizeof(*g));
    assert(g);

    g->n = n;
    g->v = malloc(sizeof(struct node) * n);
    assert(g->v);

    for(v = 0; v < n; v++) {
        g->v[v].parent = VERTEX_NULL;

        /* fill in neighbors */
        g->v[v].neighbors = malloc(sizeof(Vertex) * allocNeighbors);
        i = 0;
        if(v % 2 == 0) { g->v[v].neighbors[i++] = v/2; }
        if(3*v < n) { g->v[v].neighbors[i++] = 3*v; }
        if(v+1 < n) { g->v[v].neighbors[i++] = v+1; }
        g->v[v].neighbors[i++] = VERTEX_NULL;
    }

    return g;
}

/* output graph in dot format */
void
printGraph(const struct graph *g)
{
    Vertex u;
    size_t i;

    puts("digraph G {");

    for(u = 0; u < g->n; u++) {
        for(i = 0; g->v[u].neighbors[i] != VERTEX_NULL; i++) {
            printf("%d -> %d;\n", u, g->v[u].neighbors[i]);
        }
    }

    puts("}");
}

/* reconstruct path back to root from u */
void
printPath(const struct graph *g, Vertex u)
{
```

```
    do {
        printf(" %d", u);
        u = g->v[u].parent;
    } while(g->v[u].parent != u);
}

/* print the tree in dot format */
void
printTree(const struct graph *g)
{
    Vertex u;

    puts("digraph G {");

    for(u = 0; u < g->n; u++) {
        if(g->v[u].parent != VERTEX_NULL) {
            printf("%d -> %d;\n", u, g->v[u].parent);
        }
    }

    puts("}");
}

/* compute DFS tree starting at root */
/* this uses a recursive algorithm and will not work on large graphs! */
static void
dfsHelper(struct graph *g, Vertex parent, Vertex child)
{
    int i;
    Vertex neighbor;

    if(g->v[child].parent == VERTEX_NULL) {
        g->v[child].parent = parent;
        for(i = 0; (neighbor = g->v[child].neighbors[i]) != VERTEX_NULL; i++) {
            dfsHelper(g, child, neighbor);
        }
    }
}

void
dfs(struct graph *g, Vertex root)
{
    dfsHelper(g, root, root);
}

/* compute BFS tree starting at root */
```

```c
void
bfs(struct graph *g, Vertex root)
{
    Vertex *q;
    int head;   /* deq from here */
    int tail;   /* enq from here */
    Vertex current;
    Vertex nbr;
    int i;

    q = malloc(sizeof(Vertex) * g->n);
    assert(q);

    head = tail = 0;

    /* push root onto q */
    g->v[root].parent = root;
    q[tail++] = root;

    while(head < tail) {
        current = q[head++];

        for(i = 0; (nbr = g->v[current].neighbors[i]) != VERTEX_NULL; i++) {
            if(g->v[nbr].parent == VERTEX_NULL) {
                /* haven't seen this guy */
                /* push it */
                g->v[nbr].parent = current;
                q[tail++] = nbr;
            }
        }
    }

    free(q);
}



int
main(int argc, char **argv)
{
    int n;
    struct graph *g;

    if(argc != 3) {
        fprintf(stderr, "Usage: %s action n\nwhere action =\n  g - print graph\n  d - print
        return 1;
```

```c
    }

    n = atoi(argv[2]);

    g = makeSampleGraph(n);

    switch(argv[1][0]) {
        case 'g':
            printGraph(g);
            break;
        case 'd':
            dfs(g, 0);
            printTree(g);
            break;
        case 'b':
            bfs(g, 0);
            printTree(g);
            break;
        default:
            fprintf(stderr, "%s: unknown action '%c'\n", argv[0], argv[1][0]);
            return 1;
    }

    graphDestroy(g);

    return 0;
}
```

examples/graphSearch/search.c

The output of the program is either the graph, a DFS tree of the graph rooted at 0, or a BFS tree of the graph rooted at 0, in a format suitable for feeding to the GraphViz program `dot`, which draws pictures of graphs.

Here are the pictures for $n = 20$.

5.13.5.2 Combined implementation of depth-first and breadth-first search

These are some older implementations of BFS and DFS that demonstrate how both can be written using the same code just by changing the behavior of the core data structure. This also demonstrates how to construct DFS iteratively; for BFS, the preceding implementation is better in every respect.

```c
/* Typical usage:
 *
 *    struct searchInfo *s;
 *    int n;
 *
```

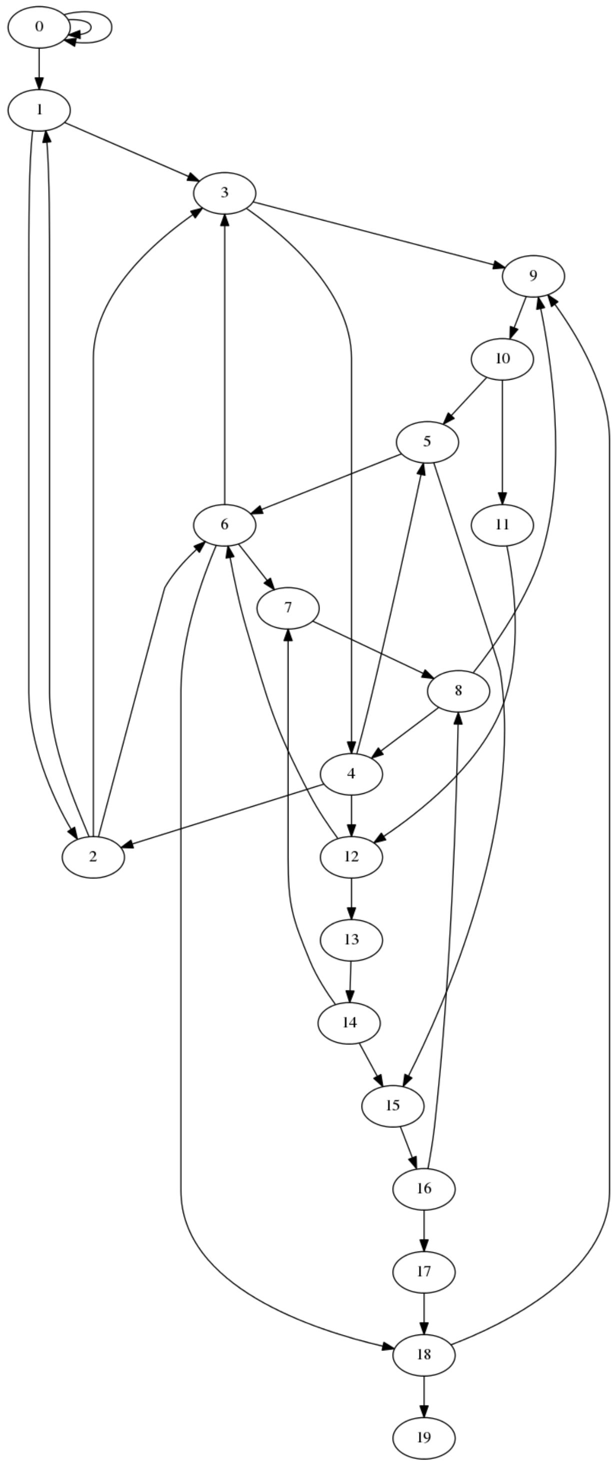


Figure 5: The full graph

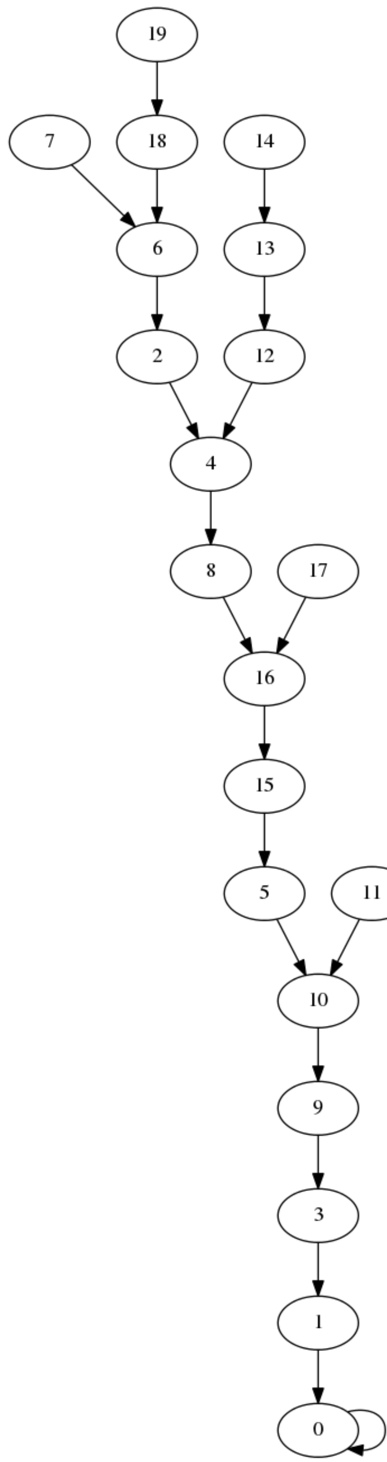


Figure 6: DFS tree

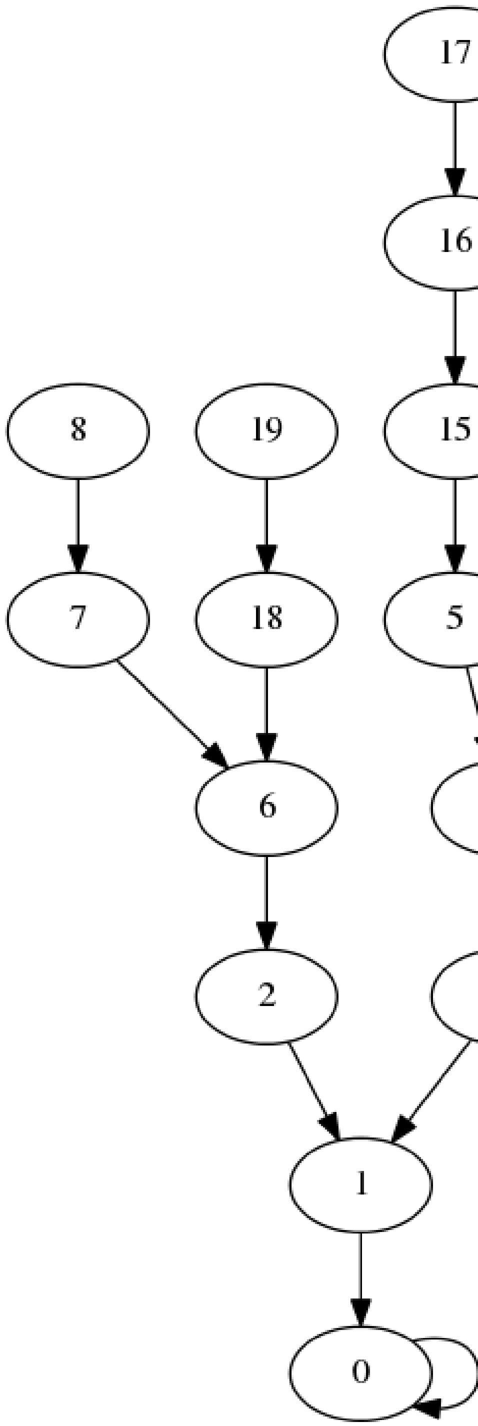


Figure 7: BFS tree

```
 *    s = searchInfoCreate(g);
 *
 *    n = graph_vertices(g);
 *    for(i = 0; i < n; i++) {
 *        dfs(s, i);
 *    }
 *
 *    ... use results in s ...
 *
 *    searchInfoDestroy(s);
 *
 */

/* summary information per node for dfs and bfs */
/* this is not intended to be opaque---user can read it */
/* (but should not write it!) */

#define SEARCH_INFO_NULL (-1) /* for empty slots */

struct searchInfo {
    Graph graph;
    int reached;         /* count of reached nodes */
    int *preorder;       /* list of nodes in order first reached */
    int *time;           /* time[i] == position of node i in preorder list */
    int *parent;         /* parent in DFS or BFS forest */
    int *depth;          /* distance from root */
};

/* allocate and initialize search results structure */
/* you need to do this before passing it to dfs or bfs */
struct searchInfo *searchInfoCreate(Graph g);

/* free searchInfo data---does NOT free graph pointer */
void searchInfoDestroy(struct searchInfo *);

/* perform depth-first search starting at root, updating results */
void dfs(struct searchInfo *results, int root);

/* perform breadth-first search starting at root, updating results */
void bfs(struct searchInfo *results, int root);
```

examples/graphs/genericSearch.h

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

#include "graph.h"
```

```c
#include "genericSearch.h"

/* create an array of n ints initialized to SEARCH_INFO_NULL */
static int *
createEmptyArray(int n)
{
    int *a;
    int i;

    a = malloc(sizeof(*a) * n);
    assert(a);

    for(i = 0; i < n; i++) {
        a[i] = SEARCH_INFO_NULL;
    }

    return a;
}

/* allocate and initialize search results structure */
/* you need to do this before passing it to dfs or bfs */
struct searchInfo *
searchInfoCreate(Graph g)
{
    struct searchInfo *s;
    int n;

    s = malloc(sizeof(*s));
    assert(s);

    s->graph = g;
    s->reached = 0;

    n = graphVertexCount(g);

    s->preorder = createEmptyArray(n);
    s->time = createEmptyArray(n);
    s->parent = createEmptyArray(n);
    s->depth = createEmptyArray(n);

    return s;
}

/* free searchInfo data---does NOT free graph pointer */
void
searchInfoDestroy(struct searchInfo *s)
```

```c
{
    free(s->depth);
    free(s->parent);
    free(s->time);
    free(s->preorder);
    free(s);
}

/* used inside search routines */
struct edge {
    int u;          /* source */
    int v;          /* sink */
};

/* stack/queue */
struct queue {
    struct edge *e;
    int bottom;
    int top;
};

static void
pushEdge(Graph g, int u, int v, void *data)
{
    struct queue *q;

    q = data;

    assert(q->top < graphEdgeCount(g) + 1);

    q->e[q->top].u = u;
    q->e[q->top].v = v;
    q->top++;
}

/* this rather horrible function implements dfs if useQueue == 0 */
/* and bfs if useQueue == 1 */
static void
genericSearch(struct searchInfo *r, int root, int useQueue)
{
    /* queue/stack */
    struct queue q;

    /* edge we are working on */
    struct edge cur;
```

```
    /* start with empty q */
    /* we need one space per edge */
    /* plus one for the fake (root, root) edge */
    q.e = malloc(sizeof(*q.e) * (graphEdgeCount(r->graph) + 1));
    assert(q.e);

    q.bottom = q.top = 0;

    /* push the root */
    pushEdge(r->graph, root, root, &q);

    /* while q.e not empty */
    while(q.bottom < q.top) {
        if(useQueue) {
            cur = q.e[q.bottom++];
        } else {
            cur = q.e[--q.top];
        }

        /* did we visit sink already? */
        if(r->parent[cur.v] != SEARCH_INFO_NULL) continue;

        /* no */
        assert(r->reached < graphVertexCount(r->graph));
        r->parent[cur.v] = cur.u;
        r->time[cur.v] = r->reached;
        r->preorder[r->reached++] = cur.v;
        if(cur.u == cur.v) {
            /* we could avoid this if we were certain SEARCH_INFO_NULL */
            /* would never be anything but -1 */
            r->depth[cur.v] = 0;
        } else {
            r->depth[cur.v] = r->depth[cur.u] + 1;
        }

        /* push all outgoing edges */
        graphForeach(r->graph, cur.v, pushEdge, &q);
    }

    free(q.e);
}

void
dfs(struct searchInfo *results, int root)
{
    genericSearch(results, root, 0);
```

```
}

void
bfs(struct searchInfo *results, int root)
{
    genericSearch(results, root, 1);
}
```

examples/graphs/genericSearch.c

And here is some test code: genericSearchTest.c. You will need to compile `genericSearchTest.c` together with both `genericSearch.c` and `graph.c` to get it to work. This Makefile will do this for you.

5.13.5.3 Other variations on the basic algorithm

Stacks and queues are not the only options for the bucket in the generic search algorithm. Some other choices are:

- A priority queue keyed by edge weights. If the edges have **weights**, the generic tree-builder can be used to find a tree containing s with minimum total edge weight.[22] The basic idea is to always pull out the lightest edge. The resulting algorithm runs in $O(n + m \log m)$ time (since each heap operation takes $O(\log m)$ time), and is known as **Prim's algorithm**. See Prim's algorithm for more details.
- A priority queue keyed by path lengths. Here we assume that edges have **lengths**, and we want to build a shortest-path tree where the length of the path is no longer just the number of edges it contains but the sum of their weights. The basic idea is to keep track of the distance from the root to each node in the tree, and assign each edge a key equal to the sum of the distance to its source and its length. The resulting search algorithm, known as **Dijkstra's algorithm**, will give a shortest-path tree if all the edge weights are non-negative. See Dijkstra's algorithm.

5.14 Dynamic programming

Dynamic programming is a general-purpose algorithm design technique that is most often used to solve **combinatorial optimization** problems, where we are looking for the best possible input to some function chosen from an exponentially large search space.

There are two parts to dynamic programming. The first part is a programming technique: dynamic programming is essentially divide and conquer run in reverse: we solve a big instance of a problem by breaking it up recursively into smaller instances; but instead of carrying out the computation recursively from the top

[22] This only works if the graph is undirected, which means that for every edge uv there is a matching edge vu with the same weight.

down, we start from the bottom with the smallest instances of the problem, solving each increasingly large instance in turn and storing the result in a table. The second part is a design principle: in building up our table, we are careful always to preserve alternative solutions we may need later, by delaying commitment to particular choices to the extent that we can.

The bottom-up aspect of dynamic programming is most useful when a straightforward recursion would produce many duplicate subproblems. It is most efficient when we can enumerate a class of subproblems that doesn't include too many extraneous cases that we don't need for our original problem.

To take a simple example, suppose that we want to compute the n-th Fibonacci number using the defining recurrence

- $F(n) = F(n-1) + F(n-2)$
- $F(1) = F(0) = 1$.

A naive approach would simply code the recurrence up directly:

```
int
fib(int n)
{
    if(n < 2) {
        return 1
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

The running time of this procedure is easy to compute. The recurrence is

- $T(n) = T(n-1) + T(n-2) + \Theta(1)$,

whose solution is $\Theta(a^n)$ where a is the golden ratio $1.6180339887498948482\ldots$. This is badly exponential.[23]

5.14.1 Memoization

The problem is that we keep recomputing values of `fib` that we've already computed. We can avoid this by **memoization**, where we wrap our recursive solution in a **memoizer** that stores previously-computed solutions in a hash table. Sensible programming languages will let you write a memoizer once and apply it to arbitrary recursive functions. In less sensible programming languages it is usually easier just to embed the memoization in the function definition itself, like this:

[23] But it's linear in the numerical value of the output, which means that `fib(n)` will actually terminate in a reasonable amount of time on a typical modern computer when run on any n small enough that $F(n)$ fits in 32 bits. Running it using 64-bit (or larger) integer representations will be slower.

```c
int
memoFib(int n)
{
    int ret;

    if(hashContains(FibHash, n)) {
        return hashGet(FibHash, n);
    } else {
        ret = memoFib(n-1) + memoFib(n-2);
        hashPut(FibHash, n, ret);
        return ret;
    }
}
```

The assumption here is that `FibHash` is a global hash table that we have initialized to map `0` and `1` to `1`. The total cost of running this procedure is $O(n)$, because `fib` is called at most twice for each value k in $0 \ldots n$.

Memoization is a very useful technique in practice, but it is not popular with algorithm designers because computing the running time of a complex memoized procedure is often much more difficult than computing the time to fill a nice clean table. The use of a hash table instead of an array may also add overhead (and code complexity) that comes out in the constant factors. But it is always the case that a memoized recursive procedure considers no more subproblems than a table-based solution, and it may consider many fewer if we are sloppy about what we put in our table (perhaps because we can't easily predict what subproblems will be useful).

5.14.2 Dynamic programming

Dynamic programming comes to the rescue. Because we know what smaller cases we have to reduce F(n) to, instead of computing F(n) top-down, we compute it bottom-up, hitting all possible smaller cases and storing the results in an array:

```c
int
fib2(int n)
{
    int *a;
    int i;
    int ret;

    if(n < 2) {
        return 1;
    } else {
        a = malloc(sizeof(*a) * (n+1));
        assert(a);
```

```
        a[1] = a[2] = 1;

        for(i = 3; i <= n; i++) {
            a[i] = a[i-1] + a[i-2];
        }
    }

    ret = a[n];
    free(a);
    return ret;
}
```

Notice the recurrence is exactly the same in this version as in our original recursive version, except that instead of computing F(n-1) and F(n-2) recursively, we just pull them out of the array. This is typical of dynamic-programming solutions: often the most tedious editing step in converting a recursive algorithm to dynamic programming is changing parentheses to square brackets. As with memoization, the effect of this conversion is dramatic; what used to be an exponential-time algorithm is now linear-time.

5.14.2.1 More examples

5.14.2.1.1 Longest increasing subsequence Suppose that we want to compute the **longest increasing subsequence** of an array. This is a sequence, not necessarily contiguous, of elements from the array such that each is strictly larger than the one before it. Since there are 2^n different subsequences of an n-element array, the brute-force approach of trying all of them might take a while.

What makes this problem suitable for dynamic programming is that any prefix of a longest increasing subsequence is a longest increasing subsequence of the part of the array that ends where the prefix ends; if it weren't, we could make the big sequence longer by choosing a longer prefix. So to find the longest increasing subsequence of the whole array, we build up a table of longest increasing subsequences for each initial prefix of the array. At each step, when finding the longest increasing subsequence of elements $0 \dots i$, we can just scan through all the possible values for the second-to-last element and read the length of the best possible subsequence ending there out of the table. When the table is complete, we can scan for the best last element and then work backwards to reconstruct the actual subsequence.

This last step requires some explanation. We don't really want to store in `table[i]` the full longest increasing subsequence ending at position `i`, because it may be very big. Instead, we store the index of the second-to-last element of this sequence. Since that second-to-last element also has a table entry that stores the index of its predecessor, by following the indices we can generate a subsequence of length $O(n)$, even though we only stored $O(1)$ pieces of information in each table entry. This is similar to the parent pointer technique used in graph search algorithms.

Here's what the code looks like:

```c
/* compute a longest strictly increasing subsequence of an array of ints */
/* input is array a with given length n */
/* returns length of LIS */
/* If the output pointer is non-null, writes LIS to output pointer. */
/* Caller should provide at least sizeof(int)*n space for output */
/* If there are multiple LIS's, which one is returned is arbitrary. */
unsigned long
longest_increasing_subsequence(const int a[], unsigned long n, int *output);
```

examples/dynamicProgramming/lis/lis.h

```c
#include <stdlib.h>
#include <assert.h>

#include "lis.h"

unsigned long
longest_increasing_subsequence(const int a[], unsigned long n, int *output)
{
    struct lis_data {
        unsigned long length;             /* length of LIS ending at this point */
        unsigned long prev;               /* previous entry in the LIS ending at this point
    } *table;

    unsigned long best;      /* best entry in table */
    unsigned long scan;      /* used to generate output */

    unsigned long i;
    unsigned long j;
    unsigned long best_length;

    /* special case for empty table */
    if(n == 0) return 0;

    table = malloc(sizeof(*table) * n);

    for(i = 0; i < n; i++) {
        /* default best is just this element by itself */
        table[i].length = 1;
        table[i].prev = n;              /* default end-of-list value */
```

```
        /* but try all other possibilities */
        for(j = 0; j < i; j++) {
            if(a[j] < a[i] && table[j].length + 1 > table[i].length) {
                /* we have a winner */
                table[i].length = table[j].length + 1;
                table[i].prev = j;
            }
        }
    }

    /* now find the best of the lot */
    best = 0;

    for(i = 1; i < n; i++) {
        if(table[i].length > table[best].length) {
            best = i;
        }
    }

    /* table[best].length is now our return value */
    /* save it so that we don't lose it when we free table */
    best_length = table[best].length;

    /* do we really have to compute the output? */
    if(output) {
        /* yes :-( */
        scan = best;
        for(i = 0; i < best_length; i++) {
            assert(scan >= 0);
            assert(scan < n);

            output[best_length - i - 1] = a[scan];

            scan = table[scan].prev;
        }
    }

    free(table);

    return best_length;
}
```

examples/dynamicProgramming/lis/lis.c

A sample program that runs `longest_increasing_subsequence` on a list of numbers passed in by `stdin` is given in test_lis.c. Here is a Makefile.

Implemented like this, the cost of finding an LIS is $O(n^2)$, because to compute each entry in the array, we have to search through all the previous entries to find the longest path that ends at a value less than the current one. This can be improved by using a more clever data structure. If we use a binary search tree that stores path keyed by the last value, and augment each node with a field that represents the maximum length of any path in the subtree under that node, then we can find the longest feasible path that we can append the current node to in $O(\log n)$ time instead of $O(n)$ time. This brings the total cost down to only $O(n \log n)$.

5.14.2.1.2 All-pairs shortest paths Suppose we want to compute the distance between any two points in a graph, where each edge uv has a length $\ell_u v$ ($+\infty$ for edges not in the graph) and the distance between two vertices s and t$ is the minimum over all s–t paths of the total length of the edges. There are various algorithms for doing this for a particular s and t, but there is also a very simple dynamic programming algorithm known as **Floyd-Warshall** that computes the distance between all n^2 pairs of vertices in $\Theta(n^3)$ time.

The assumption is that the graph does not contain a **negative cycle** (a cycle with total edge weight less than zero), so that for two connected nodes there is always a shortest path that uses each intermediate vertex at most once. If a graph does contain a negative cycle, the algorithm will detect it by reporting the distance from i to i less than zero for some i.

Negative cycles don't generally exist in distance graphs (unless you have the ability to move faster than the speed of light), but they can come up in other contexts. One example would be in currency arbitrage, where each node is some currency, the weight of an edge uv is the logarithm of the exchange rate from u to v, and the total weight of a path from s to t gives the logarithm of the number of units of t you can get for one unit of s, since adding the logs along the path corresponds to multiplying all the exchange rates. In this context a negative cycle gives you a way to turn a dollar into less than a dollar by running it through various other currencies, which is not useful, but a *positive cycle* lets you pay for the supercomputer you bought to find it before anybody else did. If we negate all the edge weights, we turn a positive cycle into a negative cycle, making a fast algorithm for finding this negative cycle potentially valuable.

However, if we don't have any negative cycles, the idea is that we can create restricted instances of the shortest-path problem by limiting the maximum index of any node used on the path. Let $L(i, j, k)$ be the length of a shortest path from i to j that uses only the vertices $0, \dots, k-1$ along the path (not counting the endpoints i and j, which can be anything). When $k = 0$, this is just the length of the i–j edge, or $+\infty$ if there is no such edge. So we can start by computing $L(i, j, 0)$ for all i. Now given $L(i, j, k)$ for all i and some k, we can compute $L(i, j, k + 1)$ by observing that any shortest i–j path that has intermediate vertices in $0 \dots k$ either consists of a path with intermediate vertices in $0 \dots k-1$, or consists of a path from i to k followed by a path from k to j, where both of these paths have intermediate vertices in $0 \dots k-1$. So we get

- $L(i,j,k+1) = \min(L(i,j,k), L(i,k,k) + L(k,j,k)$.

Since this takes $O(1)$ time to compute if we have previously computed $L(i,j,k)$ for all i and j, we can fill in the entire table in $O(n^3)$ time.

Implementation details:

- If we want to reconstruct the shortest path in addition to computing its length, we can store the first vertex for each i–j path. This will either be (a) the first vertex in the i–j path for the previous k, or (b) the first vertex in the i–k path.
- We don't actually need to use a full three-dimensional array. It's enough to store one value for each pair i, j and let k be implicit. At each step we let $L[i][j]$ be $\min(L[i][j], L[i][k] + L[k][j])$. The trick is that we don't care if $L[i][k]$ or $L[k][j]$ has already been updated, because that will only give us paths with a few extra k vertices, which won't be the shortest paths anyway assuming no negative cycles.

5.14.2.1.3 Longest common subsequence Given sequences of characters v and w, v is a *subsequence* of w if every character in v appears in w in the same order. For example, `aaaaa`, `brac`, and `badar` are all subsequences of `abracadabra`, but `badcar` is not. A longest common subsequence (LCS for short) of two sequences x and y is the longest sequence that is a subsequence of both: two longest common subsequences of `abracadabra` and `badcar` are `badar` and `bacar`.

As with longest increasing subsequence, one can find the LCS of two sequence by brute force, but it will take even longer. Not only are there are 2^n subsequences of a sequence of length n, but checking each subsequence of the first to see if it is also a subsequence of the second may take some time. It is better to solve the problem using dynamic programming. Having sequences gives an obvious linear structure to exploit: the basic strategy will be to compute LCSs for increasingly long prefixes of the inputs. But with two sequences we will have to consider prefixes of both, which will give us a two-dimensional table where rows correspond to prefixes of sequence x and columns correspond to prefixes of sequence y.

The recursive decomposition that makes this technique work looks like this. Let $L(x,y)$ be the length of the longest common subsequence of x and y, where x and y are strings. Let a and b be single characters. Then $L(xa, yb)$ is the maximum of:

- $L(x,y) + 1$, if $a = b$,
- $L(xa, y)$, or
- $L(x, yb)$.

The idea is that we either have a new matching character we couldn't use before (the first case), or we have an LCS that doesn't use one of a or b (the remaining cases). In each case the recursive call to LCS involves a shorter prefix of xa or yb, with an ultimate base case $L(x, y) = 0$ if at least one of x or y is the empty string. So we can fill in these values in a table, as long as we are careful to make sure that the shorter prefixes are always filled first. If we are smart about remembering which case applies at each step, we can even go back and extract an actual LCS, by stitching together to places where $a = b$. Here's a short C program that does this:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <limits.h>

/* compute longest common subsequence of argv[1] and argv[2] */

/* computes longest common subsequence of x and y, writes result to lcs */
/* lcs should be pre-allocated by caller to 1 + minimum length of x or y */
void
longestCommonSubsequence(const char *x, const char *y, char *lcs)
{
    int xLen;
    int yLen;
    int i;             /* position in x */
    int j;             /* position in y */

    xLen = strlen(x);
    yLen = strlen(y);

    /* best choice at each position */
    /* length gives length of LCS for these prefixes */
    /* prev points to previous substring */
    /* newChar if non-null is new character */
    struct choice {
        int length;
        struct choice *prev;
        char newChar;
    } best[xLen][yLen];

    for(i = 0; i < xLen; i++) {
        for(j = 0; j < yLen; j++) {
            /* we can always do no common substring */
            best[i][j].length = 0;
            best[i][j].prev = 0;
```

```
            best[i][j].newChar = 0;

            /* if we have a match, try adding new character */
            /* this is always better than the nothing we started with */
            if(x[i] == y[j]) {
                best[i][j].newChar = x[i];
                if(i > 0 && j > 0) {
                    best[i][j].length = best[i-1][j-1].length + 1;
                    best[i][j].prev = &best[i-1][j-1];
                } else {
                    best[i][j].length = 1;
                }
            }

            /* maybe we can do even better by ignoring a new character */
            if(i > 0 && best[i-1][j].length > best[i][j].length) {
                /* throw away a character from x */
                best[i][j].length = best[i-1][j].length;
                best[i][j].prev = &best[i-1][j];
                best[i][j].newChar = 0;
            }

            if(j > 0 && best[i][j-1].length > best[i][j].length) {
                /* throw away a character from x */
                best[i][j].length = best[i][j-1].length;
                best[i][j].prev = &best[i][j-1];
                best[i][j].newChar = 0;
            }

        }
    }

    /* reconstruct string working backwards from best[xLen-1][yLen-1] */
    int outPos;        /* position in output string */
    struct choice *p;  /* for chasing linked list */

    outPos = best[xLen-1][yLen-1].length;
    lcs[outPos--] = '\0';

    for(p = &best[xLen-1][yLen-1]; p; p = p->prev) {
        if(p->newChar) {
            assert(outPos >= 0);
            lcs[outPos--] = p->newChar;
        }
    }
}
```

```
int
main(int argc, char **argv)
{
    if(argc != 3) {
        fprintf(stderr, "Usage: %s string1 string2\n", argv[0]);
        return 1;
    }

    char output[strlen(argv[1]) + 1];

    longestCommonSubsequence(argv[1], argv[2], output);

    puts(output);

    return 0;
}
```

examples/dynamicProgramming/lcs/lcs.c

The whole thing takes $O(nm)$ time where n and m are the lengths of A and B.

5.15 Randomization

Randomization is a fundamental technique in algorithm design, that allows programs to run quickly when the average-case behavior of an algorithm is better than the worst-case behavior. It is also heavily used in games, both in entertainment and gambling. The latter application gives the only example I know of a programmer killed for writing bad code, which shows how serious good random-number generation is.

5.15.1 Generating random values in C

If you want random values in a C program, there are three typical ways of getting them, depending on how good (i.e. uniform, uncorrelated, and unpredictable) you want them to be.

5.15.1.1 The rand function from the standard library

E.g.

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

int
main(int argc, char **argv)
{
```

```
    printf("%d\n", rand());
    return 0;
}
```

examples/randomization/randOnce.c

The `rand` function, declared in `stdlib.h`, returns a random-looking integer in the range 0 to `RAND_MAX` (inclusive) every time you call it. On machines using the GNU C library `RAND_MAX` is equal to `INT_MAX` which is typically $2^{31} - 1$, but `RAND_MAX` may be as small as 32767. There are no particularly strong guarantees about the quality of random numbers that `rand` returns, but it should be good enough for casual use, and it has the advantage that as part of the C standard you can assume it is present almost everywhere.

Note that `rand` is a **pseudorandom number generator**: the sequence of values it returns is predictable if you know its starting state (and is still predictable from past values in the sequence even if you don't know the starting state, if you are clever enough). It is also the case that the initial seed is fixed, so that the program above will print the same value every time you run it.

This is a feature: it permits debugging randomized programs. As John von Neumann, who proposed pseudorandom number generators in his 1946 talk "Various Techniques Used in Connection With Random Digits," explained:

> We see then that we could build a physical instrument to feed random digits directly into a high-speed computing machine and could have the control call for these numbers as needed. The real objection to this procedure is the practical need for checking computations. If we suspect that a calculation is wrong, almost any reasonable check involves repeating something done before. At that point the introduction of new random numbers would be intolerable.

5.15.1.1.1 Supplying a seed with `srand` If you want to get different sequences, you need to **seed** the random number generator using `srand`. A typical use might be:

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

int
main(int argc, char **argv)
{
    srand(time(0));
    printf("%d\n", rand());
    return 0;
}
```

examples/randomization/srandFromTime.c

Here `time(0)` returns the number of seconds since the epoch (00:00:00 UTC, January 1, 1970, for POSIX systems, not counting leap seconds). Note that this still might give repeated values if you run it twice in the same second, and it's extremely dangerous if you expect to distribute your code to a lot of people who want different results, since two of your users *are* likely to run it twice in the same second. See the discussion of `/dev/urandom` below for a better method.

5.15.1.2 Better pseudorandom number generators

There has been quite a bit of research on pseudorandom number generators over the years, and much better pseudorandom number generators than `rand` are available. The current champion for simulation work is **Mersenne Twister**, which runs about 4 times faster than `rand` in its standard C implementation and passes a much wider battery of statistical tests. Its English-language home page is at http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html. As with `rand`, you still need to provide an initial seed value.

There are also **cryptographically secure pseudorandom number generators**, of which the most famous is Blum Blum Shub. These cannot be predicted based on their output if seeded with a true random value (under certain cryptographic assumptions: hardness of factoring for Blum Blum Shub). Unfortunately, cryptographic PRNGs are usually too slow for day-to-day use.

5.15.1.3 Random numbers without the pseudo

If you really need actual random numbers and are on a Linux or BSD-like operating system, you can use the special device files `/dev/random` and `/dev/urandom`. These can be opened for reading like ordinary files, but the values read from them are a random sequence of bytes (including null characters). A typical use might be:

```c
#include <stdio.h>

int
main(int argc, char **argv)
{
    unsigned int randval;
    FILE *f;

    f = fopen("/dev/random", "r");
    fread(&randval, sizeof(randval), 1, f);
    fclose(f);

    printf("%u\n", randval);

    return 0;
}
```

examples/randomization/devRandom.c

(A similar construction can also be used to obtain a better initial seed for `srand` than `time(0)`.)

Both `/dev/random` and `/dev/urandom` derive their random bits from physically random properties of the computer, like time between keystrokes or small variations in hard disk rotation speeds. The difference between the two is that `/dev/urandom` will always give you some random-looking bits, even if it has to generate extra ones using a cryptographic pseudo-random number generator, while `/dev/random` will only give you bits that it is confident are in fact random. Since your computer only generates a small number of genuinely random bits per second, this may mean that `/dev/random` will exhaust its pool if read too often. In this case, a read on `/dev/random` will block (just like reading a terminal with no input on it) until the pool has filled up again.

Neither `/dev/random` nor `/dev/urandom` is known to be secure against a determined attacker, but they are about the best you can do without resorting to specialized hardware.

5.15.1.4 Range issues

The problem with `rand` is that getting a uniform value between 0 and $2^{31} - 1$ may not be what you want. It could be that `RAND_MAX` is be too small; in this case, you may have to call `rand` more than once and paste together the results. But there can be problems with `RAND_MAX` even if it is bigger than the values you want.

For example, suppose you want to simulate a die roll for your video craps machine, but you don't want to get whacked by Johnny "The Debugger" when the Nevada State Gaming Commission notices that 6-6 is coming up slightly less often than it's supposed to. A natural thing to try would be to take the output of `rand` mod 6:

```
int d6(void) {
    return rand() % 6 + 1;
}
```

The problem here is that there are 2^{31} outputs from rand, and 6 doesn't divide 2^{31}. So 1 and 2 are slightly more likely to come up than 3, 4, 5, or 6. This can be particularly noticeable if we want a uniform variable from a larger range, e.g. $[0 \dots \lfloor (2/3) \cdot 2^{31} \rfloor]$.

We can avoid this with a technique called **rejection sampling**, where we reject excess parts of the output range of `rand`. For rolling a die, the trick is to reject anything in the last extra bit of the range that is left over after the largest multiple of the die size. Here's a routine that does this, returning a uniform value in the range 0 to n-1 for any positive n, together with a program that demonstrates its use for rolling dice:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

/* return a uniform random value in the range 0..n-1 inclusive */
int
randRange(int n)
{
    int limit;
    int r;

    limit = RAND_MAX - (RAND_MAX % n);

    while((r = rand()) >= limit);

    return r % n;
}

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

    srand(time(0));

    for(i = 0; i < 40; i++) {
        printf("%d ", randRange(6)+1);
    }

    putchar('\n');

    return 0;
}
```

examples/randomization/randRange.c

More generally, rejection sampling can be used to get random values with particular properties, where it's hard to generate a value with that property directly. Here's a program that generates random primes:

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

/* return 1 if n is prime */
```

```c
int
isprime(int n)
{
    int i;

    if(n % 2 == 0 || n == 1) { return 0; }

    for(i = 3; i*i <= n; i += 2) {
        if(n % i == 0) { return 0; }
    }

    return 1;
}

/* return a uniform random value in the range 0..n-1 inclusive */
int
randPrime(void)
{
    int r;

    /* extra parens avoid warnings */
    while(!isprime((r = rand())));

    return r;
}

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

    srand(time(0));

    for(i = 0; i < 10; i++) {
        printf("%d\n", randPrime());
    }

    return 0;
}
```

examples/randomization/randPrime.c

One temptation to avoid is to re-use your random values. If, for example, you try to find a random prime by picking a random x and trying x, x+1, x+2, etc., until you hit a prime, some primes are more likely to come up than others.

5.15.2 Randomized algorithms

Randomized algorithms typically make random choices to get good average worst-case performance in situations where a similar deterministic algorithm would fail badly for some inputs but perform well on most inputs. The idea is that the randomization scrambles the input space so that the adversary can't predict which possible input values will be bad for us. This still allows him to make trouble if he gets lucky, but most of the time our algorithm should run quickly.

5.15.2.1 Randomized search This is essentially rejection sampling in disguise. Suppose that you want to find one of many needles in a large haystack. One approach is to methodically go through the straws/needles one at a time until you find a needle. But you may find that your good friend the adversary has put all the needles at the end of your list. Picking candidate at random is likely to hit a needle faster if there are many of them.

Here is a (silly) routine that quickly finds a number whose high-order bits match a particular pattern:

```
int
matchBits(int pattern)
{
    int r;

    while(((r = rand()) & 0x70000000) != (pattern & 0x70000000));

    return r;
}
```

This will find a winning value in 8 tries on average. In contrast, this deterministic version will take a lot longer for nonzero patterns:

```
int
matchBitsDeterministic(int pattern)
{
    int i;

    for(i = 0; (i & 0x70000000) != (pattern & 0x70000000); i++);

    return i;
}
```

The downside of the randomized approach is that it's hard to tell when to quit if there are no matches; if we stop after some fixed number of trials, we get a Monte Carlo algorithm that may give the wrong answer with small probability. The usual solution is to either accept a small probability of failure, or interleave a deterministic backup algorithm that always works. The latter approach gives a Las Vegas algorithm whose running time is variable but whose correctness is not.

5.15.2.2 Quickselect and quicksort

Quickselect, or **Hoare's FIND** (Hoare, C. A. R. Algorithm 65: FIND, CACM 4(7):321–322, July 1961), is an algorithm for quickly finding the k-th largest element in an unsorted array of n elements. It runs in $O(n)$ time on average, which is the best one can hope for (we have to look at every element of the array to be sure we didn't miss a small one that changes our answer) and better than the $O(n \log n)$ time we get if we sort the array first using a comparison-based sorting algorithm.

The idea is to pick a random pivot and divide the input into two piles, each of which is likely to be roughly a constant fraction of the size of the original input.[24] It takes O(n) time to split the input up (we have to compare each element to the pivot once), and in the recursive calls this gives a geometric series. We can even do the splitting up in place if we are willing to reorder the elements of our original array.

If we recurse into both piles instead of just one, we get **quicksort** (Hoare, C. A. R. Algorithm 64: Quicksort. CACM 4(7):321, July 1961), a very fast and simple comparison-based sorting algorithm. Here is an implementation of both algorithms:

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

/* reorder an array to put elements <= pivot
 * before elements > pivot.
 * Returns number of elements <= pivot */
static int
splitByPivot(int n, int *a, int pivot)
{
    int lo;
    int hi;
    int temp;  /* for swapping */

    assert(n >= 0);

    /* Dutch Flag algorithm */
    /* swap everything <= pivot to bottom of array */
    /* invariant is i < lo implies a[i] <= pivot */
```

[24] The actual analysis is pretty complicated, since we are more likely to land in a bigger pile, but it's not hard to show that on average even the bigger pile has no more than 3/4 of the elements.

```
    /* and i > hi implies a[i] > pivot */
    lo = 0;
    hi = n-1;

    while(lo <= hi) {
        if(a[lo] <= pivot) {
            lo++;
        } else {
            temp = a[hi];
            a[hi--] = a[lo];
            a[lo] = temp;
        }
    }

    return lo;
}

/* find the k-th smallest element of an n-element array */
/* may reorder elements of the original array */
int
quickselectDestructive(int k, int n, int *a)
{
    int pivot;
    int lo;

    assert(0 <= k);
    assert(k < n);

    if(n == 1) {
        return a[0];
    }

    /* else */
    pivot = a[rand() % n];   /* we will tolerate non-uniformity */

    lo = splitByPivot(n, a, pivot);

    /* lo is now number of values <= pivot */
    if(k < lo) {
        return quickselectDestructive(k, lo, a);
    } else {
        return quickselectDestructive(k - lo, n - lo, a + lo);
    }
}

/* sort an array in place */
```

```c
void
quickSort(int n, int *a)
{
    int pivot;
    int lo;

    if(n <= 1) {
        return;
    }

    /* else */
    pivot = a[rand() % n];    /* we will tolerate non-uniformity */

    lo = splitByPivot(n, a, pivot);

    quickSort(lo, a);
    quickSort(n - lo, a + lo);
}


/* shuffle an array */
void
shuffle(int n, int *a)
{
    int i;
    int r;
    int temp;

    for(i = n - 1; i > 0; i--) {
        r = rand() % i;
        temp = a[r];
        a[r] = a[i];
        a[i] = temp;
    }
}

#define N (1024)

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

    srand(0);  /* use fixed value for debugging */
```

```
    for(i = 0; i < N; i++) {
        a[i] = i;
    }

    shuffle(N, a);

    for(i = 0; i < N; i++) {
        assert(quickselectDestructive(i, N, a) == i);
    }

    shuffle(N, a);

    quickSort(N, a);

    for(i = 0; i < N; i++) {
        assert(a[i] == i);
    }

    return 0;
}
```

examples/randomization/quick.c

5.15.3 Randomized data structures

Suppose we insert n elements into an initially-empty binary search tree in random order with no rebalancing. Then each element is equally likely to be the root, and all the elements less than the root end up in the left subtree, while all the elements greater than the root end up in the right subtree, where they are further partitioned recursively. This is exactly what happens in quicksort, so the structure of the tree will exactly mirror the structure of an execution of quicksort. In particular, the average depth of a node will be $O(\log n)$, giving us the same expected search cost as in a balanced binary tree.

The problem with this approach is that we don't have any guarantees that the input will be supplied in random order, and in the worst case we end up with a linked list. The solution is to put the randomization into the algorithm itself, making the structure of the tree depend on random choices made by the program itself.

5.15.3.1 Skip lists A **skip list** (Pugh, 1990) is a randomized tree-like data structure based on linked lists. It consists of a level 0 list that is an ordinary sorted linked list, together with higher-level lists that contain a random sampling of the elements at lower levels. When inserted into the level i list, an element flips a coin that tells it with probability p to insert itself in the level i+1 list as well.

Searches in a skip list are done by starting in the highest-level list and searching forward for the last element whose key is smaller than the target; the search then continues in the same way on the next level down. The idea is that the higher-level lists act as express lanes to get us to our target value faster. To bound the expected running time of a search, it helps to look at this process backwards; the reversed search path starts at level 0 and continues going backwards until it reaches the first element that is also in a higher level; it then jumps to the next level up and repeats the process. On average, we hit $1 + 1/p$ nodes at each level before jumping back up; for constant p (e.g. 1/2), this gives us $O(\log n)$ steps for the search.

The space per element of a skip list also depends on p. Every element has at least one outgoing pointer (on level 0), and on average has exactly $1/(1-p)$ expected pointers. So the space cost can also be adjusted by adjusting p. For example, if space is at a premium, setting $p = 1/10$ produces 10/9 pointers per node on average—not much more than in a linked list—but still gives $O(\log n)$ search time.

Below is an implementation of a skip list. To avoid having to allocate a separate array of pointers for each element, we put a length-1 array at the end of `struct skiplist` and rely on C's lack of bounds checking to make the array longer if necessary. A dummy head element stores pointers to all the initial elements in each level of the skip list; it is given the dummy key `INT_MIN` so that searches for values less than any in the list will report this value. Aside from these nasty tricks, the code for search and insertion is pretty straightforward. Code for deletion is a little more involved, because we have to make sure that we delete the leftmost copy of a key if there are duplicates (an alternative would be to modify `skiplistInsert` to ignore duplicates).

```
#include <stdlib.h>
#include <assert.h>
#include <limits.h>

#include "skiplist.h"

#define MAX_HEIGHT (32)

struct skiplist {
    int key;
    int height;                /* number of next pointers */
    struct skiplist *next[1];  /* first of many */
};

/* choose a height according to a geometric distribution */
static int
```

```c
chooseHeight(void)
{
    int i;

    for(i = 1; i < MAX_HEIGHT && rand() % 2 == 0; i++);

    return i;
}

/* create a skiplist node with the given key and height */
/* does not fill in next pointers */
static Skiplist
skiplistCreateNode(int key, int height)
{
    Skiplist s;

    assert(height > 0);
    assert(height <= MAX_HEIGHT);

    s = malloc(sizeof(struct skiplist) + sizeof(struct skiplist *) * (height - 1));

    assert(s);

    s->key = key;
    s->height = height;

    return s;
}

/* create an empty skiplist */
Skiplist
skiplistCreate(void)
{
    Skiplist s;
    int i;

    /* s is a dummy head element */
    s = skiplistCreateNode(INT_MIN, MAX_HEIGHT);

    /* this tracks the maximum height of any node */
    s->height = 1;

    for(i = 0; i < MAX_HEIGHT; i++) {
        s->next[i] = 0;
    }
```

```c
    return s;
}

/* free a skiplist */
void
skiplistDestroy(Skiplist s)
{
    Skiplist next;

    while(s) {
        next = s->next[0];
        free(s);
        s = next;
    }
}

/* return maximum key less than or equal to key */
/* or INT_MIN if there is none */
int
skiplistSearch(Skiplist s, int key)
{
    int level;

    for(level = s->height - 1; level >= 0; level--) {
        while(s->next[level] && s->next[level]->key <= key) {
            s = s->next[level];
        }
    }

    return s->key;
}

/* insert a new key into s */
void
skiplistInsert(Skiplist s, int key)
{
    int level;
    Skiplist elt;

    elt = skiplistCreateNode(key, chooseHeight());

    assert(elt);

    if(elt->height > s->height) {
        s->height = elt->height;
    }
```

```
    /* search through levels taller than elt */
    for(level = s->height - 1; level >= elt->height; level--) {
        while(s->next[level] && s->next[level]->key < key) {
            s = s->next[level];
        }
    }

    /* now level is elt->height - 1, we can start inserting */
    for(; level >= 0; level--) {
        while(s->next[level] && s->next[level]->key < key) {
            s = s->next[level];
        }

        /* s is last entry on this level < new element */
        /* do list insert */
        elt->next[level] = s->next[level];
        s->next[level] = elt;
    }
}

/* delete a key from s */
void
skiplistDelete(Skiplist s, int key)
{
    int level;
    Skiplist target;

    /* first we have to find leftmost instance of key */
    target = s;

    for(level = s->height - 1; level >= 0; level--) {
        while(target->next[level] && target->next[level]->key < key) {
            target = target->next[level];
        }
    }

    /* take one extra step at bottom */
    target = target->next[0];

    if(target == 0 || target->key != key) {
        return;
    }

    /* now we found target, splice it out */
    for(level = s->height - 1; level >= 0; level--) {
```

```c
        while(s->next[level] && s->next[level]->key < key) {
            s = s->next[level];
        }

        if(s->next[level] == target) {
            s->next[level] = target->next[level];
        }
    }

    free(target);
}
```

examples/trees/skiplist/skiplist.c

Here is the header file, Makefile, and test code: skiplist.h, Makefile, test_skiplist.c.

5.15.3.2 Universal hash families Randomization can also be useful in hash tables. Recall that in building a hash table, we are relying on the hash function to spread out bad input distributions over the indices of our array. But for any fixed hash function, in the worst case we may get inputs where every key hashes to the same location. **Universal hashing** (Carter and Wegman, 1979) solves this problem by choosing a hash function at random. We may still get unlucky and have the hash function hash all our values to the same location, but now we are relying on the random number generator to be nice to us instead of the adversary. We can also rehash with a new random hash function if we find out that the one we are using is bad.

The problem here is that we can't just choose a function uniformly at random out of the set of all possible hash functions, because there are too many of them, meaning that we would spend more space representing our hash function than we would on the table. The solution is to observe that we don't need our hash function h to be truly random; it's enough if the probability of collision Pr[h(x) = h(y)] for any fixed keys $x \neq y$ is $1/m$, where m is the size of the hash table. The reason is that the cost of searching for x (with chaining) is linear in the number of keys already in the table that collide with it. The expected number of such collisions is the sum of Pr[h(x) = h(y)] over all keys y in the table, or n/m if we have n keys. So this pairwise collision probability bound is enough to get the desired n/m behavior out of our table. A family of hash function with this property is called **universal**.

How do we get a universal hash family? For strings, we can use a table of random values, one for each position and possible character in the string. The hash of a string is then the exclusive or of the random values `hashArray[i][s[i]]` corresponding to the actual characters in the string. If our table has size a power of two, this has the universal property, because if two strings x and y differ in some position i, then there is only one possible value of `hashArray[i][y[i]]` (mod m) that will make the hash functions equal.

Typically to avoid having to build an arbitrarily huge table of random values, we only has an initial prefix of the string. Here is a hash function based on this idea, which assumes that the `d` data structure includes a `hashArray` field that contains the random values for this particular hash table:

```
static unsigned long
hash_function(Dict d, const char *s)
{
    unsigned const char *us;
    unsigned long h;
    int i;

    h = 0;

    us = (unsigned const char *) s;

    for(i = 0; i < HASH_PREFIX_LENGTH && us[i] != '\0'; i++) {
        h ^= d->hashArray[i][us[i]];
    }

    return h;
}
```

A modified version of the `Dict` hash table from the chapter on hash tables that uses this hash function is given here: dict.c, dict.h, test_dict.c, Makefile.

5.16 String processing

Most of the time, when we've talked about the asymptotic performance of data structures, we have implicitly assumed that the keys we are looking up are of constant size. This means that computing a hash function or comparing two keys (as in a binary search tree) takes $O(1)$ time. What if this is not the case?

If we consider an m-character string, any reasonable hash function is going to take $O(m)$ time since it has to look at all of the characters. Similarly, comparing two m-character strings may also take $O(m)$ time. If we charge for this (as we should!) then the cost of hash table operations goes from $O(1)$ to $O(m)$ and the cost of binary search tree operations, even in a balanced tree, goes from $O(\log n)$ to $O(m \log n)$. Even sorting becomes more expensive: a sorting algorithm that does $O(n \log n)$ comparisons may now take $O(mn \log n)$ time. But maybe we can exploit the structure of strings to get better performance.

5.16.1 Radix search

Radix search refers to a variety of data structures that support searching for strings considered as sequences of digits in some large base (or **radix**). These are generally faster than simple binary search trees because they usually only require examining one byte or less of the target string at each level of the tree, as compared to every byte in the target in a full string comparison. In many cases, the best radix search trees are even faster than hash tables, because they only need to look at a small part of the target string to identify it.

We'll describe several radix search trees, starting with the simplest and working up.

5.16.1.1 Tries

A **trie** is a binary tree (or more generally, a k-ary tree where k is the radix) where the root represents the empty bit sequence and the two children of a node representing sequence x represent the extended sequences $x0$ and $x1$ (or generally $x0, x1, \ldots, x(k-1)$). So a key is not stored at a particular node but is instead represented bit-by-bit (or digit-by-digit) along some path. Typically a trie assumes that the set of keys is prefix-free, i.e. that no key is a prefix of another; in this case there is a one-to-one correspondence between keys and leaves of the trie. If this is not the case, we can mark internal nodes that also correspond to the ends of keys, getting a slightly different data structure known as a **digital search tree**. For null-terminated strings as in C, the null terminator ensures that any set of strings is prefix-free.

Given this simple description, a trie storing a single long key would have a very large number of nodes. A standard optimization is to chop off any path with no branches in it, so that each leaf corresponds to the shortest unique prefix of a key. This requires storing the key in the leaf so that we can distinguish different keys with the same prefix.

The name *trie* comes from the phrase "information re*trie*val." Despite the etymology, *trie* is now almost always pronounced like *try* instead of *tree* to avoid confusion with other tree data structures.

5.16.1.1.1 Searching a trie

Searching a trie is similar to searching a binary search tree, except that instead of doing a comparison at each step we just look at the next bit in the target. The time to perform a search is proportional to the number of bits in the longest path in the tree matching a prefix of the target. This can be very fast for search misses if the target is wildly different from all the keys in the tree.

5.16.1.1.2 Inserting a new element into a trie

Insertion is more complicated for tries than for binary search trees. The reason is that a new element may add more than one new node. There are essentially two cases:

1. (The simple case.) In searching for the new key, we reach a null pointer leaving a non-leaf node. In this case we can simply add a new leaf. The cost of this case is essentially the same as for search plus $O(1)$ for building the new leaf.
2. (The other case.) In searching for the new key, we reach a leaf, but the key stored there isn't the same as the new key. Now we have to generate a new path for as long as the old key and the new key have the same bits, branching out to two different leaves at the end. The cost of this operation is within a constant factor of the cost for searching for the new leaf *after* it is inserted, since that's how long the newly-built search path will be.

In either case, the cost is bounded by the length of the new key, which is about the best we can hope for in the worst case for any data structure.

5.16.1.1.3 Implementation A typical trie implementation in C might look like this. It uses a `GET_BIT` macro similar to the one from the chapter on bit manipulation, except that we reverse the bits within each byte to get the right sorting order for keys.

```
typedef struct trie_node *Trie;

#define EMPTY_TRIE (0)

/* returns 1 if trie contains target */
int trie_contains(Trie trie, const char *target);

/* add a new key to a trie */
/* and return the new trie */
Trie trie_insert(Trie trie, const char *key);

/* free a trie */
void trie_destroy(Trie);

/* debugging utility: print all keys in trie */
void trie_print(Trie);
```

examples/trees/trie/trie.h

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

#include "trie.h"

#define BITS_PER_BYTE (8)
```

```
/* extract the n-th bit of x */
/* here we process bits within bytes in MSB-first order */
/* this sorts like strcmp */
#define GET_BIT(x, n) ((((x)[(n) / BITS_PER_BYTE]) & (0x1 << (BITS_PER_BYTE - 1 - (n) % BITS

#define TRIE_BASE (2)

struct trie_node {
    char *key;
    struct trie_node *kids[TRIE_BASE];
};

#define IsLeaf(t) ((t)->kids[0] == 0 && (t)->kids[1] == 0)

/* returns 1 if trie contains target */
int
trie_contains(Trie trie, const char *target)
{
    int bit;

    for(bit = 0; trie && !IsLeaf(trie); bit++) {
        /* keep going */
        trie = trie->kids[GET_BIT(target, bit)];
    }

    if(trie == 0) {
        /* we lost */
        return 0;
    } else {
        /* check that leaf really contains the target */
        return !strcmp(trie->key, target);
    }
}

/* gcc -pedantic kills strdup! */
static char *
my_strdup(const char *s)
{
    char *s2;

    s2 = malloc(strlen(s) + 1);
    assert(s2);

    strcpy(s2, s);
    return s2;
}
```

```
/* helper functions for insert */
static Trie
make_trie_node(const char *key)
{
    Trie t;
    int i;

    t = malloc(sizeof(*t));
    assert(t);

    if(key) {
        t->key = my_strdup(key);
        assert(t->key);
    } else {
        t->key = 0;
    }

    for(i = 0; i < TRIE_BASE; i++) t->kids[i] = 0;

    return t;
}

/* add a new key to a trie */
/* and return the new trie */
Trie
trie_insert(Trie trie, const char *key)
{
    int bit;
    int bitvalue;
    Trie t;
    Trie kid;
    const char *oldkey;

    if(trie == 0) {
        return make_trie_node(key);
    }
    /* else */
    /* first we'll search for key */
    for(t = trie, bit = 0; !IsLeaf(t); bit++, t = kid) {
        kid = t->kids[bitvalue = GET_BIT(key, bit)];
        if(kid == 0) {
            /* woohoo! easy case */
            t->kids[bitvalue] = make_trie_node(key);
            return trie;
```

```
        }
    }

    /* did we get lucky? */
    if(!strcmp(t->key, key)) {
        /* nothing to do */
        return trie;
    }

    /* else */
    /* hard case---have to extend the trie */
    oldkey = t->key;
#ifdef EXCESSIVE_TIDINESS
    t->key = 0;      /* not required but makes data structure look tidier */
#endif

    /* walk the common prefix */
    while(GET_BIT(oldkey, bit) == (bitvalue = GET_BIT(key, bit))) {
        kid = make_trie_node(0);
        t->kids[bitvalue] = kid;
        bit++;
        t = kid;
    }

    /* then split */
    t->kids[bitvalue] = make_trie_node(key);
    t->kids[!bitvalue] = make_trie_node(oldkey);

    return trie;
}

/* kill it */
void
trie_destroy(Trie trie)
{
    int i;

    if(trie) {
        for(i = 0; i < TRIE_BASE; i++) {
            trie_destroy(trie->kids[i]);
        }

        if(IsLeaf(trie)) {
            free(trie->key);
        }
```

```
        free(trie);
    }
}

static void
trie_print_internal(Trie t, int bit)
{
    int i;
    int kid;

    if(t != 0) {
        if(IsLeaf(t)) {
            for(i = 0; i < bit; i++) putchar(' ');
            puts(t->key);
        } else {
            for(kid = 0; kid < TRIE_BASE; kid++) {
                trie_print_internal(t->kids[kid], bit+1);
            }
        }
    }
}

void
trie_print(Trie t)
{
    trie_print_internal(t, 0);
}
```

examples/trees/trie/trie.c

Here is a short test program that demonstrates how to use it:

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

#include "trie.h"

/* test for trie.c */
/* reads lines from stdin and echoes lines that haven't appeared before */

/* read a line of text from stdin
 * and return it (without terminating newline) as a freshly-malloc'd block.
 * Caller is responsible for freeing this block.
 * Returns 0 on error or EOF.
 */
char *
getline(void)
```

```
{
    char *line;         /* line buffer */
    int n;              /* characters read */
    int size;           /* size of line buffer */
    int c;

    size = 1;
    line = malloc(size);
    if(line == 0) return 0;

    n = 0;

    while((c = getchar()) != '\n' && c != EOF) {
        while(n >= size - 1) {
            size *= 2;
            line = realloc(line, size);
            if(line == 0) return 0;
        }
        line[n++] = c;
    }

    if(c == EOF && n == 0) {
        /* got nothing */
        free(line);
        return 0;
    } else {
        line[n++] = '\0';
        return line;
    }
}

int
main(int argc, char **argv)
{
    Trie t;
    char *line;

    t = EMPTY_TRIE;

    while((line = getline()) != 0) {
        if(!trie_contains(t, line)) {
            puts(line);
        }

        /* try to insert it either way */
        /* this tests that insert doesn't blow up on duplicates */
```

```c
        t = trie_insert(t, line);

        free(line);
    }

    puts("===");

    trie_print(t);

    trie_destroy(t);

    return 0;
}
```

examples/trees/trie/test_trie.c

5.16.1.2 Patricia trees Tries perform well when all keys are short (or are distinguished by short prefixes), but can grow very large if one inserts two keys that have a long common prefix. The reason is that a trie has to store an internal node for every bit of the common prefix until the two keys become distinguishable, leading to long chains of internal nodes each of which has only one child. An optimization (described in this paper) known as a **Patricia tree** eliminates these long chains by having each node store the number of the bit to branch on, like this:

```c
struct patricia_node {
    char *key;
    int bit;
    struct patricia_node *kids[2];
};

typedef struct patricia_node *Patricia;
```

Now when searching for a key, instead of using the number of nodes visited so far to figure out which bit to look at, we just read the bit out of the table. This means in particular that we can skip over any bits that we don't actually branch on. We do however have to be more careful to make sure we don't run off the end of our target key, since it is possible that when skipping over intermediate bits we might skip over some that distinguish our target from all keys in the table, including longer keys. For example, a Patricia tree storing the strings `abc` and `abd` will first test bit position 22, since that's where `abc` and `abd` differ. This can be trouble if we are looking for `x`.

Here's the search code:

```c
int
patricia_contains(Patricia t, const char *key)
```

```
{
    int key_bits;

    key_bits = BITS_PER_BYTE * (strlen(key)+1);   /* +1 for the NUL */

    while(t && !IsLeaf(t)) {
        if(t->bit >= key_bits) {
            /* can't be there */
            return 0;
        } else {
            t = t->kids[GET_BIT(key, t->bit)];
        }
    }

    return t && !strcmp(t->key, key);
}
```

The insertion code is similar in many respects to the insertion code for a trie. The differences are that we never construct a long chain of internal nodes when splitting a leaf (although we do have to scan through both the old and new keys to find the first bit position where they differ), but we may sometimes have to add a new internal node between two previously existing nodes if a new key branches off at a bit position that was previously skipped over.

In the worst case Patricia trees are much more efficient than tries, in both space (linear in the number of keys instead of linear in the total size of the keys) and time complexity, often needing to examine only a very small number of bits for misses (hits still require a full scan in `strcmp` to verify the correct key). The only downside of Patricia trees is that since they work on bits, they are not quite perfectly tuned to the byte or word-oriented structure of modern CPUs.

5.16.1.3 Ternary search trees **Ternary search trees** were described by Jon Bentley and Bob Sedgewick in an article in the April 1988 issue of *Dr. Dobb's Journal*, available here.

The basic idea is that each node in the tree stores one character from the key and three child pointers `lt`, `eq`, and `gt`. If the corresponding character in the target is equal to the character in the node, we move to the *next* character in the target and follow the `eq` pointer out of the node. If the target is less, follow the `lt` pointer but stay at the *same* character. If the target is greater, follow the `gt` pointer and again stay at the same character. When searching for a string, we walk down the tree until we either reach a node that matches the terminating NUL (a hit), or follow a null pointer (a miss).

A TST acts a bit like a 256-way trie, except that instead of storing an array of 256 outgoing pointers, we build something similar to a small binary search tree for the next character. Note that no explicit balancing is done within these binary search trees. From a theoretical perspective, the worst case is that we get a 256-node deep linked-list equivalent at each step, multiplying our search time by $256 = O(1)$. In practice, only those characters that actual appear in some key at this stage will show up, so the $O(1)$ is likely to be a small $O(1)$, especially if keys are presented in random order.

TSTs are one of the fastest known data structures for implementing dictionaries using strings as keys, beating both hash tables and tries in most cases. They can be slower than Patricia trees if the keys have many keys with long matching prefixes; however, a Patricia-like optimization can be applied to give a **compressed ternary search tree** that works well even in this case. In practice, regular TSTs are usually good enough.

Here is a simple implementation of an insert-only TST. The C code includes two versions of the insert helper routine; the first is the original recursive version and the second is an iterative version generated by eliminating the tail recursion from the first.

```
typedef struct tst_node *TST;

#define EMPTY_TST (0)

/* returns 1 if t contains target */
int tst_contains(TST t, const char *target);

/* add a new key to a TST */
/* and return the new TST */
TST tst_insert(TST t, const char *key);

/* free a TST */
void tst_destroy(TST);
```

examples/trees/tst/tst.h

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

#include "tst.h"

struct tst_node {
    char key;                   /* value to split on */
    struct tst_node *lt;        /* go here if target[index] < value */
    struct tst_node *eq;        /* go here if target[index] == value */
    struct tst_node *gt;        /* go here if target[index] > value */
};

/* returns 1 if t contains key */
```

```c
int
tst_contains(TST t, const char *key)
{
    assert(key);

    while(t) {
        if(*key < t->key) {
            t = t->lt;
        } else if(*key > t->key) {
            t = t->gt;
        } else if(*key == '\0') {
            return 1;
        } else {
            t = t->eq;
            key++;
        }
    }

    return 0;
}

/* original recursive insert */
static void
tst_insert_recursive(TST *t, const char *key)
{
    if(*t == 0) {
        *t = malloc(sizeof(**t));
        assert(*t);
        (*t)->key = *key;
        (*t)->lt = (*t)->eq = (*t)->gt = 0;
    }

    /* now follow search */
    if(*key < (*t)->key) {
        tst_insert_recursive(&(*t)->lt, key);
    } else if(*key > (*t)->key) {
        tst_insert_recursive(&(*t)->gt, key);
    } else if(*key == '\0') {
        /* do nothing, we are done */
        ;
    } else {
        tst_insert_recursive(&(*t)->eq, key+1);
    }
}

/* iterative version of above, since somebody asked */
```

```c
/* This is pretty much standard tail-recursion elimination: */
/* The whole function gets wrapped in a loop, and recursive
 * calls get replaced by assignment */
static void
tst_insert_iterative(TST *t, const char *key)
{
    for(;;) {
        if(*t == 0) {
            *t = malloc(sizeof(**t));
            assert(*t);
            (*t)->key = *key;
            (*t)->lt = (*t)->eq = (*t)->gt = 0;
        }

        /* now follow search */
        if(*key < (*t)->key) {
            t = &(*t)->lt;
        } else if(*key > (*t)->key) {
            t = &(*t)->gt;
        } else if(*key == '\0') {
            /* do nothing, we are done */
            return;
        } else {
            t = &(*t)->eq;
            key++;
        }
    }
}


/* add a new key to a TST */
/* and return the new TST */
TST
tst_insert(TST t, const char *key)
{
    assert(key);

#ifdef USE_RECURSIVE_INSERT
    tst_insert_recursive(&t, key);
#else
    tst_insert_iterative(&t, key);
#endif
    return t;
}

/* free a TST */
```

```c
void
tst_destroy(TST t)
{
    if(t) {
        tst_destroy(t->lt);
        tst_destroy(t->eq);
        tst_destroy(t->gt);
        free(t);
    }
}
```

examples/trees/tst/tst.c

And here is some test code, almost identical to the test code for tries: test_tst.c.

The *Dr. Dobb's* article contains additional code for doing deletions and partial matches, plus some optimizations for inserts.

5.16.1.4 More information

- http://imej.wfu.edu/articles/2002/2/02/index.asp has some good Java-based animations of radix tries, Patricia tries, and other tree-like data structures.

5.16.2 Radix sort

The standard `quicksort` routine is an example of a **comparison-based sorting algorithm**. This means that the only information the algorithm uses about the items it is sorting is the return value of the `compare` routine. This has a rather nice advantage of making the algorithm very general, but has the disadvantage that the algorithm can extract only one bit of information from every call to `compare`. Since there are $n!$ possible ways to reorder the input sequence, this means we need at least $\log(n!) = O(n \log n)$ calls to `compare` to finish the sort. If we are sorting something like strings, this can get particularly expensive, because calls to `strcmp` may take time linear in the length of the strings being compared. In the worst case, sorting n strings of length m each could take $O(nm \log n)$ time.

5.16.2.1 Bucket sort The core idea of radix sort is that if we want to sort values from a small range, we can do it by making one bucket for each possible value and throw any object with that value into the corresponding bucket. In the old days, when Solitaire was a game played with physical pieces of cardboard, a player who suspected that that one of these "cards" had fallen under the couch might sort the deck by dividing it up into Spades, Hearts, Diamonds, and Club piles and then sorting each pile recursively. The same trick works in a computer, but there the buckets are typically implemented as an array of some convenient data structure.

If the number of possible values is too big, we may still be able to use bucket sort digit-by-digit. The resulting algorithms are known generally as **radix sort**. These are a class of algorithms designed for sorting strings in lexicographic order—the order used by dictionaries where one string is greater than another if the first character on which they differ is greater. One particular variant, **most-significant-byte radix sort** or MSB radix sort, has the beautiful property that its running time is not only linear in the size of the input in bytes, but is also linear in the smallest number of characters in the input that need to be examined to determine the correct order. This algorithm is so fast that it takes not much more time to sort data than it does to read the data from memory and write it back. But it's a little trickier to explain that the original **least-significant-byte radix sort** or LSB radix sort.

5.16.2.2 Classic LSB radix sort This is the variant used for punch cards, and works well for fixed-length strings. The idea is to sort on the least significant position first, then work backwards to the most significant position. This works as long as each sort is *stable*, meaning that it doesn't reorder values with equal keys. For example, suppose we are sorting the strings:

```
sat
bat
bad
```

The first pass sorts on the third column, giving:

```
bad
sat
bat
```

The second pass sorts on the second column, producing no change in the order (all the characters are the same). The last pass sorts on the first column. This moves the `s` after the two `b`s, but preserves the order of the two words starting with `b`. The result is:

```
bad
bat
sat
```

There are three downsides to LSB radix sort:

1. All the strings have to be the same length (this is not necessarily a problem if they are really fixed-width data types like `int`s).
2. The algorithm used to sort each position must be stable, which may require more effort than most programmers would like to take.
3. It may be that the late positions in the strings don't affect the order, but we have to sort on them anyway. If we are sorting

`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` and `baaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`, we spend a lot of time matching up `a`s against each other.

5.16.2.3 MSB radix sort For these reasons, MSB radix sort is used more often. This is basically the radix sort version of quicksort, where instead of partitioning our input data into two piles based on whether each element is less than or greater than a random pivot, we partition the input into 256 piles, one for each initial byte. We can then sort each pile recursively using the same algorithm, taking advantage of the fact that we know that the first byte (or later, the first k bytes) are equal and so we only need to look at the next one. The recursion stops when we get down to 1 value, or in practice where we get down to a small enough number of values that the cost of doing a different sorting algorithm becomes lower than the cost of creating and tearing down the data structures for managing the piles.

5.16.2.3.1 Issues with recursion depth The depth of recursion for MSB radix sort is equal to the length of the second-longest string in the worst case. Since strings can be pretty long, this creates a danger of blowing out the stack. The solution (as in quicksort) is to use tail recursion for the largest pile. Now any pile we recurse into with an actual procedure call is at most half the size of the original pile, so we get stack depth at most $O(\log n)$.

5.16.2.3.2 Implementing the buckets There is a trick we can do analagous to the Dutch flag algorithm where we sort the array in place. The idea is that we first count the number of elements that land in each bucket and assign a block of the array for each bucket, keeping track in each block of an initial prefix of values that belong in the bucket with the rest not yet processed. We then walk through the buckets swapping out any elements at the top of the good prefix to the bucket they are supposed to be in. This procedure puts at least one element in the right bucket for each swap, so we reorder everything correctly in at most n swaps and $O(n)$ additional work.

To keep track of each bucket, we use two pointers `bucket[i]` for the first element of the bucket and `top[i]` for the first unused element. We could make these be integer array indices, but this slows the code down by about 10%. This seems to be a situation where our use of pointers is complicated enough that the compiler can't optimize out the array lookups.

5.16.2.3.3 Further optimization Since we are detecting the heaviest bucket anyway, there is a straightforward optimization that speeds the sort up noticeably on inputs with a lot of duplicates: if everything would land in the same bucket, we can skip the bucket-sort and just go directly to the next character.

5.16.2.3.4 Sample implementation

Here is an implementation of MSB radix sort using the ideas above:

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

#include "radixSort.h"

/* in-place MSB radix sort for null-terminated strings */

/* helper routine for swapping */
static void
swapStrings(const char **a, const char **b)
{
    const char *temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

/* this is the internal routine that assumes all strings are equal for the
 * first k characters */
static void
radixSortInternal(int n, const char **a, int k)
{
    int i;
    int count[UCHAR_MAX+1];  /* number of strings with given character in position k */
    int mode;                /* most common position-k character */
    const char **bucket[UCHAR_MAX+1]; /* position of character block in output */
    const char **top[UCHAR_MAX+1];    /* first unused index in this character block */

    /* loop implements tail recursion on most common character */
    while(n > 1) {

        /* count occurrences of each character */
        memset(count, 0, sizeof(int)*(UCHAR_MAX+1));

        for(i = 0; i < n; i++) {
            count[(unsigned char) a[i][k]]++;
        }

        /* find the most common nonzero character */
        /* we will handle this specially */
        mode = 1;
```

```
for(i = 2; i < UCHAR_MAX+1; i++) {
    if(count[i] > count[mode]) {
        mode = i;
    }
}

if(count[mode] < n) {

    /* generate bucket and top fields */
    bucket[0] = top[0] = a;
    for(i = 1; i < UCHAR_MAX+1; i++) {
        top[i] = bucket[i] = bucket[i-1] + count[i-1];
    }

    /* reorder elements by k-th character */
    /* this is similar to dutch flag algorithm */
    /* we start at bottom character and swap values out until everything is in place
    /* invariant is that for all i, bucket[i] <= j < top[i] implies a[j][k] == i */
    for(i = 0; i < UCHAR_MAX+1; i++) {
        while(top[i] < bucket[i] + count[i]) {
            if((unsigned char) top[i][0][k] == i) {
                /* leave it in place, advance bucket */
                top[i]++;
            } else {
                /* swap with top of appropriate block */
                swapStrings(top[i], top[(unsigned char) top[i][0][k]]++);
            }
        }
    }

    /* we have now reordered everything */
    /* recurse on all but 0 and mode */
    for(i = 1; i < UCHAR_MAX+1; i++) {
        if(i != mode) {
            radixSortInternal(count[i], bucket[i], k+1);
        }
    }

    /* tail recurse on mode */
    n = count[mode];
    a = bucket[mode];
    k = k+1;

} else {

    /* tail recurse on whole pile */
```

```c
            k = k+1;
        }
    }
}

void
radixSort(int n, const char **a)
{
    radixSortInternal(n, a, 0);
}
```

examples/sorting/radixSort/radixSort.c

Some additional files: radixSort.h, test_radixSort.c, Makefile, sortInput.c. The last is a program that sorts lines on `stdin` and writes the result to `stdout`, similar to the GNU `sort` utility. When compiled with `-O3` and run on my machine, this runs in about the same time as the standard `sort` program when run on a 4.7 million line input file consisting of a random shuffle of 20 copies of `/usr/share/dict/words`, provided `sort` is run with `LANG=C sort < /usr/share/dict/words` to keep it from having to deal with locale-specific collating issues. On other inputs, `sort` is faster. This is not bad given how thoroughly `sort` has been optimized, but is a sign that further optimization is possible.

6 Other topics not covered in detail in 2015

These are mostly leftovers from previous versions of the class where different topics were emphasized.

6.1 More applications of function pointers

Here we show how to implement various mechanisms often found in more sophisticated programming languages in C using function pointers.

6.1.1 Iterators

Suppose we have an abstract data type that represents some sort of container, such as a list or dictionary. We'd like to be able to do something to every element of the container; say, count them up. How can we write operations on the abstract data type to allow this, without exposing the implementation?

To make the problem more concrete, let's suppose we have an abstract data type that represents the set of all non-negative numbers less than some fixed bound. The core of its interface might look like this:

```c
/*
 * Abstract data type representing the set of numbers from 0 to
 * bound-1 inclusive, where bound is passed in as an argument at creation.
 */
typedef struct nums *Nums;

/* Create a Nums object with given bound. */
Nums nums_create(int bound);

/* Destructor */
void nums_destroy(Nums);

/* Returns 1 if nums contains element, 0 otherwise */
int nums_contains(Nums nums, int element);
```

```c
#include <stdlib.h>
#include "nums.h"

struct nums {
    int bound;
};

Nums nums_create(int bound)
{
    struct nums *n;
    n = malloc(sizeof(*n));
    n->bound = bound;
    return n;
}

void nums_destroy(Nums n) { free(n); }

int nums_contains(Nums n, int element)
{
    return element >= 0 && element < n->bound;
}
```

From the outside, a `Nums` acts like the set of numbers from `0` to `bound - 1`; `nums_contains` will insist that it contains any `int` that is in this set and contains no `int` that is not in this set.

Let's suppose now that we want to loop over all elements of some `Nums`, say to add them together. In particular, we'd like to implement the following pseudocode, where `nums` is some `Nums` instance:

```c
sum = 0;
for(each i in nums) {
    sum += i;
```

```
}
```

One way to do this would be to build the loop into some operation in `nums.c`, including its body. But we'd like to be able to substitute any body for the `sum += i` line. Since we can't see the inside of a `Nums`, we need to have some additional operation or operations on a `Nums` that lets us write the loop. How can we do this?

6.1.1.1 Option 1: Function that returns a sequence

A data-driven approach might be to add a `nums_contents` function that returns a sequence of all elements of some instance, perhaps in the form of an array or linked list. The advantage of this approach is that once you have the sequence, you don't need to worry about changes to (or destruction of) the original object. The disadvantage is that you have to deal with storage management issues, and have to pay the costs in time and space of allocating and filling in the sequence. This can be particularly onerous for a "virtual" container like `Nums`, since we could conceivably have a `Nums` instance with billions of elements.

Bearing these facts in mind, let's see what this approach might look like. We'll define a new function `nums_contents` that returns an array of `int`s, terminated by a `-1` sentinel:

```c
int *
nums_contents(Nums n)
{
    int *a;
    int i;
    a = malloc(sizeof(*a) * (n->bound + 1));
    for(i = 0; i < n->bound; i++) a[i] = i;
    a[n->bound] = -1;
    return a;
}
```

We might use it like this:

```c
    sum = 0;
    contents = nums_contents(nums);
    for(p = contents; *p != -1; p++) {
        sum += *p;
    }
    free(contents);
```

Despite the naturalness of the approach, returning a sequence in this case leads to the *most* code complexity of the options we will examine.

6.1.1.2 Option 2: Iterator with first/done/next operations

If we don't want to look at all the elements at once, but just want to process them one at a time, we can build an *iterator*. An iterator is an object that allows you to step through the contents of another object, by providing convenient operations for getting the first element, testing when you are done, and getting the next element if you are not. In C, we try to design iterators to have operations that fit well in the top of a `for` loop.

For the `Nums` type, we'll make each `Nums` its own iterator. The new operations are given here:

```
int nums_first(Nums n) { return 0; }
int nums_done(Nums n, int val) { return val >= n->bound; }
int nums_next(Nums n, int val) { return val+1; }
```

And we use them like this:

```
    sum = 0;
    for(i = nums_first(nums); !nums_done(nums, i); i = nums_next(nums, i)) {
        sum += i;
    }
```

Not only do we completely avoid the overhead of building a sequence, we also get much cleaner code. It helps in this case that all we need to find the next value is the previous one; for a more complicated problem we might have to create and destroy a separate iterator object that holds the state of the loop. But for many tasks in C, the first/done/next idiom is a pretty good one.

6.1.1.3 Option 3: Iterator with function argument Suppose we have a very complicated iteration, say one that might require several nested loops or even a recursion to span all the elements. In this case it might be very difficult to provide first/done/next operations, because it would be hard to encode the state of the iteration so that we could easily pick up in the next operation where we previously left off. What we'd really like to do is to be able to plug arbitrary code into the innermost loop of our horrible iteration procedure, and do it in a way that is reasonably typesafe and doesn't violate our abstraction barrier. This is a job for function pointers, and an example of the *functional programming style* in action.

We'll define a `nums_foreach` function that takes a function as an argument:

```
void nums_foreach(Nums n, void (*f)(int, void *), void *f_data)
{
    int i;
    for(i = 0; i < n->bound; i++) f(i, f_data);
}
```

The `f_data` argument is used to pass extra state into the passed-in function `f`; it's a `void *` because we want to let `f` work on any sort of extra state it likes.

Now to do our summation, we first define an extra function `sum_helper`, which adds each element to an accumulator pointed to by `f_data`:

```
static void sum_helper(int i, void *f_data)
{
    *((int *) f_data) += i;
}
```

We then feed `sum_helper` to the `nums_foreach` function:

```
    sum = 0;
    nums_foreach(nums, sum_helper, (void *) &sum);
```

There is a bit of a nuisance in having to define the auxiliary `sum_helper` function and in all the casts to and from `void`, but on the whole the complexity of this solution is not substantially greater than the first/done/next approach. Which you should do depends on whether it's harder to encapsulate the state of the iterator (in which case the functional approach is preferable) or of the loop body (in which case the first/done/next approach is preferable), and whether you need to bail out of the loop early (which would require special support from the `foreach` procedure, perhaps checking a return value from the function). However, it's almost always straightforward to encapsulate the state of a loop body; just build a `struct` containing all the variables that it uses, and pass a pointer to this struct as `f_data`.

6.1.2 Closures

A **closure** is a function plus some associated state. A simple way to implement closures in C is to use a `static` local variable, but then you only get one. Better is to allocate the state somewhere and pass it around with the function. For example, here's a simple functional implementation of infinite sequences:

```
/* a sequence is an object that returns a new value each time it is called */
struct sequence {
    int (*next)(void *data);
    void *data;
};

typedef struct sequence *Sequence;

Sequence
create_sequence(int (*next)(void *data), void *data)
{
    Sequence s;

    s = malloc(sizeof(*s));
    assert(s);
```

```c
    s->next = next;
    s->data = data;

    return s;
}

int
sequence_next(Sequence s)
{
    return s->next(s->data);
}
```

And here are some examples of sequences:

```c
/* make a constant sequence that always returns x */
static int
constant_sequence_next(void *data)
{
    return *((int *) data);
}

Sequence
constant_sequence(int x)
{
    int *data;

    data = malloc(sizeof(*data));
    if(data == 0) return 0;

    *data = x;

    return create_sequence(constant_sequence_next, data);
}

/* make a sequence x, x+a, x+2*a, x+3*a, ... */
struct arithmetic_sequence_data {
    int cur;
    int step;
};

static int
arithmetic_sequence_next(void *data)
{
    struct arithmetic_sequence_data *d;

    d = data;
```

```
    d->cur += d->step;

    return d->cur;
}

Sequence
arithmetic_sequence(int x, int a)
{
    struct arithmetic_sequence_data *d;

    d = malloc(sizeof(*d));
    if(d == 0) return 0;

    d->cur = x - a;             /* back up so first value returned is x */
    d->step = a;

    return create_sequence(arithmetic_sequence_next, d);
}

/* Return the sum of two sequences */
static int
add_sequences_next(void *data)
{
    Sequence *s;

    s = data;
    return sequence_next(s[0]) + sequence_next(s[1]);
}

Sequence
add_sequences(Sequence s0, Sequence s1)
{
    Sequence *s;

    s = malloc(2*sizeof(*s));
    if(s == 0) return 0;

    s[0] = s0;
    s[1] = s1;

    return create_sequence(add_sequences_next, s);
}

/* Return the sequence x, f(x), f(f(x)), ... */
struct iterated_function_sequence_data {
    int x;
```

```
    int (*f)(int);
}

static int
iterated_function_sequence_next(void *data)
{
    struct iterated_function_sequence_data *d;
    int retval;

    d = data;

    retval = d->x;
    d->x = d->f(d->x);

    return retval;
}

Sequence
iterated_function_sequence(int (*f)(int), int x0)
{
    struct iterated_function_sequence_data *d;

    d = malloc(sizeof(*d));
    if(d == 0) return 0;

    d->x = x0;
    d->f = f;

    return create_sequence(iterated_function_sequence_next, d);
}
```

Note that we haven't worried about how to free the `data` field inside a `Sequence`, and indeed it's not obvious that we can write a generic data-freeing routine since we don't know what structure it has. The solution is to add more function pointers to a `Sequence`, so that we can get the next value, get the sequence to destroy itself, etc. When we do so, we have gone beyond building a closure to building an **object**.

6.1.3 Objects

Here's an example of a hierarchy of counter objects. Each counter object has (at least) three operations: `reset`, `next`, and `destroy`. To call the `next` operation on counter `c` we include `c` and the first argument, e.g. `c->next(c)` (one could write a wrapper to enforce this).

The main trick is that we define a basic counter structure and then extend it to include additional data, using lots of pointer conversions to make everything work.

```c
/* use preprocessor to avoid rewriting these */
#define COUNTER_FIELDS  \
    void (*reset)(struct counter *);    \
    int (*next)(struct counter *);      \
    void (*destroy)(struct counter *);

struct counter {
    COUNTER_FIELDS
};

typedef struct counter *Counter;

/* minimal counter--always returns zero */
/* we don't even allocate this, just have one global one */
static void noop(Counter c) { ; }
static int return_zero(Counter c) { return 0; }
static struct counter Zero_counter = { noop, return_zero, noop };

Counter
make_zero_counter(void)
{
    return &Zero_counter;
}

/* a fancier counter that iterates a function sequence */
/* this struct is not exported anywhere */
struct ifs_counter {

    /* copied from struct counter declaration */
    COUNTER_FIELDS

    /* new fields */
    int init;
    int cur;
    int (*f)(int);      /* update rule */
};

static void
ifs_reset(Counter c)
{
    struct ifs_counter *ic;

    ic = (struct ifs_counter *) c;
```

```
    ic->cur = ic->init;
}

static void
ifs_next(Counter c)
{
    struct ifs_counter *ic;
    int ret;

    ic = (struct ifs_counter *) c;

    ret = ic->cur;
    ic->cur = ic->f(ic->cur);

    return ret;
}

Counter
make_ifs_counter(int init, int (*f)(int))
{
    struct ifs_counter *ic;

    ic = malloc(sizeof(*ic));

    ic->reset = ifs_reset;
    ic->next = ifs_next;
    ic->destroy = (void (*)(struct counter *)) free;

    ic->init = init;
    ic->cur = init;
    ic->f = f;

    /* it's always a Counter on the outside */
    return (Counter) ic;
}
```

A typical use might be

```
static int
times2(int x)
{
    return x*2;
}

void
print_powers_of_2(void)
```

```c
{
    int i;
    Counter c;

    c = make_ifs_counter(1, times2);

    for(i = 0; i < 10; i++) {
        printf("%d\n", c->next(c));
    }

    c->reset(c);

    for(i = 0; i < 20; i++) {
        printf("%d\n", c->next(c));
    }

    c->destroy(c);
}
```

6.2 Suffix arrays

These are notes on practical implementations of suffix arrays, which are a data structure for searching quickly for substrings of a given large string.

6.2.1 Why do we want to do this?

- Answer from the old days: Fast string searching is the key to dealing with mountains of information. Why, a modern (c. 1960) electronic computer can search the equivalent of hundreds of pages of text in just a few hours. . .
- More recent answer:
  - We still need to search enormous corpuses of text (see http://www.google.com).
  - Algorithms for finding long repeated substrings or patterns can be useful for data compression) or detecting plagiarism.
  - Finding all occurrence of a particular substring in some huge corpus is the basis of statistical machine translation.
  - We are made out of strings over a particular finite alphabet GATC. String search is a central tool in computational biology.

6.2.2 String search algorithms

Without preprocessing, searching an n-character string for an m-character substring can be done using algorithms of varying degrees of sophistication, the worst of which run in time $O(nm)$ (run `strncmp` on each position in the big string), and best of which run in time $O(n+m)$ (run the Boyer-Moore string search algorithm). But we are interested in the case where we can preprocess our big string into a data structure that will let us do lots of searches for cheap.

6.2.3 Suffix trees and suffix arrays

Suffix trees and **suffix arrays** are data structures for representing texts that allow substring queries like "where does this pattern appear in the text" or "how many times does this pattern occur in the text" to be answered quickly. Both work by storing all suffixes of a text, where a *suffix* is a substring that runs to the end of the text. Of course, storing actual copies of all suffixes of an n-character text would take $O(n^2)$ space, so instead each suffix is represented by a pointer to its first character.

A suffix array stores all the suffixes sorted in dictionary order. For example, the suffix array of the string `abracadabra` is shown below. The actual contents of the array are the indices in the left-hand column; the right-hand shows the corresponding suffixes.

```
11  \0
10  a\0
 7  abra\0
 0  abracadabra\0
 3  acadabra\0
 5  adabra\0
 8  bra\0
 1  bracadabra\0
 4  cadabra\0
 6  dabra\0
 9  ra\0
 2  racadabra\0
```

A suffix tree is similar, but instead using an array, we use some sort of tree data structure to hold the sorted list. A common choice given an alphabet of some fixed size k is a trie, in which each node at depth d represents a string of length d, and its up to k children represent all $(d+1)$-character extensions of the string. The advantage of using a suffix trie is that searching for a string of length m takes $O(m)$ time, since we can just walk down the trie at the rate of one node per character in m. A further optimization is to replace any long chain of single-child nodes with a *compressed* edge labeled with the concatenation all the characters in the chain. Such compressed suffix tries can not only be searched in linear time but can also be constructed in linear time with a sufficiently clever algorithm (we won't actually do this here). Of course, we could also use a simple balanced binary tree, which might be preferable if the alphabet is large.

The disadvantage of suffix trees over suffix arrays is that they generally require more space to store all the internal pointers in the tree. If we are indexing a huge text (or collection of texts), this extra space may be too expensive.

6.2.3.1 Building a suffix array A straightforward approach to building a suffix array is to run any decent comparison-based sorting algorithm on the set of suffixes (represented by pointers into the text). This will take $O(n \log n)$ comparisons, but in the worst case each comparison will take $O(n)$ time, for a total of $O(n^2 \log n)$ time. This is the approach used in the sample code below.

The original suffix array paper by Manber and Myers ("Suffix arrays: a new method for on-line string searches," SIAM Journal on Computing 22(5):935-948, 1993) gives an $O(n \log n)$ algorithm, somewhat resembling radix sort, for building suffix arrays in place with only a small amount of additional space. They also note that for random text, simple radix sorting works well, since most suffixes become distinguishable after about $\log_k n$ characters (where k is the size of the alphabet); this gives a cost of $O(n \log n)$ to do the sort, since radix sort only looks at the bytes it needs to once. For a comparison-based sort, the same assumption gives an $O(n \log^2 n)$ running time; this is a factor of $\log n$ slower, but this may be acceptable if programmer time is more important.

The fastest approach is to build a suffix tree in $O(n)$ time and extract the suffix array by traversing the tree. The only complication is that we need the extra space to build the tree, although we get it back when we throw the tree away.

6.2.3.2 Searching a suffix array Suppose we have a suffix array corresponding to an n-character text and we want to find all occurrences in the text of an m-character pattern. Since the suffixes are ordered, the easiest solution is to do binary search for the first and last occurrences of the pattern (if any) using $O(\log n)$ comparisons. (The code below does something even lazier than this, searching for some match and then scanning linearly for the first and last matches.) Unfortunately, each comparison may take as much as $O(m)$ time, since we may have to check all m characters of the pattern. So the total cost will be $O(m \log n)$ in the worst case.

By storing additional information about the longest common prefix of consecutive suffixes, it is possible to avoid having to re-examine every character in the pattern for every comparison, reducing the search cost to $O(m + \log n)$. With a sufficiently clever algorithm, this information can be computed in linear time, and can also be used to solve quickly such problems as finding the longest duplicate substrings, or most frequently occurring strings. For more details, see (Gusfield, Dan. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997, §7.14.4]).

Using binary search on the suffix array, most searching tasks are now easy:

- Finding if a substring appears in the array uses binary search directly.

- Finding all occurrences requires two binary searches, one for the first occurrence and one for the last. If we only want to count the occurrences and not return their positions, this takes $O(m + \log n)$ time. If we want to return their positions, it takes $O(m + \log n + k)$ time, where k is the number of times the pattern occurs.
- Finding duplicate substrings of length m or more can be done by looking for adjacent entries in the array with long common prefixes, which takes $O(mn)$ time in the worst case if done naively (and $O(n)$ time if we have already computed longest common prefix information.

6.3 Burrows-Wheeler transform

Closely related to suffix arrays is the **Burrows-Wheeler transform** (Burrows and Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, DEC Systems Research Center Technical Report number 124, 1994), which is the basis for some of the best currently known algorithms for text compression (it's the technique that is used, for example, in bzip2).

The idea of the Burrows-Wheeler Transform is to construct an array whose rows are all cyclic shifts of the input string in dictionary order, and return the last column of the array. The last column will tend to have long runs of identical characters, since whenever some substring (like `the` appears repeatedly in the input, shifts that put the first character `t` in the last column will put the rest of the substring `he` in the first columns, and the resulting rows will tend to be sorted together. The relative regularity of the last column means that it will compress well with even very simple compression algorithms like run-length encoding.

Below is an example of the Burrows-Wheeler transform in action, with `$` marking end-of-text. The transformed value of `abracadabra$` is `$drcraaaabba`, the last column of the sorted array; note the long run of a's (and the shorter run of b's).

```
abracadabra$     abracadabra$
bracadabra$a     abra$abracad
racadabra$ab     acadabra$abr
acadabra$abr     adabra$abrac
cadabra$abra     a$abracadabr
adabra$abrac     bracadabra$a
dabra$abraca --> bra$abracada
abra$abracad     cadabra$abra
bra$abracada     dabra$abraca
ra$abracadab     racadabra$ab
a$abracadabr     ra$abracadab
$abracadabra     $abracadabra
```

The most useful property of the Burrows-Wheeler transform is that it can be inverted; this distinguishes it from other transforms that produce long runs like simply sorting the characters. We'll describe two ways to do this; the first is less efficient, but more easily grasped, and involves rebuilding the array one column at a time, starting at the left. Observe that the leftmost column is just all the characters in the string in sorted order; we can recover it by sorting the rightmost column, which we have to start off with. If we paste the rightmost and leftmost columns together, we have the list of all 2-character substrings of the original text; sorting this list gives the first *two* columns of the array. (Remember that each copy of the string wraps around from the right to the left.) We can then paste the rightmost column at the beginning of these two columns, sort the result, and get the first three columns. Repeating this process eventually reconstructs the entire array, from which we can read off the original string from any row. The initial stages of this process for `abracadabra$` are shown below:

```
$    a       $a   ab       $ab   abr
d    a       da   ab       dab   abr
r    a       ra   ac       rac   aca
c    a       ca   ad       cad   ada
r    a       ra   a$       ra$   a$a
a    b       ab   br       abr   bra
a -> b       ab -> br      abr -> bra
a    c       ac   ca       aca   cad
a    d       ad   da       ada   dab
b    r       br   ra       bra   rac
b    r       br   ra       bra   ra$
a    $       a$   $a       a$a   $ab
```

Rebuilding the entire array in this fashion takes $O(n^2)$ time and $O(n^2)$ space. In their paper, Burrows and Wheeler showed that one can in fact reconstruct the original string from just the first and last columns in the array in $O(n)$ time.

Here's the idea: Suppose that all the characters were distinct. Then after reconstructing the first column we would know all pairs of adjacent characters. So we could just start with the last character `$` and regenerate the string by appending at each step the unique successor to the last character so far. If all characters were distinct, we would never get confused about which character comes next.

The problem is what to do with pairs with duplicate first characters, like `ab` and `ac` in the example above. We can imagine that each `a` in the last column is labeled in some unique way, so that we can talk about the first `a` or the third `a`, but how do we know which `a` is the one that comes before `b` or `d`?

The trick is to look closely at how the original sort works. Look at the rows in the original transformation. If we look at all rows that start with `a`, the order they are sorted in is determined by the suffix after `a`. These suffixes also appear as the prefixes of the rows that *end* with `a`, since the rows that end with `a` are just the rows that start with `a` rotated one position. It follows that *all instances of the same letter occur in the same order in the first and last columns.* So if we use a stable sort to construct the first column, we will correctly match up instances of letters.

This method is shown in action below. Each letter is annotated uniquely with a count of how many identical letters equal or precede it. Sorting recovers the first column, and combining the last and first columns gives a list of unique pairs of adjacent annotated characters. Now start with `$1` and construct the full sequence `$1 a1 b1 r1 a3 c1 a4 d1 a2 b2 r2 a5 $1`. The original string is obtained by removing the end-of-string markers and annotations: `abracadabra`.

```
$1      a1
d1      a2
r1      a3
c1      a4
r2      a5
a1      b1
a2 --> b2
a3      c1
a4      d1
b1      r1
b2      r2
a5      $1
```

Because we are only sorting single characters, we can perform the sort in linear time using counting sort. Extracting the original string also takes linear time if implemented reasonably.

6.3.1 Suffix arrays and the Burrows-Wheeler transform

A useful property of the Burrows-Wheeler transform is that each row of the sorted array is essentially the same as the corresponding row in the suffix array, except for the rotated string prefix after the `$` marker. This means, among other things, that we can compute the Burrows-Wheeler transform in linear time using suffix trees. Ferragina and Manzini (http://people.unipmn.it/~manzini/papers/focs00draft.pdf) have further exploited this correspondence (and some very clever additional ideas) to design compressed suffix arrays that compress and index a text at the same time, so that pattern searches can be done directly on the compressed text in time close to that needed for suffix array searches.

6.3.2 Sample implementation

As mentioned above, this is a pretty lazy implementation of suffix arrays, that doesn't include many of the optimizations that would be necessary to deal with huge source texts.

```c
/* we expose this so user can iterate through it */

struct suffixArray {
    size_t n;                /* length of string INCLUDING final null */
    const char *string;      /* original string */
    const char **suffix;     /* suffix array of length n */
};

typedef struct suffixArray *SuffixArray;

/* construct a suffix array */
/* it is a bad idea to modify string before destroying this */
SuffixArray suffixArrayCreate(const char *string);

/* destructor */
void suffixArrayDestroy(SuffixArray);

/* return number of occurrences of substring */
/* if non-null, index of first occurrence is place in first */
size_t
suffixArraySearch(SuffixArray, const char *substring, size_t *first);

/* return the Burrows-Wheeler transform of the underlying string
 * as malloc'd data of length sa->n */
/* note that this may have a null in the middle somewhere */
char *suffixArrayBWT(SuffixArray sa);

/* invert BWT of null-terminated string, returning a malloc'd copy of original */
char *inverseBWT(size_t len, const char *s);
```

examples/suffixArray/suffixArray.h

```c
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <limits.h>

#include "suffixArray.h"

static int
saCompare(const void *s1, const void *s2)
{
    return strcmp(*((const char **) s1), *((const char **) s2));
}

SuffixArray
suffixArrayCreate(const char *s)
```

```
{
    size_t i;
    SuffixArray sa;

    sa = malloc(sizeof(*sa));
    assert(sa);

    sa->n = strlen(s) + 1;
    sa->string = s;

    sa->suffix = malloc(sizeof(*sa->suffix) * sa->n);
    assert(sa->suffix);

    /* construct array of pointers to suffixes */
    for(i = 0; i < sa->n; i++) {
        sa->suffix[i] = s+i;
    }

    /* this could be a lot more efficient */
    qsort(sa->suffix, sa->n, sizeof(*sa->suffix), saCompare);

    return sa;
}

void
suffixArrayDestroy(SuffixArray sa)
{
    free(sa->suffix);
    free(sa);
}

size_t
suffixArraySearch(SuffixArray sa, const char *substring, size_t *first)
{
    size_t lo;
    size_t hi;
    size_t mid;
    size_t len;
    int cmp;

    len = strlen(substring);

    /* invariant: suffix[lo] <= substring < suffix[hi] */
    lo = 0;
    hi = sa->n;
```

```
    while(lo + 1 < hi) {
        mid = (lo+hi)/2;
        cmp = strncmp(sa->suffix[mid], substring, len);

        if(cmp == 0) {
            /* we have a winner */
            /* search backwards and forwards for first and last */
            for(lo = mid; lo > 0 && strncmp(sa->suffix[lo-1], substring, len) == 0; lo--);
            for(hi = mid; hi < sa->n && strncmp(sa->suffix[hi+1], substring, len) == 0; hi+

            if(first) {
                *first = lo;
            }

            return hi - lo + 1;
        } else if(cmp < 0) {
            lo = mid;
        } else {
            hi = mid;
        }
    }

    return 0;
}

char *
suffixArrayBWT(SuffixArray sa)
{
    char *bwt;
    size_t i;

    bwt = malloc(sa->n);
    assert(bwt);

    for(i = 0; i < sa->n; i++) {
        if(sa->suffix[i] == sa->string) {
            /* wraps around to nul */
            bwt[i] = '\0';
        } else {
            bwt[i] = sa->suffix[i][-1];
        }
    }

    return bwt;
}
```

```c
char *
inverseBWT(size_t len, const char *s)
{
    /* basic trick: stable sort of s gives successor indices */
    /* then we just thread through starting from the nul */

    size_t *successor;
    int c;
    size_t count[UCHAR_MAX+1];
    size_t offset[UCHAR_MAX+1];
    size_t i;
    char *ret;
    size_t thread;

    successor = malloc(sizeof(*successor) * len);
    assert(successor);

    /* counting sort */
    for(c = 0; c <= UCHAR_MAX; c++) {
        count[c] = 0;
    }

    for(i = 0; i < len; i++) {
        count[(unsigned char) s[i]]++;
    }

    offset[0] = 0;

    for(c = 1; c <= UCHAR_MAX; c++) {
        offset[c] = offset[c-1] + count[c-1];
    }

    for(i = 0; i < len; i++) {
        successor[offset[(unsigned char) s[i]]++] = i;
    }

    /* find the nul */
    for(thread = 0; s[thread]; thread++);

    /* thread the result */
    ret = malloc(len);
    assert(ret);

    for(i = 0, thread = successor[thread]; i < len; i++, thread = successor[thread]) {
        ret[i] = s[thread];
    }
```

```
    return ret;
}
```

examples/suffixArray/suffixArray.c

Here is a Makefile and test code: Makefile, testSuffixArray.c.

The output of `make test` shows all occurrences of a target string, the Burrows-Wheeler transform of the source string (second-to-last line), and its inversion (last line, which is just the original string):

```
$ make test
/bin/echo -n abracadabra-abracadabra-shmabracadabra | ./testSuffixArray abra
Count: 6
abra
abra-abr
abra-shm
abracada
abracada
abracada
aaarrrdddm\x00-rrrcccaaaaaaaaaaaashbbbbbb-
abracadabra-abracadabra-shmabracadabra
```

6.4 C++

Here we will describe some basic features of C++ that are useful for implementing abstract data types. Like all programming languages, C++ comes with an ideology, which in this case emphasizes object-oriented features like inheritance. We will be ignoring this ideology and treating C++ as an improved version of C.

The goal here is not to teach you all of C++, which would take a while, but instead to give you some hints for why you might want to learn C++ on your own. If you decide to learn C++ for real, Bjarne Stroustrup's *The C++ Programming Language* is the definitive source. A classic tutorial here aimed at C programmers introduces C++ features one at a time (some of these features have since migrated into C). The web site http://www.cplusplus.com has extensive tutorials and documentation.

6.4.1 Hello world

The C++ version of "hello world" looks like this:

```
#include <iostream>

int
main(int argc, const char **argv)
```