CS 61C Lecture Notes Week 3

Topic: C: More Pointers, Arrays, Structs; Memory Allocation

Reading: Kernighan & Ritchie, Chapters 5–8

Arrays Aren't First-Class

Consider the following plausible but illegal C program fragment:

```
int a[3] = {87, 781, 440};
int b[3];

b = a;
```

We'd like this to mean, "assign to `b` the value(s) of the array `a`." (I've put the "s" of "value(s)" in parentheses because you might prefer to think of `a` as containing one array value, or three integer values.) But it doesn't work, because arrays aren't first-class data in C.

Here's a reminder of what "first-class" means. A first-class datum can be
- the value of a variable
- an argument to a procedure
- the return value from a procedure
- a member of an aggregate

But none of these is true for arrays in C. This may sound strange, because there are notations in C that look like the use of arrays as variables; as one example, I've mentioned the notation `char *argv[]` to declare `argv` as an array of pointers to characters. Isn't `argv` a variable?

The short answer: Yes, `argv` is a variable, but it isn't an array; it's a pointer. For a longer answer, consider this program fragment:

```
int foo(char *a, char b[5]) {
    char *c;
    char d[5];

    ...
    foo(c, d)
    ...
}
```

In this program, `a`, `b`, and `c` are all variables, of type pointer-to-char; `d` is not a variable, but rather an array of five characters. (The C compiler pays no attention to the `5` in the declaration of the argument `b`!)

Well, if `b` and `d` are two different kinds of thing, how do we make sense of the recursive call in the body of `foo`? It's legal, but it doesn't mean what you think it does; since there is no variable named `d`, when the array name `d` is used without a subscript, as it is in the recursive call, it is an abbreviation for the *constant* expression `&d[0]`. This expression, like all ampersand operations, gives a result of type pointer, which matches the formal parameter `b`.

This is why we can't say `b=a` in the program with which we started today. Since (in that program) `a` and `b` are both actual arrays, the statement would mean

```
&b[0] = &a[0];
```

But `&b[0]` is a *constant* expression, just like the number `4`, not a variable. (Why? Because even if you assign a new *value* to the array element `b[0]`, that element doesn't *move around* in memory; its address remains the same.) You can't use a constant expression on the left side of an assignment; you can't say

```
4 = 5;
```

Note: An array *element*, such as `b[2]`, *is* a variable. You can assign a value to it; you can use it as an argument or a return value of a procedure.

Most of the confusing nature of pointers and arrays in C can be explained as a direct consequence of what I've said so far. If you remember that arrays aren't first-class, the rest follows.

Arrays and Procedures

You'd often like to use arrays as arguments or return values of procedures. This is particularly common for character strings, which are just arrays of characters. For example, consider the following, simplified from a standard C library function:

```
char *now();
```

This function returns the current date and time, as a character string in the format

```
Sun Sep  7 21:35:12 2003
```

We'd like this procedure to return a string, which we'd like to be a first-class data type. Alas, what it returns is a *pointer* to a string — more precisely, a pointer to the first character of a string — and if you look up the model for this procedure, the C library function `ctime`, on page 256 of K&R, you'll find the ominous sentence "The next four functions return pointers to static objects that may be overwritten by other calls." What this means is that you can't say

```
char *t1, *t2;

t1 = now();
/* some lengthy computation */
t2 = now();
```

because after the second call to `now`, `t1` and `t2` will both represent the later time. In fact, the pointers `t1` and `t2` have the same value; they point to the same location in memory.

Why on earth did they design this function so badly? Why can't you keep records of various times, getting a fresh string for each?

• **Types of Storage.** When a C program is running, its memory is divided into four big chunks.

```
          +------------+
infinity  |   stack    |
          |     |      |
          |     |      |
          |     V      |
          :            :
          :            :
          :            :
          |     ^      |
          |     |      |
          |     |      |
          |    heap    |
          +------------+
          |   global   |
          |    data    |
          +------------+
          |  program   |
zero      |    code    |
          +------------+
```

First come two chunks whose size is fixed when the program is compiled:

(1) Program code. All the machine language instructions for all the procedures in the program go here.

(2) Global data. The storage reserved for global variables (including whole-program globals and same-file globals) goes here, along with local variables declared `static` within a procedure. The name "global data" is traditional, but it could better be called the "infinite-extent data area."

The rest of memory is divided into two more chunks whose size varies as the program runs. To make this possible, the third chunk starts at the lowest available address (above the first two chunks) and grows upward, whereas the fourth starts at the highest possible address and grows downward.

(3) The heap. This area of memory is used for blocks of storage that the programmer explicitly asks to allocate, calling the `malloc` library procedure.

(4) The stack. This area of memory is used for local-extent ("automatic") local variables, including procedure arguments. Space on the stack is allocated whenever a procedure is called, and automatically freed when the procedure returns to its caller.

Okay, now think about procedure `now` or its C-library cousin `ctime`. It needs to create a character array of length 26. (The format of the returned string is fixed, so its length is always the same.) In which of the four parts of memory will it live? Certainly not in the program code area. Let's think about the other three in reverse order.

(4) The procedure could allocate stack space for the string, by declaring it as a local (automatic) variable:

```
char *now() {
    char timestring[26];

    ...
    return timestring;
}
```

But this is a really bad idea. Stack space is automatically freed when the procedure returns; the next procedure call will reuse the same memory for some different purpose. So we would be returning to the caller a pointer to *volatile* memory — that is, to memory that will mysteriously change to contain something entirely different, perhaps not even characters at all, without notice.

(3) It could allocate heap space:

```
char *now() {
    char *timestring;

    timestring = (char *)malloc(26);
    ...
    return timestring;
}
```

In many ways this is the best solution. It comes closest to simulating the behavior of first-class strings; each call to `now` allocates a fresh string, and repeated calls create independent strings. When Unix was first developed, though, its inventors rejected this solution for `ctime`. I'm guessing they had two reasons, one now obsolete, and one still valid. The obsolete one is that in the days of very small memories, programmers had a terrible fear of "memory leaks" — allocating blocks of memory and not freeing them. At least in Unix this danger affected only the application program that contained the leak; one of the many, many, many things wrong with the design of Windows is that memory leaks in application programs can make the allocated memory unavailable to later programs, until the system is rebooted! (This is less true in "Professional" versions of Windows than in the version you probably have at home.) The still-valid reason to avoid the use of `malloc` in library functions is that programs can write their own heap-allocation procedures instead of using the standard `malloc`. (There are various reasons why this might be a good idea for particular applications.) The C library functions should be usable even by non-`malloc`-using programs.

(2) That leaves global-area allocation, by declaring a `static` variable:

```
char *now() {
    static char timestring[26];

    ...
    return timestring;
}
```

This avoids `malloc`, and it allocates infinite-extent space that doesn't disappear when the procedure returns. But the downside of infinite-extent space is that it's allocated only once, when the program is compiled, and the same space is shared by every invocation of the procedure.

[Of course that isn't always a downside; sometimes a local state variable is just what you want:

```
int counter() {
    static int count = 0;

    return ++count;
}
```

This procedure returns 1 the first time it's called, 2 the second time, and so on.]

Since none of these three choices is completely satisfactory, many C procedures use a fourth possibility: Make the caller allocate the memory, by taking a pointer as an argument:

```
void now(char timestring[26]) {
    ...
    return;
}
```

(As you know, the compiler ignores the `26` in this definition; it's there as a form of program documentation, telling the user of the procedure that the argument, which is actually of type `char *`, should be a pointer to a character array of length 26.)

The trouble with this solution, although it's probably the most common in C programs, is that `now` is no longer a *function!* It no longer returns a value. That doesn't matter so much in this example, but let's now consider a more interesting example: vector arithmetic. We want to represent vectors in three-dimensional space using arrays of three `double`s. We'd like vectors to be first-class data, so that once we've written a `vector_add` function, we can compute the sum of three vectors by saying

```
typedef double vector[3];
typedef double *vecptr;

vector a={1,2,3}, b={40,50,60}, c={700,800,900};
vecptr d;

d = vector_add(a, vector_add(b, c));
```

But if `vector_add` is written in the typical C manner, it'll look like this:

```
void vector_add(vector x, vector y, vector z) {
    int i;

    for (i=0; i<3; i++)
      z[i] = x[i] + y[i];
}
```

and to compute $a + b + c$ we have to say

```
vector a, b, c, d, temp;
```

```
vector_add(b, c, temp);
vector_add(a, temp, d);
```

This is, I think, uglier and less clear than the desired version

```
d = vector_add(a, vector_add(b, c));
```

(Note, by the way, that despite its header, the arguments to `vector_add` are actually not of type `vector`, since a vector is an array, and arrays aren't first-class. The arguments are actually of type `vecptr`; they are pointers to the first element of the desired vectors.)

To get proper function behavior, `vector_add` must allocate heap space for its result:

```
vecptr vector_add(vector x, vector y) {
    int i;
    vecptr z = (vecptr)malloc(sizeof(vector));

    for (i=0; i<3; i++)
      z[i] = x[i] + y[i];
    return z;
}
```

(Note that C's clever pretense of using arrays as arguments, so I can say `vector x` in declaring a formal parameter, doesn't apply to local variables within the procedure. I have to say explicitly that `z` is a pointer, not an array, because if it were an array, the `return` statement would actually return `&z[0]`, a pointer to a block of memory that will no longer exist as soon as the procedure returns! Similarly, I can't pretend that the return value is an array.)

• **But *why* aren't arrays first class?**

That's a good question. You'll have to take the answer partly on faith for now, until we study how machine language works in more detail, but basically it's this: It's easy and fast to deal with *scalar* data types, the ones that fit in a single machine word, but slower and more complicated to handle larger types. We'll see, for example, that procedure calling is easier if the procedure's arguments and return value can each fit in a single processor register, and in fact specific registers are, by convention, reserved for these purposes.

Therefore, C restricts procedure arguments and return values to be of types that fit in a single register.

Well, sort of. A value of type `double` requires two registers, and so the calling conventions allow for that. And we'll see soon that C handles structs, another kind of data aggregate, quite differently. But the two paragraphs preceding this one were true enough when the C language was first designed.

• **Call by Value vs. Call by Reference**

Review: *Call by value* means that a procedure call passes a copy of the argument into the called procedure, so that any changes to the argument variable made inside the called procedure affect only it, not the caller.

Call by reference means that a procedure call passes a pointer to the original argument into the called procedure, so that changes made inside the called procedure do affect the value in the caller.

Perhaps an example will clarify the distinction. Suppose we have

```
int foo(int x) {
    x = x+5;
    return x;
}

main() {
    int a = 7, b;
```

```
    b = foo(a);
}
```

After the call to foo, we're sure that b has the value 12. But what about a? In a language using call by value, a will still contain 7; foo can't change it by assigning to x. But in a language with call by reference, a will be 12, because foo's variable x is just a synonym (i.e., a pointer) for a.

K&R tell us that C uses call by value. (See pp. 27, 202.) But there's a hint that arrays are passed by reference: "The story is different for arrays... By subscripting this value, the function can access and alter any element of the array." [Page 28.]

We can make sense of this contradiction by remembering that arrays actually can't be arguments at all! The argument is really a *pointer*. That pointer, like all arguments, is passed by value; that is, the procedure gets a copy of the pointer value, which is the address of the array. But the array itself isn't passed by value — it isn't copied — because the array itself isn't passed as an argument at all. The copied pointer points to the one array, which is shared by the caller and the callee.

Let's rewrite the example above so that it involves an array:

```
int foo(int x[]) {
    static int s[3] = {100, 200, 300};

    x[2] = x[2]+5;
    x = s;
    return x;
}

main() {
    int a[3] = {4, 5, 6}, *b;
    b = foo(a);
}
```

After the call to foo, the elements of a are {4, 5, 11}; the elements of b are {100, 200, 300}. In other words, the assignment to x[2] in foo does affect the caller's value of a, but the assignment to x does not affect a.

[This is exactly analogous to the fact that in Scheme, which uses call by value but in which all values are pointers, set! of a formal parameter in a called procedure does not change anything in the caller, but set-car! of a formal parameter, when the actual argument is a list, does make a change that's visible to the caller. Java, a subversive effort to get stick-in-the-mud C programmers to use Lisp ideas by disguising Lisp in C syntax, also has the rule that all aggregate values are pointers, but they didn't want to use the name "pointer" lest they scare people, which is why the Java documentation talks about "references" instead.]

• Pointer Arithmetic and Array Indexing

In procedure foo above, since the argument variable x is really a pointer rather than an array, how come we can index it by saying x[2]?

For any pointer, the expression p[i] is exactly equivalent to *(p+i). In other words, subscripting involves two operations: pointer arithmetic (p+i) and dereferencing (*).

Remember that the expression p+i doesn't actually add the integer i to the contents of p. It adds i*sizeof(*p) — that is, it adds enough to skip over i elements of the array, where sizeof(*p) is the size in bytes of one element.

Actually, even for array indexing, C has to do the same sort of computation, multiplying the index by the size of an element, and adding the product to the array's starting address.

The only pointer arithmetic we've seen so far is the addition of a pointer and an integer. Other possibilities are also allowed, but not everything imaginable. For example, you can't add two pointers. This isn't much of a restriction, since most of the time you'd have no reason to do so; try to think of a case where you'd want to add two memory addresses. (The name "address" comes from house addresses on a street, and these make a fairly decent analogy. Pointer-plus-integer means something like "three houses past number 2456," but what would it mean to add two house numbers?) You can, however, subtract two pointers, provided they point to the same type. This operation is useful when you have two pointers to elements of the same array, and you want to know how far apart those two elements are. The result of `p2-p1` is the arithmetic difference divided by the size of an array element, so the result is measured in number of elements, not in number of bytes. This makes pointer arithmetic consistent; if we say

```
        p2 = p1 + i;
        j = p2 - p1;
```

then `i` and `j` will be equal.

Actually there is one situation in which you'd like to be able to add two pointers: You want to find the array element midway between two pointers; for example, this is part of the algorithm for binary (dictionary) search of a sorted array:

```
int *binsearch(int target, int array[], int size) {
    int *low = array;
    int *high = low + size;
    int *mid;

    while (low < high) {
        mid = (low + high)/2;        /* THIS DOESN'T WORK */
        if (target == *mid)
            return mid;
        if (target < *mid)
            high = mid;
        else
            low = mid+1;
    }

    return NULL;
}
```

The commented line tries to compute the average of two pointers, in the most straightforward way, but it doesn't work because adding two pointers is illegal. Instead the computation must be done this way:

```
        mid = low + (high-low)/2;
```

Subtracting two pointers is legal, and the result of the subtraction is an integer, not a pointer, so it can be divided by two, and the resulting integer can be added to `low` to compute the average.

Structs *Are* First-Class, Though!

An array is a contiguous block of same-type elements. A *struct* is a more general data aggregate: a contiguous block of multiple-type elements.

Structs are similar in flavor to objects, in that they have named fields analogous to the instance variables of an object. But of course they aren't objects, because they don't have methods. For a first example, let's redo vectors, using a struct type instead of an array:

```
struct vector {
    double x, y, z;
}
```

This declares a type, analogous to a class definition, not an instance of the type. To do that we'd say

```
struct vector foo;
```

Note that the name of the new type is `struct vector`; if you get tired of typing `struct` all the time you can say

```
typedef struct vector vector;
```

which creates a type name `vector` synonymous with `struct vector`.

Unlike arrays, structs are first class! So this is legal:

```
        struct vector a,b;
        a = b;
```

Structs – not just pointers to structs – can also be used as arguments to procedures, and can be the value returned by a procedure:

```
vector vector_add(vector a, vector b) {
    vector result;

    result.x = a.x + b.x;
    result.y = a.y + b.y;
    result.z = a.z + b.z;
    return result;
}
```

Although the local variable `result` is in the stack frame for the invocation of `vector_add`, and will therefore be freed when `vector_add` returns, the `return` statement makes a *copy* of the struct, which is returned to the caller. (Where is the storage allocated for the copy? In the caller's stack frame.)

• Pointers to Structs

The downside of having first-class structs is that it's easy to write programs that require a lot of copying of data, without realizing it. You might, therefore, choose to use pointers to structs as arguments and return values, even though you could use the structs themselves.

```
typedef struct vector *vecptr;

vecptr vector_add(vecptr a, vecptr b) {
    vecptr result = (vecptr)malloc(sizeof struct vector);

    result->x = a->x + b->x;
    result->y = a->y + b->y;
    result->z = a->z + b->z;
    return result;
}
```

The notation `ptr->field` is just an abbreviation for the commonly used combination of dereferencing a pointer and accessing a field of the resulting struct, which could be written without the abbreviation as `(*ptr).field`.

One situation in which you'll certainly focus on pointers rather than on the structs is the use of structs to build recursive data structures such as linked lists. Here's an example of a struct for a node in a linked list of integers:

```
struct intlist {
    int value;
    struct intlist *next;
};
```

The `next` field has to be a pointer, not a struct, because we can't put nodes inside of nodes. If we said

```
struct intlist {
    int value;
    struct intlist next;        /* WRONG!! */
};
```

we would be trying to create a structure requiring an infinite amount of memory! An `intlist` would contain an `intlist`, which would contain an `intlist`... and so on forever.

Here's one of the procedures we'd write to use these lists, along with a typedef to make the use of pointers more convenient:

```
typedef struct intnode *intptr;

intptr cons(int car, intptr cdr) {
    intptr result = (intptr)malloc(sizeof intlist);

    result->value = car;
    result->next = cdr;
    return result;
}
```

The name `NULL` is conventionally used for a pointer to nowhere, so it would be assigned to the `next` field of the last node of a list.

```
        intptr baz = cons(1, cons(2, cons(3, NULL)));
```

`NULL` is defined this way:

```
#define NULL ((void *)0)
```

Of course it's an error to dereference a null pointer.

• **Unions**

The lists defined above aren't nearly as good as Scheme lists, because their elements can only be integers. Real lists should be heterogeneous, with the possibility of any data type (including sublists) as elements.

To do that we need a way to say "this field can be an integer or a double or a list node." In C we say this with the `union` construct:

```
struct thing {
    enum INT, DOUBLE, NODE type;
    union {
        int i;
        double d;
        struct listnode *n;
    } value;
};

struct listnode {
    struct thing *car;
    struct thing *cdr;
};

typedef struct thing* thingptr;

thingptr make_int(int value) {
    thingptr result=(thingptr)malloc(sizeof struct thing);
    result->type = INT;
```

```
    result->value.i = value;
    return result;
}
```

This is a more complicated and realistic example of C data definition than we've seen up to now. If we're going to have the possibility of different data types as list elements, we need some way to manipulate values of arbitrary type. In particular, we need to be able to find out whether the thing we're looking at is an integer, a double, or a node.

To make this work, each thing must carry around its type information — we need tagged data. That's the purpose of the `thing` struct. It has two fields: the type and the value. The type is an `enum`, a C data type with only a small number of possible values, which are given names. Enums are essentially unsigned integers; typically, a C compiler will give the name INT the internal value 0, DOUBLE the value 1, and NODE the value 2, although this is not guaranteed.

But focus now on the other part of a thing, the `union` named `value`. The word `value` (because it appears *after* the body of the union) is the name of a variable, an actual instance of this union. The union type itself has no name, because it will be used only in the context of the `thing` type. But union types can have names, just as struct types can have names.

Each instance of a struct has all of the fields in its defining body; for example, each `listnode` has both a `car` and a `cdr`. But each instance of a union has only one of the fields in its body; any particular `thing` will have an `i` *or* a `d` *or* an `n`, not all three.

The `make_int` procedure is an example of how the union is used. In this case, we want an integer thing, so we allocate memory for a `thing`, we set its `type` to `INT`, and we set its `value.i` — in other words, the `i` field of the `value` union — to the argument value.

Since the different alternatives of a union might be different sizes (for example, an integer takes 4 bytes but a double takes 8 bytes), the size of the union is defined as the size of the largest alternative.

How Malloc Works

The fundamental idea of a memory allocator is that you should be able to ask for a block of memory:

```
void *malloc(unsigned int size);
```

and you should be able to return a block that you no longer need:

```
void free(void *block);
```

Note that `malloc` returns a generic pointer; this is the other main application, besides higher-order functions, of the `void *` type. Note also that `free`'s declaration says it returns `void`, not `void *`; in other words, it doesn't return anything.

K&R 8.7 gives an implementation of a fairly typical simple memory allocator. The idea is that memory (more precisely, the heap, which is the part of memory available for use by `malloc`) contains a bunch of different-sized blocks, some of which are in use and some of which are free at any moment:

```
------------------------------------------------------------
\\\\\\\|////|      |\\\|///////////|\\\\\|    |//////|
\\\\\\\|////|      |\\\|///////////|\\\\\|    |//////|
------------------------------------------------------------
```

In this picture the shaded blocks are in use, and there are three free blocks. (There will never be two *consecutive* free blocks, because the `free` procedure checks for this and combines them into one larger block.)

K&R discuss the issue of *alignment*. Suppose the first call to `malloc` is allocating a character string, that happens to contain five characters, so a five-byte block is all it needs. Let's say this call returns the pointer 1000, so the block is using the five bytes 1000-1004. The next call wants to allocate space for an integer, so it needs four bytes. But it can't just take the next four bytes of memory! We can't store an integer at location 1005, because the address of an integer has to be a multiple of four bytes. To solve this problem, we round every request up to the next multiple of four bytes. (Four bytes is the strictest allocation requirement on the MIPS. Some other architecture might require eight-byte alignment, and we'd have to change `malloc` accordingly.)

The whole idea of blocks of memory is in the mind of the programmer; as far as the computer is concerned, all of memory is one big block. In particular, when the programmer calls `free` with the address of an allocated block as its argument, how does `free` know where the block ends? To solve this problem, each block includes a *header* that records, at a minimum, the size of the block. (The K&R implementation uses a two-word header, with a pointer as well as the size, as we'll see in a moment.) We want this header to be invisible to the caller of `malloc`, so we put it *before* the address to which the returned pointer points.

Here's an example to make that more concrete. Someone calls `malloc(12)` to ask for 12 bytes of memory. Our version of `malloc` uses an 8-byte header. So we look for a free memory block that contains at least 12+8=20 bytes of memory. Let's say we find one at location 1000. Then we'll use bytes 1000–1007 as the header, and 1008-1019 as the block itself. The value `malloc` will return is `1008`. Later, when the user wants to return this block, s/he will call `free(1008)`. The `free` procedure knows that if the data starts at 1008, the block's header is in 1000–1007. Looking in that header, `free` can find out that the block's total size is 20 bytes (including header), so it ends at 1000+20-1 or 1019.

Besides the size, the other word in the K&R `malloc` header is a pointer to the next free block. These pointers are used to maintain a linked list of free blocks. The pointer is used only for free blocks; when a block is in use, it isn't part of the free block list. So the pointer could have been kept in the data part of the block, instead of adding a word to the header. This version is simpler to understand, though, because it keeps all the `malloc` bookkeeping information in the header.

I'm not going to go through the code in detail; read K&R for that.

• **Fragmentation**

If all calls to `malloc` requested the same size block, the management of free memory would be easy. We'd just maintain a list of free blocks, and since all free blocks would be the same size, we could just use the first block in the list to satisfy each request.

Things are more complicated when requests can be for different sizes. To take a grossly simplified, artificial example, suppose the heap is entirely filled by two blocks, each of size 100 bytes:

```
-------------------------
|\\\ 100 \\\|/// 100 ///|
-------------------------
```

Now the first block is freed:

```
-------------------------
|    100    |/// 100 ///|
-------------------------
```

Now a request is made for 92 bytes (including the header):

```
-------------------------
|\\ 92 \\| 8|/// 100 ///|
-------------------------
```

We had a large enough free block (100 bytes) to satisfy the request, but the request didn't completely use up the block. We are left with a block of eight free bytes. Now suppose we free the second 100-byte block and request 92 bytes again:

```
-------------------------
|\\ 92 \\| 8|// 92 //| 8|
```

We have a total of 16 bytes of free memory, but it takes the form of two non-contiguous blocks of eight bytes each. If a request comes in for 12 bytes, we can't satisfy it, even though the total free memory is sufficient. To describe this situation we say that the memory is *fragmented*.

Much of the complexity of practical memory allocators goes into trying to avoid fragmentation. For example, the K&R `free` takes the trouble to "coalesce" a newly-freed block with any adjacent block that happens to be free already, so that we try to maintain an inventory of larger free blocks rather than many smaller blocks.

K&R mention an approach that was once tried, but has been shown to be counterproductive: the "best fit" strategy in which, instead of taking the first block in the free list that's large enough to meet a `malloc` request, we search the entire free list to find the smallest free block that's large enough. The theory was that the leftover free space when a block is bigger than necessary to satisfy the request will probably be useless, and it's better to have a tiny useless free block than a medium-size useless free block. But it turns out that the medium-size ones can often be used to satisfy a request, and over time, the tiny slivers lead to unrecoverable fragmentation. So nobody uses best-fit these days.

• **Fragmentation: The Buddy System**

Better approaches to avoiding fragmentation tend to involve the counterintuitive decision to allocate blocks larger than what the user asked for, so as to maintain a relatively small number of block sizes. The most famous of these is the "buddy system." In this technique, all blocks must be of a size (including the header) that's a power of two.

Why powers of two? Because when we want a block of a certain size, and we find a larger free block, the leftover part is, or can be divided into, power-of-two size. For example, if we want 32 bytes and we find 64 bytes, the leftover 64-32 is another 32-byte (power of two) block. If we want 32 bytes and we find 128, we divide the leftover 128-32=96 bytes into a 64-byte block and a 32-byte block.

The advantage of having a small number of possible block sizes is that instead of maintaining a single free list of mixed-size blocks, we maintain separate free lists for each size. So if we want a 32-byte block, we first look in the 32-byte free list. With luck, we find a 32-byte free block, which can be allocated to this request with nothing left over. That makes the allocation process very simple and fast. If we're not so lucky, we look for larger and larger blocks (first 64, then 128, then 256, etc.) until we find a free block. In this case we have to divide the remainder of the large block into power-of-two-sized pieces and add them to the proper free lists.

So far we've seen the merit of a limited-number-of-sizes approach, but not the "buddy" aspect of the buddy system. That has to do with a very clever solution to the problem of coalescing small free blocks into larger ones, which can be the most time-consuming part of a memory allocator. The rule in the buddy system is that blocks must be aligned on addresses that are a multiple of their size. For example, the address of a 32-byte block must be a multiple of 32. Since sizes are powers of two, this means that the address of a block always ends in a certain number of zero bits; in the case of a 32-byte block, the rightmost five bits of the address will be zero, for the same reason that makes the rightmost two bits of a four-byte word always zero.

When a block is freed, it can be coalesced only with another block of the same size, and not just any such block; there is only one possible partner for it! To be specific, consider the 32-byte block at address `01010100000` binary. Obviously, it can only coalesce with a free block that's right next to it. That would be the block 32 bytes earlier, at address `01010000000`, or the block 32 bytes later, at address `01011000000`. But the address of the combined 64-byte block will be the smaller of the two 32-bit-block addresses. In the first case, that's `01010000000`; in the second case, it's the address of the block we're freeing now, which is `01010100000`.

But to satisfy our rules, a 64-byte block must have an address divisible by 64. That is, the address must end with six zero bits. That's true for `01010000000`, but it's *not* true for `01010100000`. Our original 32-byte block has exactly one "buddy" with which it can be coalesced, namely, the 32-byte block at 01010000000. Similarly, a 32-byte block at 01010000000 has the 32-byte block at 01010100000 as its buddy.

This sounds awfully complicated; what's the payoff? It's that the address of a block's buddy can be *computed* rather than looked up; given the size, we just flip that bit in its address.

```
address    01010100000
size       00000100000
----------------------
buddy      01010000000
```

In C:

```
void *address, *buddy;
unsigned int size;

buddy = (void *)((unsigned)address^size);
```

The ^ symbol is C's exclusive-or operator.

Given the buddy's address, we just look for it in the free list for the same size, and if it's there, we coalesce the two blocks.

There are other variants on the general idea of restricting the set of allowable block sizes. The goal is that the sum of two sizes should be another size, so one interesting variant is called the Fibonacci Buddy System, with block sizes 1, 2, 3, 5, 8, 13, ...

• Fragmentation: Exploiting Realistic Memory Demands

In practice, the size requests that a program makes in calls to `malloc` are not random. Typically, a program uses a few struct types repeatedly, so many of its calls to `malloc` will requests block sizes equal to the size of those structs. If so, the most efficient strategy, both in running time and in avoiding wasted space, is to allocate one big block of memory whose size is, say, 100 time the size of one of these structs, then have a special-purpose allocator that manages the 100 struct-sized sub-blocks. Since the free blocks are all exactly the right size, there is no fragmentation, no need to coalesce (in fact, it would be a bad idea), and really no need even for a header, since a special-purpose `free` routine could be used for each type.

The same program might also allocate some varying-length blocks, e.g., for character strings typed by the user. So it would probably use a general-purpose `malloc` in addition to the special allocators for structs. The 100-struct blocks might be allocated by `malloc`.

• Freeing is Problematic

Sometimes a block of memory is needed for a computation with a clear-cut beginning and end:

```
struct whatever *foo;

foo = (struct whatever *)malloc(sizeof struct whatever);
  /* computation goes here */
free(foo);
```

But consider an example like

```
d = vector_add(a, vector_add(b, c));
```

If we are using one of the versions of `vector_add` that heap-allocates the space for its result, what happens to the space allocated for the inner call? We don't keep a pointer to it, so it will never be freed — it will "leak." To solve this we'd have to say

```
vecptr temp = vector_add(b, c);
d = vector_add(a, temp);
```

```
free(temp);
```

We'd lose the expressive power of composition of functions.

Even worse is the opposite problem: `free`ing a block that's really still in use. This typically happens when we have two pointers to the same block, we finish using one of the pointers, and call `free` not realizing that someone else still points to the block.

• **How Good Memory Management Works**

The only good solution to this problem is to recognize that human beings aren't organized enough to be entrusted with the responsibility for freeing memory; this is just the sort of task at which computers do better than we do. The biggest advance in Java, compared to C, is not the addition of object-oriented features, but the removal of `free`.

The general idea of *garbage collection* is that you pay no attention to unused blocks of memory (letting them leak) until you run out of free heap space. You then analyze the entire heap, working out which blocks are still in use and which are free, and you reclaim the free space all at once.

How is this possible? The central insight is that all the pointers in memory form a graph structure with a few well-defined starting points: basically, the program's named variables. Some of those variables are of type pointer-to-something, and if the something is another pointer, or a struct or array containing pointers, we follow those pointers to find additional in-use blocks.

The above is a very handwavy description. The garbage collector must, for example, be able to distinguish pointer variables from non-pointer variables. Ordinarily, type information used by the compiler isn't still available when a compiled program is running; without that information, a pointer is indistinguishable from any other unsigned integer. One solution is to use tagged data throughout the program; another is to follow the Lisp model in which *all* variables are pointers to the underlying data.

One reason it took a long time for garbage collection to be widely accepted outside of artificial intelligence work is the fact that the memory management happens all at once, rather than through many brief calls to `free` as the program is running. This problem is particularly noticeable if the program is trying to do real-time animated graphics, but might also have unpleasant consequences in the control program for an airplane. Modern garbage collectors have improved this situation through two main approaches: *Generational* garbage collection is based on the observation that most blocks of memory either become unused very quickly or remain used for the entire running of the program; there is little middle ground. So a garbage collector can examine only the recently-allocated part of memory, and will still reclaim almost all of the unused space. The other approach is *incremental* garbage collection, using clever techniques to allow a multithreaded system in which one thread can be collecting garbage while other threads continue to do real work.