Applied C and C++ Programming

Alice E. Fischer
David W. Eggert
University of New Haven

Michael J. Fischer
Yale University

August 2018

Appendix A

The ASCII Code

The characters of the 7-bit ASCII code are listed, in order. The first two columns of each group give the code in base 10 and as a hexadecimal literal. The third column gives the printed form (if any) or a description of the character. Last is the escape code for the character, if one exists.

0	0x00	null \0	32	0x20	space	64	0x40	@	96	0x60	‘
1	0x01		33	0x21	!	65	0x41	A	97	0x61	a
2	0x02		34	0x22	", \"	66	0x42	B	98	0x62	b
3	0x03		35	0x23	#	67	0x43	C	99	0x63	c
4	0x04		36	0x24	$	68	0x44	D	100	0x64	d
5	0x05		37	0x25	%	69	0x45	E	101	0x65	e
6	0x06		38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	bell \a	39	0x27	’, \’	71	0x47	G	103	0x67	g
8	0x08	backspace \b	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	tab \t	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0A	linefeed \n	42	0x2A	*	74	0x4A	J	106	0x6A	j
11	0x0B	vertical tab \v	43	0x2B	+	75	0x4B	K	107	0x6B	k
12	0x0C	formfeed \f	44	0x2C	,	76	0x4C	L	108	0x6C	l
13	0x0D	carriage return \r	45	0x2D	-	77	0x4D	M	109	0x6D	m
14	0x0E		46	0x2E	.	78	0x4E	N	110	0x6E	n
15	0x0F		47	0x2F	/	79	0x4F	O	111	0x6F	o
16	0x10		48	0x30	0	80	0x50	P	112	0x70	p
17	0x11		49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12		50	0x32	2	82	0x52	R	114	0x72	r
19	0x13		51	0x33	3	83	0x53	S	115	0x73	s
20	0x14		52	0x34	4	84	0x54	T	116	0x74	t
21	0x15		53	0x35	5	85	0x55	U	117	0x75	u
22	0x16		54	0x36	6	86	0x56	V	118	0x76	v
23	0x17		55	0x37	7	87	0x57	W	119	0x77	w
24	0x18		56	0x38	8	88	0x58	X	120	0x78	x
25	0x19		57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1A		58	0x3A	:	90	0x5A	Z	122	0x7A	z
27	0x1B	escape \\	59	0x3B	;	91	0x5B	[	123	0x7B	{
28	0x1C		60	0x3C	<	92	0x5C	\	124	0x7C	|
29	0x1D		61	0x3D	=	93	0x5D	]	125	0x7D	}
30	0x1E		62	0x3E	>	94	0x5E	^	126	0x7E	~
31	0x1F		63	0x3F	?	95	0x5F	_	127	0x7F	delete

Appendix B

The Precedence of Operators in C

Arity	Operator	Meaning	Precedence	Associativity
	a[k]	Subscript	17	left to right
	fname(arg list)	Function call	17	"
	.	Struct part selection	17	"
	->	Selection using pointer	17	"
Unary	postfix ++, --	Postincrement k++, decrement k--	16	left to right
"	prefix ++, --	Preincrement ++k, decrement --k	15	right to left
"	sizeof	# of bytes in object	15	"
"	~	Bitwise complement	15	"
"	!	Logical NOT	15	"
"	+	Unary plus	15	"
"	-	Negate	15	"
"	&	Address of	15	"
"	*	Pointer dereference	15	"
"	(typename)	Type cast	14	"
Binary	*	Multiply	13	left to right
"	/	Divide	13	"
"	%	Mod	13	"
"	+	Add	12	"
"	-	Subtract	12	"
"	<<	Left shift	11	"
"	>>	Right shift	11	"
"	<	Less than	10	"
"	>	Greater than	10	"
"	<=	Less than or equal to	10	"
"	>=	Greater than or equal to	10	"
"	==	Is equal to	9	"
"	!=	Is not equal to	9	"
"	&	Bitwise AND	8	"
"	^	Bitwise exclusive OR	7	"
"	|	Bitwise OR	6	"
"	&&	Logical AND	5	"
"	||	Logical OR	4	"
Ternary	...?...:...	Conditional expression	3	right to left
Binary	=	Assignment	2	"
"	+= -=	Add or subtract and store back	2	"
"	*= /= %=	Times, divide, or mod and store	2	"
"	&= ^= |=	Bitwise operator and assignment	2	"
"	<<= >>=	Shift and store back	2	"
"	,	Left-side-first sequence	1	left to right

Figure B.1. The precedence of operators in C.

Appendix C

Keywords

C.1 Preprocessor Commands

The commands in the first group are presented in this text. The other commands are beyond its scope.

- Basic: `#include`, `#define`, `#ifndef`, `#endif`.
- Advanced: `#if`, `#ifdef`, `#elif`, `#else`, `defined()`, `#undef`, `#error`, `#line`, `#pragma`.
- Advanced macro operators: `#` (stringize), `##` (tokenize).

C.2 Control Words

These words control the order of execution of program blocks.

- Functions: `main`, `return`.
- Conditionals: `if`, `else`, `switch`, `case`, `default`.
- Loops: `while`, `do`, `for`.
- Transfer of control: `break`, `continue`, `goto`.

C.3 Types and Declarations

- Integer types: `long`, `int`, `short`, `char`, `signed`, `unsigned`.
- Real types: `double`, `float`, `long double`.
- An unknown or generic type: `void`.
- Type qualifiers: `const`, `volatile`.
- Storage class: `auto`, `static`, `extern`, `register`.
- Type operator: `sizeof`.
- To create new type names: `typedef`.
- To define new type descriptions: `struct`, `enum`, `union`.

C.4 Additional C++ Reserved Words

The following are reserved words in C++ but not in C. C programmers should either avoid using them or be careful to use them in ways that are consistent with their meaning in C++.

- Classes: `class`, `friend`, `this`, `private`, `protected`, `public`, `template`.
- Functions and operators: `inline`, `virtual`, `operator`.
- Kinds of casts: `reinterpret_cast`, `static_cast`, `const_cast`, `dynamic_cast`.
- Boolean type: `bool`, `true`, `false`.
- Exceptions: `try`, `throw`, `catch`.
- Memory allocation: `new`, `delete`.
- Other: `typeid`, `namespace`, `mutable`, `asm`, `using`.

C.5 An Alphabetical List of C and C++ Reserved Words

```
#             catch         goto                static
##            char          if                  static_cast
#define      class         inline              struct
#elif        const         int                 switch
#else        const_cast    long                template
#endif       continue      mutable             this
#error       default       namespace           throw
#if          defined()     new                 true
#ifdef       delete        operator            try
#ifndef      do            private             typedef
#include     double        protected           typeid
#line        else          public              union
#pragma      enum          register            unsigned
#undef       extern        reinterpret_cast    using
asm          false         return              virtual
auto         float         short               void
bool         for           signed              volatile
break        friend        sizeof              while
case
```

Appendix D

Advanced Aspects C Operators

This appendix describes important facts about a few C operators that were omitted in earlier chapters because they were too advanced when related material was covered.

D.1 Assignment Combination Operators

All the assignment-combination operators have the same very low precedence and associate right to left. This means that a series of assignment operators will be parsed and executed right to left, no matter what operators are used. Figure D.1 demonstrates the syntax, precedence, and associativity of the arithmetic combinations.

Notes on Figure D.1. Assignment combinations.

Box: precedence and associativity.

- This long expression shows that all the combination operators have the same precedence and that they are parsed and executed right to left.
- The `+=` is parsed before the `*=` because it is on the right. The fact that `*` alone has higher precedence than `+` alone is not relevant to the combination operators.
- The parse tree for this expression is shown in Figure D.2. Note that assignment-combination operators have two branches connected to a variable. The right branch represents the operand value used in the mathematical operation. The left branch, with the arrowhead, reflects the changing value of the variable due to the assignment action after the calculation is complete.
- The output from this program is

```
Demonstrating Assignment Combinations
Assignment operators associate right to left.
Initial values:
        k = 10 m = 5 n = 64 t = -63
Executing t /= n -= m *= k += 7 gives the values:
        k = 17 m = 85 n = -21 t = 3
```

D.2 More on Lazy Evaluation and Skipping

With lazy evaluation, when we skip, we skip the right operand. This isn't confusing when the right operand is only a simple variable. However, sometimes it is an expression with several operators. For example, look at the parse tree in Figure D.3. The left operand of the `||` operator is the expression `a < 10` and its right operand is `a >= 2 * b && b != 1`. The parse tree makes clear the relationship of operands to operators.

We can use parse trees to visualize the evaluation process. The stem of the tree represents the value of the entire expression. The stem of each subtree represents the value of the parts of the tree above it. To evaluate an expression, we start by writing the initial values of the variables above the expression. However,

We exercise the arithmetic assignment-combination operators. The parse tree for the last expression is shown in Figure D.2. Note that these operators all have the same precedence and they associate right to left.

```
#include <stdio.h>
int main( void )
{
    double k = 10.0;       double m = 5.0;
    double n = 64.0;       double t = -63.0;

    puts( "\n Demonstrating Assignment Combinations" );
    puts( " Assignment operators associate right to left.\n"
          " Initial values:  " );
    printf( "\t k = %.2g m = %.2g n = %.2g t = %.2g \n", k, m, n, t );
    t /= n -= m *= k += 7;
    puts( " Executing t /= n -= m *= k += 7 gives the values:  " );
    printf( "\t k = %.2g m = %.2g n = %.2g t = %.2g \n\n", k, m, n, t );

}
```

Figure D.1. Assignment combinations.

start the evaluation process *at the stem of the tree.* Starting at the top (the leaves) in C will give the wrong answer in many cases. As each operator is evaluated, write the answer on the stem under that operator. Figure D.3 illustrates the evaluation process.

The tree, as a whole, represents an assignment expression because the operator corresponding to the tree's stem is an `=`. Everything after the `=` in this assignment is a logical expression because the next operator, proceeding up the tree, is a logical operator. This is where we start considering the rules for lazy evaluation.

Skip the right operand. Evaluation of a logical expression proceeds left to right, skipping some subexpressions along the way. We evaluate the left operand of the leftmost logical operator first. Depending on the result, we evaluate or skip the right operand of that expression. In the example, we compute `a < 10`; if that is `true`, we skip the rest of the expression, including the `&&` operator. This case is illustrated in the upper diagram in Figure D.3. The long double bars across the right branch of the OR operator are called *pruning marks*; they are used to "cut off" the part of the tree that is not evaluated and show, graphically, where skipping begins.

A natural comment at this point is, "But I thought that `&&` should be executed first because it has higher precedence." Although precedence controls the construction of the parse tree, precedence simply is not considered when the tree is evaluated. Because of its higher precedence, the `&&` operator "captured" the operand `a >= 2 * b`. However, logical expressions are evaluated left to right, so evaluation will start with the `||` because it is to the left of the `&&`. Only if the left operand of the `||` operation is `false`, as in the lower diagram of Figure D.3, will evaluation continue with the `&&`. In this case the left operand of the `&&` is `false`, meaning its right operand can be skipped. Graphically, evaluation starts at the stem of the logical

This is the parse tree and evaluation for the last expression in Figure D.1

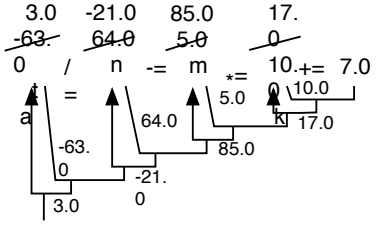
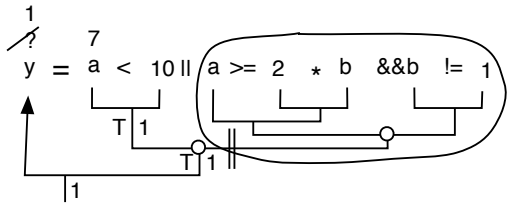


Figure D.2. A parse tree for assignment combinations.

We evaluate the expression `y = a < 10 || a >= 2 * b && b != 1` twice. Note the "pruning marks" on the tree and the curved lines around the parts of the expression that are skipped.

1. Evaluation with the values `a = 7`, `b =` anything: The `||` causes skipping



2. Evaluation with the values `a = 17`, `b = 20`: The `&&` causes skipping

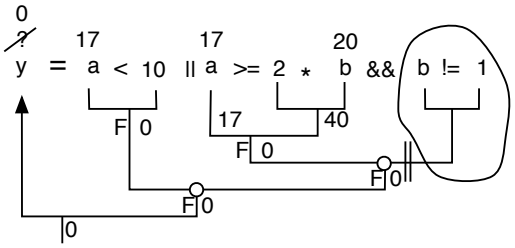


Figure D.3. Lazy evaluation.

expression and proceeds upward, doing the left side of each logical operator first and skipping the right side where possible.

The whole right operand. When skipping happens, *all* the work on the right subtree is skipped, no matter how complicated that portion of the expression is and no matter what operators are there. In our first example, the left operand (which we evaluated) was a simple comparison but the right operand was long and complex. As soon as we found that `a < 10` was `true`, we put the answer `1` on the tree stem under the `||`, skipped *all* the work on the right side of the operator, and stored the value 1 in `y`. In Figure D.4,

We evaluate the expression `y = a < 0 || a++ < b` for `a = -3`. Note that the increment operation in the right operand of `||` does not happen because the left operand is `true`.

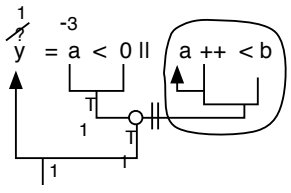


Figure D.4. Skip the whole operand.

We evaluate the expression `y = a < 0 || a > b && b > c || b > 10` for `a = 3` and `b = 17`. Note that skipping affects only the right operand of the `&&`; the parts of the expression not on this subtree are not skipped.

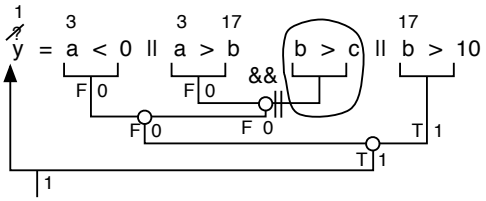


Figure D.5. And nothing but the operand.

we also skip the rest of the computation after the comparison. However, in general, we might skip only a portion of the remaining expression.

Sometimes, lazy evaluation can substantially improve the efficiency of a program. But while improving efficiency is nice, a much more important use for skipping is to avoid evaluating parts of an expression that would cause machine crashes or other kinds of trouble. For example, assume we wish to divide a number by `x`, compare the answer to a minimum value, and do an error procedure if the answer is less than the minimum. But it is possible for `x` to be 0 and that must be checked. We can avoid a division-by-0 error and do the computation and comparison in one expression by using a *guard* before the division. A guard expression consists of a test for the error-causing condition followed by the `&&` operator. The entire C expression would be

```
if (x != 0 && total / x < minimum) do_error();
```

Guarded expressions are useful in a wide variety of situations.

And nothing but the right operand. One common fallacy about C is that, once skipping starts, everything in the expression to the right is skipped. This is simply not true; the skipping involves only the right operand of the particular operator that triggered the skip. If several logical operators are in the expression, we might evaluate branches at the beginning and end but skip a part in the middle. This is illustrated by Figure D.5. In all cases, you must look at the parse tree to see what will be skipped.

D.2.1 Evaluation Order and Side-Effect Operators

A frequent cause of confusion is the relationship between logical operators, lazy evaluation, and operators such as `++` that have side effects. When used in isolation, as at the end of Figure 4.23, the increment and decrement operators are convenient and relatively free of complication. When side-effect operators are used in long, complex expressions, they create the kind of complexity that fosters errors. If a side-effect operator is used in the middle of a logical expression, it may be executed sometimes but skipped at other times. If the operator is on the skipped subtree, as in Figure D.4, that operation is not performed and the value in memory is not changed. This may be useful in a program, but it also is complex and should be avoided by beginners. Just remember, the high precedence of the increment or decrement operator affects only the shape of the parse tree; it *does not* cause the increment operation to be evaluated before the logical operator.

A second problem with side-effect operators relates to the order in which the parts of an expression are evaluated. Recall that evaluation order has nothing to do with precedence order. We have stated that logical operators are executed left to right. This also is true of two other kinds of sequencing operators: the conditional operator `?...:` and the comma, defined in the next section. Therefore, it may be a surprise to learn that C is permitted to evaluate most other operators right-side first or left-side first, or inconsistently,

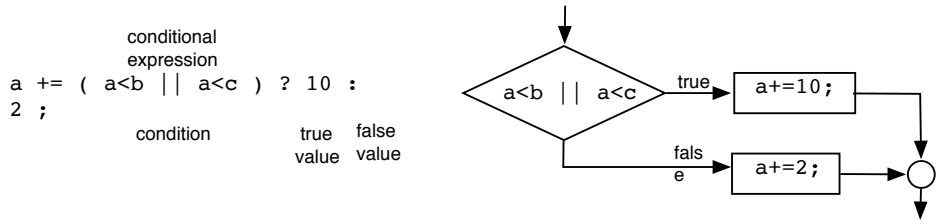


Figure D.6. A flowchart for the conditional operator.

whichever is convenient for the compiler. Technically, we say that the evaluation order for nonsequencing operators is *undefined*. This flexibility in evaluation order permits an optimizing compiler to produce faster code.

However, while the undefined evaluation order usually does not cause problems, it does lead directly to one important warning: If an expression contains a side-effect operator that changes the value of a variable V, *do not use* V anywhere else in the expression. The side effect could happen either before or after the value of V is used elsewhere in the expression and the outcome is unpredictable. Writing the expression in the order we want it executed won't help; the C compiler does not have to conform to our order.

D.3 The Conditional Operator

There is only one ternary operator in C, the *conditional operator*. It has three operands and two operator symbols (`?` and `:`). The conditional operator does almost the same thing as an `if...else` with one major difference: `if` is a statement, it has no value; but `?...:` is an operator and calculates and returns a value like any other operator.

Evaluating a Conditional Operator. We can use either a flow diagram or a parse tree to diagram the structure and meaning of a conditional operator; each kind of diagram is helpful in some ways. A flow diagram (as in Figure D.6) depicts the order in which actions happen and shows us the similarity between a conditional operator and an `if` statement, while a parse tree (Figure D.7) shows us how the value produced by the conditional operator relates to the surrounding expression.

Making a flowchart for a conditional operator is somewhat problematical since flowcharts are for statements and a conditional operator is only part of a statement. To represent the sequence of actions as we do for the `if` statement, we have to include the rest of whatever statement contains the `?...:` in the `true` and `false` boxes. Figure D.6 shows how this can be done. The condition of the `?...:` is the operand to the left of the `?`. This condition is written in the diamond-shaped box of the flowchart. The `true` clause is written between the `?` and the `:`. It is written, with the assignment operator on the left, in the `true` box. Similarly, the `false` clause is written, with another copy of the assignment operator, in the `false` box.

Looking at the flowchart, we can see that the condition of a `?...:` always is evaluated first. Then, based on the outcome, either the `true` clause or the `false` clause is evaluated and produces a result. This result then is used in the expression that surrounds the `?...:`, in this case, a `+=` statement.

Parsing a Conditional Operator. Since `?...:` can be included in the middle of an expression, it is helpful to know how to draw a parse tree for it. We diagram it with three upright parts (rather than two) and a stem as shown in Figure D.7. Note that `?...:` has very low precedence (with precedence = 3, it falls just above assignment) so it usually can be used without putting parentheses around its operands. However, parentheses are often needed around the entire unit. This three-armed treelet works naturally into the surrounding expression. The main drawback of this kind of diagram is that it does not show the sequence of execution as well as a flowchart.

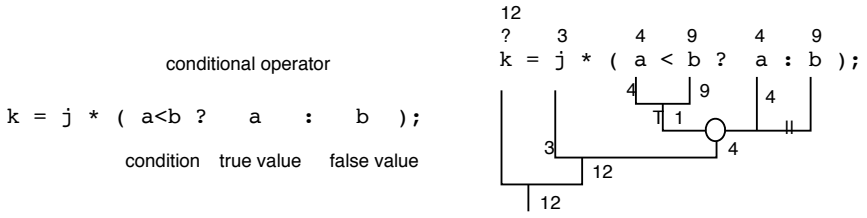


Figure D.7. A tree for the conditional operator.

Parsing a nested set of conditional operators is not hard. First parse the higher-precedence parts of the expression. Then start at the right (since the conditional operator associates from right to left) and look for the pattern:
⟨treelet⟩ `?` ⟨treelet⟩ `:` ⟨treelet⟩. Wherever you find three consecutive treelets separated only by a `?` and a `:`, bracket them together and draw a stem under the `?`. Even if the expression is complex and contains more than one `?`, this method always works if the expression is correct. If the expression is incorrect, we will find mismatched or misnested elements.

The sequence in which the parts of a conditional expression are evaluated or skipped is critical. We convey this sequencing in a parse tree by placing a sequence-point circle under the `?`. This indicates that the condition (the leftmost operand) must be evaluated first. The outcome of the condition selects either the `true` clause or the `false` clause and skips over the other. The skipping is conveyed by writing "prune marks" on either the middle branch or the rightmost branch of the parse tree, whichever is skipped. The expression on the remaining branch then is evaluated, and its value is written on the stem of the `?...:` bracket and propagated to the surrounding expression. Note that, even though evaluation *starts* by calculating a `true` or a `false` value, the value of the entire conditional operator, in general, will not be `true` or `false`.

The sequence point under the `?` has one other important effect. If the condition contains any postincrement operators, the increments must be done before evaluating the `true` clause or the `false` clause. Therefore, it is "safe" to use postincrement in a condition.

Finally, remember that evaluation order is not the same as precedence order. For example, suppose we are evaluating a conditional operator that prunes off a treelet containing some increment operators. Even though increment has much higher precedence than the conditional operator, the increment operations will not happen. This is why we must evaluate parse trees starting at the root, not the leaves. However, pruning does not change the parse tree—it merely skips part of it. We must not erase the parts that are skipped or try to get them out of the way by restructuring the whole diagram.

D.4 The Comma Operator

The comma operator, `,`, in C is used to write two expressions in a context that normally allows for only one. To be useful, the first of these expressions must have a side effect. For example, the following loop, which sums the first `n` values in the array named `data`, uses the comma operator to initialize two variables, the loop counter and the accumulator:

```
for (sum=0, k=n-1; k>=0; --k) sum += data[k];
```

The comma operator acts much like a semicolon with two important exceptions:

1. The program units before and after a comma must be non-`void` expressions. The units before and after a semicolon can be either statements or expressions.
2. When we write a semicolon after an expression, it ends the expression and the entire unit becomes a statement. When a comma is used instead, it does not end the expression but joins it to the expression that follows to form a larger expression.

3. The value of the right operand of the comma is propagated to the enclosing expression and may be used in further computations.

D.5 Summary

A number of nonintuitive aspects of C semantics have arisen in this appendix that are responsible for many programming errors. A programmer needs to be aware of these issues in order to use the language appropriately:

- *Use lazy evaluation.* The left operand of a logical operator always is evaluated, but the right operand is skipped whenever possible. Skipping happens when the value of the left operand is enough to determine the value of the expression.
- *Use guarded expressions.* Because of lazy evaluation, we can write compound conditionals in which the left side acts as a "guard expression" to check for and trap conditions that would cause the right side to crash. The right side is skipped if the guard expression detects a "fatal" condition.
- *Evaluation order is not the same as precedence order.* High-precedence operators are parsed first but they are not evaluated first and they may not be evaluated at all in a logical expression. Logical expressions are executed left to right with possible skipping. An operator on a part of a parse tree that is skipped will not be evaluated. Therefore, an increment operator may remain unevaluated even though it has very high precedence and the precedence of the logical operators is low.
- *Evaluation order is indeterminate.* The only operators that are guaranteed to be evaluated left to right are logical-AND, logical-OR, comma, and the conditional operator. With the other binary operators, either the left side or the right side, may be evaluated first.
- *Keep side-effect operators isolated.* If you use an increment or decrement operator on a variable, V, you should not use V anywhere else in the same expression because the order of evaluation of terms in most expressions is indeterminate. If you use V again, you cannot predict whether the value of V will be changed before or after V is incremented.
- Figure D.8 summarizes the complex aspects of the C operators with side effects and sequence points.

Group	Operators	Complication
Assignment combinations	`+=`, etc.	These have low precedence and strict right-to-left parsing, no matter which combination is used.
Preincrement and predecrement	`++, --`	If we use a side-effect operator, we don't use the same variable again in the same expression.
Postincrement and postdecrement	`++, --`	Remember that the postfix operators return one value for further use in the expression and leave a different value in memory. Also, don't use the same variable again in the expression.
Logical	`&&, ||, !`	Remember that negative integers are considered `true` values, and separate and different rules apply for precedence order and evaluation order.
	`&&`	Use lazy evaluation when first operand is `false`.
	`||`	Use lazy evaluation when first operand is `true`.
Conditional	`...?...:...`	The expressions both before and after the colon must produce values and those values must be the same type.
Comma	`,`	This is rarely used except in `for` loops.

Figure D.8. Complications in use of side-effect operators.

Appendix E

Dynamic Allocation in C

E.0.1 Mass Memory Allocation

When a programmer cannot predict the amount of memory that will be needed by a program, the `malloc()` function can be used at run time to allocate an array of bytes of the required size. Its prototype is

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

where `size_t` is an unsigned integer type used by the local system to store the sizes of objects. Frequently, `size_t` is defined by `typedef` to be the same as `unsigned int` or `unsigned long`. The value returned by `malloc()` is a pointer to an array of bytes. For example, to allocate a single object and an array of objects of type `LumberT`, from Chapter 13, we would write:

These functions are defined in the C standard library whose header is `stdlib.h`.

Prototype	Action
`void* malloc( size_t sz );`	Mass memory allocation. Return a pointer to an uninitialized block of memory of the specified size, `sz` bytes.
`void* calloc( size_t n, size_t sz );`	Allocate and clear memory. Return a pointer to an array of memory locations that have been cleared to 0 bits. The array has `n` slots, each of size `sz` bytes.
`void  free( void* pt );`	Recycle a memory block. Return to the operating system the block of memory that starts at the address stored in `pt`. A block should be freed after it no longer is needed by the program.
`void* realloc( void* pt, size_t sz );`	Mass memory reallocation. Given a pointer, `pt`, to a memory block that was previously allocated by `malloc()` or `calloc()`, and given a new number of bytes, `sz`, that is different from the current size of that block, resize the block to the new length. If the new block cannot start at the same location as the old one, this will involve copying the entire contents of the old block to the new one.

Figure E.1. Dynamic memory allocation functions.

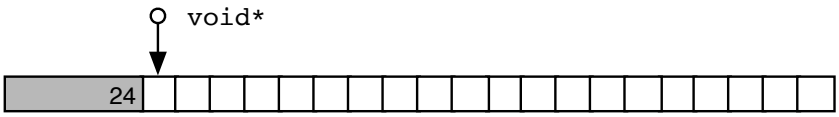
```
lumber_t * newBoard;
lumber_t * newArray;

newBoard = malloc( sizeof(lumber_t) );
newArray = malloc( 20 * sizeof(lumber_t) );
if (newBoard == NULL ||  newArray == NULL) {
    fprintf( stderr, "Insufficient memory; aborting program\n" );
    exit( 1 );
}
```

Several C principles and techniques are combined in this typical code verbatim. We will now introduce and explain these principles one at a time. The simplest way to call `malloc()` is with a literal constant, thus:

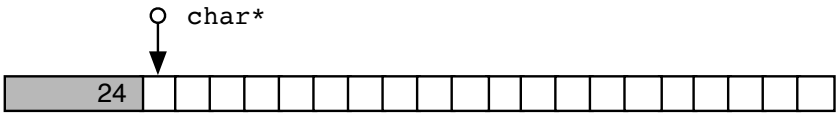
```
malloc( 20 )
```

This call allocates a memory area like the one diagrammed here:

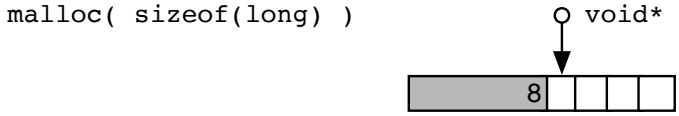


This memory is not initialized and still contains whatever data happened to be left over from a previous program. The gray area in the diagram represents additional bytes that the C system sets aside to store the total size of the allocated block (the size of a `size_t` value plus the size of the white area). The importance of these bytes becomes clear when we discuss `free()` and `realloc()`.

The type `void*` has not been discussed yet; it is a generic pointer type, which basically means "a pointer to something, but we don't know what." It is used because `malloc()` must be able to allocate memory for any type of object, and the function's prototype must specify a return type compatible with all kinds of pointers. Before the `void` pointer that is returned can be used, it must be either **explicitly cast**[1] to a specific pointer type, as shown next, or implicitly cast by storing it in a pointer variable, as shown in Figure 16.19.



Normally, `malloc()` is used to allocate space for a single object or an array of objects of some known type. Since the number of bytes occupied by a type can vary from one implementation to another, we usually do not call `malloc()` with a literal number as an argument. Instead, we use the `sizeof` operation to supply the correct size of the desired type on the local system:

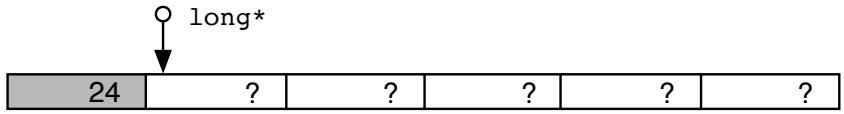


Note that the return type is still `void*`, even though we use the type name `long` in the expression. The `sizeof` expression *inside* the argument list does not affect the type of the pointer that is returned.

To allocate an array of objects, simply multiply the size of the base type by the number of array slots. The next diagram illustrates this common usage, along with a cast that converts the `void*` value to a pointer with the correct base type:

[1] The cast is not necessary, even to avoid warning messages, in ISO C. However, it was necessary in older versions of C and is a style that many older programmers follow.

```
(long*) malloc( 5 * sizeof(long) )
```

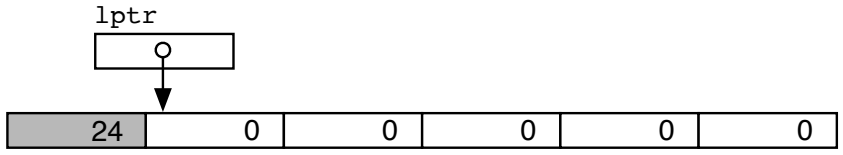


Although it is uncommon on modern systems, `malloc()` will fail if there is not enough available memory to satisfy the request. In this case, its return value is `NULL`. It is good programming practice to include a test for this condition and abort the program if it occurs, because no further meaningful processing can be done. An example of this is shown in Figure 16.19.

E.0.2 Cleared Memory Allocation

The `calloc()` function allocates an array of memory and clears all of it to 0 bits. It has two parameters, the length of the array and the size of one array element. No cast operator is needed here because the return value is stored immediately in a pointer variable of the correct type. A typical call and its results are

```
long* lptr;
lptr = calloc( 5, sizeof(long) );
```



Even though `malloc()` and `calloc()` have different numbers of parameters, they essentially do the same thing (except for initialization) and return the same type of result. Like `malloc()`, `calloc()` returns a `void*` to the first memory address in the allocated area or returns `NULL` if not enough memory is available to allocate a block of the specified size.

E.0.3 Freeing Dynamic Memory

In many applications, memory requirements grow and shrink repeatedly during execution. A program may request several chunks of memory to accommodate the data during one phase then, after using the memory, have no future need for it. Memory use and, sometimes, execution speed are made more efficient by recycling memory; that is, returning it to the system to be reused for some other purpose. Dynamically allocated memory can be recycled by calling `free()`, which uses the number of bytes in the gray area at the beginning of each allocated block. This function returns the block of memory to the system's memory manager, which adds it to the supply of available storage and eventually reassigns it when the program again calls `malloc()` or `calloc()`. The use of `malloc()` and `free()` are illustrated by the simulation program beginning in Figure **??**.

While each program is responsible for recycling its own obsolete memory blocks, a few warnings are in order. A block should be freed only once; a second attempt to free the same block is an error. Similarly, we use `free()` only to recycle memory areas created by `malloc()` or `calloc()`. Its use with a pointer to any other memory area is an error. Another common mistake, described next, is to attempt to use a block after it has been freed. These are serious errors that cannot be detected by the compiler and may cause a variety of unpredictable results at run time.

A **dangling pointer** is one whose referent has been reclaimed by the system. Any attempt to use a dangling pointer is wrong. Typically, this happens because multiple pointers often point at the same memory block. When a block is first allocated, only one pointer points to it. However, that pointer might be copied

several times as it is passed into and out of functions and stored in data structures. If one copy of the pointer is used to free the memory block, all other copies of that pointer become dangling references. A dangling reference may seem to work at first, until the block is reallocated for another purpose. After that, two parts of the program will simultaneously try to use the same storage and the contents of that location become unpredictable.

Memory leaks. If you do not explicitly free the dynamically allocated memory, it will be returned to the system's memory manager when the program completes. So, forgetting to perform a `free()` operation is not as damaging as freeing the same block twice.

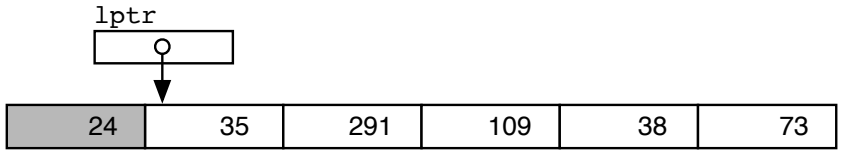
However, some programs are intended to run for hours or days at a time without being restarted. In such programs, it is much more important to keep free all dynamic memory when its useful lifetime is over. The term *memory leak* us used to describe memory that should have been recycled but was not. Memory leaks in major commercial software systems are common. The symptoms are a gradual slowdown in system performance and, eventually, a system "lock up" or crash.

Thus, it is important for programmers to learn how to free memory that is no longer is needed, and it is always good programming style to do so, especially when the memory is used for a short time by only one function. Functions often are reused in a new context, they always should clean up after themselves.

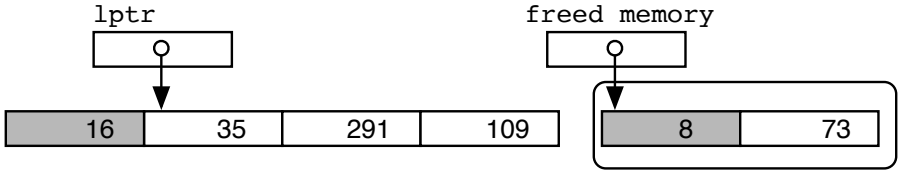
E.0.4 Resizing an Array

By using the `malloc()` function, we can defer the decision about the length of an array until run time. However, array space still must be allocated before the data are read and, perhaps, before we know how many data items actually exist. Using `malloc()` to create an array makes a program more flexible, but it still cannot accommodate an amount of data greater than expected. Fortunately, the C library provides an additional function to solve the problem of too little space for the data. The function that **resizes the data array** is called `realloc()`. When given a pointer to a memory block that was created by `malloc()` or `calloc()`, it will reallocate the array, making it either larger or smaller, according to the newly requested size.

Making an array smaller causes no physical or logical difficulties. The excess space is taken off the end of the original area and returned to the system for recycling. The length count that is kept in the gray area at the head of the block is adjusted appropriately. For example, assume we have the following memory block before reallocation:

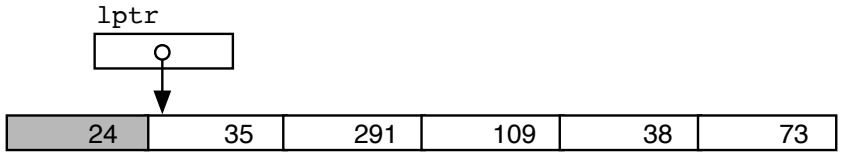


Then we give the reallocation command `lptr = realloc( lptr, 3 * sizeof(long) );`
After reallocation, the picture becomes

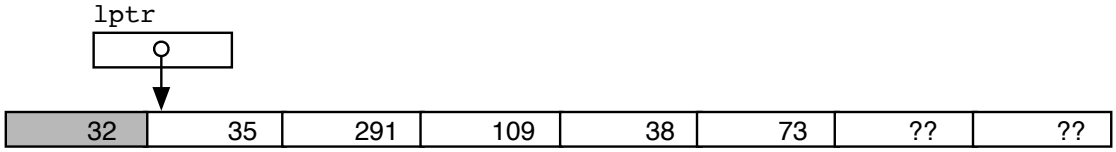


Making an array longer causes a problem because the space at the end of the array already may be in use for some other purpose. The C system keeps track of the length of every area created by `malloc()` or `calloc()` and also knows what storage is free. If there is enough empty space after the end of an array, that storage area easily and efficiently can be added onto the end. This is what `realloc()` does, if possible. When the space after the array is unavailable, `realloc()` allocates a new area that *is* large enough, then copies the data from the old area into the new one, and finally frees the old area.

For example, before reallocation, we have



After the reallocation command `lptr = realloc( lptr, 7 * sizeof(long) );` we have



Therefore, `realloc()` always succeeds unless the memory is full, but the reallocated array might start at a new memory address. The function returns the starting address of the current memory block, whether or not it has been changed. The program must store this address so it can access the storage.

Copying an entire array of information from one memory block to another is a costly operation and should be avoided if possible. It would not be a good idea to reallocate an array many times in a row, adding only a small number of slots each time. For this reason, successive calls on `realloc()` normally double the length of the array or at least add a substantial number of slots. An application of `realloc()` is given in the simulation program in Section 17.4.

Finally, `realloc()` is inappropriate for applications that have many pointers pointing into the dynamic array. Since the memory location of an array might change when it is reallocated, any pointers previously set to point at parts of the array are left dangling. If there are only a few such pointers, new memory addresses sometimes can be computed and the pointers reconnected properly. If not, `realloc()` should not be used.

Appendix F

The Standard C Environment

This appendix contains a list of standard ISO C symbols, `#include` files, and libraries. The libraries that have been used in this text are described by listing prototypes for all functions in the library. Each function is described briefly if it was used in this text or is likely to be useful to a student in the first two years of study. Alternative and traditional functions have not been listed. Readers who need more detailed information about the libraries should consult a standard reference book or the relevant UNIX manual page.

F.1 Built-in Facilities

These symbols are macros that are identified at compile time and replaced by current information, as indicated. If included in a source file, the relevant information will be inserted into that file.

- `__DATE__` is the date on which the program was compiled.
- `__FILE__` is the name of the source file.
- `__LINE__` is the current line number in the source file.
- `__STDC__` is defined if the implementation conforms to the ISO C standard.
- `__TIME__` is the time at which the program was compiled and should remain constant throughout the compilation.

F.2 Standard Files of Constants

We list the standard `#include` files that define the properties of numbers on the local system.

`limits.h`. Defines the maximum and minimum values in each of the standard integer types, as implemented on the local system. The constants defined are

- Number of bits in a character: `CHAR_BIT`.
- Type character: `CHAR_MAX`, `CHAR_MIN`.
- Signed and unsigned characters: `SCHAR_MAX`, `SCHAR_MIN`, `UCHAR_MAX`.
- Signed and unsigned short integers: `SHRT_MAX`, `SHRT_MIN`, `USHRT_MAX`.
- Signed and unsigned integers: `INT_MAX`, `INT_MIN`, `UINT_MAX`.
- Signed and unsigned long integers: `LONG_MAX`, `LONG_MIN`, `ULONG_MAX`.

Name	FLT_constant value	DBL_constant value
RADIX	2	
EPSILON	1.19209290E−07F	2.2204460492503131E−16
DIG	6	15
MANT_DIG	24	53
MIN	1.17549435E−38F	2.2250738585072014E−308
MIN_EXP	−125	−1021
MIN_10_EXP	−37	−307
MAX	3.40282347E+38F	1.7976931348623157E+308
MAX_EXP	128	1024
MAX_10_EXP	38	308

Figure F.1. Minimum values.

`float.h`. Defines the properties of each of the standard floating-point types, as implemented on the local system. The constants defined are

- The value of the radix: `FLT_RADIX`.
- Rounding mode: `FLT_ROUNDS`.
- Minimum x such that $1.0 + x \neq x$: `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`.
- Decimal digits of precision: `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`.
- Number of radix digits in the mantissa: `FLT_MANT_DIG`, `DBL_MANT_DIG`, `LDBL_MANT_DIG`.
- Minimum normalized positive number: `FLT_MIN`, `DBL_MIN`, `LDBL_MIN`.
- Minimum negative exponent for a normalized number: `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.
- Minimum power of 10 in the range of normalized numbers: `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.
- Maximum representable finite number: `FLT_MAX`, `DBL_MAX`, `LDBL_MAX`.
- Maximum exponent for representable finite numbers: `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.
- Maximum power of 10 for representable finite numbers: `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.

IEEE floating-point standard. The minimum values of the C constants that conform to the IEEE standard are listed in Figure G.1.

F.3 The Standard Libraries and `main()`

F.3.1 The Function `main()`

The `main()` function is special in two ways. First, every C program must have a function named `main()`, and that is where execution begins. A portion of a program may be compiled without a `main()` function, but linking will fail unless some other module does contain `main()`.

The other unique property of `main()` is that it has two official prototypes and is frequently used with others. The two standardized prototypes are:

- `int main( void );`
- `int main( int argc, char* argv[] );`

The int return value is intended to return a status code to the system and is needed for interprocess communication in complex applications. However, it is irrelevant for a simple program. In this case, non-standard variants, such as int main(void), can be used and will work properly. Having no return values works because most systems do not rely on a status code being returned, and simple programs have no meaningful status to report.

F.3.2 Characters and Conversions

Header file. <ctype.h>.

Functions.

- int isalnum(int ch);
  Returns true if the value of ch is a digit (0–9) or an alphabetic character (A–Z or a–z). Returns false otherwise.
- int isalpha(int ch);
  Returns true if the value of ch is an alphabetic character (A–Z or a–z). Returns false otherwise.
- int islower(int ch);
  Returns true if the value of ch is a lower-case alphabetic character (a–z). Returns false otherwise.
- int isupper(int ch);
  Returns true if the value of ch is an upper-case alphabetic character (A–Z). Returns false otherwise.
- int isdigit(int ch);
  Returns true if the value of ch is a digit (0–9). Returns false otherwise.
- int isxdigit(int ch);
  Returns true if the value of ch is a hexadecimal digit (0–9, a–f, or A–F). Returns false otherwise.
- int iscntrl(int ch);
  Returns true if the value of ch is a control character (ASCII codes 0–31 and 127). Returns false otherwise. The complementary function for a standard ASCII implementation is isprint().
- int isprint(int ch);
  Returns true if the value of ch is an ASCII character that is not a control character (32–126). Returns false otherwise.
- int isgraph(int ch);
  Returns true if the value of ch is a printing character other than space (33–126). Returns false otherwise.
- int isspace(int ch);
  Returns true if the value of ch is a whitespace character (horizontal tab, carriage return, newline, vertical tab, formfeed, or space). Returns false otherwise.
- int ispunct(int ch);
  Returns true if the value of ch is a printing character but not space or alphanumeric. Returns false otherwise.
- int tolower(int ch);
  If the value of ch is an upper-case character, returns that character converted to lower-case (a–z). Returns the letter unchanged otherwise.
- int toupper(int ch);
  If the value of ch is a lower-case character, returns that character converted to upper-case (A–Z). Returns the letter unchanged otherwise.

F.3.3 Mathematics

Header file. `<math.h>`.

Constant. `HUGE_VAL` is the a special code that represents a value larger than the largest legal floating-point value. On some systems, if printed, it will appear as `infinity`.

Trigonometric functions. These functions all work in units of radians:

- `double sin( double );`
  `double cos( double );`
  `double tan( double );`
  These are the mathematical functions sine, cosine, and tangent.
- `double asin( double );`
  `double acos( double );`
  These functions compute the principal values of the mathematical arc sine and arc cosine functions.
- `double atan( double x );`
  `double atan2( double y, double x );`
  The `atan()` function computes the principal value of the arc tangent of `x`, while `atan2()` computes the principal value of the arc tangent of `y/x`.
- `double sinh( double );`
  `double cosh( double );`
  `double tanh( double );`
  These are the hyperbolic sine, cosine, and tangent functions.

Logarithms and powers.

- `double exp( double x );`
  Computes the exponential function, e^x, where e is the base of the natural logarithms.
- `double log( double x );`
  `double log10( double x );`
  These are the natural logarithm and base-10 logarithm of x.
- `double pow( double x, double y );`
  Computes x^y. It is an error if x is negative and y is not an exact integer or if x is 0 and y is negative or 0.
- `double sqrt( double x );`
  Computes the nonnegative square root of x. It is an error if x is negative.

Manipulating number representations.

- `double ceil( double d );`
  The smallest integral value greater than or equal to `d`.
- `double floor( double d );`
  The largest integral value less than or equal to `d`.
- `double fabs( double d );`
  The absolute value of `d`. Note: `abs()` is defined in `<stdlib.h>`.
- `double fmod( double x, double y );`
  The answer, `f`, is less than `y`, has the same sign as `x`, and `f+y*k` approximately equals `x` for some integer `k`. It may return 0 or be a run-time error if `y` is 0.

- `double frexp( double x, int* nptr );`
  Splits a nonzero x into a fractional part, f, and an exponent, n, such that $|f|$ is between 0.5 and 1.0 and $x = f \times 2^n$. The function's return value is f, and n is returned through the pointer parameter. If x is 0, both values will be 0.
- `double ldexp( double x, int n );`
  The inverse of `frexp()`; it computes and returns the value of $x \times 2^n$.
- `double modf( double x, double* nptr );`
  Splits a nonzero x into a fractional part, f, and an integer part, n, such that $|f|$ is less than 1.0 and $x = f + n$. Both f and n have the same sign as x. The function's return value is f and n is returned through the pointer parameter.

F.3.4 Input and Output

Header file. `<stdio.h>`.

Predefined streams. `stdin, stdout, stderr`.

Constants.

- `EOF` signifies an error or end-of-file during input.
- `NULL` is the zero pointer.
- `FOPEN_MAX` is the number of streams that can be open simultaneously (ISO C only).
- `FILENAME_MAX` is the maximum appropriate length for a file name (ISO C only).

Types. `FILE`, `size_t`, `fpos_t`.

Stream functions.

- `FILE* fopen( const char* filename, const char* mode );`
  `int fclose( FILE* str );`
  For opening and closing programmer-defined streams.
- `int fflush( FILE* str );`
  Sends the contents of the stream buffer to the associated device. It is defined only for output streams.
- `FILE* freopen( const char* filename, const char* mode,`
  `    FILE* str );`
  Reopens the specified stream for the named file in the new mode.
- `int feof( FILE* str );`
  Returns `true` if an attempt has been made to read past the end of the stream `str`. Returns `false` otherwise.
- `int ferror( FILE* str );`
  Returns `true` if an error occurred while reading from or writing to the stream `str`.
- `void clearerr( FILE* str );`
  Resets any error or end-of-file indicators on stream `str`.
- `int rename( const char* oldname, const char* newname );`
  Renames the specified disk file.
- `int remove( char* filename );`
  Deletes the named file from the disk.

Input functions.

- `int fgetc( FILE* str );`
  `int getc( FILE* str );`
  These functions read a single character from the stream `str`.
- `int getchar( void );`
  Reads a single character from the stream `stdin`.
- `int ungetc( int ch, FILE* str );`
  Puts a single character, `ch`, back into the stream `str`.
- `char* fgets( char* ar, int n, FILE* str );`
  Reads up to `n-1` characters from the stream `str` into the array `ar`. A newline character occurring before the nth input character terminates the operation and is stored in the array. A null character is stored at the end of the input.
- `char* gets( char* ar );`
  Reads characters from the stream `stdin` into the array `ar` until a newline character occurs, then stores a null character at the end of the input. The newline is not stored as part of the string.
- `int fscanf( FILE* str, const char* format, ... );`
  Reads input from stream `str` under the control of the format. It stores converted values in the addresses on the variable-length output list that follows the format.
- `int scanf( const char* format, ... );`
  Same as `fscanf()` to stream `stdin`.
- `int sscanf( char* s, const char* format, ... );`
  Same as `fscanf()` except that the input characters come from the string `s` instead of from a stream.
- `size_t fread( void* ar, size_t size, size_t count,`
  `FILE* str );`
  Reads a block of data of `size` times `count` bytes from the stream `str`
  into array `ar`.

Output functions.

- `int fputc( int ch, FILE* str );`
  `int putc( int ch, FILE* str );`
  These functions write `ch` to stream `str`.
- `int putchar( int ch );`
  Writes a single character, `ch`, to the stream `stdout`.
- `int fputs( const char* s, FILE* str );`
  Writes `s` to stream `str`.
- `int puts( const char* s );`
  Writes string `s` and a newline character to the stream `stdout`.
- `int fprintf( FILE* str, const char* format, ... );`
  Writes values from the variable-length output list to the stream `str` under the control of the format.
- `int printf( const char* format, ... );`
  Writes values from the variable-length output list to the stream `stdout` under the control of the format.
- `int sprintf( char* ar, const char* format, ... );`
  Writes values from the variable-length output list to the array `ar` under the control of the format.
- `size_t fwrite( const void* ar, size_t size,`
  `size_t count, FILE* str );`
  Writes a block of data of `size` times `count` bytes from the array `ar` into the stream `str`.

Advanced functions. The following functions are beyond the scope of this text; their prototypes are listed without comment.

- `int setvbuf( FILE* str, char* buf, int bufmode, size_t size );`
- `void setbuf( FILE* str, char* buf );`
- Buffer mode constants `BUFSIZ`, `_IOFBF`, `_IOLBF`, `_IONBF`
- `int fseek( FILE* str, long int offset, int from );`
- `long int ftell( FILE* str );`
- `void rewind( FILE* str );`
- Seek constants `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
- `int fgetpos( FILE* str, fpos_t* pos );`
- `int fsetpos( FILE* str, const fpos_t* pos );`
- `void perror( const char* s );`
- `int vfprintf( FILE* str, const char* format, va_list arg );`
- `int vprintf( const char* format, va_list arg );`
- `int vsprintf( char* ar, const char* format, va_list arg );`
- `FILE* tmpfile( void );`
- `char* tmpnam( char* buf );` and constants `L_tmpnam`, `TMP_MAX`

F.3.5 Standard Library

Header file. `<stdlib.h>`.

Constants.

- `RAND_MAX`: the largest value that can be returned by `rand()`.
- `EXIT_FAILURE`: signifies unsuccessful termination when returned by `main()` or `exit()`.
- `EXIT_SUCCESS`: signifies successful termination when returned by `main()` or `exit()`.

`Typedefs.` `div_t` and `ldiv_t`, ISO C only, the types returned by the functions `div()` and `ldiv()`, respectively. Both are structures with two components, `quot` and `rem`, for the quotient and remainder of an integer division.

General functions.

- `int abs( int x );`
  `long labs( long x );`
  These functions return the absolute value of `x`. Note: `fabs()`, for floating-point numbers, is defined in `<math.h>`.
- `div_t div( int n, int d );`
  `ldiv_t ldiv( long n, long d );`
  These functions perform the integer division of `n` by `d`. The quotient and remainder are returned in a structure of type `div_t` or `ldiv_t`.

- `void srand( unsigned s );`
  `int rand( void );`
  The function `srand()` is used to initialize the random-number generator and should be called before using `rand()`. Successive calls on `rand()` return pseudo-random numbers, evenly distributed over the range `0...RAND_MAX`.
- `void* bsearch( const void* key, const void* base,`
  `size_t count, size_t size, int (*compar)`
  `(const void* key, const void* value) );`
  Searches the array starting at `base` for an element that matches `key`. A total of `count` elements are in the array. It uses `*compar()` to determine whether two items match. See the text for explanation.
- `int qsort( void* base, size_t count, size_t size,`
  `int (*compar)(const void* e1, const void* e2) );`
  Quicksorts the elements of the array starting at `base` and continuing for `count` elements. It uses `*compar()` to compare the elements. See the text for explanation.

Allocation functions.

- `void* malloc( size_t size );`
  Dynamically allocates a memory area of `size` bytes and returns the address of the beginning of this area.
- `void* calloc( size_t count, size_t size );`
  Dynamically allocates a memory area of `count` times `size` bytes. It clears all the bits in this area to 0 and returns the address of the beginning of this area.
- `void free( void* pt );`
  Returns the dynamically allocated area `*pt` to the system for future reuse.
- `void* realloc( void* pt, size_t size );`
  Resizes the dynamically allocated area `*pt` to `size` bytes. If this is larger than the current size and the current allocation area cannot be extended, it allocates the entire `size` bytes elsewhere and copies the information from `*pt`.

Control functions.

- `void exit( int status );`
  Flushes all the buffers, closes all the streams, and returns the status code to the operating system.
- `int atexit( void (*func)( void ) );`
  ISO C only. The function `(*func)()` is called when `exit()` is called or when `main()` returns.

String to number conversion functions.

- `double strtod( const char* str, char** p );`
  `double atof( const char* str );`
  The function `strtod()` converts the ASCII string `str` to a number of type `double` and returns that number. It leaves `*p` pointing at the first character in `str` that was not part of the number. The function `atof()` does the same thing but does not return a pointer to the first unconverted character. The preferred function is `strtod()`; `atof()` is deprecated in the latest version of the standard.
- `long strtol( const char* str, char** p, int b );`
  The function `strtol()` converts the ASCII string `str` to a number of type `long int` expressed in base `b` and returns that number. It leaves `*p` pointing at the first character in `str` that was not part of the number. This function is preferred over both `atoi()` and `atol()`, which are deprecated in the latest version of the standard.

- `int atoi( const char* str );`
  `long atol( const char* str );`
  The function `atoi()` converts the ASCII string `str` to a number of type `int` expressed in base 10 and returns that number; `atol()` converts to type `long int`. The function `strtol()` is preferred over both of these, which are deprecated in the latest version of the standard.
- `unsigned long strtoul( const char* str, char** p, int b );`
  Converts the ASCII string `str` to a number of type `long unsigned int` expressed in base `b` and returns that number. It leaves `*p` pointing at the first character in `str` that was not part of the number.

Advanced functions. The following functions are beyond the scope of this text; their prototypes are listed without comment.

- `void abort( void );`
- `char* getenv( const char* name );`
- `int system( const char* command );`

F.3.6 Strings

Header file. `<string.h>`.

String manipulation.

- `char* strcat( char* dest, const char* src );`
  Appends the string `src` to the end of the string `dest`, overwriting its null terminator. We assume that `dest` has space for the combined string.
- `char* strncat(char* dest, const char* src, size_t n);`
  This function is the same as `strcat()` except that it stops after copying `n` characters, then writes a null terminator.
- `char* strcpy( char* dest, const char* src );`
  Copies the string `src` into the array `dest`. We assume that `dest` has space for the string.
- `char* strncpy(char* dest, const char* src, size_t n);`
  Copies exactly `n` characters from `src` into `dest`. If fewer than `n` characters are in `src`, null characters are appended until exactly `n` have been written.
- `int strcmp( const char* p, const char* q );`
  Compares string `p` to string `q` and returns a negative value if `p` is lexicographically less than `q`, 0 if they are equal, or a positive value if `p` is greater than `q`.
- `int strncmp(const char* p, const char* q, size_t n);`
  This function is the same as `strcmp()` but returns after comparing at most `n` characters.
- `size_t strlen( const char* s );`
  Returns the number of characters in the string `s`, excluding the null character on the end.
- `char* strchr( const char* s, int ch );`
  Searches the string `s` for the first (leftmost) occurrence of the character `ch`. Returns a pointer to that occurrence if it exists; otherwise returns `NULL`.
- `char* strrchr( const char* s, int ch );`
  Searches the string `s` for the last (rightmost) occurrence of the character `ch`. Returns a pointer to that occurrence if it exists; otherwise returns `NULL`.
- `char* strstr( const char* s, const char* sub );`
  Searches the string `s` for the first (leftmost) occurrence of the substring `sub`. Returns a pointer to the first character of that occurrence if it exists; otherwise returns `NULL`.

Memory functions.

- `void* memchr( const void* ptr, int val, size_t len );`
  Copies `val` into `len` characters starting at address `ptr`.
- `int memcmp( const void* p, const void* q, size_t n );`
  Compares the first `n` characters starting at address `p` to the first `n` characters starting at `q`. Returns a negative value if `p` is lexicographically less than `q`, 0 if they are equal, or a positive value if `p` is greater than `q`.
- `void* memcpy(void* dest, const void* src, size_t n);`
  Copies `n` characters from `src` into `dest` and returns the address `src`. This may not work correctly for overlapping memory regions but often is faster than `memmove()`.
- `void* memmove(void* dest, const void* src, size_t n);`
  Copies `n` characters from `src` into `dest` and returns the address `src`. This works correctly for overlapping memory regions.
- `void* memset( void* ptr, int val, size_t len );`
  Copies `val` into `len` characters starting at address `ptr`.

Advanced functions. The following functions are beyond the scope of this text; their prototypes are listed without comment:

- `int strcoll( const char* s1, const char* s2 );`
- `size_t strcspn( const char* s, const char* set );`
- `char* strerror( int errnum );`
- `char* strpbrk( const char* s, const char* set );`
- `size_t strspn( const char* s, const char* set );`
- `char* strtok( char* s, const char* set );`
- `size_t strxfrm( char* d, const char* s, size_t len );`

F.3.7 Time and Date

Header file. `<time.h>`.

Constants. `CLOCKS_PER_SEC` is the number of clock "ticks" per second of the clock used to record process time.

Types.

- `time_t;`
  The integer type used to represent times on the local system.
- `clock_t;`
  The arithmetic type used to represent the process time on the local system.
- `struct tm;`
  A structured representation of the time containing the following fields, all of type `int`: `tm_sec` (seconds after the minute), `tm_min` (minutes after the hour), `tm_hour` (hours since midnight, 0–23), `tm_mday` (day of the month, 1–31), `tm_mon` (month since January, 0–11), `tm_year` (years since 1900), `tm_wday` (day since Sunday, 0–6), `tm_yday` (day since January 1, 0–365), `tm_isdst` (daylight savings time flag, `>0` if DST is in effect, `0` if not, `<0` if unknown).

Functions.

- `clock_t clock();`
  Returns an approximation to the processor time used by the current process, usually expressed in microseconds.
- `time_t time( time_t* tptr );`
  Reads the system clock and returns the time as an integer encoding of type `time_t`. Returns the same value through the pointer parameter.
- `char* ctime( const time_t* tptr );`
  `char* asctime( const struct tm* tptr );`
  These functions return a pointer to a string containing a printable form of the date and time: `"Sat Sep 14 13:12:27 1999\n"`. The argument to `ctime()` is a pointer to a `time_t` value such as that returned by `time()`. The argument to `asctime()` is a pointer to a structured calendar time such as that returned by `localtime()` or `gmtime()`.
- `struct tm* gmtime( const time_t* tp );`
  `struct tm* localtime( const time_t* tp );`
  These functions convert a time represented as a `time_t` value to a structured representation. The `gmtime()` returns Greenwich mean time; the `localtime()` converts to local time, taking into account the time zone and Daylight Savings Time.
- `time_t mktime( struct tm* tp );`
  Converts a time from the `struct tm` representation to the integer `time_t` representation.
- `double difftime( time_t t1, time_t t2 );`
  Returns the result of `t1-t2` in seconds as a value of type `double`.
- `size_t strftime( char* s, size_t max,`
  `    const char* format, const struct tm* tp );`
  `size_t wcsftime( w_char* s, size_t max,`
  `    const w_char* format, const struct tm* tp );`
  The function `strftime()` formats a single date and time value specified by `tp`, storing up to `maxsize` characters into the array `s` under control of the string `format`. The function `wcsftime()` does the same thing with wide characters.

F.3.8 Variable-Length Argument Lists

This library, known as the *vararg* facility, permits programmers to define functions with variable-length argument lists. This is an advanced feature of C and beyond the scope of this text. The list of functions is included here because this facility was used to define `say()` and `fatal()`.

Header file. `<stdarg.h>`.

Type. `va_list`.

Functions. `va_start`, `va_arg`, and `va_end`.

F.4 Libraries Not Explored

Each of the remaining libraries is named and the names of functions and constants in them are listed without prototypes or explanation. This list can serve as a starting point for further exploration of C.

Errors. Header file: `<errno.h>`. Constants: `EDOM` and `ERANGE`.
Variable: `errno`.

Nonlocal jumps. Header file: `<setjmp.h>`. Type: `jmpbuf`.
Functions: `setjmp` and `longjmp`.

Signal handling. Header file: `<signal.h>`. Type: `sig_atomic_t`. Constants: `SIG_DFL`, `SIG_ERR`, `SIG_IGN`, `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, and `SIGTERM`. Functions: `signal`, `raise`.

Control. Header file: `<assert.h>`. Constant: `NDEBUG`. Function: `assert`.

Localization. Header file: `<locale.h>`. Constants: `LC_ALL`, `LC_TIME`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, `LC_COLLATE`, and `NULL`. Type: `struct lconv`. Functions: `localeconv` and `setlocale`.

Wide-character handling. Header file: `<wctype.h>`. Functions: `iswctype`, `towctrans`, `WEOF`, `wint_t`, `wctrans`, `wctrans_t`, `wctype`, and `wctype_t`. In addition, this library contains wide analogs of all the functions in the `ctype` library, all with a `w` as the third letter of the name: `iswupper`, `towlower`, and so on.

Extended multibyte to wide-character conversion. Header file: `<wchar.h>`. Functions: `btowc`, `mbrlen`, `mbrtowc`, `mbstate_t`, `wcrtomb`, `mbsinit`, `mbsrtowcs`, `wcsrtombs`, `wcstod`, `wcstol`, `wcstoul`, and `wctob`.

Wide-string handling. Header file: `<wchar.h>`. Functions: `wcscat`, `wcschr`, `wcscmp`, `wcscoll`, `wcscpy`, `wcscspn`, `wcserror`, `wcslen`, `wcsncat`, `wcsncmp`, `wcsncpy`, `wcspbrk`, `wcsrchr`, `wcsspn`, `wcsstr`, `wcstok`, `wcsxfrm`, `wmemchr`, `wmemcmp`, `wmemcpy`, `wmemmove`, and `wmemset`.

Wide-character input and output. Header file: `<wchar.h>`. Functions: `fwprintf`, `fwscanf`, `wprintf`, `wscanf`, `swprintf`, `swscanf`, `vfwprintf`, `vwprintf`, `vwsprintf`, `fgetwc`, `fgetws`, `fputwc`, `fputws`, `getwc`, `getwchar`, `putwc`, `putwchar`, and `ungetwc`.