

C Programming Course

Tom Macke

Introduction

Course goals.

To learn enough about C to be able to write useful programs.

To learn enough to be able to learn more on your own.

Requirements.

Students need to have some familiarity with both Unix and a text editor. A C language book is necessary. I recommend Kernighan and Ritchie:

Kernighan, B.W., and Ritchie, D.M. *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1988.

Disclaimer.

Many simple tasks can be done with awk, which is simpler and easier to use than C. The point is to write a C program as the last resort!

Creating and Running C programs.

C program files have names that end with .c To compile a C program type

```
cc file.c
```

The compiled program will be written to the file a.out. To run the program type

```
a.out.
```

If the program uses the standard input and output, these may be redirected in the usual way:

```
a.out < data > output
```

C Language Elements.

Identifiers. Identifiers are used to name parts of programs. They are sequences of letters and digits beginning with a letter. The underscore ('_') is a letter. Upper and lower case letters are distinct:

```
i I EOF write_res
```

Reserved words. These identifiers have special significance in C:

```
auto double int struct  
break else long switch  
case enum register typedef  
char extern return union  
const float short unsigned  
continue for signed void  
default goto sizeof volatile  
do if static while
```

C Programming Course

Literal Constants. C supports five types of literal constants - integers, doubles, enumerated, chars and strings.

Integers are sequences of digits:

```
3 917 9882887
```

Double constants are "real" numbers. They must include a decimal point, an exponent or both:

```
3.14 1e-3 6.023E23
```

A char constant is one character enclosed in single quotes (''). To get a literal single quote precede it with a backslash ('\');

```
'a' '\'' '\\'
```

String constants are arrays of zero or more characters. They are enclosed in double quotes (""). A literal double quote is placed in a string by preceding it with a backslash:

```
"A string"  
"A string with a double quote \""  
"A string with a backslash \\"
```

Some things to remember about strings and chars:

```
"" is string with length zero
```

```
'' is an error
```

```
"A" and 'A' are very different
```

Escape sequences. These special character combinations are used to introduce control and other non-printing characters into char and string constants. They all begin with a backslash followed by a letter and/or 1, 2, or 3 digits:

```
\a bell (audible alarm)  
\b backspace  
\f formfeed  
\n newline  
\r carriage return  
\t horizontal tab  
\v vertical tab  
\? question mark (for trigraphs)  
\' single quote  
\" double quote  
\ backslash  
\ooo ooo is 1-3 octal digits  
\xhh hh is 1 or 2 hex digits
```

Comments. The two character symbol `/*` starts a comment which includes everything up to and including the first `*/`. Comments do not nest.

Preprocessor interface. Lines beginning with `#`'s are C preprocessor directives. The preprocessor will be covered later.

Operators. Operators are special characters that are used to construct expressions. C includes an extensive set of operators, something that can make C programs cryptic. The operators are:

<code>+</code>	add (can be unary)
<code>-</code>	subtract, unary negate
<code>*</code>	multiply, reference through pointer
<code>/</code>	divide
<code>%</code>	modulus
<code><</code>	less than
<code><=</code>	less than or equal
<code>></code>	greater than
<code>>=</code>	greater than or equal
<code>==</code>	equal
<code>!=</code>	not equal
<code>&</code>	Bit AND, unary address of
<code>^</code>	Bit EXCLUSIVE OR
<code> </code>	Bit INCLUSIVE OR
<code>~</code>	Bit COMPLEMENT
<code><<</code>	left shift
<code>>></code>	right shift
<code>&&</code>	Boolean AND
<code> </code>	Boolean OR
<code>!</code>	Boolean NOT
<code>?:</code>	Conditional operator
<code>=</code>	Assignment
<code>+=</code>	+=
<code>-=</code>	-=
<code>*=</code>	*=
<code>/=</code>	/=
<code>%=</code>	%=
<code>&=</code>	&=
<code>^=</code>	^=
<code> =</code>	=
<code><<=</code>	<<=
<code>>>=</code>	>>=
<code>++</code>	increment
<code>--</code>	decrement
<code>-></code>	point to field
<code>.</code>	select field
<code>[]</code>	select array element
<code>(type)</code>	type "cast"
<code>sizeof</code>	return size in bytes of operand
<code>()</code>	group subexpression
<code>{ }</code>	separate subexpressions

Other symbols. Statements are terminated with a semicolon `;`. They curly braces `{ ' ' }` are used to group statements and to define structures. Only three printable characters `'@', '$', '.'` are not used.

Two simple C programs.

These two programs demonstrate many features that are present in nearly all C programs. They both begin with the line

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

This tells the C preprocessor to replace this line with the contents of the file `stdio.h`. This file contains many useful definitions for performing simple I/O. Almost every C program will contain this line.

Both programs are called `main`. Every C program begins with the function `main`.

The curly braces `{ ' ' }` are used to group things. They are used in both programs to define the function `main`. They are also used in the second program to group two statements together as the body of the `while` loop.

The first program includes a string with escape sequence `\n`. This indicates that the string is to end with a newline.

Hello world program. This is the traditional first C program. It writes the line "Hello, world" on your terminal screen.

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

```
main()
{
    printf( "Hello, world\n" );
}
```

File copy program. This copies characters from the standard input to the standard output. Input redirection allows it copy files.

```
#include <stdio.h>

main()
{
    int    c;

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

Variables, Expressions, if, printf().

Variables and declarations.

A variable is named region of the computer's main memory that can hold data. This data is accessed symbolically via the variable's name. Every variable has a type which determines its size and how to interpret the bit pattern it holds.

A declaration is a program construct that associates a variable name with a type. C requires that all variables be declared before they are used. Declarations are restricted to two places in a C program. They must either be outside of any function, (i.e. not between any { } or if inside a function, they must precede any executable or action statement.

Fundamental types.

C supports four data types. In order of increasing size they are char, int, float and double. The syntax of a simple type declaration is:

```
type    var1, var2, ... ;
```

where type is one of the four simple type names and the var's are identifiers:

```
char    c1, c2;
int     i, j, k;
float   x, y, z;
double  dx;
```

char. Variables of type char are used to hold single characters. They are really small integers with the range of -128 to 127. By preceding a char declaration with the reserved word unsigned, the range is changed to the non negative integers 0 to 255.

int. C provides several types of integers. There are standard integers declared with int. The reserved words short and long may be used in place of int to get short or long integers respectively. While a C compiler could support three types of integers, most support only two as int is a synonym for either short or long. short integers are either shorter or the same size as ordinary (int) integers; long integers are either longer or the same size as ordinary (int) integer; but short integers are always shorter than long integers. Integers are signed unless preceded by the reserved word unsigned.

On most Unix workstations int and long are both four byte integers with range -2^{31} to $2^{31}-1$. short integers use two bytes with range -2^{15} to $2^{15}-1$. The unsigned ranges are 0 to $2^{32}-1$ and 0 to $2^{16}-1$ respectively.

float and double. C provides two approximations of real numbers. double variables are always larger than float variables. On most Unix workstations the approximate ranges are 10^{-45} to 10^{38} for float and 10^{-324} to 10^{308} for double.

Expressions.

Expressions are composed of variables, literals and operators. They are used to create and/or transform data. They resemble expressions from algebra although there are some differences that arise from any computer's finite precision. C support six types of expressions: address, arithmetic, bit, relational, Boolean and assignment. Address and bit expressions will be discussed later.

Every operator has a precedence level which determines when the subexpression involving that operator will be evaluated. The higher the precedence the sooner the evaluation. C has a mind numbing 13 precedence levels! Most operators behave naturally although there are traps for the unwary. Parentheses () can and should be used to group subexpressions to assure they will be evaluated correctly. The approximate precedence order is address, arithmetic, bit, relational, Boolean and finally assignment. The complete precedence table is in Class 3.

Arithmetic expressions. Five arithmetic operators, + and - for addition and subtraction and *, / and % for multiplication, division and modulus. + and - have both infix (a + b) and prefix (-a) forms. As prefix operators they have the highest precedence and are evaluated first. The multiplying operators, *, / and % are evaluated next and then the infix adding operators + and -. The arithmetic operators associate in the natural fashion, from left to right.

```
int    a, b, c;

a = 2;    b = 3;    c = 5;
a + b * c    is 17
(a + b) * c    is 25
a + b + c is (a + b) + c    is 10
a + -b    is -1
a / b    is 0!
b / a    is 1!
a / 3.0    is 0.66666667
```

Note that if both operands to divide integers, the result is integer, i.e. truncated. If at least one operand to divide is float or double the result will be float or double.

Type conversion. C permits expressions involving mixed char, int, float and double variables (and literals). The rule is that if two operands have different types, the one with the smaller type is converted to the larger type. Thus char to int to float to double. The only to this rule is for assignment where the expression is converted to the type of the assigned variable.

Six comparison operators ==, !=, <, <=, >, >=, which represent equal, not equal, less than, less than or equal, greater than or equal and greater. Any two values of the same type may be tested for equality but only the basic types plus pointers may be tested for order.

Three Boolean operators &&, || and !, which stand for "and-fi", "or-else" and not. Not has the highest possible precedence along with the unary plus and minus. And-if has higher precedence then or-else. And-if and or-else have unusual behavior in that the evaluation stops as soon as the truth value of the expression is known. Thus a && b will not evaluate b if a is false; similarly a || b will not evaluate b if a is true. This can lead to some very strange program behavior so care is advised in using complex Boolean expressions.

An assignment takes the value of an expressions and stores it in a variable. Many computer languages like FORTRAN and Pascal treat the assignment as a statement and not an expression. C treats the assignment operator = as an ordinary expression operator and assignment expression can be used in any place where an expression is required including the tests in if, while, for and switch statements.

Several C language constructs, i.e. the tests in `if`, `while` and `for` statements require an expression that is either true or false. C considers zero to be false and not zero to be true. Not zero is anything that is not zero, so don't assume that its 1 or some other value.

`if` statement.

The `if` statement is used to select an action based on the value of an expression. A simple `if` statement executes the action if the expression is true and skips it if false. An `if/else` executes one of two mutually exclusive actions based on the expression. If true, the statement following the `if` part is executed, if false the statement following the `else` part is executed.

Simple `if` statement.

```
if( cond ) stmt_1
```

`if/else` statement

```
if( cond ) stmt_1
else      stmt_2
```

`cond` is any C language expression. If its value is not zero, the condition is true and `stmt_1` is executed. If the value of `cond` is zero, it is false and `stmt_1` is skipped. In the simple `if`, execution resumes at the statement following `stmt_1`; if the `if/else stmt_2` is executed.

`stmt_1` and `stmt_2` represent any single C statement including other `if` statements. In order for an `if` statement to control more than one statement they must be grouped into a block, i.e. surrounded by the curly braces, '{' and '}'.

```
if( len == LINELEN ) {
    printf( "%s\n", line );
    len = 0;
}
```

A problem arises when `stmt_1` is an `if` statement. Consider this program fragment:

```
if( a > 0 )
    if( a > b ) c = a;
    else      c = b;
```

Which `if` is the `else` paired with. The rule is that an `else` goes with nearest unpaired `if`. To force the `else` to be paired with the first `if` statement the inner `if` would have to be surrounded by {}:

```
if( a > 0 ) {
    if( a > b ) c = a;
} else      c = b;
```

A multi-way branch can be built by replacing `stmt_2` with an `if` statement:

```
if( c == 'a' )
    acnt++;
else if( c == 'c' )
    ccnt++;
else if( c == 'g' )
    gcnt++;
else if( c == 'u' )
    ucnt++;
else
```

`printf( "ERROR: unexpected char %c\n", c );`

The final `else` can be omitted or left in to catch "impossible" conditions.

`printf` statement.

The `printf()` function is used to write characters, including characters representing internal data to the standard output. This type of data conversion is called formatted I/O.

`printf()` has 1 or more arguments. The first argument is format string. `printf()` uses this format string to write an appropriate character representation of the values of the arguments following the format string.

A format string consists of ordinary characters and format descriptors. Format descriptors are paired left to right with arguments 2, 3, ... The number of format descriptors and arguments must agree. `printf()` works by examining the first character of the format string. If it is an ordinary character it is written to the standard output. If it is a '%', it signals the beginning of a format descriptor. `printf()` then fetches the value of the indicated argument, formats it and writes these character to the standard output. When the format string is exhausted, `printf()` returns.

Useful format descriptors:

%d	write an integer
%4d	write an integer right justified in a field width 4
%Nd	write an integer right justified in a field of width N
%f	write a floating point number in scientific notation
%5.1f	write a float right justified in width 5 with 1 decimal place, e.g. 3.4. The 5 and 1 can be other positive integers
%s	write a string
%%	write a %.

Sample program Unix utility "wc".

The Unix utility `wc` counts the number of characters, words and lines in a file. A word is defined as a sequence of non white space characters delimited by white space or the beginning or end of the file.

Count chars, Version 1.0. Here is the skeleton. It counts the number of characters in standard input. Notice how it close it to the copy file program from Class 1.

```
#include <stdio.h>

main()
{
    int    c, count;

    count = 0;
    c = getchar();
    while( c != EOF ){
        count = count + 1;
        c = getchar();
    }
    printf( "%4d\n", count );
}
```

Count chars, Version 1.1, written as real C.

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

```
main()
{
    int    c, count;

    count = 0;
    while( ( c = getchar() ) != EOF ){
        count++;
    }
    printf( "%4d\n", count );
}
```

Here the call to `getchar()` has been moved into the test of the while loop. This is possible because assignments are simple expressions. Also the expression `count = count + 1` has been replaced by `count++`, where the `++` means increment by 1.

Count chars, Version 2.0 add count lines.

```
#include <stdio.h>

main()
{
    int    c, count, lcount;

    lcount = count = 0;
    while( ( c = getchar() ) != EOF ){
        if( c == '\n' )
```

```
        lcount++;
        count++;
    }
    printf( "%4d %4d\n", lcount, count );
}
```

Added the new variable, `lcount` and an `if` statement that checks for newline `'\n'`.
Count chars, Version 3.0 add count "words".

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

```
main()
{
    int    c, count, wcount, lcount;
    int    inword;

    lcount = wcount = count = 0;
    inword = 0;
    while( ( c = getchar() ) != EOF ){
        if( c == '\n' ){
            lcount++;
            inword = 0;
        }
        else if( isspace( c ) )
            inword = 0;
        else if( !inword ){
            wcount++;
            inword = 1;
        }
        count++;
    }
    printf( "%4d %4d %4d\n", lcount, wcount, count );
}
```

Several changes. A new preprocessor line

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

This file contains several functions that are useful for classifying characters. Here I've used `isspace()` which is true if its argument is "white space". The original test for `'\n'` is left if because `isspace( '\n' )` is true. The variable `inword` is used to mark the first character of a word to insure that `wcount` is only incremented at the beginning of a word instead of at each character of a word.

Arrays, structs, pointers, operators

Structured types.

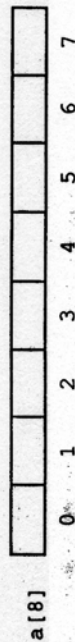
Arrays. An array is an area of storage containing one or more elements of the same type. This type can be any valid C type including structures, pointers and other arrays. The elements are numbered beginning with 0 and increase by 1 to $N-1$ where N is the size or number of elements in the array. Data in the array is accessed by evaluating an expression involving the base of the array and the number or index of the desired element.

Variables are declared to be arrays by following the variable name with square brackets ('[', ']'). The square brackets surround an optional expression specifying the size of the array. If the expression is present storage will be allocated for the array at compile time. If the expression is missing, storage must be explicitly allocated at run time before using the array. The size expression must be known at compile time. Here are some examples:

```

type      var [ size ]; syntax
int       a [ 8 ];    array of 8 integers
float     pos [ 3 ];  array of 3 floats
char      pname [];   pname is the name of a character
                        array, but no storage is allocated
    
```

Here is how memory for a is organized:



Multi-dimension arrays. An array can be associated with more than one index expression. Such arrays are multi-dimensional where the number of dimensions is equal to the number of indices required to access an element. A two dimensional array is sometimes called a matrix. Multi-dimensional arrays required N pairs of brackets. All brackets except the left most must surround an expression which specifies the size of that dimension. These expressions are required so the compiler can map indices onto the correct elements. If the leftmost expression is missing, the array is declared but no storage is allocated to it. Such an array must be allocated before it can be used.

```

float     xform [ 4 ] [ 4 ];    4 x 4 matrix
float     xform [ ] [ 4 ];      4 x 4 matrix, no storage
    
```

Strings. A string is special type of array of char. The difference is that is that the string ends with the first element that contains a 0. The length of the string including the zero byte must be less than or equal to the length of the underlying character array. This zero byte is part of all C strings including literals. Thus the size of a literal is always one more than the number of characters between the double quotes. The empty string "" contains 1 value, the number 0. When declaring character arrays that will hold strings always make them large enough to hold the final NUL.

Structures. A structure is a area of storage containing one of more elements of possibly different types. The elements are named and are accessed via an expression involving the structure variable and the element name.

A structure declaration consists of the reserved word struct, followed by the member list followed by the variable name. The member list is enclosed in braces '{, '}'. An optional identifier, called the structure tag may be inserted between struct and {.

```

struct      complex {          "tagged" structure
float       real;
float       imag;
} z;        "complex" variable z

struct      {
char        a_name [ A_NAME_SIZE ];
int         a_attr;
float       a_pos [ 3 ];      structure variable a1
} a1;
    
```

Pointers. A pointer is the address of a program element. Most pointers are pointers to data but C also supports pointers to functions (code). All pointers have a type and all pointers in an expression must have the same type. Every pointer type includes the value NULL, which points to nothing.

A variable is declared to be a pointer to type by preceding the variable name in the declaration with *, or by following it with []:

type	*var;	syntax
type	var [];	
int	*ip;	pointer to an int
char	cp [];	pointer to a char

Arrays and Pointers. Pointers and one dimensional arrays are very similar in C. In fact the array access operation a [i] is the same as *(a + i), a pointer operation.

Address Operators.

Indirection and Address of. Indirection is used to access data through pointers. The value of the pointer is read and used as the address of the data. Indirection is indicated by the unary prefix operator *. C also provides a way to determine the address of a variable. Do so by preceding the variable name with &. Taking the address of a variable generates a pointer to that variable.

```

int        i, j;
int        *ip;                pointer to an integer

i = 2;
ip = &i;                        take address of i and store in ip
j = *ip;                        indirect reference, j is now 2
    
```

Indexing. Indexing is used to access data in arrays. The array name is followed by the index expression surrounded by brackets '[', ']''. The value of the index expression must be between 0 and $N-1$, where N is the size of array. C does NOT do bounds checking so it is up to you to ensure that you follow this rule.

```
int    a[ 8 ];
int    i, j;
```

```
i = 2;
a[ 3 ] = 4;
j = a[ i ];
j = a[ 5 * i + 2 ]; fetch element 2
                    fetch element 12, an error!
```

Selection. Selection is used to access data in the members of a structure. C has two different selection operators, both infix: the dot '.', and the "points to" '->'. The left operand is the structure variable and the right operand is the member name.

```
struct complex {
    float    real, imag;
} c, *pc;
// declare structure var c and pointer pc
```

```
c.real = 3.0;
( *pc ).imag = -1.0;
pc->imag = -1.0; // same thing as preceding line
```

Since structure variables are often used with pointers, C provides the shorthand *ptr->member* for the longer *(*ptr). member*.

Bit operators.

C has six bit operators which are used to manipulate the bits of character and integer expressions. They are & for And, ^ for Exclusive Or, | for Inclusive Or, ~ for Complement and << and >> for left and right shift. A right shift of a signed expression replicates the sign bit.

```
/* NOT REAL C!!! */
char    a, b;
unsigned char ua;

ua = a = 10011011 // "bits" of a, ua
b = 01110010      // "bits" of b
a & b             is 00010010
a ^ b             is 11101001
a | b             is 11111011
~a               is 01100100
a << 1           is 00110110
a >> 1           is 11001101
b >> 1           is 00111001
ua >> 1          is 01001101
```

Reflexive operators.

The operation of updating the value of a variable is so common that C provides several special operators especially for this purpose. These reflexive assignments can be used to shorten expressions like

```
var = var op expr // (which is really) var = var op ( expr )
to
```

```
var op= expr.
```

Here *var* stands for any C variable, *expr* is a restricted C expression that does not involve *var* and *op* is one of + - * / % & ^ | << >>. No space is permitted between *op* and =. The parentheses are not needed if *expr* will be evaluated before *op*. An example will help:

```
i = i * j + k // is not the same as
i * = j + k    // because * is evaluated before + in the first
                // statement but after + in the second statement.
i = i * ( j + k ) // The reflexive assignment would have been
                  // correct only if this was the original assignment.
```

Increment and decrement operators.

C provides two special operators to be used where a reflexive assignments consists of adding or subtracting 1 to or from a variable. They are ++ for increment by 1 and -- for decrement by 1. These operators can be used as prefix or postfix operators with subtly different behavior:

```
int    i, j, k;

i = 1; j = 2; k = 0;
i++;    // i is now 2
++j;    // j is now 3
k = i++; // k is now 2, i is now 3 (use then add)
k = ++j; // k is now 4, j is now 4 (add then use)
```

Remember that in C, assignment is just another operator, and that an expression involving assignment not only updates a variable but also has a value. The operators ++ and -- are forms of assignment and they too update their operand and have a value. The difference is whether the value is the value before or after the update. If a ++ or -- is used as a prefix operator, the value is the new value, i.e. after the update; used as postfix ++ or -- the value is original value, i.e. before the update. In both prefix and postfix the updated value is the same..

Ternary Operator.

Another common program action is to conditionally update a variable based on the value of some other expression:

```
if( a > b )
    c = a;
else
    c = b;
```

The effect of the *if/else* statement is to assign *c* the maximum of *a* and *b*. C provides a special ternary operator to do this:

```
c = a > b ? a : b;
```


Unlike the `if` statement where the `else` part is optional, the "`else`" part of this operator, i.e. the expression after the `:` must be present. An empty `else` can be simulated by replicating the left hand side:

```
if ( a > b )
    c = a;
```

becomes

```
c = a > b ? a : c;
```

Ternary operators can be nested but the resulting expression can be quite complicated and hard to understand. A nested ternary operator is used to define `putc()` in the file `/usr/include/stdio.h`.

"Sequence Operator".

C uses the comma, `,` to mean evaluate in left to right order. This operator is only legal inside parentheses and brackets. It is mainly used in macro definitions and for loops. The comma used to separate parameters in a function call is NOT a sequence operator. Use with care. The comma can cause problems especially in multi-dimensional array references:

```
float mat[ 4 ][ 4 ];
```

```
mat[ 2 ][ 3 ] = 2.0;      correct
mat[ 2, 3 ] = 2.0        error, as it means row 2 then 3.
```

C does not have shorthand array references like many other languages. What the error line does is evaluate the expression 2, then evaluate the expression 3. The comma operator returns a single index 3 which selects a row of `mat`. This row is not a `float` (or any other numeric type) and so assignment is an error.

Type cast.

The prefix cast operator `( type )` tells the compiler that it should consider its operand to be a value of `type`. No conversion is done

```
int    i;
float  f;
```

```
f = ( float ) i;      interpret (don't convert) i as float
```

`sizeof`.

This prefix operator returns the size in bytes of its operand which is either a variable, expression or `type`.

```
int
sizeof i           i;
sizeof int         is 4
sizeof ( i + 1.0 ) is 8 ( 1.0 is double)
```

Pointer arithmetic.

C supports a limited form of pointer arithmetic. Two pointers of the same type may be added, subtracted and compared. Pointers are totally ordered meaning that if `pa` and `pb` are pointers to `type`, then either `pa < pb`, `pa == pb` or `pa > pb`. Integer constants are also permitted in pointer expressions where they are understood to mean `N * sizeof type`. Thus adding `N` to a pointer advances it `N-1` elements; subtracting `N` backs it up `N-1` elements.

Operator Summary.

Operators	Associativity
<code>() [] -></code>	left to right
<code>! ~ ++ -- + - * & (type) sizeof</code>	right to left
<code>* / %</code>	left to right
<code>+</code>	left to right
<code><< >></code>	left to right
<code>< <= > >=</code>	left to right
<code>== !=</code>	left to right
<code>& ^</code>	left to right
<code> </code>	left to right
<code>&& </code>	left to right
<code>?:</code>	left to right
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	right to left
<code>,</code>	left to right

Statements

Branching.

C provides two statements for conditionally changing the flow of control, an `if` statement providing a two way branch and a `switch` statement offering a multi-way branch.

```
if statement.
if ( expr ) stmt_1
or
if ( expr ) stmt_1
else stmt_2
```

The upper construction will execute `stmt_1` if `expr` is nonzero and skip it otherwise.

The lower construction will execute `stmt_1` if `expr` is nonzero and `stmt_2` otherwise.

Two things to remember. The object of an `if` (or `else`) is *one* statement. To control more than one statement they must be enclosed in `{ }`. And an `else` is paired with the nearest unpaired `if`. Braces must be used to change this pairing.

`switch` statement.

```
switch ( expr ) {
case ce1 : stmt_list
break;
case ce2 :
case ce3 : stmt_list
break;
more cases
```

```
default : stmt_list
break;
}
```

This is the most general form of the `switch` statement. The statement begins with an expression and contains a list of actions enclosed in braces. An action begins with the reserved word `case` followed by a constant integer or character valued expression (`ce1`, etc.) followed by a colon (`:`). This is followed by a statement list specifying the action. The action is terminated with a `break` statement. The individual statement lists are *not* enclosed in braces. The statement list labelled with the reserved word `default` is optional. A `switch` statement must contain at least one case. Each case specifies a *single* value. If a statement list is associated with multiple values it must be preceded by a case for each of those values. All case expressions must be dis-

tinct. The effect of the `break` statement following a statement list is to leave the `switch` statement and resume execution following the closing brace.

`switch` works by evaluating `expr`. This value is compared against the choices specified in the case expressions and if it finds a match executes the selected statement list. If no case expression is matched the statement list labeled with `default` is executed. If the `switch` has no `default` clause, the `switch` is skipped.

Most computers can make a multi-way branch based on the value of an integer in one operation, so a `switch` is generally faster than an equivalent `if/else` tree. Keep in mind that the `switch` expression must be an integer and probably has only a small range (say under 255).

The `switch` statement contains one potential trap. The `break` statement that I have shown ending each case is optional! If a `break` is omitted between two cases and the first is selected, execution will continue on into the second case.

Loops.

C offers three types of loops, a `while` loop where the test is done before entering the loop body, a `do/while` loop where the test is done after the loop body and a `for` loop, a test before loop with initialization and increment code.

`while` statement.

```
while ( expr )
    stmt
```

The object or body of the loop is a single statement. Braces must be used to construct a loop with more than one statement. The `while` statement works by evaluating `expr`. If it is nonzero `stmt` is executed. Control then returns to the `while` and `expr` is reevaluated. Looping continues as long as `expr` is nonzero. If `expr` is originally zero the loop is skipped.

`do` statement.

```
do
    stmt
while ( expr )
```

This loop also has body of a single statement. Braces are again used to construct a loop body with more than one statement. The action of this loop is to execute `stmt`. Then `expr` is evaluated and if it is nonzero control returns to the `do` and the loop is executed again. If it is zero execution resumes following the `while`. Thus this loop is always executed once, no matter what the value of `expr`. `do/while` loops are much less useful than `while` loops.

`for` statement.

```
for ( expr_1; expr_2; expr_3 )
    stmt
```

This is equivalent to this `while` loop:

```
expr_1;
while ( expr_2 ) {
    stmt
```

```
    expr_3;
```

```
    )
```

Any or all of the *expr*'s can be absent. If *expr_2* is missing, it is considered to have the value TRUE so

```
    for ( ; ; ) stmt
```

is an "infinite" loop.

The *for* statement provides a convenient way to collect all the information about a loop in one place. *expr_1* is the initial conditions, *expr_2* the termination condition and *expr_3* is the "increment" or how to compute the value of the next iteration. Here are two examples.

Set an array to 0:

```
for ( i = 0; i < A_SIZE; i++ )
    a[ i ] = 0.0;
```

Find the end of a list:

```
for ( pt = NULL, p = list; p; p = p->next )
    pt = p;
```

Notice that the "increment" is not restricted to numeric values.

break statement.

Used to alter the flow of control in a loop or switch. break exits the immediately enclosing loop or switch. break can only be used inside a loop or switch.

Used to alter the flow of control in a loop. continue skips over the loop body to either the increment code (for statements) or the test (while and do/while statements), continue can only be used inside a loop.

Statement labels and goto statement.

Any C statement can be preceded by a label, i.e. an identifier followed by a colon. Control can be transferred to this statement by executing a goto label statement.

statements

```
goto err;
```

statements

```
err : statements
```

gotos and labels should be used sparingly if at all. The goto statement and its target statement must be in the same function.

Functions.

A function is a named block, i.e. a list of declarations followed by a list of statements enclosed in braces. Either the declarations, the statements or both may be omitted. All C functions are typed.

When a function name is encountered in the evaluation of an expression, it is "called". This results in the execution of the statement list in the function's block to produce a value of the function's type. This value is used to continue the evaluation of the expression.

Functions may contain special variables called parameters that are used to communicate between the function's environment and its interior. Parameters allow the same function to operate on different data. This permits a single piece of code to be used in many places saving memory and reducing program complexity.

return statement.

While all C functions have a type and return a value of this type it is up to the programmer to assign the correct value to the function. This is done with the return statement. Its syntax is

```
return expr;
```

The action is to evaluate *expr* and assign its value to the function, then return control to point immediately following the current call. If *expr* is missing, no value is assigned and the only action is to resume execution at point following the call.

Some functions, called procedures or subroutines in other languages may not return values or those values may not be suitable for use in expressions. C permits the value of a function to be ignored if the function call is the only element of an expression. Naturally these functions use return statements without an expression.

Function definition. C functions may be defined in two ways, the original or "old style" or the ANSI or "new style". They differ in the treatment of parameters. In this course we will only use the old style.

A function definition has four parts. They are in order, an optional type specifier, the function name, parameter information and a block. A type specifier is any valid C type except array types. A pointer to an array is valid. If the type is not given, the function is given the type int. The name is an identifier. Parameter information consists of a parameter list enclosed in parentheses. The parameter list is a comma separated list of identifiers and may be empty. Each parameter in this must be declared between the right parenthesis that follows the parameter list and the left brace that opens the block. These declarations are standard C declarations. The block is a list of declarations followed by a list of statements. A block can be empty. Here is an example.

```
int    incr ( i )
int    i;
{
    int    j;

    j = i + 1;
    return j;
}
```

This is perhaps the simplest useful function. It does contain all of the elements of a definition. It begins with type of the function, int. Since int is the default type it could have been left out. Next is the function name, incr, followed by a parameter list containing one parameter i. Then i is declared to be an int. Next comes the block which contains one additional declaration and

two statements. The action of the function is to add 1 to value of `i`. The new value is computed in the statement

```
j = i + 1;
and the result is assigned to incr in statement
return j;
```

Here is how the function can be used:

```
int      k, l;

k = 2;
l = incr( k );    l is set to 3, k is still 2
k = incr( -2 );   k is now -1
incr( 0 );        the new value 1 is ignored.
```

Function declaration. In C every identifier except those naming `int` valued functions must be declared before it is used. A function definition also declares the function. But there are cases where it may not be desirable or even possible to define a function before it is used. For example, two functions may be mutually recursive. Or more likely they are defined in different files. To avoid the undeclared identifier problem C requires a function declaration. This is an abbreviated form of the function definition consisting of the type the function name followed by parentheses and a semicolon. New or ANSI style declarations are different but will not be described here. Here is an example:

```
double   cos();
char     *fgets();
FILE     *fopen();
```

These declare that `cos` returns a value of type `double`, `fgets` a pointer to a `char` and `fopen` a pointer to a `FILE`, a struct defined in `stdio.h` and used for file I/O.

Parameters. Parameters are special function variables that are used to pass data from the call to the inside of the function. When a function is called the expressions in the parameter list are evaluated and the values are *copied* into the corresponding function parameters where they will be used in ordinary C expression to compute the function's value.

This type of parameter passing is known as "call by value" as the parameter contains a *copy* of the calling value. Any changes to the parameter made in the function only change this copy, not the original data. Does this mean that a C function can not modify data in its parameter list. No, the answer is to pass not the data, but a pointer to it. Then using indirect addressing in the function the original data is accessed and modified. This example will demonstrate this:

```
int      incr( i )
int
{
```

```
    i = i + 1;    i is now increased by 1; no effect
    return( i ); outside of incr()
```

```
    }
    j = 2;
    k = incr( j );    k is 3, j is still 2

Now consider a new version of incr(), this one directly incrementing its argument:

void
incr( pi )
int
{
```

```
    *pi += 1;    Add 1 to the value pointed at by pi
    no return statement
}
```

```
j = 2;
incr( &j );    take the address of j, j is now 3
```

The call to `incr()` contains a copy of the *address* of the variable `j`. `void` is a reserved word describing a type that has no values. This prevents `incr()` from being assigned a value in a return statement or a call of `incr()` from providing a value in an expression. In real C, `incr()` would have omitted the type specifier and the (undefined) `int` value it returned would have been ignored.

Blocks and scope.

A list of statements enclosed in braces is more than just a compound statement. It is actually an unnamed block and like a block in a function the statement in it may be preceded by a list of variable declarations. These variables are local to the block and may be used only inside it. Any data associated with these variables becomes unavailable and undefined once the block is exited. It is possible to get a label in an unnamed block, but the value of any local variables in the block would then be undefined. Such jumps are a very bad idea!

The *scope* of a variable is the region of a program that the variable is visible. C variables have either local or global scope. A global variable is declared outside of all functions including `main()`. It becomes visible immediately following its declaration and remains visible unless the variable name is redeclared in a function or unnamed block. A variable declared in a function or unnamed block has local scope; it is visible only in that function or block unless redeclared in an internal block. Note that in C unnamed blocks can be nested but functions must be global.

Preprocessor, argc, argv, stdio

The C language environment.

C is a small language. It lacks several features provided by most other languages. C has no symbolic constants. It has a string type, but no string operations. C has pointers for dynamic data but memory allocation is not part of the language. It doesn't even have I/O! Instead C relies on other programs to accomplish these tasks. It uses a preprocessor to provide symbolic constants. String operations, memory allocation and I/O are provided through calls to external functions in the standard C library.

The preprocessor.

The preprocessor, called `cpp`, is a separate program that runs ahead of the actual C language compiler. `cpp` is an essential part of the C language *environment* and its rules are different from C. `cpp` provides macro processing, file inclusion and conditional compilation. When a user compiles a C program, each `.c` file is first passed through `cpp` and the processed source is sent to the C compiler.

`cpp` reads a source file line by line. Lines beginning with a '#' in column 1 are `cpp` lines and are not part of the C program. The '#' is followed by an identifier specifying the action to be taken. Space may precede this identifier. Some actions require arguments which follow this identifier. A `cpp` line can be broken into many physical lines by preceding all but the last newline with a backslash ('\'). The `cpp` lines are:

```
# define
# undef
# include
# if
# ifdef
# ifndef
# elif
# else
# endif
# line
# error
# pragma
#
```

(ignored)

Use of any other word following the '#' is an error and will terminate compilation. These words are keywords and are only special if they immediately follow the '#'. They have no significance when used elsewhere. They are *not* C language reserved words (except `if` and `else`), but are simply identifiers. All other lines are ordinary lines and the only action taken by `cpp` is to scan for macro names and expand them.

Macro processing. A macro is named piece of text. Once the macro has been defined, each subsequent use of its name will be replaced by its text. Macros can have arguments allowing each use to reflect local conditions. Macro names have the same syntax as C identifiers. A letter followed by zero or more letters and/or digits. As in C, the underscore ('_') is a letter and upper and lower

case letters are distinct. Generally macro names don't use lower case letters so they can be distinguished from C variables. Here are some simple macros

#define	name	definition	Syntax
#define	NULL	0	
#define	EOF	(-1)	
#define	PI	3.14159265358979323846	
#define	VERSION	\	

"Stgreg 1.2.2 24-July-1992"

These lines act to define the symbolic constants `NULL`, `EOF`, `PI` and `VERSION`. Note that `VERSION` was continued over two lines with the backslash. Once a macro is defined, each subsequent time it name appears, whether in other `cpp` line or ordinary lines, the name is replaced by its definition. This replacement is simple string substitution. The context of the macro name is ignored. If it was in an expression, this may result in an illegal expression or an incorrect evaluation order:

```
#define DECR -1 Dangerous!

k = j-DECR;      Input to cpp
k = j--1;        Output from cpp, error!
```

What has happened is that simple substitution placed the two minus signs together and produced the decrement operator which takes only one argument. If a space had separated the minus and `DECR`, the `cpp` output would have been legal C:

```
#define DECR (-1) Safe

k = j-DECR;      Input to cpp
k = j-(-1);      Output from cpp, correct.
```

A more subtle error occurs when the substitution is legal but changes the meaning of the original expression:

```
#define LEN 80
#define SIZE LEN + 1 Dangerous!

k = 5 * SIZE;      Input to cpp
k = 5 * 80 + 1     Output from cpp
```

Here the user intended for `k` to be `5 * 81` or `405` but since `SIZE` was an expression, the multiply was done first and then the add resulting in `k` being assigned `401`. The take home lesson is that if a macro definition is an expression, surround it with parentheses which will insure that the substitution will produce both legal C and preserve your expression order.

Macros can have arguments. The syntax of the definition is changed to include a parameter list following the macro name. No space is permitted between the name and the left parenthesis and the first argument. The parameter list is a comma separated list of identifiers. These identifiers are used in the macro definition. A space must follow the closing parentheses of the parameter list:


```
#define ABS(a) ((a)>=0?(a):- (a))
#define MAX(a,b) ((a)>= (b)? (a) : (b))
```

After these definitions, ABS produces the absolute value of its argument and MAX the larger of its argument. Note that both definitions are surrounded by parentheses to insure that after replacement the expression in the macro definition is kept together. Also the macro parameters are always surrounded by parentheses in the definition. This is required because when macro is expanded, the strings in the call are substituted in the definition and the parentheses insure legal and correct substitution:

```
k = ABS(1& j);          input to cpp
k = ((1& j)>=0)? (1& j) : - (1& j); output from cpp
```

Had the parenthesis been omitted the resulting expression would have been legal but very different

```
k = 1& j>=0?1& j:-1& j;          (no parens)
```

One final caution about macros. Since the arguments are simple substitution, an expression with a "side-effect" can cause problems:

```
k = ABS(1++);          input to cpp
k = ((1++)>=0)? (1++) : - (1++); output from cpp
```

Here 1 is incremented twice, once in the test, then again in the final assignment.

Like most tools, macros are useful when used correctly. Simple definitions without arguments are invaluable. They increase both the reliability and readability of a program. A test against EOF is much clearer than the equivalent test against -1. If the value of EOF is changed, only the definition need to be changed, instead of all its uses. Array sizes should always be declared with symbolic constants:

```
#define LINELEN 80
char line[ LINELEN + 1 ];
```

These two lines insure that line is always one larger than LINELEN.

File inclusion. When cpp encounters a line of the form:

```
#include <file>
```

or

```
#include "file"
```

it replaces the line with the contents of *file*. In the first case, file must be in the directory /usr/include. In the second case it is interpreted as a path name and if not rooted (i.e. beginning with a '/') in the current directory. File includes can be nested.

Conditional compilation. This can be used to prepare different versions of a program based on the value of a cpp symbol:

```
#ifdef XGL
include these lines if symbol XGL is defined.
#else
```

include these lines if symbol XGL is not defined.

```
#endif
```

The `#else` part is optional. A more complete description of conditional compilation can be found in the text.

Command arguments.

Every C program begins execution with function `main()`. This function has two arguments `argc` and `argv`. `argc` is an integer and `argv` is an array of strings. When a shell executes a program, it breaks the command line up into words separated by spaces. A pointer to the string containing each word is placed in `argv[]`. The number of words on the command line is placed in `argc`. `argc` is at least one and `argv[0]` is the name of the file containing the current program. Arguments are either options which set the initial value(s) of program variables or the names of the files the program will use. A Unix convention is that options begin with a minus '-'. This program prints its arguments:

```
main( argc, argv )
int  argc;
char *argv[];
{
    int  ac;

    for( ac = 0; ac < argc; ac++ )
        printf( "argv[%2d] = %s\n",
                ac, argv[ ac ] );
}
```

If this program is compiled and left in `a.out` then running it produces this output:

```
macke@euler% a.out
argv[ 0] = a.out
macke@euler% a.out one two three
argv[ 0] = a.out
argv[ 1] = one
argv[ 2] = two
argv[ 3] = three
```

Standard I/O.

A file is a named sequence of zero or more bytes. The data in the file usually resides in the computer system's permanent storage so that its existence continues after the program that created it terminates. Files are not part of C, but a service provided by the underlying operating system, which in this course is Unix. C appears to have file operations by providing routines in the standard C library that can create and destroy, connect to and disconnect from and read from and write to files.

Every C process or running program begins execution connected to three files. They are called `stdin`, `stdout` and `stderr` and are used for input, output and error reporting. For C programs that are run from a terminal, `stdin` is the keyboard and `stdout` and `stderr` are the screen.

The standard library provides two methods of accessing Unix files: file pointers and file descriptors. A file descriptor is a small integer that is the index into a special data structure in the operating system. File descriptor I/O is "low level" and much less convenient than file pointer I/O. Since almost any I/O operation can be done using file pointers, only they will be discussed.

Creating and opening (connecting to) a file.

```
FILE *fopen( path, mode )
char path[];
char mode[];
```

path is the name of the file to open. mode is the type of access. Mode is usually "r" for read access, "w" for write access or "a" for append access. Other modes are possible and are described in the text. A file opened for reading must exist prior to the call to fopen(). A file opened for writing or appending is created with zero length if it does not exist before the call. If the file exists and is opened for write it truncated (current contents lost). Opening an existing file for append positions it so that new data is added to the end. fopen() returns a NULL pointer if an error occurred during the call.

Closing (disconnecting from) a file.

```
int fclose( fp )
FILE *fp;
```

The file pointed to by fp is closed. All files are closed automatically on program exit. fclose() returns 0 if successful and EOF on error. Generally not checked.

Reading from a file.

```
getc( fp ) a macro, not a function
```

Return the next character from the file pointed to by fp. Return EOF if the file is at its end.

```
int ungetc( c, fp )
int c;
FILE *fp;
```

Push the character c back on to the file pointed at by fp. This routine is used to get one character of lookahead on a file. ungetc() returns c. Pushing EOF has no effect. Most files including terminals support ungetc().

```
char *fgets( buf, size, fp )
char buf[];
int size;
FILE *fp;
```

Read at most size - 1 characters or until a newline is read from or end of file is encountered on fp and place them in buf. Append a zero byte to buf and return. fgets() returns buf if some data was read and NULL on error or immediate EOF.

Writing to a file

```
putc( c, fp ) a macro, not a function
```

Write the character c to the file pointed to by fp.

```
int fputs( str, fp )
char str[];
FILE *fp;
```

Write the contents of str to the file pointed to by fp. The zero byte that terminates str is not written. fputs() returns EOF on error and 0 on success.

Destroying a file.

```
int unlink( path )
char path[];
```

The file with name path is unlinked from its containing directory. If no other links to path exist, it is deleted. unlink() returns 0 on success and -1 on failure.

File programming example.

To tie all this together here is how to modify the word count program from Class 2 to read from a named file. The file name will be an optional command line argument. If present, the program operates on the data in this file; if absent the program uses standard input. The program informs the user if the named file could not be opened.

```

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

/* Define some symbolic constants:*/
#define USAGE "usage: wc [ file ]\n"
#define SUCCESS 0
#define ERROR (-1)

main( argc, argv )
int  argc;
char *argv[];
{
    FILE *ifp;
    int count, wcount, lcount;

    /* file processing:*/
    if( argc == 1 )
        ifp = stdin;
    else if( argc > 2 ) {
        fprintf( stderr, USAGE );
        exit( ERROR );
    }
    else if( ( ifp = fopen( argv[ 1 ], "r" ) )
              == NULL ) {
        fprintf( stderr,
                "can't open input file %s\n",
                argv[ 1 ] );
        exit( ERROR );
    }

    /* do the work:*/
    wordcount( ifp, &lcount, &wcount, &count );

    printf( "%4d %4d %4d\n", lcount, wcount, count );

    /* files are valuable, don't close input*/
    if( ifp != stdin )
        fclose( ifp );

    exit( SUCCESS );
}

wordcount( fp, lc, wc, cc )
FILE *fp;
int *lc;
int *wc;
int *cc;
{
    int c;
    int inword;

    *lc = *wc = *cc = 0;
    inword = 0;
    while( ( c = getc( fp ) ) != EOF ) {
        if( c == '\n' ) {
            ( *lc )++;
            inword = 0;
        }
        else if( isspace( c ) )
            inword = 0;
        else if( !inword ) {
            ( *wc )++;
            inword = 1;
        }
        ( *cc )++;
    }
}

```


Formatted I/O, Strings

Formatted I/O

Formatted I/O is the process that converts between the computer's internal data representation and the ordinary character representation used in text files. C provides a powerful formatted I/O facility with the six routines, `printf()`, `fprintf()` and `sprintf()` for output and `scanf()`, `fscanf()` and `sscanf()` for input. The two functions `printf()` and `scanf()` "write" to or "read" from a string instead of a file. The two routines `printf()` and `scanf()` do not have a target/source argument as they always write to stdout and read from stdin. All other actions, including return values are identical to `printf()` and `fscanf()`.

All formatted I/O routines have similar parameter list. The first two arguments are the source/destination (if required) and the format string. Subsequent arguments are either expression(s) whose values are to be converted (`printf()`, `fprintf()`, `sprintf()`), or addresses of variables that will receive the converted data (`scanf()`, `fscanf()`, `sscanf()`).

Similar format strings are used for both input and output. The format string is a mixture of ordinary characters and format descriptors. A format descriptor begins with a '%' and is followed by optional flags and ends with a type specifier. The type specifier is either a letter, or for reading strings a set of characters permitted in the string enclosed in square brackets ('[]'). This course covers only the most common format descriptors, a complete description can be found in Appendix B of the text.

Formatted output.

```
int      fprintf( fp, format, ... )
FILE     *fp;
char     format[];
char     *sprintf( str, format, ... )
char     str[];
char     format[];
```

`fprintf()` uses `format` to convert the values of the expressions that follow it (if any) to their character representations and writes those characters to the file pointed at by `fp`.

`sprintf()` works exactly the same except that the convert characters are written to `str`, which must be large enough to hold them plus the final zero byte.

`fprintf()` returns the number of characters written or a negative number if an error occurred during processing. `sprintf()` returns a system dependent value.

The action of both routines is to examine the characters in `format` from left to right. If the character is not a '%', it is written to the destination. If it is a '%', then the it begins a format descriptor. This descriptor will be used to convert the corresponding expression in the parameter list into characters which are then written to the destination.

An output format descriptor begins with a '%'. This can be followed by several optional fields in any order. They are the flags, the width, a period, the precision and a length modifier. The format descriptor ends with a conversion character, a letter specifying the type of the data to be converted.

C Programming Course

Two useful flags are:

-

0

left justify the converted characters
pad numbers with leading zeros.

The width is either an integer or a star (*). A star indicates that the width is the value of the current expression in the parameter list. A period is used to separate the width and the precision. The precision is also either an integer or a star. Again a star means that the precision is the value of the current parameter. The precision is used to control the actually number of characters printed for a string conversion or the number of digits following the decimal point in for a double. It is ignored in other conversions. The length modifier is a letter used to indicate that an int is really short or long (h,l) or a double is really a long double (L).

The most useful conversion characters are:

d

convert int signed decimal.

c

convert int to a character.

f

convert double to [-]mmmm.ddd where the number of places after the decimal point is specified by the precision.

e

convert double to [-]m.ddddeLnn where the number of places after the decimal pointer is specified by the precision.

s

convert a string. Characters are written until a zero byte is encountered.

%

write a '%'.
encountered.

Integer examples:

```
fprintf( fp, "%d", 123 );      prints      123
fprintf( fp, "%2d", 123 );    prints      123
fprintf( fp, "%4d", 123 );    prints      123
fprintf( fp, "%04d", 123 );   prints     0123
fprintf( fp, "%-4d", 123 );   prints     123
```

In the last example, the 3 is followed by a space. The entire integer is always printed, no matter what the specified field width. If no field width is specified, the width used is exactly the width of the integer.

Floating point examples

```
fprintf( fp, "%f", 1.23 );    prints      1.230000
fprintf( fp, "%5.2f", 1.23 ); prints      1.23
fprintf( fp, "%5.1f", 1.23 ); prints      1.2
fprintf( fp, "%-5.1f", 1.23 ); prints      1.2
```

The 1.2 is followed by two spaces.

String Examples:

```
fprintf( fp, "%s", "abcd" );  prints      abcd
fprintf( fp, "%4.3s", "abcd" ); prints      abc
fprintf( fp, "%5s", "abcd" ); prints      abcd
```

```
fprintf( fp, "%-5s", "abcd" ); prints abcd
fprintf( fp, "%-5.3s", "abcd" ); prints abc
```

The last two examples are followed by one and two spaces. A precision of 0 prints only width blanks. If the width is missing, it is the larger of the width of the string or the precision.

Runtime format:

```
fprintf( fp, "%*d", wid, i );
```

uses the value in *wid* as the field width to use for printing *i*. Similarly, the next format uses the values of *wid*, and *prec* to determine the width and precision to be used in printing the string:

```
fprintf( fp, "%*.s", wid, prec, "abcdef" );
```

The same conversion could be performed leaving the results in a string by using `fprintf( str, ... )`; where *str* is a character array large enough to hold the results.

Formatted input.

```
int      fscanf( fp, format, ... )
FILE     *fp;
char     format[];

int      sscanf( str, format, ... )
char     str[];
char     format[];
```

`fscanf()` uses format to interpret the next several characters read from the file pointed to by *fp*. The values of these fields are assigned to corresponding variables following format in the parameter list. The parameters following format are the *addresses* of the variable to be assigned. `sscanf()` works exactly the same except the source of the characters to convert is the string *str*.

Both functions return the number of fields read or EOF if the source was exhausted before the format was satisfied or an error occurred.

The interpretation of the format string in `fscanf()` is somewhat different from that in `fprintf()`. In an `fscanf()` format, blanks and tabs are ignored. Nonblank ordinary (not %) characters must match the next non blank character in the source. Format descriptors match the next *field* in the input where a field is defined as a sequence of non white space characters. This means that `fscanf()` will skip lines to find enough fields to satisfy the format string.

Input format descriptors are simpler than output format descriptors. Input format descriptors begin with % and can be followed by an optional * indicating "assignment suppression", an optional length or maximum size of the field, a optional length modifier (l,h,L) and end with a conversion character. Assignment suppression reads the designated field but does not assign the value to the corresponding variable.

Most of the output conversion characters can be used in input format. The big difference is in reading a string. In addition to the %s, a set of characters can be given. This is enclosed in brackets ('[', ']').

```
%d      read an integer
%f      read a float
%c      read a single character - don't skip spaces!
%s      read a string
%[...]  read a string composed of a run of the characters between the
brackets. %[...] includes ] in this set.
%[^...] read a string composed of a run of the characters not between
the brackets. %[^...] includes ] in this set.
```

String examples. Let *abcde fg* be the next 8 character in the file *fp. Then

```
fscanf( fp, "%s", str ); sets str to abcde
fscanf( fp, "%3s", str ); sets str to abc
fscanf( fp, "%[ab]", str ); sets str to ab
fscanf( fp, "%[cd]", str ); doesn't set str
fscanf( fp, "%*s", str ); doesn't set str
```

The same conversions could have been done using input from a string by using `sscanf()` where *fp* is replaced by the name of the source string.

Strings.

C defines the special data type, string as an array of *char* with the understanding that the last element in the array has value 0. However, the only string operation directly provided by the language is the creation of string literals. Yet C is a good language for string processing. How is the done?

The solution is to use C's separate compilation facility to construct a library of C functions that operate on strings. These functions are ordinary C user functions. Then any other C program can use that functionality by referencing these string functions in expressions. After the program is compiled it is linked with the appropriate library and the language appears to support strings. The standard library includes string functions that determine the length of a string, copy a string, compare strings and search strings. The most useful of these functions are described below and a complete description of the string library can be found in the text.

Length of a string

```
int      strlen( s )
char     s[];
```

Return the number of non-null characters in *s*.

Copying strings.

```
char     *strcpy( s, t )
char     s[];
char     t[];
```


Copy the string in *t* to *s* which must be large enough to hold. Return *s*.

```
char *strncpy( s, t, n )
char s[];
char t[];
int n;
```

Copy at most *n* characters of *t* to *s* which must be large enough to hold them. Copying stops when either *n* characters have been copied or a 0 is copied, whichever is first. If exactly *n* non-null characters are transferred, *s* may not be properly terminated, i.e. end with a 0. Return *s*.

```
char *strcat( s, t )
char s[];
char t[];
```

Copy *t* to the end of *s* which must be large enough to hold the combined string. Return *s*.

```
char *strncat( s, t, n )
char s[];
char t[];
int n;
```

Copy at most *n* characters of *t* to the end of *s* which must be large enough to hold them. Copying stops when either *n* characters have been copied or a 0 is copied, whichever is first. If exactly *n* non-null characters are transferred, *s* may not be properly terminated, i.e. end with a 0. Return *s*.

Comparing strings.

```
int strcmp( s, t )
char s[];
char t[];
```

Compare *s* and *t*. Return a negative, 0 or positive integer if *s* is less than, equal to or greater than *t*.

```
int strncmp( s, t, n )
char s[];
char t[];
int n;
```

Compare at most *n* characters of *s* and *t*. Return a negative, 0 or positive integer if *s* is less than, equal to or greater than *t*. Comparison stops when either *n* characters have been tested or a 0 value is found in *s* or *t*.

Searching for a character.

```
char *strchr( s, c )
char s[];
char c;
```

Return a pointer to the first occurrence of the character *c* in *s*. Return NULL if *c* is not in *s*.

```
char *strchr( s, c )
char s[];
char c;
```

Return a pointer to the last occurrence of the character *c* in *s*. Return NULL if *c* is not in *s*.
Programming Example, developing `strcpy()`

The functions that provide string operation are ordinary C functions. Here we will develop `strcpy()` beginning with the most direct version and after three revisions a highly efficient non obvious version that shows why C is so useful.

Copy a string, Version 1 - Direct

```
char *strcpy( s, t )
char s[];
char t[];
{
    int i, len;

    len = strlen( t ) + 1;
    for( i = 0; i < len; i++ )
        s[ i ] = t[ i ];
    return( s );
}
```

The direct approach. Determine the length of *t*, and copy it element by element to *s*. `strlen()` returns the number of non-null characters in a string, but `strcpy()` must copy the entire string including the last null character. The return statement insures that `strcpy()` points to *s*.

Copy a string, Version 2 - Implicit source length. In C, unlike some other languages, the assignment operator, '=', has a value, the value it has just assigned to its left operand. Now if that value is zero, then C considers it false and if non-zero true. These two capabilities let us dispense with the call to `strlen()` as the copy loop itself can find the end of string *t*:

```
s[ i ] = t[ i ];    if t[ i ] was zero, s[ i ] is zero,
                  but the expression has value false.
```

```
char *strcpy( s, t )
char s[];
char t[];
{
    int i;

    for( i = 0; s[ i ] = t[ i ]; i++ )
        return( s );
}
```


As long as a non-zero character is copied from `t` to `s`, the middle expression of the `for` loop is true; copying a zero, terminates `s` and the loop. Note that the loop has no body – all the work is done in test.

Copy a string, Version 3 – Pointers vs. Indices. In C pointers are used much more often than the equivalent array index expression. This is in part because originally pointers allow less sophisticated compilers to produce fairly good code. But pointers are also used because they offer a very compact notation, especially in the accessing of highly complex data.

If `s` is an array of `char` and `i` is an `int`, then

```
s[ i ] is the same as *( s + i )
```

Both expressions take the value of `s` which is the address of its first element and add `i` to it, then use this sum as the address of the desired element. Now introduce a new variable `sp`, declared as

```
char *sp;
```

and as C supports pointer arithmetic assign `s + i` to it:

```
sp = s + i;
```

then

```
s[ i ] is *( s + i ) is *sp
```

Instead of incrementing the index, `i`, and computing the address of the next element, we just increment `sp` and get this address directly. So:

```
char *strcpy( s, t )
char s[];
char t[];
{
```

```
    char *sp, *tp;
```

```
    for( sp = s, tp = t; *sp++ = *tp++; )
```

```
        return( s );
```

```
}
```

Here we assign `sp` and `tp` to point to the first elements of `s` and `t`. The loop condition assigns the current element of `t` to `s`, advances both `sp` and `tp` and if the newly assigned value was zero, terminates the loop!

The loop test expression while cryptic is really straight forward to work out. Begin with `*tp++`. This says take the value of `tp`, get the data there (`*tp`) and then increment `tp` to point at the value for the next time around the loop. The left side `*sp++` takes the value of `sp`, stores the just fetched value, there and then advances `sp` so it too is ready for the next trip around the loop. The entire expression is either zero or not and so can be used to correctly terminate the `for` loop.

Copy a string, Version 4 – C as its really written. The final optimization takes advantage of the fact that all C parameters are called by value which means they are copies of the original data. Any changes made by the function to the parameters can not alter the original data. The two argu-

ments of `strcpy()` are copies of the addresses of `s` and `t`. Thus these parameters can be used to do the assignments instead of required `sp` and `tp`. `s0` is required to remember the original value of `s`.

```
char *strcpy( s, t )
char s[];
char t[];
{
    char *s0;

    for( s0 = s; *s++ = *t++; )
        return( s0 );
}
```

Dynamic Variables, lists.

Dynamic data structures.

Up to this point all the code discussed has used statically allocated data. This meant that the sizes of the variables were fixed when the program was compiled. However, there are many programs that can not know the size or even the shape of their data until they are running. Varying sizes, but not shapes can be accommodated by making the program variables large enough to hold the maximum input size, but large nearly empty data structures waste computer resources resulting in longer running times or fewer concurrent processes.

The solution is to use "dynamic" data. Here the allocation of memory to a program's variables is deferred until the space is actually required. Since only the memory required is assigned to the program, inefficiencies associated with large nearly empty data structures are avoided. And with clever use of structures and pointers data structures resembling arbitrary graphs can be built.

C provides support for dynamic data in the form of pointers and structures. Acquisition of memory however is a service provided by the operating system and not part of the C language. Access to this service is provided through named functions in the standard library that get and free memory. Three of the most useful memory functions are `malloc()`, `calloc()` and `free()`. Others are described in Appendix B of the text.

Get a block of memory.

```
char *malloc( size )
      unsigned size;
```

Return a pointer to a block of at least size bytes. Return NULL if the request could not be satisfied.

```
char *calloc( nelem, elsize )
      unsigned nelem;
      unsigned elsize;
```

Return a pointer to a block of at least `nelem * elsize` bytes. Return NULL if the request could not be satisfied. Each byte of the newly allocated block is set to 0.

Free up a block of memory.

```
int free( ptr )
      char *ptr;
```

Return the previously allocated block of memory pointed to by `ptr` to the operating system. Return 1 on success and 0 on failure.

Declaring Dynamic Variables.

The simplest dynamic variable consists of a pointer to an elementary type. After the variable has been allocated with a call to `malloc()`, it behaves like an array, differing from the arrays we've seen before only in the fact that the size of the array was determined when the variable was allocated. A much more interesting case occurs when the type of dynamic variable is a pointer to a special kind of structure, one that contains not only data but pointer(s) to other structures of the same or different types. This type of data is used to make lists, trees and arbitrary graphs.

Declare

```
char *cp;
int *ip;
float *fp;
```

declare three pointer variables which after allocation, would be used just like ordinary arrays:

```
cp[ 1 ] = 'a';
ip[ 1 ] = getw( fp );
x = fp[ 37 ];
```

Declaring list variables is a little more complicated. The problem is that a structure used in a list (or tree, etc.) must contain pointers. But not just any pointers, they have to be pointers to the very type still being declared! The solution is to give such a structure a "tag" or name, so that the incomplete structure can be referenced during its declaration:

```
struct tag
{
    struct tag    tag
      more fields
    *ptr;
};
```

`tag` is an identifier. The field `ptr` is declared to be a pointer to a structure of type `tag`, allowing variables of this type to point to each other. Only tagged structures can reference themselves.

Adding types with typedef.

C allows users to define and name a data type. This type name can then be used to declare variables and functions of that type. This ability is especially important for structure data, as without it, each structure variable (or function) would have to be preceded by a copy of the structure definition. Not only would this add keystrokes, but it invites errors. The definition must be copied correctly each time and any changes to the definition must be made through out the program.

C uses the reserved word `typedef` to introduce a type definition. Any C type can be named although the syntax of some type definitions can be obscure. Here, we will only discuss type definitions of structure which are both obvious and essential. A full description of `typedef` can be found in the text.

The syntax of a structure `typedef` is simple:

```
typedef struct tag
      declare structure members
} name;
```

The structure definition is preceded by `typedef` and followed by the name of the new type. Following the type definition, `name` acts like any simple C type and is used to declare variables, functions or allocate space:

```
typedef struct item
      struct item *next;
      char name[ 10 ];
      char *data;
} ITEM;
```

defines a new type, ITEM, which can be used in declarations:

```
ITEM    temp;
ITEM    *list;
ITEM    *make_item();
```

Here temp is an ITEM, i.e. it is a variable with the three fields, next, name, and data. list is pointer to an ITEM, and make_item() is a function that returns a pointer to an ITEM. Without the typedef each use of the word ITEM would have had to be replaced with a complete copy of the structure definition.

Allocating and freeing dynamic variables.

Assigning space to dynamic variables is done via calls to either malloc() or calloc(). Here is where we finally use the two unusual operators, type-cast and sizeof that were introduced in Class 3. C is fairly strongly typed as far as pointers go, in that if p1 is a pointer to one type and p2 is a pointer to a different type, then these two variables can't be used in the same expression. So, how can malloc() which returns a pointer to char, be used to assign all pointers? The answer is the type-cast, or to tell the compiler, that yes malloc() is a pointer to char, but here it should be considered as a pointer to some other type. The other difficulty is how large a block to allocate? Simple, use sizeof applied to a type and the compiler will figure it out! An example will help:

```
list = ( ITEM * ) malloc( sizeof ITEM );
```

This statement calls malloc() to request a block of memory. The size of the block is the value of the expression sizeof ITEM, which is the size of one variable of type ITEM. The (ITEM *) preceding malloc() informs the compiler that the type of this expression is now a pointer to an ITEM which matches the that of the left hand side, list, so the assignment is legal. Three things to remember: the argument to malloc() is the size of the structure, sizeof ITEM * would have allocated a block the size of a pointer to an ITEM; the function malloc() must be cast to have the same value as the variable to which it assigned (here char * to ITEM *) and malloc() does not assign any part of the new variable - the programmer must explicitly do this.

Programming Example - Lists.

In this section, we develop a set of functions that operate on a "linked" list. Each item in the list contains a field that points to the next item in the list. The last item has no successor so its link has value NULL. The first element of the list is called the head. The items in the list are structure variables which generally contain the "name" or key of the item plus other data. There are both ordered and unordered lists. A list is ordered if for any two elements i, j, the keys of i and j are consistently ordered. A simple linked list is "one-way". It is easy to find the successor of any item but not possible to find its predecessor. A "doubly-linked" list would have pointers to both the next and previous items. Our list will be a singly linked ascending order list using an integer as the key.

Declarations.

```
typedef      struct      item      {
              struct      item      *next;
              int          key;
              char         *data;
            } ITEM;
```

ITEM

```
*malloc();
```

An ITEM has contains three things, the pointer to the next ITEM, next, the name or key of the ITEM, an integer, key and a catch all pointer to other data data. The memory allocation function malloc() was declared to return a pointer to an ITEM so that casting the return value will not be necessary.

Create an item.

```
ITEM      *make_item( key, data )
int      key;
char     *data;
{
```

```
    ITEM      *p;
```

```
    if ( p = malloc( sizeof ITEM ) != NULL ) {
        p->next = NULL;
        p->key = key;
        p->data = data;
    }
    return( p );
}
```

This function is straight forward. The new item is only assigned if the allocation was successful. Notice how compact the code is because C allows the allocation to be inside the test.

Insert an item into list.

```
ITEM      *insert( list, ip )
ITEM      *list;
ITEM      *ip;
{
    ITEM      *p, *pl;

    for( pl = NULL, p = list;
        p && p->key < ip->key;
        pl = p, p = p->next )
        ;

    ip->next = p;
```



```

if( p1 )
    p1->next = ip;
else
    list = ip;

return( list );
}

```

All the work is done in the expression of the for loop. In general, the new ITEM will be inserted between two ITEMS. The loop terminates when list is exhausted or it reaches the first ITEM that should follow the new ITEM. Since only successors are available from list, p1 is used to remember the previous list element. The && or "and-if" operator is put to good use in the loop test. First it tests p against NULL, and only if p is a valid pointer is the second test carried out. When the loop exits, p points to the element that must follow ip. This is always true, even if list is originally empty, or p becomes the first or last element. If p1 is NULL on loop exit, then ip is to become the first element of list, otherwise insert it after p1. The last statement returns list so that any changes to the head of list will be sent back to the calling function. A call to update list would have list as an argument and a left hand side:

```
list = insert( list, ip );
```

Find an ITEM with key having a certain value in list.

```

ITEM *find_item( list, key )
ITEM *list;
int key;
{
    ITEM *p;

    for( p = list; p; p = p->next )
        if( p->key == key )
            return( p );

    return( NULL );
}

```

Return a pointer to the first ITEM in list that matches key. If no such ITEM can be found, return NULL. If list is originally empty, the for loop is never entered and NULL is returned. Delete an ITEM from list.

```

ITEM *delete( list, ip )
ITEM *list;
ITEM *ip;
{
    ITEM *p, *pl;

    for( pl = NULL, p = list; p; p = p->next ) {
        if( p == ip ) {

```

```

        if( pl )
            pl->next = ip->next;
        else
            list = ip->next;
        free( ip );
        return( list );
    }
    pl = p;
}

return( list );
}

list = delete( list, ip );

```

Search list for the ITEM pointed to by ip and remove it if found. If this element is not on list, return it unchanged. Again, ip will generally be between two ITEMS and p1 is used to remember element before ip. If p1 is NULL, then ip is the first element of list. If list contained a single ITEM, removing it sets it to NULL. delete() is used like insert():

Trees and Sorting

Trees.

A tree is special sort of linked list where 1) every node or element of the list can have more than one successor and 2) every node except one has exactly one predecessor. This special node which has no predecessors is called the *root* and is used to access the tree's contents. Trees have interesting properties. Because each node can have multiple successors, trees grow both longer (deeper) and wider as nodes are added to them. Trees don't have cycles meaning there is no path from any node that leads back to that node. A node with no successors is called a *leaf*; all other nodes are *interior* nodes. Most trees have nodes with a fixed number of successors. Nodes with two successors, called binary trees are the most common. A tree in which every node except one has either the maximum number of successors or zero successors is called a balanced tree. A balanced m -way tree of N elements has height $\lceil \log_m N \rceil$ implying that any element is no more than $\lceil \log_m N \rceil$ links from the root.

Examples of trees. Trees are used throughout computer programming. Expressions have a tree structure where variables and constants are the leaves and the operators are the interior nodes. Searching often involves trees. At each stage of the search there are new candidates and as each candidate is tested it will in turn generate new candidates producing a tree like structure.

Compilers often use trees to hold both a program's identifiers (the symbol table) and an intermediate form of the program called a parse tree. A symbol table is usually an ordered binary tree where the symbol's name is used to order the tree. It is ordered so that for any node all the symbol's in that node's left subtree are less than that node's symbol; all the symbols on the right subtree are greater. Trees are useful here because the number of symbols in a program varies greatly and if the tree is relatively balanced a symbol can be retrieved in $\log_2 N$ steps where N is the current number of symbols.

Traversal order. Many tree algorithms involve visiting some or all of the nodes in a certain order. This is called traversing the tree. There are two standard tree traversals defined for trees with any number of branches. A *preorder* traversal visits the root then the subtrees in some order. A *postorder* traversal visits the subtrees then the root. A special kind of traversal called *inorder* is defined for binary trees. This visits the left subtree, then the root then the right subtree. Searching usually uses preorder traversal: test the current candidate which is the current root; use it to generate new candidates or subtrees and then visit them in some order. Expression evaluation on the other hand uses postorder traversal: evaluate the subtrees which produce the values required for the current operator or root. The inorder traversal gets its name because an inorder traversal of a binary ordered tree will sort it.

Recursion.

Many things can be defined recursively or in terms of themselves. Trees are a natural example - a binary tree is either empty or node with two subtrees. In computer programming, recursion means that a function can call itself, either directly as in the tree processing functions in the next section or indirectly through intermediate functions. All C functions including `main()` can be used recursively.

Recursive functions make use of special area in memory called a *stack*. Data can be entered into and taken out of a stack only in a certain order - last in/first out. When data is added to a stack it is *pushed* onto the stack. The stack is *popped* to remove data. A stack behaves like an array that can only be accessed at one end. This is exactly the kind of structure required for recursion. Each time a function is called the values of its parameters and local variables are pushed onto the stack. Then when the function is completed, this data is popped, and the function returns, leaving the caller's data accessible on the top of the stack.

Variables declared inside a function body i.e. between the open and closing braces of the function definition are local variables. They can only be used inside the function. Each time a function calls itself the values of its local variables are pushed onto the stack the new call begins with a new set of local variables. Thus the local data at each level in a recursive invocation is isolated from all other levels in the call. Keep in mind that this applies only to local variables. If any local variables are pointers the data they point to is *not* pushed onto the stack and can be accessed and modified throughout the call.

Recursion vs. iteration. Many mathematical relationships are expressed as recurrences, where the i th value is some function of the previous values. Here is a second order recurrence. Explicit val-

$$R_i = f(R_{i-1}, R_{i-2})$$

ues would be given for the R_1 and R_2 . Is such a function a candidate for recursive evaluation? In general no. Most arithmetic recurrences can be easily evaluated in loops, where local variable are used to "remember" the required previous values. Recursion should be used where a significant amount of local context is required for each level. The position in a search tree deserves recursion, the index of loop does not.

Tree programming example.

This section develops a set of routines that can be used to create, use and maintain a binary ordered tree. This type of tree might be used by a C compiler to maintain information about the identifiers that have been declared. In such a case the generic field `data` would contain symbol information or might be replaced by several more specific fields, but the basic tree processing would be unchanged.

Declarations.

```
#define      NAMESIZE      10
typedef struct node {
    char      name[ NAMESIZE ];
    struct node *left;
    struct node *right;
    char      *data;
} NODE;
```

```

        else
            return( find( root->left, name ) );
        }
        return( find( root->right, name ) );
    }

```

Return NULL if no node has the specified name.

Delete a node from a tree. Deleting a node from a tree is harder than from a simple list. In general the node to be deleted will have two successors but since no node can have more than one predecessor, this requires that its two subtrees be combined into one. In an ordered binary tree, there are two possibilities. The left subtree can become the new left most subtree of the right subtree or the right subtree can become the right most subtree of the left subtree. After the two subtrees have been combined a pointer to the new tree is returned and the specified node is deleted.

The complexity of deletion suggests that trees may not be an appropriate data structure if nodes are to be both inserted and deleted. A further problem with deletion is that the tree can become very unbalanced and access time will no longer be logarithmic. If only a few deletions are required then marking a node as invalid but leaving it in the tree can be worthwhile. Large fluctuations in the size and shape of a tree may require explicit rebalancing to maintain acceptable performance. By keeping information about the height differences of each subtree at each node, the basic insert/delete algorithms can be modified to update and rebalance the tree in $\lg_2 N$ time. These trees are known as AVL trees after their inventors, Abelson, Velski' and Landis.

```

    NODE *delete( root, name )
    NODE *root;
    char name[];
    {

```

```

        int cv;
        NODE *lp;

```

```

        if( !root )
            return( NULL );
        else if( (cv=strcmp( root->name, name ))==0 ) {
            if( !root->left )
                return( root->right );
            else if( !root->right )
                return( root->left );
            else {
                for( lp = root->right;
                    lp->left;
                    lp = lp->left )
                {
                    lp->left = root->left;
                    return( root->right );
                }
            }
        }
        else if( cv > 0 )

```

Each node has a left and right subtree. The value of character string in NAME will be used to order the tree so that every NAME in the left subtree is less than the NAME in the root and every NAME in the right subtree is greater. malloc() is declared to return NODE * to avoid type casting in the examples.

Insert a NODE into a tree.

```

    NODE *insert( root, name, data )
    NODE *root;
    char name[];
    char *data;
    {
        if( !root ) {
            root = malloc( sizeof NODE );
            strcpy( root->name, name );
            root->left = NULL;
            root->right = NULL;
            root->data = data;
        }
        else if( ( cv=strcmp( root->name, name ))>0 )
            root->left = insert( root->left, name );
        else if( cv < 0 )
            root->right = insert( root->right, name );
        return( root );
    }

```

Duplicates are *not* inserted into this tree. This is not the only strategy but in order for this function to work correctly a *value* can only appear at one place in the tree. Possibilities include keeping track of a count of duplicates or storing them in a sublist from the node with that value.

Find the NODE with name.

```

    NODE *find( root, name )
    NODE *root;
    char name[];
    {
        int cv;

        if( !root )
            return( NULL );
        else if( ( cv=strcmp( root->name, name ))==0 )
            return( root );
        else if( cv > 0 )

```


Trees and Sorting

```

        root->left = delete( root->left, name );
    else
        root->right = delete( root->right, name );
    return( root );
}

```

Sorting.

No computer programming course would be complete without at least mentioning sorting. In fact sorting was implicitly mentioned because an inorder traversal of an ordered binary tree will sort it. Sorting can be a "hard" topic. Its either taken for granted, in which case it is boring or if it is analyzed mathematically where the resulting equations are difficult to understand. The important things about sorting are that it can be done fast - N items can be sorted in $N \log_2 N$ time and that many problems can work much faster on sorted data.

Improving calculation of non-bonded interactions with sorting.

For example, in molecular mechanics it is necessary to compute the non-bonded interaction between pairs of atoms. Since the atoms are *not* bonded, determination of the interacting atoms requires that *all* atoms be checked. Fortunately non-bonded interactions fall off quickly with distance so that atoms that are separated by some threshold distance can be considered not to interact. Even so, how can the correct atoms be identified.

Sort them along one axis, say the Z axis. Then select each atom in increasing Z value. Because of non-bonded interactions are assumed to be zero once two atoms are separated by some threshold, each atom needs to be compared against atoms in the Z-index until their Z values differ by more than this threshold. Originally it would take $O(n^2)$ steps to examine all pairs but after sorting it takes only $O(N \cdot Z_{den} + N \log_2 N)$ steps where Z_{den} is the number of atoms in the slice from Z to $Z + threshold$. The second term is the time required to sort the molecule. For molecules of any size both $\log_2 N$ and Z_{den} are substantially smaller than N .

Sorting with qsort () from the standard library.

The standard library provides a routine qsort () which will sort an array of N fixed size elements in $N \log_2 N$ time.

```

void qsort( base, nel, width, compare )
char *base;
int nel;
int width;
int ( *compare ) ();

```

base is the address of the array, nel is the number of elements in it, width is the sizeof each element in bytes (they must all be the same size) and compare is a function used to order the elements. This function takes pointers to the two array elements that are to be compared. It must return a negative, zero or positive integer value if the first element is less than, equal to or larger than the second. This code will sort an array of floats:

```

float a[ 10 ];
static int compare( f1, f2 )
float *f1, *f2;
{
    float fv;
    if( ( fv = *f1 - *f2 ) < 0 )
        return( -1 );
    else if( fv > 0 )
        return( 1 );
    else
        return( 0 );
}

qsort( a, 10, sizeof float, compare );

```

Arrays with natural bounds, using Unix

Arrays with natural bounds, using Unix

Arrays with natural bounds.

C arrays are very limited. All indices begin at 0, and for multi-dimensional arrays, all the sizes except the left most must be specified at compile time. The zero index limitation is a nuisance but the compile time size restriction is what makes C inadequate for serious numeric computation. It is not possible to write a C function that can work directly on arbitrary higher dimensional arrays. A C function can be written to invert 4×4 matrices or 512×512 matrices or any other fixed size, but no C function can *directly* invert $N \times N$ matrices where N is not known until the program is running.

It is always possible to map a multi-dimensional array onto a one dimensional array that has the same number of elements. The indices are combined in an address polynomial to select the desired elements. It works, but the resulting one dimensional indices can be formidable. A better solution makes uses of C's pointer facilities to simulate arrays with runtime bounds. The following is adapted from *Numerical Recipes in C*¹.

In C pointers and one dimensional arrays are closely related. In fact the array access operator is defined as a pointer reference:

```
a[i] is really *(a + i)
```

The value of a , the base of the array is added to the value of i , the index and the sum is the address of the desired element. The value of a is fixed and i ranges from 0 to $N-1$ where N is the number of elements in the array. The real difference between pointers and array is a consequence of how they are declared.

```
int aa[ 10 ];
int *ap;
```

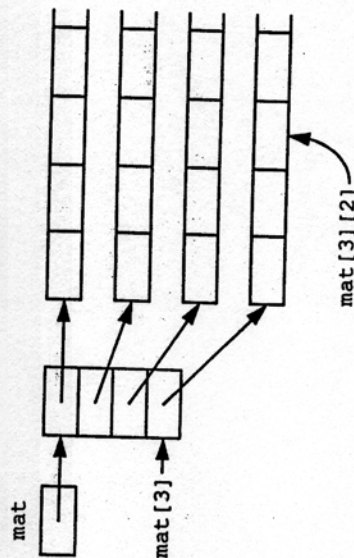
In the first declaration, aa is more than just a pointer, it is the address of an area in memory that holds 10 consecutive integers. This address is fixed when the program is compiled and can not be changed. These integers are at $*(a + 0)$, $*(a + 1)$, etc. This is why all C arrays begin at 0. The second declaration, declares that ap will hold an address, an address to assigned later and one that can be changed. This suggests a way in which the 0 lower bound may be circumvented. It is the value of the complete expression $(ap + i)$, the sum of the pointer plus the index that must be the address of the desired element. So if the index naturally ranges from *low* to *high* where *low* is not 0, then simply subtract *low* from ap , and then $ap[i]$ will behave as though the array had bounds *low* to *high* instead of 0 to $high-low+1$.

Now having built one dimensional arrays with arbitrary bounds can arbitrary matrices be built? Simple. C allows pointers to pointers, so allocate a group of ap like arrays and put all their addresses into an array of pointers. If this new pointer array had ap like bounds the result would be a matrix with arbitrary bounds.

¹ W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, Port Chester N.Y., 1988, pp. 15-21.

float

**mat;



The figure shows how mat points to an array of pointers, each of which in turn points to an array of floats. In this example, all the lower bound are 0, but by adjusting the contents of mat , and the elements of the array it points to, arbitrary lower bounds can be simulated. The "matrix" mat would be allocated in two steps, first the array of pointers, the rows of the matrix. The non standard function `matrix()` allocates variable bound two dimensional array:

```
float **matrix( nrl, nrh, ncl, nch )
{
    int i;
    float **m;

    m = (float **)
        malloc( (nrh-nrl+1)*sizeof float);
    m -= nrl;
    for( i = nrl; i <= nrh; i++ )
        m[i] = ( float *)
            malloc( (nch-ncl+1)*sizeof float);
        m[i] -= ncl;
    }
}
```

Each pointer has the appropriate lower bound subtracted from it after the allocation. A real subroutine should check that each call to `malloc()` was successful. To declare mat to be an $N \times N$ matrix, the user would write:

```
mat = matrix( 1, N, 1, N );
```


Useful Unix Services.

Accessing the environment. Unix provides three ways to get data into a program. In addition to files and command arguments programs can access environment variables, symbols that were defined in the shell that is currently running the program. Some of these variables, like HOME, TERM and SHELL are defined by the shell and describe the context or environment of a program. Other environment variables, set by the user are useful for setting program parameters that change infrequently like the location of libraries or working directories. The value of an environment variable is access via `getenv()`.

```
char *getenv( name )
char name[];
```

The argument is the name of the desired variable; the function returns a pointer to its value if the variable is defined and NULL if not. The actual value is stored in location internal to `getenv()` and is overwritten on each call. The following code fragment gets the value of the symbol RESLIB and writes it to standard output:

```
char *rp;
char reslib[ 40 ];

if ( ( rp = getenv( "RESLIB" ) ) ) {
    strcpy( reslib, rp );
    printf( "using res. library %s\n",
           reslib );
}
else {
    printf( "RESLIB not defined\n" );
    exit( 1 );
}
```

The value of the symbol RESLIB is copied to the local array `reslib`, so that any future calls to `getenv()` can not change it.

Getting the date and time. Unix provides several ways to access the current date and time. The simplest is to use `time()` followed by `ctime()`.

```
#include <sys/types.h>
```

```
time_t time( tloc )
time_t *tloc;
```

```
char *ctime( tloc )
time_t *tloc;
```

`time()` returns the number of seconds since 00:00:00 GMT, Jan 1, 1970. If `tloc` is not NULL, store this value in at the address pointed at by `tloc`. `ctime()` takes a pointer to a `time_t` variable and converts its value into a 26 character string like:

```
Sun Aug 9 14:26:43 1992\n\0
```

Single digit days and hours are preceded by a space so that each field has a constant width. `ctime()` returns a pointer to an internal area containing this string and must be copied to avoid having it overwritten on subsequent calls. `time_t` is defined in the file `/usr/include/sys/types.h`. This program gets and prints the time.

```
#include <stdio.h>
#include <sys/types.h>

time_t time_t;
char *ctime();

main()
{
    time_t t;

    t = time( NULL );
    printf( "%s", ctime( &t ) );
}
```

Executing shell commands from a program. Unix allows a program to execute a shell command via a call to the function `system()` in the standard library.

```
int system( cmd )
char cmd[];
```

`cmd` contains a command line for the sh (the Bourne shell, not the cshell). It differs from a typed command in that no newline is required. The return value is related to the exit status of the new shell, not the command it executed and is generally worthless. Some things are very easy to do with a shell but hard to do in a program. `system()` is an easy way to add this kind of functionality to a program.

As an example, a program could have a help file and write it out in response to the user command `help`. However if it was longer than the screen size, some sort of paging would be required to insure that the user saw the needed part. And of course the user should be able to list the help file line by line and search for things, etc. This sounds a lot like the Unix code, more. `system()` lets your code use more to provide help:

```
system( "more helpfile" );
```

This command starts a shell, then uses more to page through *helpfile*. Be sure that *helpfile* is properly named so that more can find it no matter where the program is running.

Another good use for `system()` is file name expansion. In C it is relatively difficult to read each entry in a directory, select those that match a pattern and sort the results. On the other hand, it is one shell command:

```
ls pattern
```

The direct solution is have `system()` execute the appropriate `ls` command into a temporary file, get the desired file names, one per line from this file and then delete it. The take home lesson

is to solve the problem and `system()` allows you to access some very powerful shell features from a C program.

Temporary files. Many programs require temporary files to hold large amounts of data. While temporary files are no different from ordinary files, some care must be taken in naming them. Each time a program creates a temporary file, it must use a new name. This is required so that concurrent instances of the same program can run without interference. Let's say that some program, say `search`, needs a temporary file and always uses `/tmp/search.tmp`. When the user starts the program, this file will be created. But since Unix is a multi-processing system, the same user can start another instance of `search` before the first one finishes. This new instance will create the same temporary file and both processes will write to it. Since neither process knows about the other, each uses the file as if it had exclusive access and the results will be wrong.

What is needed is some way to make unique temporary file names. The solution is to include the process id a number that is different for every running program as part of the file name. Several functions are provided in the standard library that generate temporary files. Here we consider only the simplest, `tmpnam()`

```
char *tmpnam( str )
char *str;
```

This function generates a file in the directory `P_tmpdir` which is defined in `<stdio.h>`. If `str` is `NULL`, the new file name is left in an internal area and will be overwritten on subsequent call. If `str` is not `NULL`, the new file name is written into it. `str` must be at least `L_tmpnam` bytes where `L_tmpnam` is also defined in `<stdio.h>`. After the file name has been created, it open via `fopen()` and accessed in the normal fashion. A program should delete all of its temporary files before exiting. The following program reads a pattern from input and lists all files that match that pattern.

```
#include <stdio.h>

char *tmpnam();

main()
{
    FILE *tfp;
    char tfname[ L_tmpnam ];
    char pattern[ 40 ], line[ 80 ];

    scanf( "%s", pattern );
    tmpnam( tfname );
    printf( line, "ls %s > %s", pattern, tfname );
    system( line );
    tfp = fopen( tfname, "r" );
    while( fgets( line, sizeof( line ), tfp ) )
        printf( line );
    fclose( tfp );
}
```

```
    unlink( tfp );
}
```

Errors and error messages. All C library routines that access the Unix operating system return a value indicating if the access was successful. For functions that do not return pointers, this value is 0 if the access was successful and not 0, if the request failed. For pointers, a `NULL` pointer (0 value) indicates failure and non zero success. In addition to the return value, the actual error number is written into the integer variable `errno`. This variable is accessed by adding this line near the beginning of the source:

```
#include <errno.h>

A cryptic description of each the values is given in the file
/usr/include/sys/errno.h
```

Longer but still cryptic descriptions are given in section two of the manual:

```
man 2 intro
```

The function `perror()` can be used to write a short message describing the current error to the standard error.

```
void perror( msg )
char msg[];
```

If `msg` is not `NULL` write it followed by a colon and then the system message. Otherwise write only the system message.

Separate Compilation, filters, data formats, further reading

Separate compilation.

C was designed for separate compilation. This means that the source of a program can be in more than one file. This offers several advantages. When a program is modified, only those files containing modified source must be recompiled which saves time. By placing a fully developed and debugged function or group of related functions in a file, they can be shared by different programs which increases reliability. And several people can work on different parts of a large program without having to worry about source file consistency. There is however, some cost to these advantages. Care must be taken about the visibility of names; that all the separate modules share the same assumptions; and that changes to a part of source result in recompilation of all the other parts of the source that are dependent on it. C addresses the first through storage classes, the C preprocessor handles the second and yet another separate program, `make` takes care of the third.

Storage classes. In C every variable has a name, a type and a storage class. The storage class specifies two things, the type of memory to use for the variable and the visibility of the variable's name. C has four storage classes, `auto`, `extern`, `register` and `static`. Syntactically, `typedef` behaves like a storage class but its action is completely different. Every variable declaration can be preceded by an optional storage class. In small programs, the default class is almost always acceptable but in larger programs and programs using separate compilation an explicit storage may be required.

The idea of visibility was introduced in section 4.5 along with scoping. Variables with global scope are visible in the current file at any point following their declaration. But that's not all, some global scope variables are visible *outside* of the current file. This is how functions in separate files share data - if they both contain a global declaration with the same name, then when they are linked together, both declarations reference the same variable. Simple.

Only local variables can have any storage class, and they default to `auto`. This means that each time the function (or block) is entered, new copies of these variables are created. These variables remain in existence until the block or function is exited. The names of `auto` variables are visible only inside the block that contains their declarations.

Declaring a local variable to have storage class `register` is a *hint* to the compiler that this variable will be used extensively and that it should be stored in one of the computer's fast registers. This is somewhat of a relic from the days when sophisticated code optimizers were not available. Now, most workstation compilers can generate better code without your help than with it and user interference in register assignment may actually slow things down! Obviously, it is not possible to take the address of a `register` variable.

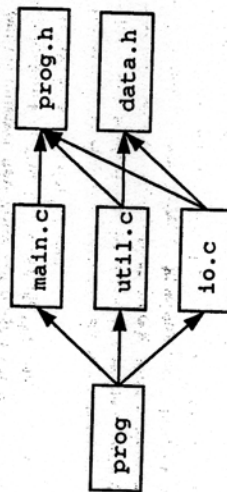
Both local and global variables may have storage class `static`. There is only one copy of any `static` variable. Thus if a `static` variable is declared inside a function, each time that function is called, the `static` declaration refers to the *same* variable. The variable exists for the entire program execution, not just the time the function is active. Thus local `static` variables can be used to *remember* things between function calls. The name of a `static` local variable has local scope.

Declaring a global variable to be `static` has a slightly different effect. First, there is only one copy of *any* global variable. When a global is declared `static`, this tells the compiler that the name of this variable is not available outside the current file. It is a private variable that can not be shared between source files.

The last class is `extern`, short for external. Preceding a declaration with this word tells the compiler the name and type of the variable and that the storage for the variable is declared in another file. If a "local" variable is declared `extern`, it restricts the visibility of its name to the current function in the current file. A global `extern` declaration also indicates that the storage will be defined and assigned in another file, but the visibility of the name is the same as any other global - the rest of the current file following the declaration. A global variable shared among files should be declared as `extern` in all but one of them. Declaring it `extern` in all of them will result in an undefined symbol message when the program is linked. The `extern` is optional, but good practice. It is up to the user to insure that all shared variables have the same (or consistent) declarations as this is not checked by the linker.

Header files. The C preprocessor is used to insure that each source file of a program contains a consistent set of declarations. Each source begins with preprocessor directives that include one or more "header" files. These files should contain symbolic constants and macros (`#defines`) and `typedefs` used throughout the source. Header files can also contain function declarations and declarations of `extern` variables. They should not contain ordinary global variable declarations or function definitions.

`make`, the program that generates commands. The input is usually a graph of file dependencies and the output is the set of commands required to insure that the files are consistent. Consider a program, `prog`, that has its source in three files, `main.c`, `util.c` and `io.c`. `prog` thus depends on these three files. Further assume that all of the `.c` files include the file `prog.h` and that `util.c` and `io.c` both include the file `data.h`. The relationships among these files can be represented as a graph. Here an arrow points from a file to another file that it depends on. Changing



ing a file at a node requires that all its predecessors be recompiled for the change to take effect. The `make` program takes a program dependency graph, checks the modification dates of all the files in the graph and "makes" all the predecessors that have more recent successors. For example if the file `io.c` was modified, `make` would recompile it, then link the new `io.o` and the old


```
cc -c file.c
```

So our third makefile is

```
# makefile version 3.0, just about right
prog: main.o util.o io.o
    cc -o prog main.o util.o io.o

main.o: prog.h
util.o:  prog.h data.h
io.o:   prog.h data.h
```

There is one last improvement. The list of files after `prog` has to be written twice, a possible source of error. `make` has a macro facility that allows a symbol to be defined for the list. Then this symbol is used in place of it. Our final makefile is

```
# makefile version 3.1, includes macro
OBJS = main.o util.o io.o
prog: $(OBJS)
    cc -o prog $(OBJS)

main.o: prog.h
util.o:  prog.h data.h
io.o:   prog.h data.h
```

The macro name `OBJS` must be enclosed in the parentheses and preceded by the dollar sign. Macro names are generally upper case. Macro lines can be continued across lines by ending each line but the last with a backslash.

Filter programming.

A filter is program that reads from the standard input, transforms the data and writes it to the standard output. As such it can be used at any stage in a pipeline. Some of the most useful Unix commands, like `awk`, `grep` and `sort` are filters. The importance of filters comes from their ability to be combined in a large number of ways allowing a small set of them to provide a lot of functionality.

A filter program needs to follow certain guidelines. It needs to produce the same style of output no matter if the input was from `stdin`, a named file or a pipeline. Any error messages must be written to `stderr`, not `stdout`. This insures that they come to the user's attention instead of disappearing into a file or down a pipe. And most important, the filter should do one thing. If its description includes an "and" or an "or", it might be a useful program but its not a filter. It is often worthwhile to add code to a filter to allow it to read one or more named files if it is the first stage of a pipeline.

The big reason to write filter programs is to take advantage of already existing code. For example, a user might write a code that performs some operation on each line a file. If the lines contain identifying symbols, then the same code can process subsets of these files by placing it in a pipeline following `grep` or `awk` to select the desired lines. Your program still processes every, you just made Unix supply the selection.

`main.o` and `util.o` to rebuild `prog`. If the file `prog.h` had been changed all three `.c` files would be recompiled and the new objects linked to make an updated `prog`.

`make` expects to find these dependency graphs in a file named `makefile` or `Makefile`. The syntax of the commands in this file can be downright baffling and those readers who wish a full description of `make`'s capabilities should refer to *Managing Projects with Make*.¹ However for simple and even medium size codes the `makefile`s are relatively straightforward.

The dependency graphs, as a `makefile` can hold more than one, consist of one entry for each interior node of the graph(s). This entry begins with a dependency line followed by zero or more command lines. The dependency line begins with the target, the filename of the node followed by a colon followed by the names of the files the target depends on. The commands line(s) that follow the dependency line must begin with a `tab` and describe how to update (or make) the target. A long line can be continued over several real lines by ending all but the last line with a backslash. ('\\'). The dependents on the first target have their own entries and this continues until every interior node has an entry. Thus the first `makefile` for `prog` is

```
# makefile version 1.0, naive approach
prog: main.c util.c io.c
    cc -o prog main.c util.c io.c

main.c: prog.h
util.c:  prog.h data.h
io.c:   prog.h data.h
```

When the user types `make` everything works, the three `.c` files are compiled and linked. Unfortunately this file does not always do what we want. It works but consider what happens if `io.c` is modified. All *three* files are recompiled! What's the point? The problem is that `prog` was said to be dependent on the `.c` files which is true. But this ignores the fact that there is a two step process in going from the `.c` files to `prog`. First the `.c` files are compiled into object or `.o` files. Then the `.o` files are linked to make `prog`. So our next `makefile` is

```
# makefile version 2.0, wordy, but what we want
prog: main.o util.o io.o
    cc -o prog main.o util.o io.o

main.o: main.c prog.h
    cc -c main.c

util.o: util.c prog.h data.h
    cc -c util.c

io.o:   io.c prog.h data.h
    cc -c io.c
```

This time when the user modifies one of the source files, only that file is recompiled, the others are left untouched. However, we typed way too much. Most of the commands in this file are not needed. `Make` assumes that a `.o` file depends on a `.c` file with the same name. And `make` just assumes that `.c` files are converted to `.o` files with the command

1. S. Talbot, *Managing Projects with Make*, O'Reilly & Associates, Sebastopol, CA, 1990.

Data file formats.

If a program is to be used repeatedly, some thought should be given to the data formats. Many scientific codes process columns of data and this section applies to them.

Use an ASCII file. Keep the data organized as (short) lines as ordinary text. It will take longer to read and write as numbers must be converted between internal and external formats, but you can read and *correct* the data files with *vi*. Organization as columns and lines also allows you to use Unix filters on your data extending your programs capabilities.

Define "comment" lines. These are lines that are read by the user, not the program. A '#' in column 1 is easy and widely used in the Unix world. Put the data's creation date in a comment. This can be a lifesaver. Every Unix file contains its creation and last modification date, but these don't always reflect the creation date of the data in the file. With the date in a comment, you'll always know.

Ignore blank lines.

Explicitly separate all the columns by white space. Do this by including at least one space between each format descriptor in the format strings that write the date. Don't assume that numbers will always fit in the field width in the format. By including a space between each format descriptor, some lines may be longer than usual, but two fields will never run together.

Keep in mind the difference between a report and a database. A database should have only data, no headings or summary lines. Every line should have the same format with the possible exception that the one or two lines at the beginning of the file describe the format of the data. By using this protocol it can be processed by filters where the database is recreated between stages in a pipe. The only differences are the number of data lines and columns keeps changing. A report is generated by a program that is the last stage of pipe. It can add headings and summary information, what ever is needed for easy reading, because its contents will never be reprocessed only printed. Where to go for more.

Wirth, N. *Algorithms + Data Structures = Programs*.

Very clear introduction to most concepts used in non numerical computing, including, sorting, searching, recursion and compiling.

Kernighan, B. and Plauger, X., *Software Tools*.

How to design real programs. Most of the programs developed in the book are available as standard Unix commands. Each chapter takes a topic, like filter programs, or pattern matching and develops one or more programs that use that topic. The programs are useful, but the real value of the book is in the text that explains how and why each program was developed.

Kernighan, B. and Pike R., *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

"Heavy sledding." This book describes most services available under Unix, files, filter processing, shell programming and document preparation. Most useful in seeing how to design a set of programs that work well together.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T., *Numerical Recipes in C*, Cambridge University Press, Port Chester N.Y., 1988.

Contains a good introduction to most numerical algorithms, plus references if you need more. The programs are available on magnetic media. Most work, although some bugs have been reported. The programs make some unusual assumptions about data storage that you may not agree with. Aho, A. and Ullman X., *The Dragon Book*. The only book on compiler design.

Knuth, D., *The Art of Computer Programming*, v. 1. Fundamental Algorithms. — v. 2. Seminumerical Algorithms. — v. 3. Sorting and Searching. Addison-Wesley, Reading MA, 1981.

Contains detailed mathematical analysis of random numbers, arithmetic, and searching and sorting. If you really need to know why, then look here.