Notes on C

Version 0.0

John H Remmers
Dept. of Computer Science
Eastern Michigan University
(remmers@emunix.emich.edu)

These notes are intended for programmers who are experienced in a high-level structured language such as Pascal and who need a basic reference and "quick start" for programming the C Language. They are not a complete reference to the C language; however, it is hoped that a person who has mastered these notes can learn additional features and fine points quickly as needed.

We use the ANSI version of C.

1. Identifiers

An identifier in C is a letter or underscore followed by any number of further letters, underscores, and decimal digits. Identifiers are used to denote entities such as constants, variables, arrays, functions, and macros.

In addition, C has a number of keywords consisting of lower-case letters but whose meaning is reserved by the language and which may not be used as identifiers. The keywords in ANSI C are

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

Upper and lower case are different in C. Thus, `While` and `WHILE` are valid but different identifiers, and both are distinct from the reserved word `while`.

An identifier may be of any length. Some older C compilers may recognize only the first 8 characters as significant, but modern compilers usually treat every character of very long identifiers (e.g. 64 characters or longer) as significant.

Examples: The following are all valid identifiers.

```
fubaz          xy13          TheLength      theLength      the_length
```

2. Comments

Comments in C are delimited by the character sequences `/*` and `*/`. Comments may coexist on the same line as code and may extend over any number of lines.

Examples:

```
/*
 * this is a
 * multi-line comment
 */

i = 2 * i + 1;       /* This is an end-of-line comment */
```

3. Primitive Data Types

C supports integer, real, and character data. Unlike many languages, it supports multiple flavors of each, such as short and long integers, single- and double-precision reals, and signed and unsigned characters.

One creates simple variables of a primitive data type via a *declaration list*, consisting of a type specifier followed by a comma-list of identifiers naming the variables being declared. The simplest type specifiers are

`int`	"normal" integer. Typically 16 or 32 bits, depending on machine architecture, but not guaranteed to be either.
`char`	character. Normally 1 byte (8 bits). Left-most bit may or may not be interpreted ad a sign, depending on the compiler.
`float`	normal precision real. Typically about 7 or 8 significant decimal digits.
`double`	extended-precision real. Typically 15 or 16 significant decimal digits.

By prepending the modifiers `long`, `short`, `unsigned`, and `signed`, one specifies variations on these types, as in

`long int`	integers which might be longer (i.e. more bits) than a normal `int`. Typical size is 32 bits. On machines with 32-bit architectures, often the same as `int`.
`short int`	integers which might be shorter than a normal `int`. Typical size is 16 bits.
`unsigned int`	integer in which every bit is treated as a data bit, rather than one bit being reserved as a sign. An unsigned `int` can never test as negative.

Meaningful combinations of modifiers are allowed, such as "`long unsigned int`".

There are additional modifiers that affect the scope of a declared object or the way in which memory is allocated for it. These will be discussed later.

A declaration consists of a type specifier followed by a comma-separated list of the objects being declared, terminated by a semicolon. **Examples** of simple variable declarations:

```
int alpha, beta, gamma;
double length_of_beam;
unsigned short int mask1, mask2;
char ch;
```

Depending on the placement of a declaration in a program, the identifiers may be local to a function or block, or global to the entire program. Again, more on this later.

4. Literals

In addition to more-or-less standard notation for literal constants, one can indicate octal and hexadecimal constants in C, as well as primitive data values of modified types.

Examples:

`1036`	decimal integer one thousand three hundred and six
`01036`	octal value $1(8^3) + 3(8) + 6 = 542$ decimal (string of digits preceded by 0 always represents octal)
`0x1036`	hexadecimal value equivalent to `4150` decimal. (string of digits preceded by `0x` or `0X` always represents hexadecimal)
`1036L`	1036 stored as a `long int`.
`'f'`	character value `f`.
`136.45`	float value with non-zero decimal part
`13.2e-3`	equivalent to `.0132`. (`e-3` is a scale factor)

Certain character literals are *escape sequences* representing commonly-occurring control characters. Each escape sequence starts with the backslash character `'\'`. The most important are

`'\n'`	newline
`'\t'`	horizontal tab
`'\0'`	null character
`'\b'`	backspace
`'\a'`	audible bell

String literals, represented by sequences of characters enclosed in double quotes, are a type of array and will be considered after arrays have been discussed.

5. Arrays

As in many other languages, an *array* in C is a sequence of values, all of the same type, addressable by an integer *index* or *subscript*. An array is created by specifying the component type and the size in a declaration. Array subscripts *always* start with `0` and go up to one less than the declared size. Arrays are not dynamic; the size must be a constant known at compile-time. (However, this restriction can be gotten around by using dynamic memory allocation, discussed later.)

Examples of array declarations:

```
int tb[20];
```
- array of 20 integers; first element `tb[0]`, last `tb[19]`

```
char str[64];              - array of 64 characters
double length[17628]       - array of 17628 extended-precision reals
```

Array references are formed by following the array name by an integer expression in square brackets.

Multi-dimensional arrays are formed as follows:

```
int tbl[20][50];           - array of 20 rows and 50 columns
```

and referenced similarly: `tbl[i][j]` is the element in the `i`th row and `j`th column of `tb`.

6. Strings

A string literal is a sequence of characters enclosed in double quotes:

```
"Hello"
"This is a test."
"F"
"This is a string\ncontaining three\nlines of text."
```

The string `"F"` is not the same as the character literal `'F'`. A common mistake in C programming is to use one when the other is appropriate.

Escape sequences such as `\n` have the same significance in string literals as in character literals.

the literal `""` denotes the *empty string*.

To embed the double quote character itself in a string, precede it by a backslash:

```
"She said \"Hello\"."
```

At runtime, the characters of a string are stored in consecutive bytes of memory, terminated by a null character. Thus the string `"Hello"` occupies 6 bytes of memory, not 5. In fact, a string is just a constant character array, and individual characters can be referenced by subscript notation:

```
"Hello"[0] = 'H'
"Hello"[4] = 'o'
"Hello"[5] = '\0'
```

The opening and closing quotes of a string literal must be on the same line.

In ANSI C, string literals may be concatenated to form a single null terminated string; the literals are separated by "white space" (blanks, tabs, newlines). For example

```
"hello" " " "there"
```

represents the same string as

```
"hello there"
```

and the same string as

```
"hell"
"o t"
"here"
```

The main reason for this construction is to facilitate construction of long strings not limited in length to a single line.

7. Structures

A **structure** in C is a heterogeneous collection of data objects. "*Heterogeneous*" means that the objects contained in a structure may be of different data types. The concept of a structure in C is essentially the same as a Pascal *record*; identifiers are used to reference the objects, or *fields*, of a structure. A structure type is defined by a *template* that gives a name to the type and specifies the names and data types of the fields; here is an example of a structure template that might be appropriate for creating objects that have a name, a weight, and an age:

```
struct item {
   char name[20];
   float weight;
   int age;
};
```

Here, `struct` is a keyword, `item` is a name, or *tag*, for the structure type, and `name`, `weight`, and `age` are the field names. All the braces and semicolons shown are required.

The above example is essentially a type definition. It does not by itself create any structure objects; this can be done by declaration, such as:

```
struct item obj, list[100], tabl[10][15];
```

This declaration creates a simple `item` structure called `obj`, an array `list` of *100* `item` structures, and a two-dimensional array of `item` structures called `tabl`.

Reference to fields of an item structure can be made using the *selector* operator ".", just as in Pascal. The following are legal field references, given the above declarations:

`obj.weight`	- the `weight` field of `obj`, of type `float`
`obj.age`	- the `age` field of `obj`,of type `int`
`obj.name[0]`	- the initial character of the `name` field of `obj`
`list[3].weight`	-the `weight` field of component *3* of `list`
`tabl[0][14].name[1]`	- the 2nd character of the `name` field of the item structure in row 0, column 14 of `tabl`

Structures may be nested; that is, a field of a structure may itself be a structure.

The fields of a structure may be given initial values by attaching an initializer to the definition, as in

```
struct item widget = {"SUPER WIDGET", 26.6, 15};
```

8. Program Form

The term *function* in C means the same thing as it does in many other program languages: a block of code that can be *called*, receive *parameters*, execute *actions*, and return a *value* to the caller. A function has a *name* given by an identifier, and can contain declarations of local objects (variables, arrays, structures) not accessible outside the function. A function may be recursive, either directly by calling itself, or indirectly through a chain of calls to other functions.

A *program* in C is a collection of functions and global data objects. (An object is global if it is defined outside the scope of any function.) A complete program must contain a function called `main`; this is where execution always begins.

Thus, a C program in broad outline, looks like this:

data object definition
data object definition
...
function definition
function definition
function definition
...
data object definition
data object definition
data object definition
...
function definition
(etc.)

Data and function definitions may appear in any order, although the most common convention is to define all the data objects first, then the functions. In contrast to Pascal and some other block-structured languages, function definitions in C may not be nested; that is, a function definition may not contain the definition of another function inside it.

The general form of a function definition is

<*return-type-and-name*> (<*parameters*>)
<*body*>

The *return-type-and-name* specifies the name of the function (an identifier) and the data type of the value returned to the caller. In typical cases, *return-type-and-name* is just a designator for the data type followed by the name, although for complicated data types it can have a more elaborate form. If the function returns no value (i.e., is like a Pascal procedure), this is indicated by specifying the keyword `void` as the return type.

In ANSI C, *parameters* is a comma-separated list of parameter declarations, syntactically the same as variable declarations except that the terminating semicolons are absent. If *parameters* is either the empty list or the keyword `void`, the function passes no parameters. The enclosing parentheses are always required however, even for an empty parameter list.

A function *body* is a compound statement, delimited by braces `{` and `}`, that contains local data declarations as well as *statements* that define what actions the function is to perform. We will discuss this in detail in the section dealing with statements.

Here are some sample function definitions; bodies are only indicated for now:

Function `foo` accepting one integer and two character parameters, and returning an integer:

```
int foo (int bar, char ch, char ch2)
```

```
{  …  }
```

or equivalently

```
foo (int bar, char ch, char ch2)
{ … }
```

(A missing return type defaults to integer. However, it is recommended that you always specify the return type explicitly and not use this feature.)

Function `sort` accepting an integer array parameter and an integer parameter, returning no value. Note that the array size is not given explicitly. Any integer array of any size may be passed when the function is called.

```
void sort (int a[], int n)
{  …  }
```

Function `bloop` that takes a `struct item` parameter (see previous section) and returns a value of the same type:

```
struct item bloop(struct item thing)
{  …  }
```

Function `msg` that receives no parameters and returns no value:

```
void msg(void)
{  …  }
```

or

```
void msg()
{  …  }
```

9. Function prototypes, scope, and storage class

C uses static scoping rules such as are commonly found in other block-structured programming languages, such as Pascal. A function may be referenced (e.g., called) at any point in a source file after a definition or prototype (explained below) for the function appears. Similarly, a global data declaration applies from the point it appears to the end of the source file, unless overridden by a local declaration of the same name. Local declarations are visible at compile time only in the block where they appear; a local declaration of a name always overrides a more global one for the same name for the duration of the block.

A *function prototype*, a concept introduced in ANSI C, is roughly the same as what in some other languages is called a *forward declaration*: A specification of the function's name, return-type, and parameters, but without the body. A function prototype is terminated with a semicolon. The full definition of a function given by a prototype may appear either later in the same source file as the prototype, or in a different source file.

Here are function prototypes for the example functions of the previous section:

```
int foo(int, char, char);
```
- note that only the types of the parameters need to be specified, not the names

```
void sort(int[], int);

struct item bloop(struct item thing);       - on the other hand, you can specify
                                              parameter names if you want to

void msg(void);
```

Use of function prototypes is optional, but it is recommended that you use them. At the beginning of a source file, list prototypes for all functions defined or called in that file (except for standard library functions, in which this is done in associated header files like `stdio.h`). This is good documentation and also enables the compiler to check consistency of parameter passing in function calls.

C allows the parts of a program to be split up among several different source files. This facilitates modular programming, in which a collection of closely related functions and global data structures are put together in a single source fie. There are some issues of scope connected with this. Ordinarily, a function or global data structure defined in one source file is accessible from another. For functions, this is accomplished by supplying a function prototype in the second source file. For data objects, one puts an *external declaration* in the second file. For example, if one source file contains definitions of a global item structure `obj` and a function `sort` as follows:

```
struct item obj;

void sort(int a[], int n)
{
   <interior of function body>
}
```

Then a second source file containing

```
extern struct item obj;
void sort(int[], int);
```

could manipulate `obj` and call `sort`

Often it is desirable to make a function or data object *private*, that is, accessible only in the file in which it is defined. One can do this by using the `static` attribute, as follows, at the point of definition:

```
static struct item obj;

static void sort(int a[], int n)
{
   <interior of function body>
}
```

If this is done, attempts to refer to `obj` and `sort` from a different source file will fail.

The attribute `static` may also be given to local variables of a function, but the effect is different than for globals: A local data object declared as `static` has its value *saved* between function calls (although it is still referenceable by name only within the function body), whereas a local object without the static attribute is discarded when the function returns. If the declaration contains an initializer, the initialization is done for static objects only once, when the program is loaded; for local objects that are not static – called *automatic* – the initialization is done every time the function is called.

10. Expressions and Operators

An *expression* in C is built from constants, identifiers, operators, and grouping symbols such as parentheses. Expressions specify rules for computing values at run time.

C has an unusually large number of operators. Here are the standard ones, grouped by category:

Arithmetic Operations:

`+` (add), `-` (subtract), `*` (multiply), `/` (divide), `%` (remainder)

The division operator denotes *integer* division if its operands are integer, otherwise *real*.

Relational and Logical Operators:

`>` `=` `<` `<=`	(greater, greater or equal, less, less or equal)
`==` `!=`	(equality, non-equality (*note the double equals!!*))
`&&` `||`	(logical and, logical or -- *left-to-right short-circuit evaluation*)
`!`	(logical negation)

These operators always yield a value of 0 (false) or 1 (true). However, they accept values other than these as operands, treating any non-0 as true. For example, the value of `6 || 0` is `1`.

Increment and Decrement Operators:

`++`	(post-increment when used as postfix operator, pre-increment when used as prefix operator)
`--`	(post-decrement when used as prefix operator, pre-decrement when used as prefix operator)

Explanation: if `i` is a variable, both `++i` and `i++` have the effect of increasing `i` by `1`. However, the *value* of `++i` is the *new* value of `i`, whereas the value of `i++` is the *old* value. For example, if `i` is `6`, then `printf("%d\n", ++i)` will display `7`, but `printf("%d\n", i++)` will display `6`. Both will leave the value of `i` at `7`.

Bitwise Operators:

`&` `|` `^`	((bitwise and, or and exclusive or)
`<<` `>>`	(left shift, right shift)
`~`	(ones complement)

In contrast to the logical operators, the bitwise operators work bit-by-bit, and can yield results other than 0 and 1. For example, `5 & 7` is equal to `5`. (Write out the binary representations of 5 and 7 to see this.)

Assignment Operators:

`=`	(Assigns value of right-hand operand to the left-hand operand. the latter must be a variable.)

C views assignment as an *expression* whose value is the same as that of the right-hand-side expression. That means that assignments can be part of a larger expression. For example, if `a, b, c` are `int` variables, then the expression

```
c = ( a = 2) + (b = 3)
```

sets `a` to `2`, `b` to `3`, and `c` to `5`. (This is not intended as an example of good coding practice!) A more common use of "embedded assignment" is exemplified by the code fragment

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

where a variable is simultaneously set and tested.

C has a number of other assignment operators that have the effect of "updating" the left-hand-side variable according to a specified operation and right-hand-side expression. The updating assignment operators are

```
+=   -=   *=   /=   %=   <<=   >>=   &=   ^=   !=
```

For example, the assignment

```
i += j
```

adds the value of variable `j` to variable `i`.

Here are three different ways to increment variable `i` by `1`:

```
i = i + 1          ++i          i += 1
```

Conditional Expressions:

A conditional expression is formed from three operands and the characters `?` and `:` thusly:

expr1 `?` *expr2* `:` *expr3*

It is evaluated as follows: *expr1* is evaluated. If it is non-zero (true), then *expr2* is evaluated and becomes the value of the conditional expression. If it is 0 (false), then *expr3* is evaluated and becomes the value of the expression. In all cases, exactly one of *expr2* and *expr3* are evaluated.

Thus, a conditional expression can be thought of as an expression-level version of *if-then-else*.

Example: `p *=  ((n%2) ?  2 : 3)`

where `n` and `p` are integer. This multiplies `p` by `2` if n is odd and by `3` if n is even.

Other Operators:

The array-subscripting operator `[]` and the field-selector operator `.` were covered in earlier sections.

Two important operators used in connection with pointers -- the *address-of* operator `&` and the *dereferencing* operator `*` will be covered in the section on pointers.

11 Statements

Statements in C specify actions to be performed at runtime. They are found inside function definitions.

Most of the statements in C have close analogs in other well-known block-structured languages, such as Pascal. Since we are assuming that the reader is familiar with at least one such language, our treatment here will be brief and expand only on features that are "unusual".

Expression Statements and the null statement:

The simplest statements in C are the *expression statements,* formed by appending the semicolon character `;` to an expression. This can be done to *any* expression. An expression statement is executed by evaluating the expression. ***Examples:***

`x = ( -b + sqrt(b*b - 4*a*c) / ((2*a);` – assignment statement that solves a quadratic equation.

`++index;` – increment statement that adds 1 to variable `index`

`scanf("%d", &n);` – function-call statement that reads an integer value into `n`.

`42;` – do-nothing statement that expresses the meaning of life.

In general, semicolons function in C as *terminators*, in contrast to Pascal where they function as *separators*. This means that semicolons are often required in C where they are not in Pascal, or required in Pascal where they are not in C.

The official *null statement* in C is denoted by a semicolon `;` all by itself. One finds it in places where syntax requires a statement but there is nothing to be done, e.g. a while-loop with an empty body (which comes up more frequently than one might think).

Compound Statements (Blocks):

A *compound statement* or *block* is a sequence of zero or more statements enclosed in braces { and }. The statement may be preceded by declarations local to the block:

```
{
    optional declarations
    statement-1
    statement-2
    .....
    statement-n
}
```

A compound statement specifies *sequential execution:* The statements are executed in the order that they appear. Note that semicolons are not used to separate the component statements, although a semicolon may be required as a terminator to some of the compound statements.

Example:

```
{
    int i, j;
```

```
    scanf("%d %d", &i, &j);       /* read integers from stdin */
    k = i * i + j * j;            /* compute sum of their squares */
    printf("%d\n", k);            /* write this on stdout */
}
```

Note that `i` and `j` are local to the block; their values will be lost when control flows out of the block. Since `k` is not declared in the block, it must either be global, local to some surrounding compound statement, or a parameter of the function in which the block is located. Its value will still be available when the block shown above finishes executing.

The body of a function must always be a block. Here is an example of a complete function definition:

```
void starmess(char m[])    /*print message surrounded by *** */
    {
    printf("*** %s ***\n", m);
    }
```

This function could be called in an expression statement:

```
starmess("hello");
```

This would print the output

```
*** hello ***
```

to the standard output.

If:

The *if-statement* has one of the two forms

`if (`*expression*`)` *statement*

or

`if (`*expression*`)` *statement-1* `else` *statement-2*

In the first case, the statement is executed if the expression if non-0 (true) and skipped otherwise. In the second, the first statement is executed and the second skipped if the expression is non-0; otherwise, the second statement is executed and the first one is skipped. *Statement-1* and *statement-2* are often called the *then-clause* and *else-clause*, respectively.

Note that parentheses surrounding the expression are *required*. Note also the absence of a "then" keyword.

A clause of an if-statement may be any valid statement incuding another if-statement This means that constructions like this work:

```
if (expression-1)
        statement-1
else if (expression-2)
        statement-2
else if (expression-3)
        statement-3
else
        statement-4
```

Style note: When a clause of an if-statement is a compound statement, it is common (although not universal) practice to put the opening `{` brace on the same line as the keyword that announces the clause, and to put the closing brace at the same level of indentation as the keyword. For example:

```
if (n > 0) {
    ++i;
    k += i;
}
else {
    --i;
    k -= i;
}
```

The author of these notes favors this style, as it tends to make programs shorter (fewer lines) without detracting from readability.

Switch and Break:

The *switch* statement allows multi-way decisions. Its general syntax is rather complicated; we present it only in its most commonly used form, which is similar to Pascal's "case" statement:

```
switch (expression) {
    case-labels
        statements
    case-labels
        statements
    .....
    case-labels
        statements
}
```

Here, the *expression* should be of some integer type (`int`, `long int`, `char`, etc.). Each case label has the form

```
case constant :
```

where the constant is of the same type as the leading expression. The switch statement branches to a case-label whose associated constant matches the current value of the expression and executes the statements which appear from that point on. One of the case labels may be

```
default:
```

where control will branch if none of the constants match the expression value. The default label can appear anywhere that a case-label is allowed, although in most cases it is the last case-label.

If there is no default label and no case-label matches the expression, all the statements in the switch-statement are skipped.

An **important** point to be aware of is that execution of statements in a switch statement does not terminated when another case-label is encountered – in other words, execution of one case will "flow into" the next case. This is not usually the desired behavior, and special statement called a *break statement* will override it. Execution of a `break` statement causes immediate termination of the `switch`.

Example: The following `switch` statement increments one of three different counters according as the current value of `c` is a vowel, consonant, or something else.

```
switch (c) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
    case 'A': case 'E': case 'I': case 'O': case 'U':
        ++vcount;
        break;
    default:
        if (isalpha(c))
         ++ccount;
        else
         ++ocount;
        break
}
```

While:

The *while* statement implements a test-first condition-controlled loop, with essentially the same semantics as the while constructions in other structured languages, such as Pascal. The syntax is

```
while (expression)
    statement
```

Execution is as follows: (1) The expression is evaluated. If it is zero (false), execution of the `while` statement terminates. If it is non-zero (true), then (2) the statement is executed and step (1) is repeated.

The statement may be any valid C statement, but is perhaps most commonly a compound statement. In this case, it is fairly common practice to put the opening brace on the same line as the `while` keyword:

```
while (expression) {
    .....
}
```

Break and continue:

We have already encountered the *break* statement as a way of "breaking out" of a switch statement. It can also be used to terminate a loop, such as while statement, as in the following simple linear search of an array:

```
i = 0;
while (b[i] != x) {
    ++i;
    if (i  ==  n)
        break;
}
```

The loop will terminate if either the value *x* is found or the entire array has been searched without finding *x*.

The semicolon following the keyword `break` is part of the statement and is always required.

The *continue* statement starts the next iteration of a loop. (The author of these notes seldom uses it.) Its syntax is

```
continue;
```
(semicolon required)

For:

The *for* statement sets up a loop. It is somewhat similar to Pascal's *for* statement and Fortran's *do* statement, but more flexible. The syntax is

```
for (expr1; expr2; expr3)
        statement
```

In most cases it is equivalent to

```
expr1;
while (expr2) {
        statement
        expr3;
}
```

In other words, *expr1* does loop initialization, *expr2* determines loop continuation and *expr3* is an upgrading operation performed immediately before *expr2* is evaluated again.

For example

```
for (i = 0; i < 10; ++i)
        b[i] = 0;
```

is a counter-controlled loop that sets the first *n* elements of array *b* to zero. (What is the value of *i* when the loop terminates?)

Because the expression components of a `for` statement can be any valid expressions of type integer, the C `for` statement offers more flexibility than its counterparts in other well-known languages. For example:

```
for (i = 0, j = n-1; i < j; ++i, --j) {
        tmp = b[i];
        b[i] = b[j];
        b[j] = tmp;
        }
```

controls two counters, one increasing and the other decreasing, to reverse the elements of an array. (The commas in the expressions above are uses of the comma operator, to be described in the next section.)

The expressions in the `for` statement may be omitted, but the separating semi-colons are always required. For example:

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

is the minimum *for* statement: no expressions and the null statement as the body. A missing *expr2* is treated as identically true; thus the above example is an infinite loop.

Do:

The *do* statement is like *while,* except that it tests the continuation condition at the bottom rather than the top. Syntax:

```
do
        statement
while (expr);
```

Thus the statement is always executed at least once. Execution of the *do* statement terminates when the expression becomes zero.

Return:

The *return* statement is used to return control from functions. It has two forms:

```
return;
```

and

```
return expression;
```

The first form is appropriate for functions of return type `void` that return no value to the caller. For other functions, the second form should normally be used; the value of the expression is returned to the caller. The expression should be of the same data type as the function's return type. ***Example:***

```
int max3 (int x, int y, int z) {
      int m;
      m = x;
      if (y > m)
            m = y;
      if (z > m)
            m = z;
      return m;
}
```

12. Two unusual operators: *sizeof* and *comma*.

In this section we cover two useful C operators for which many other commonly used programming languages fail to have analogs.

The *sizeof* operator yields the size of a data object or type in bytes. The forms are

```
sizeof (expression)
```

and

```
sizeof(data type)
```

Note that parenthesis around the argument are required in the second form but not the first. For example, given the declarations

```
char c, str[50];
int n, list[20];
```

and assuming (as is true on many machines) that a character occupies 1 byte (8 bits) and an integer 4 bytes (32 bits), we would have

```
sizeof c    =     1
sizeof str  =     50
sizeof n    =     4
sizeof list =     80
sizeof (int)=     4
```

The values returned by the *sizeof* operator are system-dependent. Sizeof is probably most often used in conjunction with the memory allocator `malloc()` to allocate sufficient amount of memory for a new object.

Another unusual but useful operator is the *comma* operator, denoted simply by the comma character. Its form is

expr1 , expr2

where *expr1* and *expr2* are expressions of the same data type. This is called a comma-expression. It is evaluated by evaluating first *expr1*, then *expr2*. The value of *expr2* is the value of the comma-expression. Comma-expressions may be nested; one can write

expr1 , expr2 , ... , exprn

to evaluate each of the expressions in the order listed. Again, the value of the whole thing is the value of the last expression.

The comma operator is useful in places where one expression is required but one needs to evaluate several, e.g., as the components of a `for` statement. (See the example from the `for` statement section where the comma operator was used.)

13. Pointers

A *pointer* is a value that denotes the *location* of an object. A synonym is *address*. Pointers are fundamental in C programming; knowing how to use them is essential.

The *address-of* operator:

Given a named object that corresponds to a location in memory, the *address-of* operator can be used to generate a pointer to it. For example, given the declarations

```
int i;
char str[256];
struct thing foo;
```

the following expressions are valid:

`&i`	- address of variable `i`
`&str[0]`	- address of initial element (i.e., the "base address) of `str`
`&foo`	- address of structure `foo`

The following are *not* valid:

`&356`	- can't take address of a constraint
`&&i`	- can't take address of an address

C considers pointers to objects of different data type to themselves be of different data type. In the above example, `&i` is of type "pointer to `int`", `&str[0]` is of type "pointer to `char`", and `&foo` is of type "pointer to struct `foo`". The distinction is important in pointer arithmetic, discussed later in these notes.

Just what a pointer "is" (i.e., the implementation) depends on the architecture of the underlying machine. If the machine has a flat 32-bit address space, pointers are most likely just 32-bit quantities, i.e. indistinguishable at the implementation level from `long int` quantities. If the address space is segmented (as in the older Intel 8086 and 8088 microprocessors), pointer structure is more complicated, and in fact the C implementation may distinguish different kinds of pointers ("near pointers, "far" pointers, etc.) – this is seldom a concern in a Unix environment, however.

The address-of operator is often needed when passing an object as a function parameter, if the function is supposed to change the value stored in the object. The reason is that parameters are always passed by value in C; hence a function needs the address of an object in order to change the object. This is commonly seen in the standard `scanf` function, which stores values read from memory into variables:

```
scanf("%d", &i);          - read value into variable i
scanf("%c", &str[255]); - read character into last element of array str
```

Failure to use the `&` in the above examples will result in `scanf` interpreting the *value* of the variables as an address, usually with disastrous results. ("Segmentation violation" is a common error message here; it means that the program tried to access memory outside its address space.)

In contexts where a pointer is required, C interprets an unsubscripted array name as a pointer to the initial element of the array. Thus,

```
scanf("%c", str)
```

is legal and equivalent to

```
scanf("%c", &str[0])
```

i.e., a `scanf` call that reads a single character into the initial element of str.

Note that the symbol used for the *address-of* operator is identical to that used for the bit-wise *and* operator. The C compiler can always tell from context which meaning is intended, since *address-of* takes a single argument and bitwise *and* takes two.

The *dereferencing* operator:

The dereferencing operator is the inverse of address-of: Applied to a pointer, it yields the object pointed to. It is denoted by the symbol *.

For example, if `i` is a variable of type `int`, then the expression (`*&i`) is equivalent to `i`, and the assignment

```
*&i = 35
```

has the same effect as

```
i = 35
```

(One would always write it the second way; the example illustrates only the concept of dereferencing, not a practical use.)

The dereferencing operator is necessary, for example, in functions which receive pointer parameters, in order that the function can manipulate the objects pointed to. Before discussing this in detail, we need to talk about pointer variables and declarations.

Pointer variables

A *pointer variable* is a variable whose value can be a pointer. The type of a pointer variable involves the type of the object pointed to – e.g. a variable of type "pointer to `int`" should only be assigned addresses of `int` objects, not of other types of objects such as chars.

Pointer variables are declared using dereferencing syntax, e.g.

```
int *ip;                - declaration of variable ip of type "pointer to int"
struct thing *tp;       - variable of type pointer to struct thing
int *iplist[100];       - array each of whose element is a "pointer to int"
```

Declaring a pointer variable does not in and of itself allocate an object for the variable to point to. This must normally be done by assignment. *Example:*

```
int i, *ip;             - declare int variable i and pointer to int ip
...
i = 45;
ip = &i;                - store address of i in ip
printf("%d\n",  *ip);   - prints 45
```

If a pointer does not point to an object , it is an error to dereference it (and may well give a segmentation violation.)

Here is a complete function that swaps the values of two `int` variables. The parameters must be pointers to the variables in order for the function to change the values of the variables:

```
void swap (int *ip, int *jp) {
      int tmp;
      tmp = *ip;
      *ip = *jp;
      *jp = tmp;
}
```

It is the caller's responsibility to ensure that the pointers point to valid integer objects when the function is called. For example, if `i` and `j` are `int` variables, the following is a valid call to `swap()`:

```
swap(&i, &j);
```

Pointers to pointers to pointers . . . :

A pointer variable can be a pointer to another pointer object. For example:

```
char **ipp;             - ipp is pointer to pointer to int
char **table[50];       - table is an array of pointers to pointers to characters
```

We'll say more about the uses of indirect pointers when we discuss pointer arithmetic.

Pointer arithmetic

In C, one can do arithmetic on pointers in much the same way that one does "address arithmetic" in assembly languages. This unusual feature of C is one reason why C finds wide use in application areas formerly thought suitable only for assembly – such as writing operating systems or device drivers.

One can write expressions of the form

pointer + *offset*
or *pointer* – *offset*

where *pointer* points into an array and *offset* is an integer. One can also increment and decrement pointers into arrays using `++, --,` and *assignment* operators. It must be emphasized that these arithmetic operations are defined *only* for pointers which point to array elements.

Example: The declarations

```
int a[50], *p;
```

define `a` to be an integer array and `p` to be a pointer to an integer. The loop

```
p = a;                    /* point a to the initial element of a */
while ( p < a + 100 ) {
      *p = 0;
      ++p;
}
```

sets all the entries of `a` to 0. One can also do this with a for loop:

```
for ( p = a; p < a + 100; ++p)
      *p = 0;
```

The above example illustrates comparison of pointers: the relational operator < can be used between two pointers provided they both point into the same array. The result is `1` (true) if the first operand points to an earlier element of the array than the second operand, and `0` (false) otherwise. Definitions of the other comparison operators (`<=, >, >=, ==, !=`) on pointers are similar.

WARNING: The effect of doing pointer arithmetic is *undefined* if the pointers involved do not point into an array, or the result of any arithmetic operation points outside the bounds of the array. It is in general the **programmer's responsibility** to guard against undefined operations of this kind, as C does not generally detect such errors either at compile time or run time.

The null pointer.

The symbol `0`, when used in a context where a pointer is expected, has a special meaning: the so-called *null pointer*, a pointer value which is guaranteed not to point to any object. (Pascal programmers will recognize this as being the same concept as `nil` in Pascal.) For clarity, the symbol `NULL` is defined in `stdio.h` to be equivalent to `0`.

The null pointer is useful as a sentinel to mark the end of a list of pointers.

Arrays of pointers and pointers to pointers

Arrays of pointers and indirect pointers (pointers to pointers) have many uses.
A typical one involves building and manipulating tables of variable-length character string.

Example: Consider the declarations

```
char *strtab[50]; /* array of 50 character pointers */
char **stp;       /* used to point into strtab[] */
```

The statements

```
strtab[0] = "hello";
```

```
strtab[1] = "goodbye";
strtab[2] = "why";
strtab[3] = "never";
strtab[4] = NULL;
```

store pointers to some null-terminated strings in the first four elements of the array, and the null pointer in the fifth. We can now use the pointer `stp` to scan the strings. For example, the following will print the strings on the standard output, all on one line, each string followed by a space character:

```
stp = strtab;
while (*stp != NULL) {
      printf("%s ", *stp);
      ++stp;
}
```

One could also accomplish the same thing using an integer index `i` and a *for*-loop:

```
for (i = 0; strtab[i] != NULL; ++i)
      printf("%s ", strtab[i]);
```

In order for the comparisons with `NULL` to successfully terminate the above loops, it is necessary to have explicitly stored `NULL` in the array previously. You ***can not assume*** that an uninitialized pointer is equal to `NULL`.

14. The preprocessor; `#include`; `#define`;

Before actual compilation, a C source file is run through a ***preprocessor*** that processes special directives. A preprocessor directive is a line that begins with the character # followed by an identifier that specifies the type of director. (Many compilers require that the # appear in column 1, while others allow it to be indented as long as it is preceded only by blanks. To be safe, always put the # in column 1.)

The preprocessor makes certain *textual replacements* in the source code before passing it to the compiler.

The most frequently used preprocessor directives are `#define` and `#include`. The first is used to define symbolic constants and macros; the second to include other files (typically so-called "header files") as if they were part of the source file.

`#define`:
This directive has the form

`#define` *identifier sequence-of-tokens*

or

`#define` *identifier(parameter-list) sequence-of-tokens*

The effect is to replace every occurrence of the *identifier* in the source by the sequence of *sequence-of-tokens* following. The most common use is to introduce symbolic names for constants:

```
#define PI              3.1415926
#define LISTMAX         1000
#define PROMPT          "What next? "
#define ShowPrompt(msg) printf("%s", msg)
```

The preprocessor will then literally substitute the definition for each occurrence of the symbol, e.g., in the code fragments:

```
int list[LISTMAX];            /* declare array of size 1000 */

for (i = 0; i < LISTMAX; ++i)
      printf("%d\n", list[i]);      /* print whole array */

area = PI * radius * radius;  /*compute circle area */

ShowPrompt(PROMPT);           /* display "What next? " */
```

`#include`:

The `#include` directive causes another file (usually a text file containing C source code) to be inserted as if its contents actually appeared in the file being compiled. Valid forms are

`#include <`*name-of-file*>

and

`#include "`*name-of-file*"

The symbols delimiting the file name determine where the preprocessor searches for the file. If brackets < and > are used, the preprocessor first looks in the current directory; if it can't find the file there, it looks in a standard system for header files (usually `/usr/include` in Unix systems). On the other hand, if quotes are used to delimit the filename, only the current directory is searched.

By far the most common use of the `#include` directive is to make certain common definitions and declarations available to more than one source file. Typical examples:

```
#include <stdio.h>      /* include defs needed for */
                        /* standard i/o library */

#include <string.h>     /* defs for standard string */
                        /* functions, like strcmp() */

#include <ctype.h>      /* defs of character-testing */
                        /* macros, like isupper() */

#include <signal.h>     /* symbolic definition of Unix */
                        /* signal values, such as SIGINT, */
                        /* SIGKILL, etc. */

#include "mystuff.h"    /* do this for your own common */
                        /* definitions and declarations */
```

And so on – there are *many* standard header files in a Unix environment. Typically, `#include` directives appear at the beginning of a source file that need to access the facilities declared and defined in the respective header files.

15. Naming conventions.

Although usage is not completely standard, it is reasonably widespread practice to adhere to the following case conventions in making up names for things in programs:

Variables, arrays, structures:	all lower case
#define constants	all upper case
#define macros with parameters:	capitalize mixed upper/lower case

We have followed these conventions in all of the examples in these notes. We stress, however, that they are only conventions and not requirements of the language.

Compiling, Linking, Executing under Unix

In a Unix environment, one creates executable programs from C source by compiling and linking. The standard command for accomplishing this is "`cc`". Arguments to `cc` may include source files to be compiled, object files to be linked, and additional options that control compilation or linking.

An alternative to `cc` on many systems is `gcc`, the Free Software Foundation or "GNU" C compiler. It is often to be preferred when it is available, one reason for this being that `gcc` accepts full ANSI C, whereas many versions of `cc` do not.

The `cc` and `gcc` commands are actually front ends that run several programs – the several passes of the compiler, the assembler to produce object files and the linker to merge object files and runtime libraries into a single executable file. Once an executable file has been produced, it can usually be executed simply by typing its name followed by any arguments it might require.

The name of a C source file in general *must* end in "`.c`" for `cc` or `gcc` to recognize it. These commands will also accept object files with names ending in "`.o`" and run the linker to produce an executable file.

Examples

(1) `cc fubar.c`

This produces an executable file named (for obscure historical reasons) `a.out`. During compilation, an object file was also produced, but this is normally deleted before `cc` terminates.

(2) `cc fubar.c -o fubar`

This does the same thing as Example (1), except that the `-o` flag causes the executable filename to be the following argument, in this case `fubar`, rather than `a.out`.

(3) `cc -c fubar.c`

The `-c` flag instructs the compiler to produce an object file but to suppress execution of the linker, so that an execcutable file is not produced. In this example, the compiler produces (and does not delete) an object file named `fubar.o`.

(4) `cc fubar.c -lm`

The `-l` flag (lower case "`ell`", not the digit "`1`"), followed immediately by a library names, causes the designated object library to be searched for definitions of functions and external data not defined in the source files. In this case "`m`" denotes the math function library, which contains definitions of routines such as `sqrt()`.

(5) `cc f1.o f2.o f3.c f4.c -o prog`

This causes source files `f3.c` and `f4.c` to be compiled, then linked with previously compiled object files `f1.o` and `f2.o` to produce the executable program `prog`.

(6) `cc -S fubar.c`

This produces an assemby language translation, stored in the file `fubar.s`, useful if you want to examine the code produced by the compiler.

Epilogue

version 0.0 John Remmers' original notes
very lightly edited (spelling, consistent style)
Table of Contents

Table of Contents