Essential C

By Nick Parlante

This Stanford CS Education document tries to summarize all the basic features of the C language. The coverage is pretty quick, so it is most appropriate as review or for someone with some programming background in another language. Topics include variables, int types, floating point types, promotion, truncation, operators, control structures (if, while, for), functions, value parameters, reference parameters, structs, pointers, arrays, the pre-processor, and the standard C library functions.

The most recent version is always maintained at its Stanford CS Education Library URL http://cslibrary.stanford.edu/101/ . Please send your comments to nick.parlante@cs.stanford.edu.

I hope you can share and enjoy this document in the spirit of goodwill in which it is given away -- Nick Parlante, 4/2003, Stanford California.

Table of Contents

The C Language

C is a professional programmer's language. It was designed to get in one's way as little as possible. Kernighan and Ritchie wrote the original language definition in their book, The C Programming Language (below), as part of their research at AT&T. Unix and C++ emerged from the same labs. For several years I used AT&T as my long distance carrier in appreciation of all that CS research, but hearing "thank you for using AT&T" for the millionth time has used up that good will.

Section 1
Basic Types and Operators

C provides a standard, minimal set of basic data types. Sometimes these are called "primitive" types. More complex data structures can be built up from these basic types.

Integer Types

The "integral" types in C form a family of integer types. They all behave like integers and can be mixed together and used in similar ways. The differences are due to the different number of bits ("widths") used to implement each type -- the wider types can store a greater ranges of values.

`char` ASCII character -- at least 8 bits. Pronounced "car". As a practical matter `char` is basically always a byte which is 8 bits which is enough to store a single ASCII character. 8 bits provides a signed range of -128..127 or an unsigned range is 0..255. `char` is also required to be the "smallest addressable unit" for the machine -- each byte in memory has its own address.

`short` Small integer -- at least 16 bits which provides a signed range of -32768..32767. Typical size is 16 bits. Not used so much.

`int` Default integer `--` at least 16 bits, with 32 bits being typical. Defined to be the "most comfortable" size for the computer. If you do not really care about the range for an integer variable, declare it `int` since that is likely to be an appropriate size (16 or 32 bit) which works well for that machine.

`long` Large integer -- at least 32 bits. Typical size is 32 bits which gives a signed range of about -2 billion ..+2 billion. Some compilers support "long long" for 64 bit ints.

The integer types can be preceded by the qualifier `unsigned` which disallows representing negative numbers, but doubles the largest positive number representable. For example, a 16 bit implementation of `short` can store numbers in the range -32768..32767, while `unsigned short` can store 0..65535. You can think of pointers as being a form of `unsigned long` on a machine with 4 byte pointers. In my opinion, it's best to avoid using `unsigned` unless you really need to. It tends to cause more misunderstandings and problems than it is worth.

Extra: Portability Problems

Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware. Unfortunately it occasionally leads to bugs where a program runs differently on a 16-bit-int machine than it runs on a 32-bit-int machine. In particular, if you are designing a function that will be implemented on several different machines, it is a good idea to use typedefs to set up types like `Int32` for 32 bit int and `Int16` for 16 bit int. That way you can prototype a function `Foo(Int32)` and be confident that the typedefs for each machine will be set so that the function really takes exactly a 32 bit int. That way the code will behave the same on all the different machines.

char Constants

A `char` constant is written with single quotes (') like 'A' or 'z'. The `char` constant 'A' is really just a synonym for the ordinary integer value 65 which is the ASCII value for

uppercase 'A'. There are special case char constants, such as '\t' for tab, for characters which are not convenient to type on a keyboard.

`'A'`	uppercase 'A' character
`'\n'`	newline character
`'\t'`	tab character
`'\0'`	the "null" character -- integer value 0 (different from the char digit '0')
`'\012'`	the character with value 12 in octal, which is decimal 10

int Constants

Numbers in the source code such as `234` default to type `int`. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a `long` such as 42L. An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal -- `0x10` is way of expressing the number 16. Similarly, a constant may be written in octal by preceding it with "0" -- `012` is a way of expressing the number 10.

Type Combination and Promotion

The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, `char` and `int` can be combined in arithmetic expressions such as `('b' + 5)`. How does the compiler deal with the different widths present in such an expression? In such a case, the compiler "promotes" the smaller type (`char`) to be the same size as the larger type (`int`) before combining the values. Promotions are determined at compile time based purely on the **types** of the values in the expressions. Promotions do not lose information -- they always convert from a type to compatible, larger type to avoid losing information.

Pitfall -- int Overflow

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

Floating point Types

`float`	Single precision floating point number	typical size: 32 bits
`double`	Double precision floating point number	typical size: 64 bits
`long double`	Possibly even bigger floating point number (somewhat obscure)	

Constants in the source code such as 3.14 default to type `double` unless the are suffixed with an 'f' (float) or 'l' (long double). Single precision equates to about 6 digits of

precision and double is about 15 digits of precision. Most C programs use `double` for their computations. The main reason to use `float` is to save memory if many numbers need to be stored. The main thing to remember about floating point numbers is that they are inexact. For example, what is the value of the following `double` expression?

```
(1.0/3.0 + 1.0/3.0 + 1.0/3.0)    // is this equal to 1.0 exactly?
```

The sum may or may not be 1.0 exactly, and it may vary from one type of machine to another. For this reason, you should never compare floating numbers to eachother for equality (==) -- use inequality (<) comparisons instead. Realize that a correct C program run on different computers may produce slightly different outputs in the rightmost digits of its floating point computations.

Comments

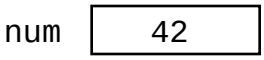
Comments in C are enclosed by slash/star pairs: `/* .. comments .. */` which may cross multiple lines. C++ introduced a form of comment started by two slashes and extending to the end of the line: `// comment until the line end`
The // comment form is so handy that many C compilers now also support it, although it is not technically part of the C language.

Along with well-chosen function names, comments are an important part of well written code. Comments should not just repeat what the code says. Comments should describe what the code **accomplishes** which is much more interesting than a translation of what each statement does. Comments should also narrate what is tricky or non-obvious about a section of code.

Variables

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable. The following declares an `int` variable named "`num`" and the 2nd line stores the value 42 into `num`.

```
int num;
num = 42;
```



A variable corresponds to an area of memory which can store a value of the given type. Making a drawing is an excellent way to think about the variables in a program. Draw each variable as box with the current value inside the box. This may seem like a "beginner" technique, but when I'm buried in some horribly complex programming problem, I invariably resort to making a drawing to help think the problem through.

Variables, such as `num`, do not have their memory cleared or set in any way when they are allocated at run time. Variables start with random values, and it is up to the program to set them to something sensible before depending on their values.

Names in C are case sensitive so "x" and "X" refer to different variables. Names can contain digits and underscores (_), but may not begin with a digit. Multiple variables can be declared after the type by separating them with commas. C is a classical "compile time" language -- the names of the variables, their types, and their implementations are all flushed out by the compiler at compile time (as opposed to figuring such details out at run time like an interpreter).

```
float x, y, z, X;
```

Assignment Operator =

The assignment operator is the single equals sign (=).

```
i = 6;
i = i + 1;
```

The assignment operator copies the value from its right hand side to the variable on its left hand side. The assignment also acts as an expression which returns the newly assigned value. Some programmers will use that feature to write things like the following.

```
y = (x = 2 * x);     // double x, and also put x's new value in y
```

Truncation

The opposite of promotion, truncation moves a value from a type to a smaller type. In that case, the compiler just drops the extra bits. It may or may not generate a compile time warning of the loss of information. Assigning from an integer to a smaller integer (e.g.. `long` to `int`, or `int` to `char`) drops the most significant bits. Assigning from a floating point type to an integer drops the fractional part of the number.

```
char ch;
int i;

i = 321;
ch = i;     // truncation of an int value to fit in a char
// ch is now 65
```

The assignment will drop the upper bits of the `int` 321. The lower 8 bits of the number 321 represents the number 65 (321 - 256). So the value of `ch` will be (`char`)65 which happens to be 'A'.

The assignment of a floating point type to an integer type will drop the fractional part of the number. The following code will set `i` to the value 3. This happens when assigning a floating point number to an integer or passing a floating point number to a function which takes an integer.

```
double pi;
int i;

pi = 3.14159;
i = pi;     // truncation of a double to fit in an int
// i is now 3
```

Pitfall -- int vs. float Arithmetic

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

```
{
   int score;
...// suppose score gets set in the range 0..20 somehow
```

```
   score = (score / 20) * 100;         // NO -- score/20 truncates to 0
...
```

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

```
score = ((double)score / 20) * 100;    // OK -- floating point division from cast

score = (score / 20.0) * 100;          // OK -- floating point division from 20.0

score = (int)(score / 20.0) * 100;     // NO -- the (int) truncates the floating
                                       // quotient back to 0
```

No Boolean -- Use int

C does not have a distinct boolean type-- `int` is used instead. The language treats integer 0 as false and all non-zero values as true. So the statement...

```
i = 0;
while (i - 10) {
   ...
```

will execute until the variable `i` takes on the value 10 at which time the expression `(i - 10)` will become false (i.e. 0). (we'll see the while() statement a bit later)

Mathematical Operators

C includes the usual binary and unary arithmetic operators. See the appendix for the table of precedence. Personally, I just use parenthesis liberally to avoid any bugs due to a misunderstanding of precedence. The operators are sensitive to the type of the operands. So division (`/`) with two integer arguments will do integer division. If either argument is a float, it does floating point division. So `(6/4)` evaluates to 1 while `(6/4.0)` evaluates to 1.5 -- the 6 is promoted to 6.0 before the division.

`+` Addition

`-` Subtraction

`/` Division

`*` Multiplication

`%` Remainder (mod)

Unary Increment Operators: ++ --

The unary ++ and -- operators increment or decrement the value in a variable. There are "pre" and "post" variants for both operators which do slightly different things (explained below)

var++	increment	"post" variant
++*var*	increment	"pre" variant

var-- decrement "post" variant

--var decrement "pre" variant

```
int i = 42;
i++;     // increment on i
// i is now 43
i--;     // decrement on i
// i is now 42
```

Pre and Post Variations

The Pre/Post variation has to do with nesting a variable with the increment or decrement operator inside an expression -- should the entire expression represent the value of the variable before or after the change? I **never** use the operators in this way (see below), but an example looks like...

```
int i = 42;
int j;

j = (i++ + 10);
// i is now 43
// j is now 52 (NOT 53)

j = (++i + 10)
// i is now 44
// j is now 54
```

C Programming Cleverness and Ego Issues

Relying on the difference between the pre and post variations of these operators is a classic area of C programmer ego showmanship. The syntax is a little tricky. It makes the code a little shorter. These qualities drive some C programmers to show off how clever they are. C invites this sort of thing since the language has many areas (this is just one example) where the programmer can get a complex effect using a code which is short and dense.

If I want j to depend on i's value before the increment, I write...

```
j = (i + 10);
i++;
```

Or if I want to j to use the value after the increment, I write...

```
i++;
j = (i + 10);
```

Now then, isn't that nicer? (editorial) Build programs that do something cool rather than programs which flex the language's syntax. Syntax -- who cares?

Relational Operators

These operate on integer or floating point values and return a 0 or 1 boolean value.

== Equal

`!=`	Not Equal
`>`	Greater Than
`<`	Less Than
`>=`	Greater or Equal
`<=`	Less or Equal

To see if x equals three, write something like:

```
if (x == 3) ...
```

Pitfall = ≠ ==

An absolutely classic pitfall is to write assignment (=) when you mean comparison (==). This would not be such a problem, except the incorrect assignment version compiles fine because the compiler assumes you mean to use the value returned by the assignment. This is rarely what you want

```
if (x = 3) ...
```

This does not test if `x` is 3. This sets `x` to the value 3, and then returns the 3 to the `if` for testing. 3 is not 0, so it counts as "true" every time. This is probably the single most common error made by beginning C programmers. The problem is that the compiler is no help -- it thinks both forms are fine, so the only defense is extreme vigilance when coding. Or write "= ≠ ==" in big letters on the back of your hand before coding. This mistake is an absolute classic and it's a bear to debug. Watch Out! And need I say: "Professional Programmer's Language."

Logical Operators

The value 0 is false, anything else is true. The operators evaluate left to right and stop as soon as the truth or falsity of the expression can be deduced. (Such operators are called "short circuiting") In ANSI C, these are furthermore guaranteed to use 1 to represent true, and not just some random non-zero bit pattern. However, there are many C programs out there which use values other than 1 for true (non-zero pointers for example), so when programming, do not assume that a true boolean is necessarily 1 exactly.

`!`	Boolean not (unary)
`&&`	Boolean and
`||`	Boolean or

Bitwise Operators

C includes operators to manipulate memory at the bit level. This is useful for writing low-level hardware or operating system code where the ordinary abstractions of numbers, characters, pointers, etc... are insufficient -- an increasingly rare need. Bit manipulation code tends to be less "portable". Code is "portable" if with no programmer intervention it compiles and runs correctly on different types of computers. The bitwise operations are

typically used with unsigned types. In particular, the shift operations are guaranteed to shift 0 bits into the newly vacated positions when used on unsigned values.

`~`	Bitwise Negation (unary) – flip 0 to 1 and 1 to 0 throughout
`&`	Bitwise And
`|`	Bitwise Or
`^`	Bitwise Exclusive Or
`>>`	Right Shift by right hand side (RHS) (divide by power of 2)
`<<`	Left Shift by RHS (multiply by power of 2)

Do not confuse the Bitwise operators with the logical operators. The bitwise connectives are one character wide (`&`, `|`) while the boolean connectives are two characters wide (`&&`, `||`). The bitwise operators have higher precedence than the boolean operators. The compiler will never help you out with a type error if you use `&` when you meant `&&`. As far as the type checker is concerned, they are identical-- they both take and produce integers since there is no distinct boolean type.

Other Assignment Operators

In addition to the plain = operator, C includes many shorthand operators which represents variations on the basic =. For example "+=" adds the right hand side to the left hand side. `x = x + 10;` can be reduced to `x += 10;`. This is most useful if x is a long expression such as the following, and in some cases it may run a little faster.

```
person->relatives.mom.numChildren += 2;      // increase children by 2
```

Here's the list of assignment shorthand operators...

`+=, -=`	Increment or decrement by RHS
`*=, /=`	Multiply or divide by RHS
`%=`	Mod by RHS
`>>=`	Bitwise right shift by RHS (divide by power of 2)
`<<=`	Bitwise left shift RHS (multiply by power of 2)
`&=, |=, ^=`	Bitwise and, or, xor by RHS

Section 2
Control Structures

Curly Braces {}

C uses curly braces ({}) to group multiple statements together. The statements execute in order. Some languages let you declare variables on any line (C++). Other languages insist that variables are declared only at the beginning of functions (Pascal). C takes the middle road -- variables may be declared within the body of a function, but they must follow a '{'. More modern languages like Java and C++ allow you to declare variables on any line, which is handy.

If Statement

Both an if and an if-else are available in C. The <*expression*> can be any valid expression. The parentheses around the expression are required, even if it is just a single variable.

```
if (<expression>) <statement>    // simple form with no {}'s or else clause


if (<expression>) {        // simple form with {}'s to group statements
   <statement>
   <statement>
}


if (<expression>) {        // full then/else form
   <statement>
}
else {
   <statement>
}
```

Conditional Expression -or- The Ternary Operator

The conditional expression can be used as a shorthand for some if-else statements. The general syntax of the conditional operator is:

```
<expression1> ? <expression2> : <expression3>
```

This is an expression, not a statement, so it represents a value. The operator works by evaluating expression1. If it is true (non-zero), it evaluates and returns expression2 . Otherwise, it evaluates and returns expression3.

The classic example of the ternary operator is to return the smaller of two variables. Every once in a while, the following form is just what you needed. Instead of...

```
if (x < y) {
   min = x;
}
else {
   min = y;
}
```

You just say...

```
min = (x < y) ? x : y;
```

Switch Statement

The `switch` statement is a sort of specialized form of `if` used to efficiently separate different blocks of code based on the value of an integer. The switch expression is evaluated, and then the flow of control jumps to the matching const-expression case. The case expressions are typically int or char constants. The `switch` statement is probably the single most syntactically awkward and error-prone features of the C language.

```
switch (<expression>) {
   case <const-expression-1>:
      <statement>
      break;

   case <const-expression-2>:
      <statement>
      break;

   case <const-expression-3>:    // here we combine case 3 and 4
   case <const-expression-4>:
      <statement>
      break;

   default:    // optional
      <statement>
}
```

Each constant needs its own `case` keyword and a trailing colon (:). Once execution has jumped to a particular case, the program will keep running through all the cases from that point down -- this so called "fall through" operation is used in the above example so that expression-3 and expression-4 run the same statements. The explicit `break` statements are necessary to exit the `switch`. Omitting the `break` statements is a common error -- it compiles, but leads to inadvertent fall-through behavior.

Why does the switch statement fall-through behavior work the way it does? The best explanation I can think of is that originally C was developed for an audience of assembly language programmers. The assembly language programmers were used to the idea of a jump table with fall-through behavior, so that's the way C does it (it's also relatively easy to implement it this way.) Unfortunately, the audience for C is now quite different, and the fall-through behavior is widely regarded as a terrible part of the language.

While Loop

The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

```
while (<expression>) {
   <statement>
}
```

Do-While Loop

Like a while, but with the test condition at the bottom of the loop. The loop body will always execute at least once. The do-while is an unpopular area of the language, most everyone tries to use the straight `while` if at all possible.

```
do {
   <statement>
} while (<expression>)
```

For Loop

The for loop in C is the most general looping construct. The loop header contains three parts: an initialization, a continuation condition, and an action.

```
for (<initialization>; <continuation>; <action>) {
   <statement>
}
```

The initialization is executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true (like a `while`). After every execution of the loop, the action is executed. The following example executes 10 times by counting 0..9. Many loops look very much like the following...

```
for (i = 0; i < 10; i++) {
   <statement>
}
```

C programs often have series of the form 0..(some_number-1). It's idiomatic in C for the above type loop to start at 0 and use < in the test so the series runs up to but not equal to the upper bound. In other languages you might start at 1 and use <= in the test.

Each of the three parts of the `for` loop can be made up of multiple expressions separated by commas. Expressions separated by commas are executed in order, left to right, and represent the value of the last expression. (See the string-reverse example below for a demonstration of a complex for loop.)

Break

The `break` statement will move control outside a loop or switch statement. Stylistically speaking, `break` has the potential to be a bit vulgar. It's preferable to use a straight `while` with a single test at the top if possible. Sometimes you are forced to use a `break` because the test can occur only somewhere in the midst of the statements in the loop body. To keep the code readable, be sure to make the break obvious -- forgetting to account for the action of a break is a traditional source of bugs in loop behavior.

```
while (<expression>) {
   <statement>
   <statement>

   if (<condition which can only be evaluated here>)
      break;

   <statement>
   <statement>
}
// control jumps down here on the break
```

The `break` does not work with `if`. It only works in loops and switches. Thinking that a break refers to an `if` when it really refers to the enclosing `while` has created some high quality bugs. When using a `break`, it's nice to write the enclosing loop to iterate in the most straightforward, obvious, normal way, and then use the `break` to explicitly catch the exceptional, weird cases.

Continue

The `continue` statement causes control to jump to the bottom of the loop, effectively skipping over any code below the continue. As with `break`, this has a reputation as being vulgar, so use it sparingly. You can almost always get the effect more clearly using an if inside your loop.

```
while (<expression>) {
   ...
   if (<condition>)
      continue;
   ...
   ...
   // control jumps here on the continue
}
```

Section 3
Complex Data Types

C has the usual facilities for grouping things together to form composite types-- arrays and records (which are called "structures"). The following definition declares a type called "struct fraction" that has two integer sub fields named "numerator" and "denominator". If you forget the semicolon it tends to produce a syntax error in whatever thing follows the struct declaration.

```
struct fraction {
   int numerator;
   int denominator;
};          // Don't forget the semicolon!
```

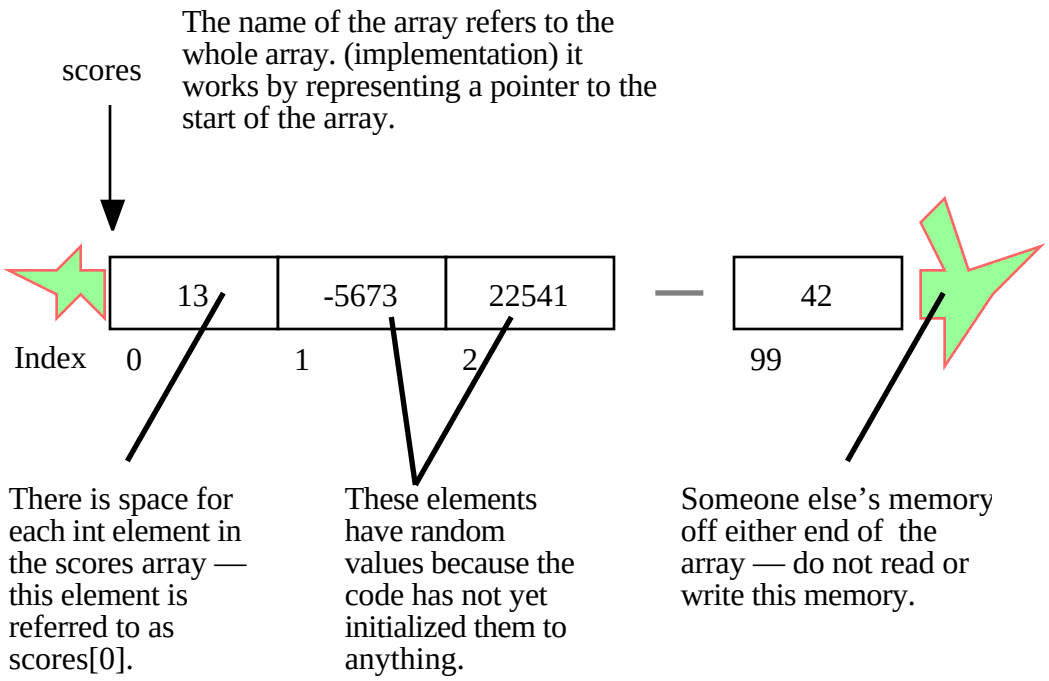
This declaration introduces the type `struct fraction` (both words are required) as a new type. C uses the period (.) to access the fields in a record. You can copy two records of the same type using a single assignment statement, however == does not work on structs.

```
struct fraction f1, f2;          // declare two fractions

f1.numerator = 22;
f1.denominator = 7;

f2 = f1;    // this copies over the whole struct
```

Arrays

The simplest type of array in C is one which is declared and used in one place. There are more complex uses of arrays which I will address later along with pointers. The following declares an array called `scores` to hold 100 integers and sets the first and last elements. C arrays are always indexed from 0. So the first `int` in `scores` array is `scores[0]` and the last is `scores[99]`.

```
int scores[100];

scores[0]  = 13;           // set first element
scores[99] = 42;           // set last element
```



It's a very common error to try to refer to non-existent `scores[100]` element. C does not do any run time or compile time bounds checking in arrays. At run time the code will just access or mangle whatever memory it happens to hit and crash or misbehave in some unpredictable way thereafter. "Professional programmer's language." The convention of numbering things `0..(number of things - 1)` pervades the language. To best integrate with C and other C programmers, you should use that sort of numbering in your own data structures as well.

Multidimensional Arrays

The following declares a two-dimensional 10 by 10 array of integers and sets the first and last elements to be 13.

```
int board [10][10];

board[0][0] = 13;
board[9][9] = 13;
```

The implementation of the array stores all the elements in a single contiguous block of memory. The other possible implementation would be a combination of several distinct one dimensional arrays -- that's not how C does it. In memory, the array is arranged with the elements of the rightmost index next to each other. In other words, `board[1][8]` comes right before `board[1][9]` in memory.

(highly optional efficiency point) It's typically efficient to access memory which is near other recently accessed memory. This means that the most efficient way to read through a chunk of the array is to vary the rightmost index the most frequently since that will access elements that are near each other in memory.

Array of Structs

The following declares an array named "numbers" which holds 1000 `struct fraction`'s.

```
struct fraction numbers[1000];

numbers[0].numerator = 22;       /* set the 0th struct fraction */
numbers[0].denominator = 7;
```

Here's a general trick for unraveling C variable declarations: look at the right hand side and imagine that it is an expression. The type of that expression is the left hand side. For the above declarations, an expression which looks like the right hand side (`numbers[1000],` or really anything of the form `numbers[...]`) will be the type on the left hand side (`struct fraction`).

Pointers

A pointer is a value which represents a reference to another value sometimes known as the pointer's "pointee". Hopefully you have learned about pointers somewhere else, since the preceding sentence is probably inadequate explanation. This discussion will concentrate on the syntax of pointers in C -- for a much more complete discussion of pointers and their use see http://cslibrary.stanford.edu/102/, Pointers and Memory.

Syntax

Syntactically C uses the asterisk or "star" (*) to indicate a pointer. C defines pointer types based on the type pointee. A `char*` is type of pointer which refers to a single `char`. a `struct fraction*` is type of pointer which refers to a `struct fraction`.

```
int* intPtr;   // declare an integer pointer variable intPtr

char* charPtr; // declares a character pointer --
               // a very common type of pointer


// Declare two struct fraction pointers
// (when declaring multiple variables on one line, the *
//  should go on the right with the variable)
struct fraction *f1, *f2;
```

The Floating "*"

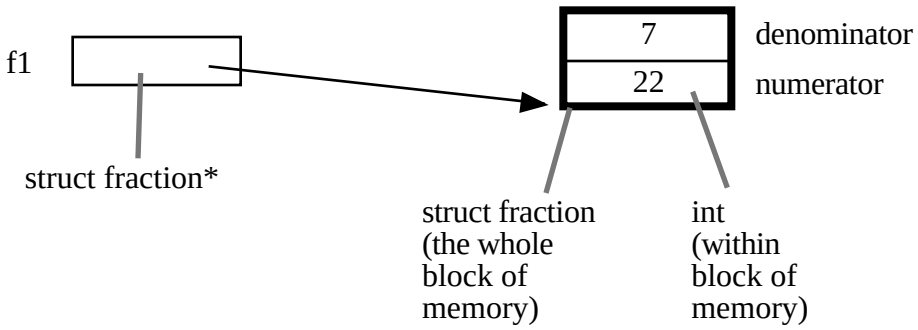
In the syntax, the star is allowed to be anywhere between the base type and the variable name. Programmer's have their own conventions-- I generally stick the * on the left with the type. So the above declaration of intPtr could be written equivalently...

```
int  *intPtr;        // these are all the same
int * intPtr;
int*  intPtr;
```

Pointer Dereferencing

We'll see shortly how a pointer is set to point to something -- for now just assume the pointer points to memory of the appropriate type. In an expression, the unary * to the left of a pointer dereferences it to retrieve the value it points to. The following drawing shows the types involved with a single pointer pointing to a struct fraction.

```
struct fraction* f1;
```



Expression	Type
f1	struct fraction*
*f1	struct fraction
(*f1).numerator	int

There's an alternate, more readable syntax available for dereferencing a pointer to a struct. A "`->`" at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written `f1->numerator`.

Here are some more complex declarations...

```
struct fraction** fp;      // a pointer to a pointer to a struct fraction

struct fraction fract_array[20];       // an array of 20 struct fractions

struct fraction* fract_ptr_array[20];  // an array of 20 pointers to
                                       // struct fractions
```

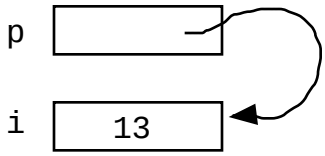
One nice thing about the C type syntax is that it avoids the circular definition problems which come up when a pointer structure needs to refer to itself. The following definition defines a node in a linked list. Note that no preparatory declaration of the node pointer type is necessary.

```
struct node {
   int data;
   struct node* next;
};
```

The & Operator

The & operator is one of the ways that pointers are set to point to things. The & operator computes a pointer to the argument to its right. The argument can be any variable which takes up space in the stack or heap (known as an "LValue" technically). So `&i` and `&(f1->numerator)` are ok, but `&6` is not. Use & when you have some memory, and you want a pointer to that memory.

```
void foo() {
   int* p;  // p is a pointer to an integer
   int i;   // i is an integer

   p = &i;  // Set p to point to i
   *p = 13; // Change what p points to -- in this case i -- to 13

   // At this point i is 13. So is *p. In fact *p is i.
}
```



When using a pointer to an object created with &, it is important to only use the pointer so long as the object exists. A local variable exists only as long as the function where it is declared is still executing (we'll see functions shortly). In the above example, `i` exists only as long as `foo()` is executing. Therefore any pointers which were initialized with `&i` are valid only as long as `foo()` is executing. This "lifetime" constraint of local memory is standard in many languages, and is something you need to take into account when using the & operator.

NULL

A pointer can be assigned the value 0 to explicitly represent that it does not currently have a pointee. Having a standard representation for "no current pointee" turns out to be very handy when using pointers. The constant NULL is defined to be 0 and is typically used when setting a pointer to NULL. Since it is just 0, a NULL pointer will behave like a boolean false when used in a boolean context. Dereferencing a NULL pointer is an error which, if you are lucky, the computer will detect at runtime -- whether the computer detects this depends on the operating system.

Pitfall -- Uninitialized Pointers

When using pointers, there are two entities to keep track of. The pointer and the memory it is pointing to, sometimes called the "pointee". There are three things which must be done for a pointer/pointee relationship to work...

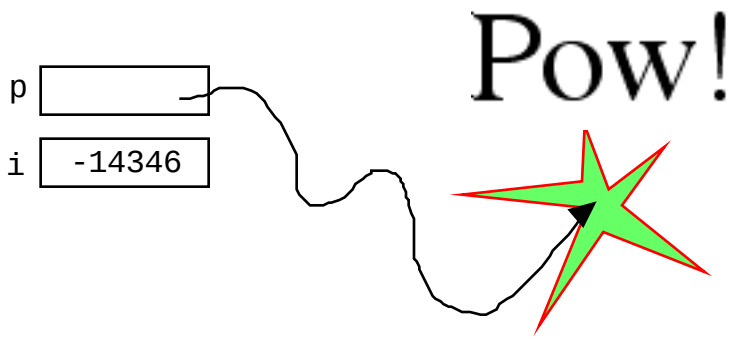
(1) The pointer must be declared and allocated

(2) The pointee must be declared and allocated

(3) The pointer (1) must be initialized so that it points to the pointee (2)

The most common pointer related error of all time is the following: Declare and allocate the pointer (step 1). Forget step 2 and/or 3. Start using the pointer as if it has been setup to point to something. Code with this error frequently compiles fine, but the runtime results are disastrous. Unfortunately the pointer does not point anywhere good unless (2) and (3) are done, so the run time dereference operations on the pointer with * will misuse and trample memory leading to a random crash at some point.

```
{
   int* p;

   *p = 13;    // NO NO NO p does not point to an int yet
               // this just overwrites a random area in memory
}
```



Of course your code won't be so trivial, but the bug has the same basic form: declare a pointer, but forget to set it up to point to a particular pointee.

Using Pointers

Declaring a pointer allocates space for the pointer itself, **but it does not allocate space for the pointee**. The pointer must be set to point to something before you can dereference it.

Here's some code which doesn't do anything useful, but which does demonstrate (1) (2) (3) for pointer use correctly...

```
int* p;     // (1) allocate the pointer
int i;      // (2) allocate pointee
struct fraction f1;  // (2) allocate pointee

p = &i;     // (3) setup p to point to i
*p = 42;    // ok to use p since it's setup

p = &(f1.numerator);       // (3) setup p to point to a different int
*p = 22;

p = &(f1.denominator);     // (3)
*p = 7;
```

So far we have just used the & operator to create pointers to simple variables such as `i`. Later, we'll see other ways of getting pointers with arrays and other techniques.

C Strings

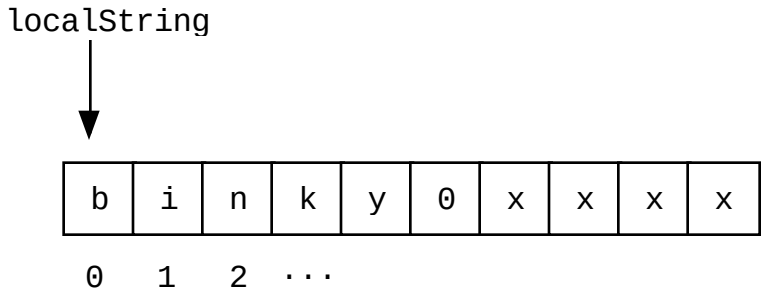
C has minimal support of character strings. For the most part, strings operate as ordinary arrays of characters. Their maintenance is up to the programmer using the standard facilities available for arrays and pointers. C does include a standard library of functions which perform common string operations, but the programmer is responsible for the managing the string memory and calling the right functions. Unfortunately computations involving strings are very common, so becoming a good C programmer often requires becoming adept at writing code which manages strings which means managing pointers and arrays.

A C string is just an array of `char` with the one additional convention that a "null" character ('\0') is stored after the last real character in the array to mark the end of the string. The compiler represents string constants in the source code such as "binky" as arrays which follow this convention. The string library functions (see the appendix for a partial list) operate on strings stored in this way. The most useful library function is `strcpy(char dest[], const char source[]);` which copies the bytes of one string over to another. The order of the arguments to strcpy() mimics the arguments in of '=' -- the right is assigned to the left. Another useful string function is `strlen(const char string[]);` which returns the number of characters in C string not counting the trailing '\0'.

Note that the regular assignment operator (=) does **not** do string copying which is why strcpy() is necessary. See Section 6, Advanced Pointers and Arrays, for more detail on how arrays and pointers work.

The following code allocates a 10 `char` array and uses strcpy() to copy the bytes of the string constant "binky" into that local array.

```
{
   char localString[10];

   strcpy(localString, "binky");
}
```



The memory drawing shows the local variable `localString` with the string "binky" copied into it. The letters take up the first 5 characters and the '\0' char marks the end of the string after the 'y'. The x's represent characters which have not been set to any particular value.

If the code instead tried to store the string "I enjoy languages which have good string support" into localString, the code would just crash at run time since the 10 character array can contain at most a 9 character string. The large string will be written passed the right hand side of localString, overwriting whatever was stored there.

String Code Example

Here's a moderately complex `for` loop which reverses a string stored in a local array. It demonstrates calling the standard library functions strcpy() and strlen() and demonstrates that a string really is just an array of characters with a '\0' to mark the effective end of the string. Test your C knowledge of arrays and `for` loops by making a drawing of the memory for this code and tracing through its execution to see how it works.

```
{

char string[1000];   // string is a local 1000 char array
int len;

strcpy(string, "binky");
len = strlen(string);

/*
 Reverse the chars in the string:
 i starts at the beginning and goes up
 j starts at the end and goes down
 i/j exchange their chars as they go until they meet
*/
int i, j;
char temp;
for (i = 0, j = len - 1; i < j; i++, j--) {
   temp = string[i];
   string[i] = string[j];
   string[j] = temp;
}

// at this point the local string should be "yknib"

}
```

"Large Enough" Strings

The convention with C strings is that the owner of the string is responsible for allocating array space which is "large enough" to store whatever the string will need to store. Most routines do not check that size of the string memory they operate on, they just assume its big enough and blast away. Many, many programs contain declarations like the following...

```
{
   char localString[1000];
   ...
}
```

The program works fine so long as the strings stored are 999 characters or shorter. Someday when the program needs to store a string which is 1000 characters or longer, then it crashes. Such array-not-quite-big-enough problems are a common source of bugs, and are also the source of so called "buffer overflow" security problems. This scheme has the additional disadvantage that most of the time when the array is storing short strings, 95% of the memory reserved is actually being wasted. A better solution allocates the string dynamically in the heap, so it has just the right size.

To avoid buffer overflow attacks, production code should check the size of the data first, to make sure it fits in the destination string. See the strlcpy() function in Appendix A.

char*

Because of the way C handles the types of arrays, the type of the variable `localString` above is essentially `char*`. C programs very often manipulate strings using variables of type `char*` which point to arrays of characters. Manipulating the actual chars in a string requires code which manipulates the underlying array, or the use

of library functions such as strcpy() which manipulate the array for you. See Section 6 for more detail on pointers and arrays.

TypeDef

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

```
typedef <type> <name>;
```

The following defines `Fraction` type to be the type (`struct fraction`). C is case sensitive, so `fraction` is different from `Fraction`. It's convenient to use typedef to create types with upper case names and use the lower-case version of the same word as a variable.

```
typedef struct fraction Fraction;

Fraction fraction;   // Declare the variable "fraction" of type "Fraction"
                     //  which is really just a synonym for "struct fraction".
```

The following typedef defines the name Tree as a standard pointer to a binary tree node where each node contains some data and "smaller" and "larger" subtree pointers.

```
typedef struct treenode* Tree;
struct treenode {
   int data;
   Tree smaller, larger;   // equivalently, this line could say
};                         // "struct treenode *smaller, *larger"
```

Section 4
Functions

All languages have a construct to separate and package blocks of code. C uses the "function" to package blocks of code. This article concentrates on the syntax and peculiarities of C functions. The motivation and design for dividing a computation into separate blocks is an entire discipline in its own.

A function has a name, a list of arguments which it takes when called, and the block of code it executes when called. C functions are defined in a text file and the names of all the functions in a C program are lumped together in a single, flat namespace. The special function called "main" is where program execution begins. Some programmers like to begin their function names with Upper case, using lower case for variables and parameters, Here is a simple C function declaration. This declares a function named `Twice` which takes a single `int` argument named `num`. The body of the function computes the value which is twice the `num` argument and returns that value to the caller.

```
/*
 Computes double of a number.
 Works by tripling the number, and then subtracting to get back to double.
*/
static int Twice(int num) {
   int result = num * 3;
   result = result - num;
   return(result);
}
```

Syntax

The keyword "`static`" defines that the function will only be available to callers in the file where it is declared. If a function needs to be called from another file, the function cannot be static and will require a prototype -- see prototypes below. The `static` form is convenient for utility functions which will only be used in the file where they are declared. Next , the "`int`" in the function above is the type of its return value. Next comes name of the function and its list of parameters. When referring to a function by name in documentation or other prose, it's a convention to keep the parenthesis () suffix, so in this case I refer to the function as "`Twice()`". The parameters are listed with their types and names, just like variables.

Inside the function, the parameter `num` and the local variable `result` are "local" to the function -- they get their own memory and exist only so long as the function is executing. This independence of "local" memory is a standard feature of most languages (See CSLibrary/102 for the detailed discussion of local memory).

The "caller" code which calls `Twice`() looks like...

```
int num = 13;
int a = 1;
int b = 2;
a = Twice(a);        // call Twice() passing the value of a
b = Twice(b + num);  // call Twice() passing the value b+num
// a == 2
// b == 30
// num == 13 (this num is totally independent of the "num" local to Twice()
```

Things to notice...

(vocabulary) The expression passed to a function by its caller is called the "actual parameter" -- such as "a" and "b + num" above. The parameter storage local to the function is called the "formal parameter" such as the "num" in "static int Twice(int num)".

Parameters are passed "by value" that means there is a single copying assignment operation (=) from each actual parameter to set each formal parameter. The actual parameter is evaluated in the caller's context, and then the value is copied into the function's formal parameter just before the function begins executing. The alternative parameter mechanism is "by reference" which C does not implement directly, but which the programmer can implement manually when needed (see below). When a parameter is a struct, it is copied.

The variables local to `Twice()`, `num` and `result`, only exist temporarily while `Twice()` is executing. This is the standard definition for "local" storage for functions.

The `return` at the end of `Twice()` computes the return value and exits the function. Execution resumes with the caller. There can be multiple return statements within a function, but it's good style to at least have one at the end if a return value needs to be specified. Forgetting to account of a `return` somewhere in the middle of a function is a traditional source of bugs.

C-ing and Nothingness -- void

`void` is a type formalized in ANSI C which means "nothing". To indicate that a function does not return anything, use `void` as the return type. Also, by convention, a pointer which does not point to any particular type is declared as `void*`. Sometimes `void*` is used to force two bodies of code to not depend on each other where `void*` translates roughly to "this points to something, but I'm not telling you (the client) the type of the pointee exactly because you do not really need to know." If a function does not take any parameters, its parameter list is empty, or it can contain the keyword `void` but that style is now out of favor.

```
void TakesAnIntAndReturnsNothing(int anInt);

int TakesNothingAndReturnsAnInt();
int TakesNothingAndReturnsAnInt(void); // equivalent syntax for above
```

Call by Value vs. Call by Reference

C passes parameters "by value" which means that the actual parameter values are copied into local storage. The caller and callee functions do not share any memory -- they each have their own copy. This scheme is fine for many purposes, but it has two disadvantages.

1) Because the callee has its own copy, modifications to that memory are not communicated back to the caller. Therefore, value parameters do not allow the callee to communicate back to the caller. The function's return value can communicate some information back to the caller, but not all problems can be solved with the single return value.

2) Sometimes it is undesirable to copy the value from the caller to the callee because the value is large and so copying it is expensive, or because at a conceptual level copying the value is undesirable.

The alternative is to pass the arguments "by reference". Instead of passing a copy of a value from the caller to the callee, pass a pointer to the value. In this way there is only one copy of the value at any time, and the caller and callee both access that one value through pointers.

Some languages support reference parameters automatically. C does not do this -- the programmer must implement reference parameters manually using the existing pointer constructs in the language.

Swap Example

The classic example of wanting to modify the caller's memory is a `swap()` function which exchanges two values. Because C uses call by value, the following version of Swap will not work...

```
void Swap(int x, int y) {        // NO does not work
   int temp;

   temp = x;
   x = y;      // these operations just change the local x,y,temp
   y = temp;   // -- nothing connects them back to the caller's a,b
}


// Some caller code which calls Swap()...
int a = 1;
int b = 2;
Swap(a, b);
```

`Swap()` does not affect the arguments `a` and `b` in the caller. The function above only operates on the copies of `a` and `b` local to `Swap()` itself. This is a good example of how "local" memory such as (x, y, temp) behaves -- it exists independent of everything else only while its owning function is running. When the owning function exits, its local memory disappears.

Reference Parameter Technique

To pass an object X as a reference parameter, the programmer must pass a pointer to X instead of X itself. The formal parameter will be a pointer to the value of interest. The caller will need to use & or other operators to compute the correct pointer actual parameter. The callee will need to dereference the pointer with * where appropriate to access the value of interest. Here is an example of a correct Swap() function.

```
static void Swap(int* x, int* y) {     // params are int* instead of int
   int temp;

   temp = *x;        // use * to follow the pointer back to the caller's memory
   *x = *y;
   *y = temp;
}
```

```
// Some caller code which calls Swap()...
int a = 1;
int b = 2;

Swap(&a, &b);
```

Things to notice...

- The formal parameters are int* instead of int.
- The caller uses & to compute pointers to its local memory (a,b).
- The callee uses * to dereference the formal parameter pointers back to get the caller's memory.

Since the operator & produces the address of a variable -- `&a` is a pointer to `a`. In `Swap()` itself, the formal parameters are declared to be pointers, and the values of interest (a,b) are accessed through them. There is no special relationship between the **names** used for the actual and formal parameters. The function call matches up the actual and formal parameters by their order -- the first actual parameter is assigned to the first formal parameter, and so on. I deliberately used different names (a,b vs x,y) to emphasize that the names do not matter.

const

The qualifier const can be added to the left of a variable or parameter type to declare that the code using the variable will not change the variable. As a practical matter, use of `const` is very sporadic in the C programming community. It does have one very handy use, which is to clarify the role of a parameter in a function prototype...

```
void foo(const struct fraction* fract);
```

In the foo() prototype, the `const` declares that foo() does not intend to change the struct fraction pointee which is passed to it. Since the fraction is passed by pointer, we could not know otherwise if foo() intended to change our memory or not. Using the `const`, foo() makes its intentions clear. Declaring this extra bit of information helps to clarify the role of the function to its implementor and caller.

Bigger Pointer Example

The following code is a large example of using reference parameters. There are several common features of C programs in this example...Reference parameters are used to allow the functions Swap() and IncrementAndSwap() to affect the memory of their callers. There's a tricky case inside of IncrementAndSwap() where it calls Swap() -- no additional use of & is necessary in this case since the parameters x, y inside InrementAndSwap() are already pointers to the values of interest. The names of the variables through the program(a, b, x, y, alice, bob) do not need to match up in any particular way for the parameters to work. The parameter mechanism only depends on the types of the parameters and their order in the parameter list -- not their names. Finally this is an example of what multiple functions look like in a file and how they are called from the main() function.

```
static void Swap(int* a, int* b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

static void IncrementAndSwap(int* x, int* y) {
   (*x)++;
   (*y)++;
   Swap(x, y);       // don't need & here since a and b are already
                     // int*'s.
}


int main()
{
   int alice = 10;
   int bob = 20;

   Swap(&alice, &bob);
   // at this point alice==20 and bob==10

   IncrementAndSwap(&alice, &bob);
   // at this point alice==11 and bob==21

   return 0;
}
```

Section 5
Odds and Ends

main()
The execution of a C program begins with function named main(). All of the files and libraries for the C program are compiled together to build a single program file. That file must contain exactly one main() function which the operating system uses as the starting point for the program. Main() returns an int which, by convention, is 0 if the program completed successfully and non-zero if the program exited due to some error condition. This is just a convention which makes sense in shell oriented environments such as Unix or DOS.

Multiple Files
For a program of any size, it's convenient to separate the functions into several separate files. To allow the functions in separate files to cooperate, and yet allow the compiler to work on the files independently, C programs typically depend on two features...

Prototypes
A "prototype" for a function gives its name and arguments but not its body. In order for a caller, in any file, to use a function, the caller must have seen the prototype for that function. For example, here's what the prototypes would look like for `Twice()` and `Swap()`. The function body is absent and there's a semicolon (;) to terminate the prototype...

```
int Twice(int num);
void Swap(int* a, int* b);
```

In pre-ANSI C, the rules for prototypes where very sloppy -- callers were not required to see prototypes before calling functions, and as a result it was possible to get in situations where the compiler generated code which would crash horribly.

In ANSI C, I'll oversimplify a little to say that...

1) a function may be declared `static` in which case it can only be used in the same file where it is used below the point of its declaration. Static functions do not require a separate prototype so long as they are defined before or above where they are called which saves some work.

2) A non-static function needs a prototype. When the compiler compiles a function definition, it must have previously seen a prototype so that it can verify that the two are in agreement ("prototype before definition" rule). The prototype must also be seen by any client code which wants to call the function ("clients must see prototypes" rule).(The require-prototypes behavior is actually somewhat of a compiler option, but it's smart to leave it on.)

Preprocessor
The preprocessing step happens to the C source before it is fed to the compiler. The two most common preprocessor directives are #define and #include...

#define

The #define directive can be used to set up symbolic replacements in the source. As with all preprocessor operations, #define is extremely unintelligent -- it just does textual replacement without understanding. #define statements are used as a crude way of establishing symbolic constants.

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

Later code can use the symbols MAX or SEVEN_WORDS which will be replaced by the text to the right of each symbol in its #define.

#include

The "#include" directive brings in text from different files during compilation. #include is a very unintelligent and unstructured -- it just pastes in the text from the given file and continues compiling. The #include directive is used in the .h/.c file convention below which is used to satisfy the various constraints necessary to get prototypes correct.

```
#include "foo.h"     // refers to a "user" foo.h file --
                     //    in the originating directory for the compile

#include <foo.h>     // refers to a "system" foo.h file --
                     //    in the compiler's directory somewhere
```

foo.h vs foo.c

The universally followed convention for C is that for a file named "foo.c" containing a bunch of functions...

- A separate file named `foo.h` will contain the prototypes for the functions in `foo.c` which clients may want to call. Functions in `foo.c` which are for "internal use only" and should never be called by clients should be declared `static`.

- Near the top of `foo.c` will be the following line which ensures that the function definitions in `foo.c` see the prototypes in `foo.h` which ensures the "prototype before definition" rule above.
  ```
  #include "foo.h"    // show the contents of "foo.h"
                      // to the compiler at this point
  ```

- Any `xxx.c` file which wishes to call a function defined in `foo.c` must include the following line to see the prototypes, ensuring the "clients must see prototypes" rule above.
  ```
  #include "foo.h"
  ```

#if

At compile time, there is some space of names defined by the #defines. The #if test can be used at compile-time to look at those symbols and turn on and off which lines the compiler uses. The following example depends on the value of the FOO #define symbol. If it is true, then the "aaa" lines (whatever they are) are compiled, and the "bbb" lines are ignored. If FOO were 0, then the reverse would be true.

```
#define FOO 1

...

#if FOO
   aaa
   aaa
#else
   bbb
   bbb
#endif
```

You can use `#if 0 ...#endif` to effectively comment out areas of code you don't want to compile, but which you want to keeep in the source file.

Multiple #includes -- #pragma once

There's a problem sometimes where a .h file is #included into a file more than one time resulting in compile errors. This can be a serious problem. Because of this, you want to avoid #including .h files in other .h files if at all possible. On the other hand, #including .h files in .c files is fine. If you are lucky, your compiler will support the #pragma once feature which automatically prevents a single file from being #included more than once in any one file. This largely solves multiple #include problems.

```
// foo.h
// The following line prevents problems in files which #include "foo.h"
#pragma once

<rest of foo.h ...>
```

Assert

Array out of bounds references are an extremely common form of C run-time error. You can use the assert() function to sprinkle your code with your own bounds checks. A few seconds putting in assert statements can save you hours of debugging.

Getting out all the bugs is the hardest and scariest part of writing a large piece of software. Assert statements are one of the easiest and most effective helpers for that difficult phase.

```
#include <assert.h>
#define MAX_INTS 100
{
   int ints[MAX_INTS];
   i = foo(<something complicated>);   // i should be in bounds,
                                       // but is it really?
   assert(i>=0);           // safety assertions
   assert(i<MAX_INTS);

   ints[i] = 0;
```

Depending on the options specified at compile time, the assert() expressions will be left in the code for testing, or may be ignored. For that reason, it is important to only put expressions in assert() tests which do not need to be evaluated for the proper functioning of the program...

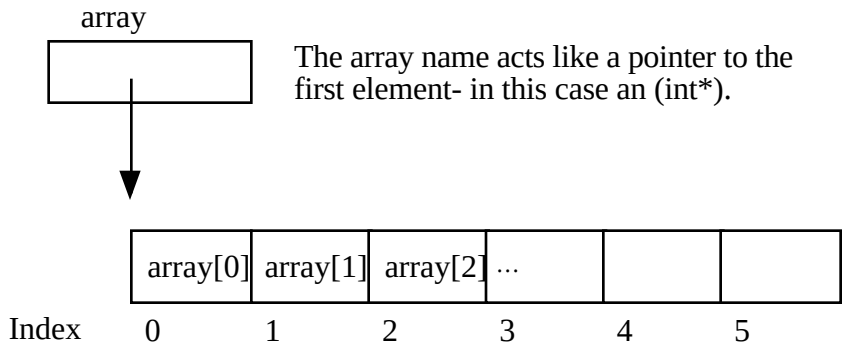
```
int errCode = foo();       // yes
assert(errCode == 0);


assert(foo() == 0);        // NO, foo() will not be called if
                           // the compiler removes the assert()
```

Section 6
Advanced Arrays and Pointers

Advanced C Arrays

In C, an array is formed by laying out all the elements contiguously in memory. The square bracket syntax can be used to refer to the elements in the array. The array as a whole is referred to by the address of the first element which is also known as the "base address" of the whole array.

```
{
   int array[6];

   int sum = 0;
   sum += array[0] + array[1];   // refer to elements using []
}
```

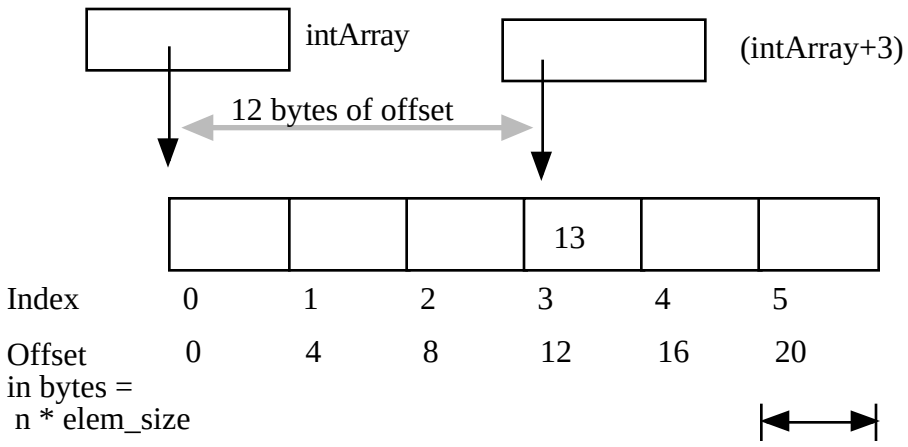


The programmer can refer to elements in the array with the simple [ ] syntax such as `array[1]`. This scheme works by combining the base address of the whole array with the index to compute the base address of the desired element in the array. It just requires a little arithmetic. Each element takes up a fixed number of bytes which is known at compile-time. So the address of element `n` in the array using 0 based indexing will be at an offset of (n * element_size) bytes from the base address of the whole array.

address of nth element = address_of_0th_element + (n * element_size_in_bytes)

The square bracket syntax [ ] deals with this address arithmetic for you, but it's useful to know what it's doing. The [ ] takes the integer index, multiplies by the element size, adds the resulting offset to the array base address, and finally dereferences the resulting pointer to get to the desired element.

```
{
   int intArray[6];

   intArray[3] = 13;
}
```



'+' Syntax

In a closely related piece of syntax, a + between a pointer and an integer does the same offset computation, but leaves the result as a pointer. The square bracket syntax gives the nth element while the + syntax gives a pointer to the nth element.

So the expression `(intArray + 3)` is a pointer to the integer `intArray[3]`. `(intArray + 3)` is of type `(int*)` while `intArray[3]` is of type `int`. The two expressions only differ by whether the pointer is dereferenced or not. So the expression `(intArray + 3)` is exactly equivalent to the expression `(&(intArray[3]))`. In fact those two probably compile to exactly the same code. They both represent a pointer to the element at index 3.

Any `[]` expression can be written with the + syntax instead. We just need to add in the pointer dereference. So `intArray[3]` is exactly equivalent to `*(intArray + 3)`. For most purposes, it's easiest and most readable to use the `[]` syntax. Every once in a while the + is convenient if you needed a pointer to the element instead of the element itself.

Pointer++ Style -- strcpy()

If `p` is a pointer to an element in an array, then `(p+1)` points to the next element in the array. Code can exploit this using the construct p++ to step a pointer over the elements in an array. It doesn't help readability any, so I can't recommend the technique, but you may see it in code written by others.

(This example was originally inspired by Mike Cleron) There's a library function called `strcpy(char* destination, char* source)` which copies the bytes of a C string from one place to another. Below are four different implementations of strcpy() written in order: from most verbose to most cryptic. In the first one, the normally straightforward while loop is actually sortof tricky to ensure that the terminating null character is copied over. The second removes that trickiness by moving assignment into the test. The last two are cute (and they demonstrate using ++ on pointers), but not really the sort of code you want to maintain. Among the four, I think strcpy2() is the best stylistically. With a smart compiler, all four will compile to basically the same code with the same efficiency.

```
// Unfortunately, a straight while or for loop won't work.
// The best we can do is use a while (1) with the test
// in the middle of the loop.
void strcpy1(char dest[], const char source[]) {
   int i = 0;

   while (1) {
      dest[i] = source[i];
      if (dest[i] == '\0') break;      // we're done
      i++;
   }
}


// Move the assignment into the test
void strcpy2(char dest[], const char source[]) {
   int i = 0;

   while ((dest[i] = source[i]) != '\0') {
      i++;
   }
}


// Get rid of i and just move the pointers.
// Relies on the precedence of * and ++.
void strcpy3(char dest[], const char source[])
{
   while ((*dest++ = *source++) != '\0') ;
}


// Rely on the fact that '\0' is equivalent to FALSE
void strcpy4(char dest[], const char source[])
{
   while (*dest++ = *source++) ;
}
```

Pointer Type Effects

Both [ ] and + implicitly use the compile time type of the pointer to compute the element_size which affects the offset arithmetic. When looking at code, it's easy to assume that everything is in the units of bytes.

```
int *p;

p = p + 12;    // at run-time, what does this add to p? 12?
```

The above code does not add the number 12 to the address in p-- that would increment p by 12 **bytes**. The code above increments p by 12 **ints**. Each int probably takes 4 bytes, so at run time the code will effectively increment the address in p by 48. The compiler figures all this out based on the type of the pointer.

Using casts, the following code really does just add 12 to the address in the pointer p. It works by telling the compiler that the pointer points to `char` instead of `int`. The size of `char` is defined to be exactly 1 byte (or whatever the smallest addressable unit is on the computer). In other words, `sizeof(char)` is always 1. We then cast the resulting

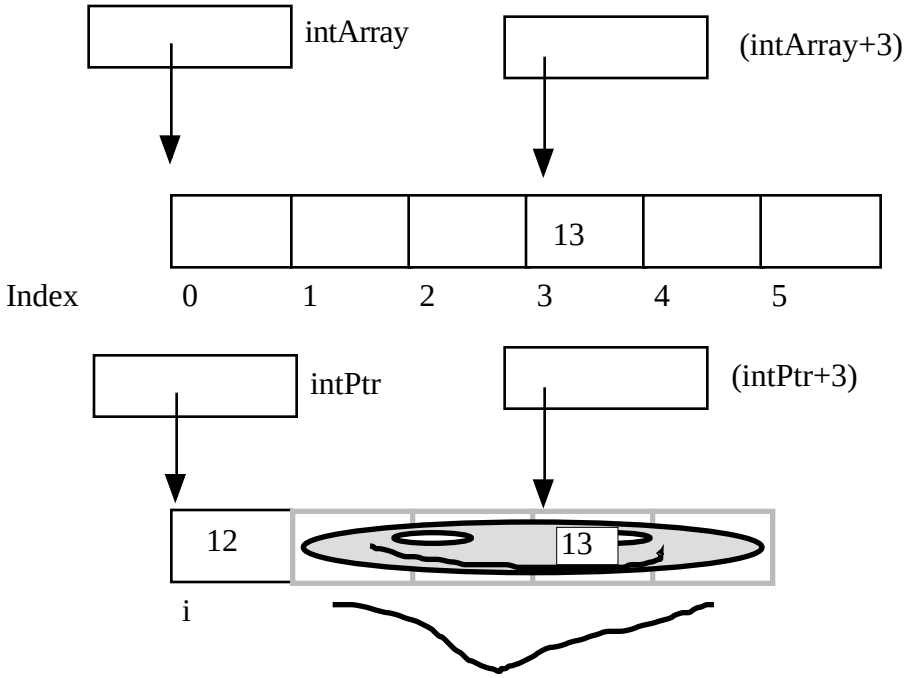
`(char*)` back to an `(int*)`. The programmer is allowed to cast any pointer type to any other pointer type like this to change the code the compiler generates.

```
p = (int*) ( ((char*)p) + 12);
```

Arrays and Pointers

One effect of the C array scheme is that the compiler does not distinguish meaningfully between arrays and pointers-- they both just look like pointers. In the following example, the value of `intArray` is a pointer to the first element in the array so it's an `(int*)`. The value of the variable `intPtr` is also `(int*)` and it is set to point to a single integer `i`. So what's the difference between `intArray` and `intPtr`? Not much as far as the compiler is concerned. They are both just (`int*`) pointers, and the compiler is perfectly happy to apply the `[]` or `+` syntax to either. It's the programmer's responsibility to ensure that the elements referred to by a `[]` or `+` operation really are there. Really its' just the same old rule that C doesn't do any bounds checking. C thinks of the single integer `i` as just a sort of degenerate array of size 1.

```
{
   int intArray[6];
   int *intPtr;
   int i;

   intPtr = &i;

   intArray[3] = 13;       // ok
   intPtr[0] = 12;         // odd, but ok. Changes i.
   intPtr[3] = 13;         // BAD! There is no integer reserved here!
}
```



These bytes exist, but they have not been explicitly reserved. They are the bytes which happen to be adjacent to the memory for i. They are probably being used to store something already, such as a smashed looking smiley face. The 13 just gets blindly written over the smiley face. This error will only be apparent later when the program tries to read the smiley face data.

Array Names Are Const

One subtle distinction between an array and a pointer, is that the pointer which represents the base address of an array cannot be changed in the code. The array base address behaves like a `const` pointer. The constraint applies to the name of the array where it is declared in the code-- the variable `ints` in the example below.

```
{
   int ints[100]
   int *p;
   int i;

   ints = NULL;            // NO, cannot change the base addr ptr
   ints = &i;              // NO
   ints = ints + 1;        // NO
   ints++;                 // NO

   p = ints;         // OK, p is a regular pointer which can be changed
                     // here it is getting a copy of the ints pointer

   p++;              // OK, p can still be changed (and ints cannot)
   p = NULL;         // OK
   p = &i;           // OK

   foo(ints);        // OK (possible foo definitions are below)
}
```

Array parameters are passed as pointers. The following two definitions of foo look different, but to the compiler they mean exactly the same thing. It's preferable to use whichever syntax is more accurate for readability. If the pointer coming in really is the base address of a whole array, then use [ ].

```
void foo(int arrayParam[]) {
   arrayParam = NULL;      // Silly but valid. Just changes the local pointer
}

void foo(int *arrayParam) {
   arrayParam = NULL;      // ditto
}
```

Heap Memory

C gives programmers the standard sort of facilities to allocate and deallocate dynamic heap memory. A word of warning: writing programs which manage their heap memory is notoriously difficult. This partly explains the great popularity of languages such as Java and Perl which handle heap management automatically. These languages take over a task which has proven to be extremely difficult for the programmer. As a result Perl and Java programs run a little more slowly, but they contain far fewer bugs. (For a detailed discussion of heap memory see http://cslibrary.stanford.edu/102/, Pointers and Memory.)

C provides access to the heap features through library functions which any C code can call. The prototypes for these functions are in the file <stdlib.h>, so any code which wants to call these must #include that header file. The three functions of interest are...

`void* malloc(size_t size)` Request a contiguous block of memory of the given size in the heap. malloc() returns a pointer to the heap block or NULL if the request could not be satisfied. The type `size_t` is essentially an `unsigned long` which indicates how large a block the caller would like measured in bytes. Because the block pointer returned by malloc() is a void* (i.e. it makes no claim about the type of its pointee), a cast will probably be required when storing the void* pointer into a regular typed pointer.

`void free(void* block)` The mirror image of malloc() -- free takes a pointer to a heap block earlier allocated by malloc() and returns that block to the heap for re-use. After the free(), the client should not access any part of the block or assume that the block is valid memory. The block should not be freed a second time.

`void* realloc(void* block, size_t size);` Take an existing heap block and try to relocate it to a heap block of the given size which may be larger or smaller than the original size of the block. Returns a pointer to the new block, or NULL if the relocation was unsuccessful. Remember to catch and examine the return value of realloc() -- it is a common error to continue to use the old block pointer. Realloc() takes care of moving the bytes from the old block to the new block. Realloc() exists because it can be implemented using low-level features which make it more efficient than C code the client could write.

Memory Management

All of a program's memory is deallocated automatically when the it exits, so a program only needs to use free() during execution if it is important for the program to recycle its memory while it runs -- typically because it uses a lot of memory or because it runs for a

long time. The pointer passed to free() must be exactly the pointer which was originally returned by malloc() or realloc(), not just a pointer into somewhere within the heap block.

Dynamic Arrays

Since arrays are just contiguous areas of bytes, you can allocate your own arrays in the heap using malloc(). The following code allocates two arrays of 1000 ints-- one in the stack the usual "local" way, and one in the heap using malloc(). Other than the different allocations, the two are syntactically similar in use.

```
{
   int a[1000];

   int *b;
   b = (int*) malloc( sizeof(int) * 1000);
   assert(b != NULL);      // check that the allocation succeeded

   a[123] = 13;      // Just use good ol' [] to access elements
   b[123] = 13;      // in both arrays.

   free(b);
}
```

Although both arrays can be accessed with [ ], the rules for their maintenance are very different....

Advantages of being in the heap

- Size (in this case 1000) can be defined at run time. Not so for an array like "a".
- The array will exist until it is explicitly deallocated with a call to free().
- You can change the size of the array at will at run time using realloc(). The following changes the size of the array to 2000. Realloc() takes care of copying over the old elements.

```
...
b = realloc(b, sizeof(int) * 2000);
assert(b != NULL);
```

Disadvantages of being in the heap

- You have to remember to allocate the array, and you have to get it right.
- You have to remember to deallocate it exactly once when you are done with it, and you have to get that right.
- The above two disadvantages have the same basic profile: if you get them wrong, your code still looks right. It compiles fine. It even runs for small cases, but for some input cases it just crashes unexpectedly because random memory is getting overwritten somewhere like the smiley face. This sort of "random memory smasher" bug can be a real ordeal to track down.

Dynamic Strings

The dynamic allocation of arrays works very well for allocating strings in the heap. The advantage of heap allocating a string is that the heap block can be just big enough to store the actual number of characters in the string. The common local variable technique such as `char string[1000];` allocates way too much space most of the time, wasting the unused bytes, and yet fails if the string ever gets bigger than the variable's fixed size.

```
#include <string.h>

/*
 Takes a c string as input, and makes a copy of that string
 in the heap. The caller takes over ownership of the new string
 and is responsible for freeing it.
*/
char* MakeStringInHeap(const char* source) {
   char* newString;

   newString = (char*) malloc(strlen(source) + 1); // +1 for the '\0'
   assert(newString != NULL);
   strcpy(newString, source);
   return(newString);
}
```

Section 7
Details and Library Functions

Precedence and Associativity

```
function-call() [] -> .                                  L to R

! ~ ++ -- + - *(ptr deref) sizeof &(addr of)             R to L
(all unary ops are the same)

* / %                                                   L to R
(the top tier arithmetic binary ops)

+ -                                                     L to R
(second tier arithmetic binary ops)

< <= > >=                                               L to R

== !=                                                   L to R

in order: & ^ | && ||                                   L to R
(note that bitwise comes before boolean)

= and all its variants                                  R to L

, (comma)  .                                            L to R

A combinations which never works right without parens: *structptr.field
You have to write it as (*structptr).field or structptr->field
```

Standard Library Functions

Many basic housekeeping funcions are available to a C program in form of standard library functions. To call these, a program must #include the appropriate .h file. Most compilers link in the standard library code by default. The functions listed in the next section are the most commonly used ones, but there are many more which are not listed here.

stdio.h	file input and output
ctype.h	character tests
string.h	string operations
math.h	mathematical functions such as sin() and cos()
stdlib.h	utility functions such as malloc() and rand()
assert.h	the assert() debugging macro
stdarg.h	support for functions with variable numbers of arguments
setjmp.h	support for non-local flow control jumps
signal.h	support for exceptional condition signals
time.h	date and time

limits.h, float.h constants which define type range values such as INT_MAX

stdio.h

Stdio.h is a very common file to #include -- it includes functions to print and read strings from files and to open and close files in the file system.

`FILE* fopen(const char* fname, const char* mode);`
Open a file named in the filesystem and return a FILE* for it. Mode = "r" read,"w" write,"a"append, returns NULL on error. The standard files `stdout`, `stdin`, `stderr` are automatically opened and closed for you by the system.

`int fclose(FILE* file);`
Close a previously opened file. Returns EOF on error. The operating system closes all of a program's files when it exits, but it's tidy to do it beforehand. Also, there is typically a limit to the number of files which a program may have open simultaneously.

`int fgetc(FILE* in);`
Read and return the next unsigned char out of a file, or EOF if the file has been exhausted. (detail) This and other file functions return ints instead of a chars because the EOF constant they potentially is not a `char`, but is an `int`. getc() is an alternate, faster version implemented as a macro which may evaluate the FILE* expression more than once.

`char* fgets(char* dest, int n, FILE* in)`
Reads the next line of text into a string supplied by the caller. Reads at most n-1 characters from the file, stopping at the first '\n' character. In any case, the string is '\0' terminated. The '\n' is included in the string. Returns NULL on EOF or error.

`int fputc(int ch, FILE* out);`
Write the char to the file as an unsigned char. Returns ch, or EOF on err. putc() is an alternate, faster version implemented as a macro which may evaluate the FILE* expression more than once.

`int ungetc(int ch, FILE* in);`
Push the most recent fgetc() char back onto the file. EOF may not be pushed back. Returns ch or EOF on error.

`int printf(const char* format_string, ...);`
Prints a string with values possibly inserted into it to standard output. Takes a variable number of arguments -- first a format string followed by a number of matching arguments. The format string contains text mixed with % directives which mark things to be inserted in the output. %d = int, %Ld=long int, %s=string, %f=double, %c=char. Every % directive must have a matching argument of the correct type after the format string. Returns the number of characters written, or negative on error. If the percent directives do not match the number and type of arguments, printf() tends to crash or otherwise do the wrong thing at run time. fprintf() is a variant which takes an additional FILE* argument which specifies the file to print to. Examples...

printf("hello\n");	prints: hello
printf("hello %d there %d\n", 13, 1+1);	prints: hello 13 there 2
printf("hello %c there %d %s\n", 'A', 42, "ok");	prints: hello A there 42 ok

`int scanf(const char* format, ...)`

Opposite of printf() -- reads characters from standard input trying to match elements in the format string. Each percent directive in the format string must have a matching pointer in the argument list which scanf() uses to store the values it finds. scanf() skips whitespace as it tries to read in each percent directive. Returns the number of percent directives processed successfully, or EOF on error. scanf() is famously sensitive to programmer errors. If scanf() is called with anything but the correct pointers after the format string, it tends to crash or otherwise do the wrong thing at run time. sscanf() is a variant which takes an additional initial string from which it does its reading. fscanf() is a variant which takes an additional initial FILE* from which it does its reading. Example...

```
{
   int num;
   char s1[1000];
   char s2[1000];

   scanf("hello %d %s %s", &num, s1, s2);
}
```

Looks for the word "hello" followed by a number and two words (all separated by whitespace). scanf() uses the pointers `&num`, `s1`, and `s2` to store what it finds into the local variables.

ctype.h

ctype.h includes macros for doing simple tests and operations on characters

`isalpha(ch)` // ch is an upper or lower case letter

`islower(ch), isupper(ch)` // same as above, but upper/lower specific

`isspace(ch)` // ch is a whitepace character such as tab, space, newline, etc.

`isdigit(ch)` // digit such as '0'..'9'

`toupper(ch), tolower(ch)` // Return the lower or upper case version of a alphabetic character, otherwise pass it through unchanged.

string.h

None of these string routines allocate memory or check that the passed in memory is the right size. The caller is responsible for making sure there is "enough" memory for the operation. The type `size_t` is an unsigned integer wide enough for the computer's address space -- most likely an `unsigned long`.

```
size_t strlen(const char* string);
```
Return the number of chars in a C string. EG strlen("abc")==3

```
char* strcpy(char* dest, const char* source);
```
Copy the characters from the source string to the destination string.

```
size_t strlcpy(char* dest, const char* source,
                   size_t dest_size);
```
Like strcpy(), but knows the size of the dest. Truncates if necessary. Use this to avoid memory errors and buffer-overflow security problems. This function is not as standard as strcpy(), but most sytems have it. Do not use the old strncpy() function -- it is difficult to use correctly.

```
char *strcat(char* dest, const char* source);
```
Append the characters from the source string to the end of destination string. (There is a non-standard strlcat() variant that takes the size of the dest as third argument.)

```
int strcmp(const char* a, const char* b);
```
Compare two strings and return an int which encodes their ordering. zero:a==b, negative:a<b, positive:a>b. It is a common error to think of the result of strcmp() as being boolean true if the strings are equal which is, unfortunately, exactly backwards.

```
char* strchr(const char* searchIn, char ch);
```
Search the given string for the first occurence of the given character. Returns a pointer to the character, or NULL if none is found.

```
char* strstr(const char* searchIn, const char* searchFor);
```
Similar to strchr(), but searches for an entire string instead of a single character. The search is case sensitive.

```
void* memcpy(void* dest, const void* source, size_t n);
```
Copy the given number of bytes from the source to the destination. The source and destination must not overlap. This may be implemented in a specialized but highly optimized way for a particular computer.

```
void* memmove(void* dest, const void* source, size_t n);
```
Similar to memcpy() but allows the areas to overlap. This probably runs slightly slower than memcpy().

stdlib.h

`int rand();`
Returns a pseudo random integer in the range 0..RAND_MAX (limits.h) which is at least 32767.

`void srand(unsigned int seed);`
The sequence of random numbers returned by rand() is initially controlled by a global "seed" variable. srand() sets this seed which, by default, starts with the value 1. Pass the expression `time(NULL)` (time.h) to set the seed to a value based on the current time to ensure that the random sequence is different from one run to the next.

`void* malloc(size_t size);`
Allocate a heap block of the given size in bytes. Returns a pointer to the block or NULL on failure. A cast may be required to store the void* pointer into a regular typed pointer. [ed: see the Heap Allocation section above for the longer discussion of malloc(), free(), and realloc()]

`void free(void* block);`
Opposite of malloc(). Returns a previous malloc block to the system for reuse

`void* realloc(void* block, size_t size);`
Resize an existing heap block to the new size. Takes care of copying bytes from the old block to the new. Returns the new base address of the heap block. It is a common error to forget to catch the return value from realloc(). Returns NULL if the resize operation was not possible.

`void exit(int status);`
Halt and exit the program and pass a condition int back to the operating sytem. Pass 0 to signal normal program termination, non-zero otherwise.

`void* bsearch(const void* key, const void* base, size_t len, size_t elem_size, <compare_function>);`
Do a binary search in an array of elements. The last argument is a function which takes pointers to the two elements to compare. Its prototype should be:
int compare(const void* a, const void* b);, and it should return 0, -1, or 1 as strcmp() does. Returns a pointer to a found element, or NULL otherwise. Note that strcmp() itself cannot be used directly as a compare function for bsearch() on an array of char* strings because strcmp() takes char* arguments and bsearch() will need a comparator that takes pointers to the array elements -- char**.

`void qsort(void* base, size_t len, size_t elem_size, <compare_function>);`
Sort an array of elements. Takes a function pointer just like besearch().

Revision History

11/1998 -- original major version. Based on my old C handout for CS107. Thanks to Jon Becker for proofreading and Mike Cleron for the original inspiration.

Revised 4/2003 with many helpful typo and other suggestions from Negar Shamma and A. P. Garcia