The C Programming Language

Types, Operators, and Expressions

Constants

A character constant is an integer, written as one character within single quotes such as ‘x’. The value of a character constant is the numeric value of the character in the machine’s character set. For example, ‘0’ has the value 48 in ASCII.

Certain characters can be expressed in character and string constants by escape sequences like `\n`. In addition, an arbitrary byte-sized bit pattern can be specified by `\ooo`, where `ooo` is one to three octal digits (0…7) or by `\xhh` where `hh` is one or more hexadecimal digits (0..9, a…f, A…F). So as an example, you can write:

```
#define VTAB '\013' // ASCII vertical tab
#define BELL '\007' // ASCII bell character
```

or in hexadecimal,

```
#define VTAB '\xb'  // ASCII vertical tab
#define BELL '\x7'  // ASCII bell character
```

The complete set of escape sequences is:

`\a`	`alert (bell) character`
`\b`	`backspace`
`\f`	`formfeed`
`\n`	`newline`
`\r`	`carriage return`
`\t`	`horizontal tab`
`\v`	`vertical tab`
`\\`	`backslash`
`\?`	`question mark`
`\`	`single quote`
`\"`	`double quote`
`\0`	`null character`
`\ooo`	`octal number`
`\xhh`	`hexadecimal number`

Increment and Decrement Operators

An example of when to use `++i` rather than `i++`:

```
/* squeeze:  delete all c from s */

void squeeze(char s[], int c)

{

    int i, j;



    for (i = j = 0; s[i] != \0 ; i++)

        if (s[i] != c)

            s[j++] = s[i];

    s[j] = \0 ;
}
```

This is the same code as:

```
if (s[i] != c)
{

    s[j] = s[i];

    j++;

}
```

Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`, whether signed or unsigned.

```
&    bitwise AND
|    bitwise inclusive OR
^    bitwise exclusive OR
<<   left shift
>>   right shift
~    one's complement (unary)
```

~ yield's one's compliment of an integer; that is, it converts each 1-bit into a 0-bit and vice versa.

Control Flow

Break and Continue

The `break` statement provides an early exit from `for`, `while`, and `do`, just as it does from `switch`. **A `break` causes the innermost enclosing loop or switch to be exited immediately.**

The `continue` statement is related to `break`, but less often used; **it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin.** In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. This statement only applies to loops, not to a `switch`.

GoTo and Labels

`goto` is almost never needed, but it can be useful in situations such as abandoning a process in some deeply nested structure, such as breaking out of two or more loops at once as shown below:

```
for ( ... )

    for ( ... ) {

        ...

        if (disaster)

            goto error;

    }

...

error:
    // Clean up the mess
```

Functions and Program Structure

Static Variables

The `static` declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled

Types, Operators, and Expressions

Constants

A character constant is an integer, written as one character within single quotes such as 'x'. The value of a character constant is the numeric value of the character in the machine's character set. For example, '0' has the value 48 in ASCII.

Certain characters can be expressed in character and string constants by escape sequences like `\n`. In addition, an arbitrary byte-sized bit pattern can be specified by `\ooo`, where ooo is one to three octal digits (0…7) or by `\xhh` where `hh` is one or more hexadecimal digits (0..9, a…f, A…F). So as an example, you can write:

```
#define VTAB '\013' // ASCII vertical tab
#define BELL '\007' // ASCII bell character
```

or in hexadecimal,

```
#define VTAB '\xb'  // ASCII vertical tab
#define BELL '\x7'  // ASCII bell character
```

The complete set of escape sequences is:

\a	alert (bell) character
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\?	question mark
\	single quote
\"	double quote
\0	null character
\ooo	octal number
\xhh	hexadecimal number

Increment and Decrement Operators

An example of when to use ++i rather than i++:

```
/* squeeze:  delete all c from s */

void squeeze(char s[], int c)

{

    int i, j;



    for (i = j = 0; s[i] !=  \0 ; i++)

        if (s[i] != c)

            s[j++] = s[i];

    s[j] =  \0 ;

}
```

This is the same code as:

```
if (s[i] != c)
{

    s[j] = s[i];

    j++;
}
```

Bitwise Operators

C provides six operators for bit manipulation; these may only be applied to integral operands, that is, `char`, `short`, `int`, and `long`, whether signed or unsigned.

```
&    bitwise AND
|    bitwise inclusive OR
^    bitwise exclusive OR
<<   left shift
>>   right shift
~    one's complement (unary)
```

~ yield's one's compliment of an integer; that is, it converts each 1-bit into a 0-bit and vice versa.

Control Flow

Break and Continue

The `break` statement provides an early exit from `for`, `while`, and `do`, just as it does from `switch`. **A `break` causes the innermost enclosing loop or switch to be exited immediately.**

The `continue` statement is related to `break`, but less often used; **it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin.** In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. This statement only applies to loops, not to a `switch`.

GoTo and Labels

`goto` is almost never needed, but it can be useful in situations such as abandoning a process in some deeply nested structure, such as breaking out of two or more loops at once as shown below:

```
for ( ... )

    for ( ... ) {

        ...

        if (disaster)

            goto error;

    }

...

error:
    // Clean up the mess
```

Functions and Program Structure

Static Variables

«««< Updated upstream The `static` declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. ======= The `static` declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled

Pointers and Arrays

In C, there is a strong relationship between pointers and arrays. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will generally be faster, but at least to the uninitiated, somewhat harder to understand.

If you look at an array, indexing it via `array[i]`, is the same way as indexing it via `&array[0] + i`. `&array[0]`, is the pointer to the first item in the array, if we have a value I added on to it, that will move down the array as shown in Figure 1

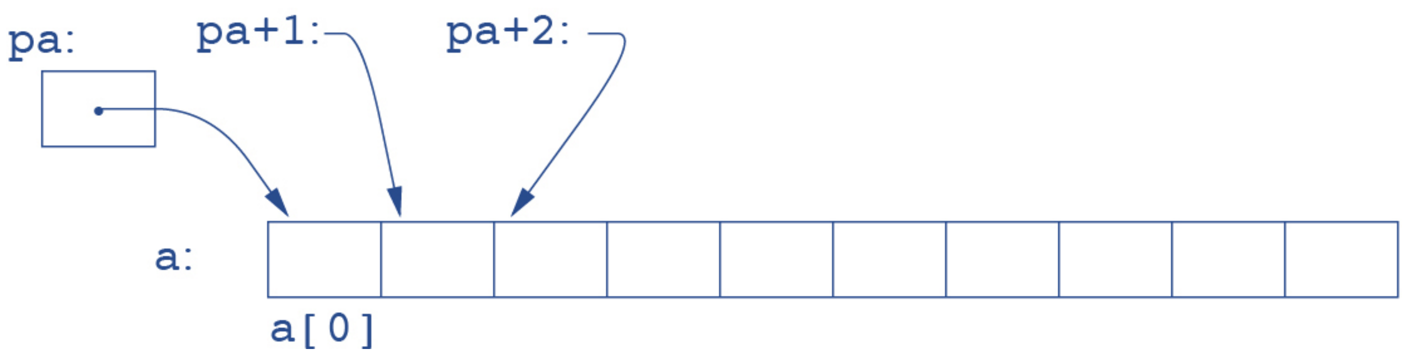


Figure 1: Using pointers to index an array

Using this line of thinking, it is possible to then pass part of an array to a function:

```
foo(&a[2]);
// or
foo(a+2);
```

Where the function `foo`'s arguments have the form of:

```
foo(int arr[]) {...}
// or
fo(int* arr) {...}
```

As far as `foo` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

Address Arithmetic

Because memory is allocated sequentially, we can use mathematical comparisons to see if we have enough memory allocated in a buffer, to see a position of a value in an array, etc. For example:

```
if (allocbuf + ALLOCSIZE - allocp >= n)      // Checks if the buffer is large enough
// or
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)  // Checks if the pointer is within the buffer range
```

Character Pointers and Functions

A string constant is written as `"I am a string"`. It is represented as an array of characters and is terminated with a \0.

When a character string appears, it is accessed through a character pointer. For example

```
printf("Hello, World!\n");
```

takes the pointer of the start of the character string.

If you declare a character pointer as follows:

```
char* pmessage;
```

then the statement

```
pmessage = "Hello, World!";
```

Assigns `pmessage` to the "Hello, World!". This is *not* a copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

Pointer Arrays; Pointers to Pointers

To create an array of pointers, we can say:

```
char* lineptr[10];  // Array of character pointers
```

Multi-Dimensional Arrays

C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers.

**If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows.

```
foo (int daytab[2][13] {...}
// or
foo (int daytab[][13] {...}
// or
foo (int (*daytab)[13] {...}
```

It is necessary to have the parenthesis in the last version since brackets [] have higher precedence than *. Without the parenthesis

```
int* daytab[13]
```

is an array of 13 pointers to integers. More generally, only the first dimension of an array is free; all others have to be specified.

Initialization of Pointer Arrays

```
char* array_of_pointers[] =
{
    "Item 1",
    "Item 2",
    ...
}
```

Pointers vs. Multi-Dimensional Array

The big difference between

```
int a[10][20];
// and
int* b[10];
```

Is that the first one allocates all 10x20 array, the second only allocates 10 integer pointers. Initialization must be done explicitly either statically or with code. The important advantage of the pointer array is that the rows of the array may be of different lengths.

Command Line Arguments

When `main` is called, it is called with two arguments. The first (conventionally called `argc` for argument count) is the number of command-line arguments. The second (`argv`, for argument vector), is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

Accepting Command Line Flags

```
#include <stdio.h>

#include <string.h>

#define MAXLINE 1000



int getline(char *line, int max);



/* find:  print lines that match pattern from 1st arg */

main(int argc, char *argv[])

{

    char line[MAXLINE];

    long lineno = 0;

    int c, except = 0, number = 0, found = 0;
    while (--argc > 0 && (*++argv)[0] ==  - )
        while (c = *++argv[0])
            switch (c) {
            case  x :
                except = 1;
                break;
            case  n :
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

Pointers To Functions

In C, it is possible to define pointers to functions which can be assigned, placed in arrays, passed to functions, returned by functions, and so on.

The generic `void*` is used for pointer arguments. Any pointer can be cast to `void*` and back again without loss of information.

When creating a function that accepts function pointers:

```
void qsort (void* v[], int left, int right, int (*comp)(void*, void*))
```

Says that comp is a pointer to a function that has two `void*` as arguments.

Complicated Declarations

```
char **argv

    argv:  pointer to pointer to char

int (*daytab)[13]

    daytab:  pointer to array[13] of int

int *daytab[13]

    daytab:  array[13] of pointer to int

void *comp()

    comp:  function returning pointer to void

void (*comp)()

    comp:  pointer to function returning void

char (*(*x())[])()

    x: function returning pointer to array[] of

    pointer to function returning char

char (*(*x[3])())[5]

    x: array[3] of pointer to function returning

    pointer to array[5] of char
```

Structures

A `struct` declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct {...} x, y, z;
```

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or the shape of a structure. If the declaration is tagged, however; the tag can be used later in definitions of instances of the structure. For example:

```
struct point pt;
```

Is analogous to `int c`.

Arrays of Structures

When defining an array of structures, it is acceptable to define it in the following manner:

```
struct key {

    char *word;

    int count;

} keytab[] = {
    "auto", 0,

    "break", 0,

    "case", 0,

    "char", 0,

    "const", 0,

    "continue", 0,

    "default", 0,

    /* ... */

    "unsigned", 0,

    "void", 0,

    "volatile", 0,

    "while", 0

};
```

However, it is more precise to enclose the initializers for each "row" or structure in braces:

```
 { "auto", 0 },

 { "break", 0 },

 { "case", 0 },

...
```

Table Lookup

This section will be about writing the innards of a table-lookup package.

There are two routines that manipulate the names and replace texts. `install(s,t)` records the name s and the replacement text `t` in a table; `s` and `t` are just character strings. `lookup(s)` searches for `s` in the table, and returns a pointer to the place where it was found, or `NULL` if it wasn't there.

The algorithm is a hash search—the incoming name is converted into a small non-negative integer, which is then used to index into an array of pointers. An array element points to the beginning of a linked list of blocks describing names that have that hash value. It is NULL if no names have hashed to that value.

This is shown in Figure 2.

A block in the list is a structure containing pointers to the name, the replacement text, and the next blocks in the list. A `NULL` next-pointer marks the end of the list.

```
struct nlist {       /* table entry: */

    struct nlist *next;   /* next entry in chain */

    char *name;           /* defined name */

    char *defn;           /* replacement text */

};
```

The pointer array is just

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE];    /* pointer table */
```

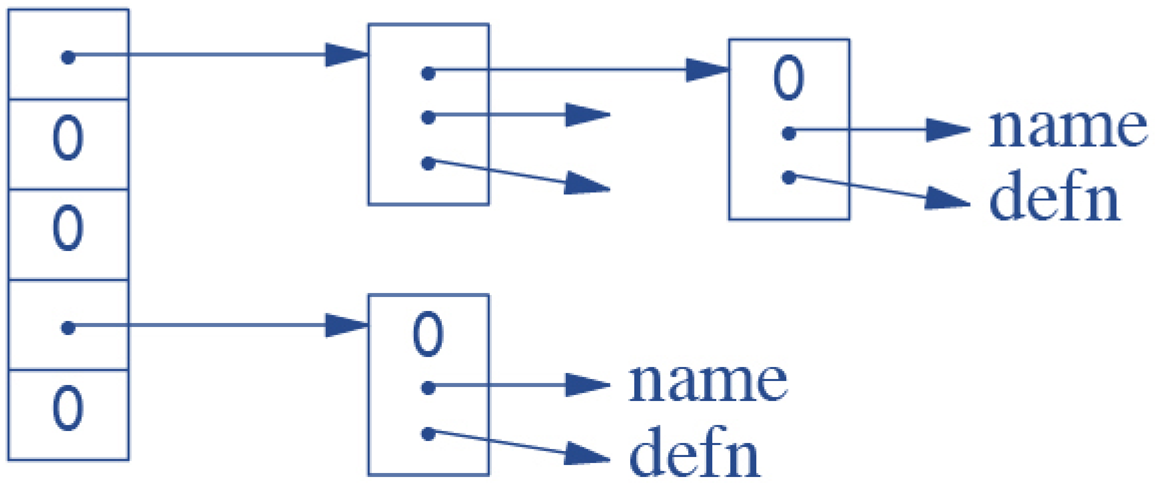


Figure 2: Diagram of table lookup

The hashing function, which is used by both `lookup` and `install`, adds each character value in the string to a scrambled combination of the previous ones and returns the remainder modulo the array size.

```
/* hash:  form hash value for string s */

unsigned hash(char *s)

{

    unsigned hashval;



    for (hashval = 0; *s !=  \0 ; s++)

        hashval = *s + 31 * hashval;

    return hashval % HASHSIZE;

}
```

The reason we return `hashval % HASHSIZE` is because `hastab` only has a size of `HASHSIZE`. In this particular instance, the hash is placing the item in a arbitrary position from 0 to `HASHSIZE` - 1.

The hashing process produces a staring index in the array `hashtab`; if the string is to be found anywhere, it will be int he list of blocks beginning there. By this, it is meant that if there is a collision in the hashing, a list will be produced at that hash from which the for loop will search for the string name.

```
/* lookup:  look for s in hashtab */

struct nlist *lookup(char *s)

{

    struct nlist *np;



    for (np = hashtab[hash(s)]; np != NULL; np = np->next)

        if (strcmp(s, np->name) == 0)

            return np;  /* found */

    return NULL;        /* not found */

}
```

`install` returns a `NULL` if for any reason there is no room for a new entry.

```c
struct nlist *lookup(char *);

char *strdup(char *);



/* install:  put (name, defn) in hashtab */

struct nlist *install(char *name, char *defn)

{

    struct nlist *np;

    unsigned hashval;


    if ((np = lookup(name)) == NULL) {  /* not found */

        np = (struct nlist *) malloc(sizeof(*np));

        if (np == NULL || (np->name = strdup(name)) == NULL)

            return NULL;

        hashval = hash(name);

        np->next = hashtab[hashval];

        hashtab[hashval] = np;

    } else      /* already there */

        free((void *) np->defn);  /* free previous defn */

    if ((np->defn = strdup(defn)) == NULL)

        return NULL;

    return np;

}
```

Input and Output

Command Execution

The function `system(char* s)` executes the commands contained in the character string `s`, then resumes execution of the current program.

```c
system("date");
```

Storage Management

```c
void* malloc(size_t n); // returns a pointer to n bytes of uninitialized storage
void* calloc(size_t n, size_t size); // returns a pointer to enough space for an array of n objects of the
free(p); // frees the space appointed to p where p was originally obtained by calling malloc or calloc.
```

The UNIX System Interface

Open, Creat, Close, Unlink

Other than the default standard input, output and error, you must explicitly open files in order to read or write them (such as with I/O manipulation with external hardware).

There is a limit (often about 20) on the number of files that a program may have open simultaneously.

Random Access - Lseek

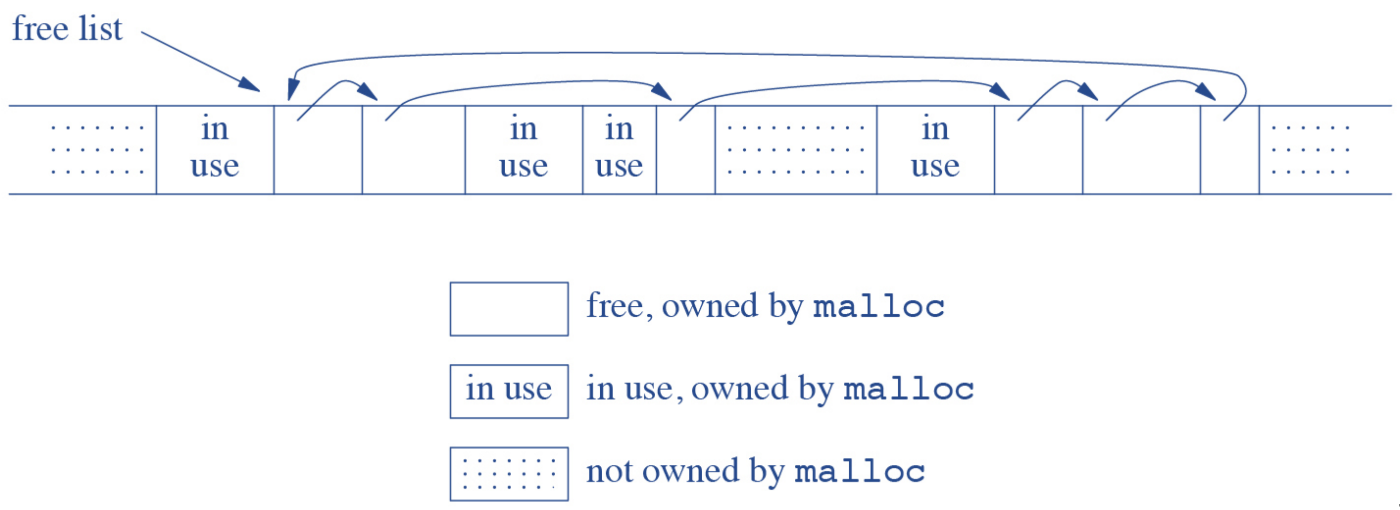
When necessary, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without reading or writing any data:

```
long lseek(int fd, long offset, int origin);
```

`lseek` sets the position in the file whose descriptor is given. Subsequent reads and write will start at this point.

Example - A Storage Allocator

This example creates a memory allocator that allows for non-continuous and/or segmented memory allocation as shown in Figure **??**.



`freep` is a list of pointers to the start of free areas of memory. The list contains structures of the form:

```
typedef long Align;  /* for alignment to long boundary */


union header {       /* block header: */

    struct {

        union header *ptr; /* next block if on free list */

        unsigned size;     /* size of this block */

    } s;

    Align x;         /* force alignment of blocks */

};


typedef union header Header;
```

The overridden `malloc` method has the form of:

```
static Header base;        /* empty list to get started */

static Header *freep = NULL;     /* start of free list */
```

```c
/* malloc:  general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *morecore(unsigned);
    unsigned nunits;

    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {  /* no free list yet */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {    /* big enough */
            if (p->s.size == nunits)     /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {             /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p+1);
        }
        if (p == freep)  /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;   /* none left */
    }
}
```

Where morecore is used to obtain more storage from the operating system.

```c
#define    NALLOC    1024    /* minimum #units to request */
```

```
/* morecore:  ask system for more memory */

static Header *morecore(unsigned nu)

{

    char *cp, *sbrk(int);

    Header *up;



    if (nu < NALLOC)

        nu = NALLOC;

    cp = sbrk(nu * sizeof(Header));

    if (cp == (char *) -1)    /* no space at all */

        return NULL;

    up = (Header *) cp;

    up->s.size = nu;

    free((void *)(up+1));

    return freep;

}
```

free is the last thing. It scans the free list looking for the place to insert the free block. This is either between two existing blocks or at one end of the list. In any case, if th block being freed is adjacent to either neighbor, the adjacent blocks are combined.

```
/* free:  put block ap in free list */

void free(void *ap)

{

    Header *bp, *p;



    bp = (Header *)ap - 1;    /* point to block header */

    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)

        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))

            break;  /* freed block at start or end of arena */



    if (bp + bp->s.size == p->s.ptr) { /* join to upper nbr */

        bp->s.size += p->s.ptr->s.size;

        bp->s.ptr = p->s.ptr->s.ptr;

    } else
```

```
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {          /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```