DCC Compiler System: User's Guide

Edited by Jeff Teunissen

The latest version of DCC may be found at
`https://github.com/Deek/CoCoC/`

Edition 0, December 2022
Not finished yet, input welcome!

This book wouldn't be possible without the amazing work of Dennis M. Ritchie, Ken Thompson, Brian W. Kernighan, Alfred Aho, Rob Pike, Stephen Bourne, and a host of others. The co-creations of Unix and C have had impossible-to-overstate impacts on the development of the modern world. It is upon their shoulders we all stand.

The information contained in this book is believed to be accurate as of the date of publication; however, the CoCoC Project disclaims any responsibility for any damages, including indirect or consequential, from the use of the DCC compiler system or from reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

The original code for this compiler is believed to have been written by James McCosh in 1982-1983, with OS-9 implementation assistance from Terry Crane and Kim Kempf. The Relocatable Macro Assembler, Linker, and Profiler were edited by Wes Camden and Ken Kaplan.

This book was typeset in Adobe Minion Pro, Adobe Myriad Pro, and DejaVu Sans Mono using LyX and XeLaTeX.

Contents

1
Getting Started

Introduction

It's no great exaggeration to say that the C programming language is the foundation for the modern computing world. At various times, it is either the most popular or second most popular programming languages in the world. The rapid rise of C as the go-to systems programming language is not surprising, since C is an incredibly versatile and efficient language that can handle tasks that previously would have required complex assembly language programming.

C was originally developed at the AT&T Bell System Laboratories as an implementation language for the Unix operating system by Dennis Ritchie. Later, Ritchie and his colleague Brian W. Kernighan wrote a book *(The C Programming Language)* that became universally accepted as the standard for the language until it was formally standardized by ANSI[1] and later ISO[2]. It is an interesting reflection on C that even before the ANSI standard, C programs tended to be far more *portable* (taking source from one machine and running it on another) between radically different computer systems than "standard" languages such as BASIC, COBOL, and Pascal.

The biggest reason C is very portable is that the language is *very simple*. You can learn everything you need to know to produce working, useful programs in a few days, or at most a week. A full description of C's syntax can fit in a few pages (you'll find such a description towards the end of this book). Further, the C language has very little in the way of runtime requirements – that is, the code that is needed in every program just to make it function. For Basic09, the `RunB` module needed to run programs is 12 kilobytes in size, while the code needed to set up a C program, process its arguments, and set up the standard file-based I/O is only a few hundred bytes.

Because the language is itself very simple, its creators made it inherently expandable. If you need new functions, you can just write them exactly like the functions made by the creators of the language. Compare this to the older practice of adding additional keywords to a language when new functionality was needed; for example, the number of BASIC dialects in existence, most just because different machines had slightly different capabilities, defies all

[1] ANSI X3.159-1989

[2] ISO 9899:1990, 1999, 2011, 2018

reason. A lesser factor is the Unix operating system, which is *also* quite versatile on its own and serves to allow programs to work together. Indeed, C and Unix are intimately related.

Fortunately, the 6809 microprocessor, the OS-9 operating system, and the C language form an outstanding combination. The 6809 was specifically designed to efficiently run high-level languages, and its stack-oriented instruction set and versatile repertoire of addressing modes handle the C language very well. As mentioned previously, Unix and C are closely related, and OS-9's similarity to Unix allows it to support C programs better than most similar small-computer operating systems. Most applications written in C can be transported from a Unix system to an OS-9 system, recompiled, and correctly executed.

Who is this book for?

If you don't know *any* programming, this book is probably not for you. We're going to assume you have at least some experience programming in at least one other language, such that you won't be lost when you see lines like `x=x+1`. Also, while we try to use some useful programming techniques, this is not a book about algorithms and data structures — it's specifically a book to help you learn C, as implemented in the DCC compiler.

The first C program

The best way to learn a new programming language is to write programs using it, and the first program almost everyone writes is a program that prints the words `Hello, world!` to the output device (usually the screen).

To many new programmers, this seems quite a silly program, but it's a basic test of capabilities: you must be able to create a text file, compile it into an executable, load it, and run it. Once you've gotten those skills, the rest isn't a lot more difficult.

In C, the most basic version of the first program looks like this:

```
#include <stdio.h>

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

There are a number of different text editors available in OS-9 and NitrOS-9, but for this first program we'll use the most basic text editor possible, a program that comes on almost every OS-9 disk: `build`. This program is quite primitive, offering little in the way of editing other than what the "read line" handler gives for free, so try not to make any mistakes typing.

At the `OS9:` shell prompt, type: `build hello.c` and press the ENTER key. You should be met with a different prompt, a simple `?`. For each line you enter at the prompt, that line will be written to the file `hello.c`; to finish, press ENTER on an empty line. To add an empty line,

use a single space and *then* hit ENTER. Once finished, the screen should look something like this:

```
OS9: build hello.c
? main()
? {
?     printf("Hello, world!\n");
? }
?
OS9: ▉
```

To compile this new file `hello.c`, run the command `dcc hello.c`; if you have entered everything correctly, `dcc` will then run a few programs and exit cleanly, leaving a program file `hello` in the current execution directory[3]. Running that program by typing `hello` at the prompt, should produce the expected output:

```
OS9: hello
Hello, world!
OS9: ▉
```

Let's explain the parts of the program now. A C program is composed of one or more *functions*, which describe the actual computations that the program does. C functions are similar to the procedures of a BASIC09 program, and somewhat like the lines between the target of a **GOSUB** and the next **RETURN** in a Color BASIC program. In our example, `main` is a function.

You can name functions almost anything you like, subject to a few rules we'll outline later, but `main` is special – it's where every C program starts running. This means every program written in C will, *and must*, have exactly one function named `main` somewhere in it. A program's main function usually invokes other functions to perform its job, some being other functions you or someone on your team has written, and others from libraries of functions that came with the compiler, or have been installed from other sources.

The usual method of communicating between functions, is using *arguments*, sometimes called parameters. A function's *argument list* is enclosed in parentheses; in our example, our `main` function doesn't use any arguments, so its argument list is empty. this is indicated by `()`. "Curly" braces, `{}`, enclose the statements that are the code of the function. If you have used Pascal before, they're like the words **BEGIN** and **END** in that language in that they form the bounds of a series of statements.

A function is invoked, or *called*, by giving its name, followed by a parenthesized list of arguments that are to be passed to the function. There is no need to **RUN** a function, like in BASIC09; you simply give the function's name. The parentheses are required however, even if the argument list passed to the function is empty.

[3] This is usually the directory `/DD/CMDS`

The line `printf("Hello, world!\n");` is a function call, calling the standard library function `printf`, with the *string constant* "Hello, world!\n" as an argument. That sequence `\n` inside the string is a C notation for the *newline character*. When output to the display, a newline returns the cursor to the first column of the next line. If you had left it out, the program would still *print* `Hello, world!` on the screen but instead of returning to the OS9: prompt on the next line, the output might look something like:

```
OS9: hello
Hello, world!OS9: ▉
```

The only way to get a newline into the `printf` argument is by using `\n`[4]; if you tried to insert an actual newline character into the string by doing

```
printf("Hello, world!
");
```

then the compiler would complain about an incomplete string.

Note that the two-character sequence `\n` actually represents a single character. An *escape sequence* such as `\n` is a general mechanism used to represent characters that are hard to insert with the keyboard or not visible when printed. Among the escape sequences provided by the C language are the backspace character `\b`, the carriage return character `\r`, tab character `\t`, `\"` for a double quote, and `\\` to represent the backslash character itself. On OS-9 systems, the compiler adds one more: `\l` (that's a lowercase L) to refer to the *linefeed* character. On Unix systems, the newline character is the same as (and in fact is) a linefeed, while on OS-9 systems the newline character is the same as a carriage return.

The `printf` function never adds a newline on its own, so it's possible to call it multiple times to build a single output line. Our first program could just as easily been written as

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

and while the program might run more slowly this way, its output would be identical.

Comments, variables, and arithmetic

Our next simple program prints all odd numbers smaller than 100. You can find it in Listing 1.1 on the facing page

[4] This is not strictly true — there are other codes that *can* be used, but they are not portable.

```c
/* print odd numbers */
main()
{
    int i, low, limit;

    low = 1;        // start here
    limit = 100;    // stop here

    i = low;
    while (i < limit) {
        if (i % 2)
            printf("%2d ", i);
        i = i + 1;
    }
    printf("\n");
}
```

Listing 1.1: Printing odd numbers

The first line of the program is a *comment*, which in this case explains what the program is supposed to do. Any characters between `/*` and the next `*/` are completely ignored by the compiler. Every C compiler supports this comment form. DCC and more recent C compilers also support a second type of comment, which goes from the sequence `//` until the end of the current line. Comments can be used freely to make your programs easier for you or others to understand. A comment may appear anywhere a blank or newline may appear. Note, though, that `/* */` style comments *do not nest*, so you should be careful when using them to disable code.

Unlike some languages that automatically allocate storage for variables when they are first used, in C all variables *must* be declared before you can use them. This is usually done at the top of a function, before you place any executable statements. However, because the compiler can't always know if a variable is declared in another file of the program, it can't always detect that you have used an undeclared variable. If you forget to declare a variable, the compiler might *seem* to allow it, only to prevent the source code from being turned into a program module at the end.

A *declaration* consists of a named type and a list of one or more variables that have that type, as in `int i, low, limit;`. The type **`int`** tells the compiler that the listed variables are *integers*. The actual size of any integer type is *machine-dependent*. Machine-dependent features may vary between machines, but there are certain minimum limits that such features must always meet. For example, on the 6809, an **`int`** is a 16-bit signed number that may take the value of any whole number in the range −32768 to 32767; while on a 32-bit Linux system, the "same" **`int`** is 32 bits wide and can hold numbers of much higher magnitude. Another kind of number, a **`float`**, is a *floating point* type. Floating point number types have

an integer part and a fractional part. The single-precision **float** type may take values very close to almost any real number with about seven or fewer significant digits (though there are numbers that can never be exactly represented in floating point formats).

C supplies several basic types besides **int** and **float**:

Type	Description	Size on 6809
char	one byte, the smallest integer	1 byte
short	short integer	2 bytes
int	integer of the most convenient size	2 bytes
long	long integer	4 bytes
float	single-precision floating point	4 bytes
double	double-precision floating point	8 bytes

The sizes of these types are also machine-dependent, however all integral types must be defined such that (**char** = 1) ≤ **short** ≤ **int** ≤ **long**. Likewise, **double** must always be at least as large as **float**. In addition to the basic types, there are *arrays*, *structures*, and *unions* of these basic types; there are *pointers* to them, and *functions* that return them. We'll cover all of these later.

The actual computation in our counting program starts with the *assignments*

```
low = 1;
limit = 100;
```

These statements set the variables to the values we want them to start out with. Any statement is terminated by the semicolon character `;` only – never the end of a line.

Each number being printed is calculated the same way, so we use a loop that repeats once per entry. This is the purpose of the **while** statement

```
while (i < limit) {
    ...
}
```

First, the expression inside the parentheses is checked. If the expression is true (the value of `i` is less than the value of `limit`), the contents of the loop (everything between the braces) is executed once, the expression is checked again, and if true the loop is executed again, etc. Only when the expression becomes false (`i` is no longer less than `limit`) does the loop end.

The "body" of a **while** may be a *compound statement* with braces, like in our example above, or a *simple statement* without them, as in

```
while (i < limit)
    i = i * 2;
```

In either version, we choose to indent the statement(s) "inside" the **while** by one tab stop, to make it clear what's inside and what's outside the loop.

> **A note on source code formatting:** C makes few demands on how source code must be formatted, and it can be a very expressive language. It gives you many ways to be creative in formatting…but programs are also used to communicate with others. It's important to make good use of indentation and white space to make programs easy to read and understand, so *like* Kernighan & Ritchie, we recommend writing only one statement per line, and usually leaving blank space around operators. The positioning of braces is not actually very important, but it has become one of the "holy wars" programmers argue about. Feel free to pick a style that suits you and use it consistently, but if you're working with someone else's code, try to work with what's already there – very few people enjoy inconsistency.

You may notice that the loop is running once for each number between 1 and 99, but it's only displaying the odd numbers. What gives? Well, the expression

```
if (i % 2)
    printf(...);
```

is the key. In an **`if`** statement, the condition to be tested is an expression, enclosed in parentheses, and followed by a statement. The expression within the **`if`** is evaluated, and if its value is true (not equal to zero), the statement is executed. In our example, the `%` operator is the *mod operator*, which gives the remainder of a division operation. When the value of `i` is even, dividing it by 2 has no remainder, so the value of the expression is zero (false), and the `printf` function is not called.

Our example also includes another use of the `printf` function. Its first argument is a string of characters to be printed, with each `%` character introducing a format code describing how another of its arguments is to be interpreted, substituted, or formatted. In the statement

```
printf("%2d ", i);
```

the *conversion specifier* `%2d` says the next argument is an **`int`** to be interpreted as a decimal value, and printed in a space at least 2 spaces wide. `printf` also recognizes `%o` for octal, `%x` for hexadecimal, `%f` for a floating point value, `%c` for a character, `%s` for a string, and `%%` for a literal %. Each format code must be paired with its corresponding argument; if they don't line up properly by number and type, you'll get unexpected output or even a program crash.

By the way, `printf` is *not* actually part of the C language. The language specification does not include any input or output definitions, and there is nothing magic about `printf`. It's just an extremely useful function, part of the standard library that's *usually* available to C programs. If you have to *input* data, look up the `scanf` function. It's a lot like `printf`, except it reads instead of writes.

Trying another way: The `for` statement

As you can imagine, there are lots of ways to write a program; Listing 1.2 on the next page shows a different version of the same program. This version of the program produces exactly

the same output, but it looks vastly different. The major change is that we eliminated all but one of the variables — only `i`, the *iterator*, remains. The lower and upper limits are now just constant numbers in a **`for`** statement, a new construction we introduce.

```
/* print odd numbers */
main()
{
    int i;

    for (i = 1; i < 100; i += 2) {
        printf("%2d ", i);
    }

    printf("\n");
}
```

Listing 1.2: Odd numbers using **`for`**

The **`for`** statement is another kind of loop, a more general version of **`while`**. If you compare the **`for`** to the **`while`** we used before, the way it works should be fairly clear…but if it isn't, we'll describe it now. A **`for`** statement contains three parts, separated by semicolons. The first part, `i = 1`, is done once before the loop is entered for the first time. The second part is the expression that controls the loop, `i < 100`. Each time the loop begins, if this expression is true, then the body of the loop will be run; otherwise the loop exits. The third part, `i += 2`, executes each time the loop is run to its end, after the loop completes but before the condition is evaluated again.

Choosing whether to use **`for`** or **`while`** is essentially arbitrary; choose whichever one makes the most sense to you for what you're doing, and you'll almost certainly be right. **`for`** is usually good in loops when the initial setup and the re-setup of the loop condition are both related to one another, and especially if they are both single statements. Even though the eventual code is equivalent to **`while`**, **`for`** loops are more visually compact and keep the statements that control the loop right next to one another, often making the loop's state more clear.

Assignment operators

You may have noticed a construct you're not used to if you haven't yet used a language based on C: in the construct `i += 2`, += is an *assignment operator*. An assignment operator is any kind of operator that assigns the result to a variable; = is the simplest one, it assigns the value of the expression on the right to the location described on the left. You have probably figured this out by context, but `i += 2` means exactly the same thing as `i = i + 2`. This same shorthand works for all of the arithmetic operations in C. We'll cover these in more detail later.

Symbolic constants

In the previous example with the **`for`** loop, we replaced two variables with two basic numbers. This is often more efficient, because every time the value of a variable is looked up it must be fetched from memory, while a number can often be embedded right into a CPU instruction. This is all well and good, but we lost something in doing that — we lost the clarity of knowing what the numbers 1 and 100 meant in context. This might seem a strange objection, but it's sometimes an important one. We also lost the ability to use the same numbers in more than one place. That wasn't important in our example, but it often can be. For these reasons, it's considered bad practice to use such "magic" numbers in a program. Fortunately for us, C gives us a mechanism to avoid these magic numbers without giving up their speed benefits.

```
/* print odd numbers */
#define LOW     1   /* starting number */
#define HIGH    100 /* ending number */
#define STEP    2   /* step size */

main()
{
    int i;

    for (i = LOW; i < HIGH; i += STEP) {
        printf("%2d ", i);
    }

    printf("\n");
}
```

Listing 1.3: Odd numbers using constants

With the **`#define`** construct, at the beginning of your program you can define a *symbolic name*, or *macro* to be a particular string of text. From that point on, the compiler replaces each occurrence of that symbolic name with the corresponding text. The replacement for the name can be anything you like; it's not limited to numbers. Listing 1.3 shows the same program using constants instead of bare numbers. The quantities `LOW`, `HIGH`, and `STEP` are constants, so they don't need to be declared as variables. By convention, symbolic names are usually written all in upper case, to distinguish them from other identifiers like variables and functions.

Notice that there is no semicolon at the end of a definition. This is important, because all text after the named constant is replaced (once any comments are removed). If we added a semicolon after the number 1 in the line **`#define`** `LOW 1`, when we used it in our **`for`** loop, there would be too many semicolons and it would generate an error.

Assignments as expressions

This very simple program copies its input, one character at a time, to its output. If you have used other languages, the most obvious way to write it would probably look something like Listing 1.4.

```
#include <stdio.h>

/* copy input to output -- naive version */
main()
{
    int c;

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

Listing 1.4: A tiny `cat`

More experienced C programmers would usually write the code a bit more concisely by taking advantage of the fact that in C, every assignment is also an expression. That is, the statement `c = getchar()` doesn't just assign the value returned by the `getchar` function to the variable `c`, it's also an expression that has a value (specifically, the new value of `c`). This means you may do things like chaining assignments together to assign a single value to a whole group of variables, as in `a = b = c = d = 0;` but more importantly, we can use it to make our code more concise and somewhat more efficient as well.

```
/* copy input to output -- better version */
main()
{
    int c;

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

In this version, each time we go through the loop, we call `getchar`, assign its returned value to the variable `c`, and check that the value wasn't equal to the constant `EOF`. As long as that statement remains true, we keep calling `putchar(c)`.

This version of the program isn't just shorter; it also compiles into a smaller program, because there is only one call to `getchar` instead of two.

If, Else, and Huh?

The little program shown in Listing 1.5 is a very simple version of the Unix `wc` (word count) program. It reads the standard input one character at a time, and acts upon which characters it sees. There are a number of new language features introduced in this program, and we'll describe them now.

```
#define YES 1
#define NO  0

/* count lines, words, and chars in input */
main()
{
    int     c, nl, nw, inword;
    long    nc;

    inword = NO;
    nc = nw = nl = 0;

    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c = '\t')
            inword = NO;
        else if (!inword) {
            inword = YES;
            ++nw;
        }
    }
    printf("%d %d %ld\n", nl, nw, nc);
}
```

Listing 1.5: Counting Words

For the first time, we've brought in the other half of **`if`**, **`else`**. An **`if`** statement lets you execute a statement if an expression is true, but **`else`** adds the ability to execute a second statement only if the **`if`**'s expression is *not* true.

One common (indeed, very common) idiom is to attach a second **`if`** to an **`else`**, and a second **`else`** to that second **`if`**, and so on. Because there can be at most one **`else`** for each **`if`**, we don't need a special `elif` or `elseif` keyword for this construction used in some other languages, and the logic can be much simplified.

Logical operators

This program also introduces a new class of operations, the *logical operators*. In the expression `c == ''|| c == '\n'|| c = '\t'`, the logical OR operator `||` joins the results of two sub-expressions such that if either of them is true, the value of the joined expression is true. Further, logical operators use *short circuit evaluation*; the left side is tested first, and if it's true (or false, in the case of the logical AND operator `&&`), the right side is not evaluated at all. This mechanism can be used to avoid doing expensive operations when they aren't necessary. Also, note that while we say "left side" and "right side" here, operators can be chained arbitrarily in this way, and they will be evaluated from left to right, stopping as soon as the value of the expression as a whole has been "decided."

The logical OR operator isn't the only one we've introduced; there's one more — the logical *negation operator*, `!` (usually pronounced as "not"). In the expression `!inword`, the negation operator inverts the logical meaning of the expression to its right. If `inword` is false (zero), `!inword` has the value 1; and if `inword` is true, `!inword` has the value 0. This is equivalent to the expression `inword != 0`, but since it's easier to read as a sentence, most programmers prefer it.

Character constants

It might at first appear that this program is comparing strings, but it isn't; the things we're comparing here aren't strings, they're *character constants*. Character constants, written inside single quotes, are another way to refer to a small integer value. For example, `'F'`, in the ASCII character set, has the value 70, but it's clear that it refers to the letter F instead of just a number. Just like in strings, escape sequences like `\n` work in character literals, but remember that this is exactly one number, not a sequence — `'\n'` refers to the ASCII value of the newline character, not the sequence used to make it.

Prefix and postfix operators

This program contains another new thing we haven't discussed yet. The expression `++nc` uses the *increment operator*, `++`; there's also the *decrement operator*, `--`. You can use them before or after the name of a variable to increment (add to) or decrement (subtract from) that variable. When placed before the variable name, as in `++nc`, the value of the expression is the new value of the variable. But when placed after the variable, as in `nc++`, the value of the expression is the *original* value of that variable, before you changed it.

In our example, the difference between pre-incrementing and post-incrementing the variable `nc` has no effect on the result of the operation (we're not checking the value of the variable inside the expression), so for now we can treat it as just another way to add 1 to a variable. That said, there are times when you may want exactly the behavior of the postfix version.

Parentheses and precedence

In the expression `(c = getchar())!= EOF`, the variable `c` is being assigned the return value of the function `getchar`, and then `c` is compared with the symbolic constant `EOF` (defined in the file `<stdio.h>`). The parentheses are important, because different operations have differing priorities, or *precedence*.

The `!=` operator has a higher precedence than `=`, so the expression `c = getchar()!= EOF` is actually equivalent to the expression `c = (getchar()!= EOF)`. If *that* expression had been evaluated, `c` would have been assigned a value of 0 or 1 depending on whether `getchar` returned a value that was the same as `EOF`. Needless to say, that's not very useful if we want to do something with the actual value returned by `getchar`.

This Chapter Is Not Complete Yet

2
DCC Characteristics

The language implementation

DCC's dialect of C is implemented mostly as described in the First Edition of *The C Programming Language*, by Kernighan and Ritchie (hereafter referred to as *K&R*).

Although this version of C follows the specification faithfully, there are some differences. The differences mostly reflect parts of C that are obsolete or constraints imposed by memory size limitations.

Differences from the specification

- Bit fields are not supported.
- Constant expressions for initializers may include arithmetic operators only if all the operands are of type **`int`** or **`char`**.
- The archaic forms of assignment operators, such as =+ or =*, which are still recognized by some C compilers, are not supported. Use the newer forms +=, *=, and so forth.
- The escape sequence for new-line, `\n`, refers to the ASCII "Carriage Return" (`13`, hex `$0D`) character used by OS-9 for end-of-line, not the ASCII "Linefeed" (`10`, hex `$0A`) character used by Unix. Programs which use `\n` for end-of-line (all programs found in *K&R*), will work normally, but programs from DOS or Windows which use `\r\n` to end a line will print two carriage returns.

Enhancements and extensions

The `direct` storage class

The 6809 microprocessor instructions for accessing memory via an index register or the stack pointer can be relatively short and fast when they are used in C programs to access **`auto`** (function local) variables or function arguments. The instructions used to access global variables are normally not very nice and are four bytes long (and, correspondingly, not fast).

However, the 6809 has a nice feature which helps considerably. Memory anywhere in a single *page*[1] may be accessed with fast, two-byte instructions.

This is called the *direct page*, and at any time its location is specified by the contents of the *direct page register* within the processor. The linkage editor sorts out where this could be, and the programmer only needs to specify for the compiler which variables should be in the direct page to give the maximum benefit in code size and execution speed.

To this end, a new storage class specifier is recognized by the compiler. In the manner of *K&R* page 192, the sc-specifier list is extended as follows:

sc–specifier:
```
auto
static
extern
register
typedef
direct
static direct
extern direct
```

The **direct** keyword may be used in place of one of the other sc-specifiers, and its effect is that the variable will be placed in the direct page. **direct** creates a global direct page variable. **extern direct** references an external-type direct page variable; and **static direct** creates a direct page variable without global scope. These new storage classes may *not* be used to declare function arguments.

Direct variables may be initialized, but will (as with other variables not explicitly initialized) have the value zero at the start of program execution. 255 total bytes are available on the direct page (the linker reserves one byte). If all direct variables occupy less than the full 255 bytes, any remaining global variables will occupy the balance and memory above if necessary. If too many bytes of storage are requested on the direct page, the linker will report an error and the programmer must reduce the number of direct variables to fit the 256 bytes addressable by the 6809. Library functions should not use direct variables, they are for user programs.

It should be kept in mind that direct is unique to this compiler, and it may not be possible to transport programs written using the direct storage class to other environments without modification.

Embedded assembly language

As versatile as C is, occasionally there are some things that can only be done (or done at maximum speed) in assembly language. The OS-9 C compiler permits user-supplied assembly-language statements to be directly embedded in C source programs.

[1] A 256-byte block of memory, aligned at an 8-bit boundary

A line beginning with `#asm` switches the preprocessor into a mode which passes all subsequent lines directly to the assembly language output, until a line beginning with **`#endasm`** is encountered. **`#endasm`** switches the mode back to normal. Care should be exercised when using this directive so that the correct code section is adhered to. Normal code from the compiler is in the **psect** (code) section. If your assembly code uses the **vsect** (variable) section, be sure to put an **endsect** directive at the end to leave the state correct for later compiler-generated code.

Control character escape sequences

The escape sequences for non-printing characters in character constants and strings (see *K&R* page 181) are extended as follows:

linefeed (LF): `\l` (lower case L)

This is to distinguish LF (hex 0A) from \n which on OS-9 is the same as \r (hex 0D).

bit patterns:

`\NNN`	(octal constant)
`\dNN`	(decimal constant)
`\xNN`	(hexadecimal constant)

For example, the following all have a value of 255 (decimal):

```
\377    \xff    \d255
```

Implementation-dependent characteristics

K&R frequently refer to characteristics of the C language whose exact operations depend on the architecture and instruction set of the computer actually used. This section contains specific information regarding this version of C for the 6809 processor.

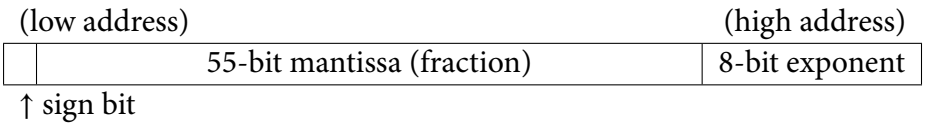
Data representation and storage requirements

Each variable type requires a specific amount of memory for storage. The sizes and formats of the basic types in bytes are illustrated in the table on the current page.

	Size	Internal form
char	1	two's complement binary
int	2	
long	4	
unsigned char	1	unsigned binary
unsigned int	2	
unsigned long	4	
float	4	binary floating point
double	8	

This compiler follows the PDP-11 implementation and format in that integral types are converted to their next highest convenient type through sign extension, or simple widening

in the case of unsigned types. That is, a **char** is converted to **int** by sign extension, while an **unsigned char** is widened to **unsigned int** without conversion. As on PDP-11, **int** (the "most convenient" integral type) is the same size as **short**, and **long float** is a synonym for **double**. The format for **double** values is as follows:



The format of the mantissa is sign-and-magnitude with an *implied* 1 bit at the sign bit position for a total of 56 bits of mantissa. The exponent is biased by 128. The format of a **float** is identical, except that the mantissa is three bytes long instead of 7. Conversion from **double** to **float** is carried out by truncating the least significant (right-most) four bytes of the mantissa. The reverse conversion is done by padding the least significant four mantissa bytes with zeros.

Register variables

One register variable may be declared in each function. The types permitted for register variables are **int**, **unsigned int**, and any pointer type. Invalid register variable declarations are ignored; i.e. the storage class is made **auto**. For further details, see *K&R* page 81.

Considerable savings in code size and speed can be made by judicious use of a register variable. The most efficient use is made of it for a pointer or a counter for a loop. However, if a register variable is used for a complex arithmetic expression, there is no saving. The U register is assigned to register variables.

Access to command-line parameters

The standard C argument vector `argc` and `argv` is available to the program's `main` function as described in *K&R* page 110. The start-up routine for C programs ensures that the parameter string passed to it by the parent process is converted into null-terminated strings as expected by the program. In addition, it will run together as a single argument any strings enclosed between single or double quotes (`''` or `""`). If either is a required part of a string argument, then the other should be used as a delimiter.

System calls and the standard library

Operating system calls

The system interface supports almost all the system calls of OS-9, and many of those of UNIX as well. In order to facilitate the portability of programs from UNIX, some of the calls use UNIX names rather than OS-9 names for the same function. There are a few UNIX calls that do not have exactly equivalent OS-9 calls. In these cases, the library function simulates the function of the corresponding UNIX call. In cases where there are OS-9 calls that do

not have UNIX equivalents, the OS-9 names are used. Details of the calls and a name cross-reference are provided in the DCC Library Reference manual.

The standard library

The C compiler includes a very complete library of standard functions. It is essential for any program which uses functions from the standard library to have the statement:

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

See the standard library manual for details on the functions provided.

Important Note If you need to format any long integers (using `printf` or any of its variations), the program **must** include a reference to the `pflinit` function somewhere. This is a memory-saving feature, so that programs that *don't* need to format **`long`**s don't waste memory on code to format them. Similarly, if values of type **`float`** or **`double`** are to be printed, the `pffinit` function must be referenced. These functions do nothing; existence of calls to them in a program informs the linker that the relevant routines are also needed.

Run-time arithmetic error handling

K&R leave the treatment of various arithmetic errors open, merely saying that it is machine dependent. This implementation deals with a limited number of error conditions in a special way; it should be assumed that the results of other possible errors are undefined.

Three new system error numbers are defined in `<errno.h>`:

```
#define EFPOVR  40 /* floating point overflow of underflow */
#define EDIVERR 41 /* division by zero */
#define EINTERR 42 /* overflow on conversion of float to long */
```

If one of these conditions occur, the program will send a signal to itself with the value of one of these errors. If not caught or ignored, the will cause termination of program with an error return to the parent process. However, the program can catch the interrupt using `signal()` or `intercept()` (see C System Calls), and in this case the service routine has the error number as its argument.

Achieving maximum program performance

Programming considerations

Because the 6809 is an 8/16 bit microprocessor, the compiler can generate efficient code for 8 and 16 bit objects (**`char`**s, **`int`**s, etc.). However, code for 32 and 64 bit values (**`long`**s, **`float`**s, **`double`**s) can be at least four times longer and slower. Therefore, don't use **`long`**, **`float`**, or **`double`** where an **`int`** or **`unsigned int`** will do.

The compiler can perform extensive evaluation of constant expressions provided they involve only constants of type **char**, **int**, and/or **unsigned** versions of those types. There is no constant expression evaluation at compile-time (except single constants and "casts" of them) where there are constants of type **long**, **float**, or **double**. Therefore, complex constant expressions involving these types are evaluated at run time by the compiled program. You should manually compute the value of constant expressions of these types if speed is essential.

The optimizer pass

The optimizer pass automatically occurs after the compilation pass. It analyzes the assembly source code text, removes redundant code, and searches for code sequences that can be replaced by shorter and faster equivalents. The optimizer tends to shorten object code by about 11%, with a significant increase in program execution speed. The optimizer is recommended for production versions of debugged programs. Because this pass takes additional time, the `-O` compiler option can be used to inhibit it during error-checking-only compilations.

The profiler

The profiler is an optional method used to determine the frequency of execution of each function in a C program. It allows you to identify the most-frequently used functions where algorithmic or C source code programming improvements will yield the greatest gains.

When the `-P` compiler option is selected, code is generated at the beginning of each function to call the profiler module (called `_prof`), which counts invocations of each function during program execution. When the program has terminated, the profiler automatically prints a list of all functions and the number of times each was called. The profiler slightly reduces program execution speed. See "prof.c" source for more information.

C compiler components and files

Compilation of a C program by DCC requires that the following files be present in the current execution directory (usually `/DD/CMDS`):

`dcc`	Compiler executive. Runs other programs.
`dcpp`	Macro preprocessor. Performs token replacement.
`dcc68`	Compiler proper. Generates assembly code.
`dco68`	Assembly optimizer. Cleans up assembly to improve speed and code size.
`rma`	Assembler. Converts assembly to machine code, saves in object files.
`rlink`	Linkage editor (linker). Links object files to form programs.

In addition, a file called `clib.l` contains the standard library, math functions, and systems library. The file `cstart.r` is the setup code for compiled programs. Both of these files

must be located in a directory named `LIB` on the *default system drive*[2].

When specifying `#include` files for the pre-processor to read in, if you use angle brackets (< and >), instead of quotation marks ", the file will be sought starting at the `DEFS` directory on the default system drive.

Temporary files

During compilation, a number of temporary files are created. If your system has a RAM disk mounted on `/R` or `/R0`, or if the directory `/DD/TMP` exists, then the compiler will place its temporary files in that location — otherwise, in the current data directory. It is important to ensure that enough space is available on the disk used for temporary files. As a rough guide, at least three times the number of blocks in the largest source file (and its included files) should be free.

The special identifiers `etext`, `edata`, and `end` are predefined in the linkage editor and may be used to establish the addresses of the end of executable text, initialized data, and uninitialized data respectively.

Running the compiler

The syntax of the command line which calls the compiler is:

`dcc` [options] *`file`*... [options]

One file at a time can be compiled, or a number of files may be compiled together. The compiler manages the compilation up to four stages: pre-processor, compilation to assembler code, assembly to relocatable code, and linking to binary executable code (in OS-9 memory module format).

The compiler accepts three types of source files, provided each name on the command line has the relevant postfix as shown below. Any of the file types below may be mixed on the command line.

Suffix	Usage
.c	C source file
.a	assembly language source file
.r	relocatable module
(none)	executable binary (OS-9 memory module)

Table 2.1: File Name Suffix Conventions

There are two modes of operation: multiple source file and single source file. The compiler selects the mode by inspecting the command line. The usual mode is single source and is specified by having only one source file name on the command line. Of course, more than one source file may be compiled together by using the `#include` facility in the source

[2]The system's default mass storage device, specified in the OS-9 `Init` module and usually the disk drive the system was booted from.

code. In this mode, the compiler will use the name obtained by removing the postfix from the name supplied on the command line, and the output file (and the memory module produced) will have this name. For example:

```
dcc prg.c
```

will leave an executable file called `prg` in the current execution directory.

The multiple source mode is specified by having more than one source file name on the command line. In this mode, the object code output file will have the name `output` in the current execution directory, unless a name is given using the `-f=` option (see below). Also, in multiple source mode, the relocatable modules generated as intermediate files will be left in the same directories as their corresponding source files with the postfixes changed to `.r`. For example:

```
dcc prg1.c /d0/fred/prg2.c
```

will leave an executable file called `output` in the current execution directory, one file called `prg1.r` in the current data directory, and `prg2.r` in the directory `/d0/fred`.

Compiler option flags

The compiler recognizes several command-line option flags which modify the compilation process where needed. All flags are recognized before compilation commences so the flags may be placed anywhere on the command line. Flags may be run together as in `-ro`, except where a flag is followed by something else; see `-f=` and `-d` for examples.

-a suppresses assembly, leaving the output as assembler code in a file whose name is postfix ".a".

-e=<number> Set the edition number constant byte to the *number* given. This is an OS-9 convention for memory modules.

-o inhibits the assembly code optimizer pass. The optimizer will shorten object code by about 11% with a comparable increase in speed and is recommended for production versions of debugged programs.

-p invokes the profiler to generate function frequency statistics after program execution.

-r suppresses linking library modules into an executable program. Outputs are left in files with postfix `.r`.

-M=<size> will instruct the linker to allocate *size* for data, stack, and parameter area. Memory size may be expressed in pages (an integer), or in kilobytes by appending `k` to an integer. For more details of the use of this option, see the Memory management section of this manual.

-l=<path> specifies a library to be searched by the linker ahead of the standard library and system interface.

-f=<path> overrides the above output file naming. The output will be put into a file at *path*. This flag does not make sense in multiple source mode if either the -a or -r flag is also present. The module will be called the last name in *path*.

-c will output the source code as comments inside the assembly code.

-s stops the generation of stack-checking code. -s should only be used with great care when the application is extremely time-critical and when the use of the stack by compiler-generated code is fully understood.

-d<sym>[=val] is equivalent to `#define sym val` written in the source file. -d is useful where different versions of a program are maintained in one source file and differentiated by means of the `#ifdef` or `#ifndef` preprocessor directives. If no *val* is supplied, the symbol will be expanded as the value 1. If a *val* is supplied, the expansion will be the value of *val*.

Example command lines

`dcc prg.c` compiles the C source file prg.c into an executable program named prg in the current execution directory.

`dcc -a prg.c` compiles the C source file prg.c into assembly source file prg.a in the current working directory.

`dcc -r prg.c` compiles the C source file prg.c into relocatable module prg.r in the current working directory.

`dcc prg1.c prg2.c prg3.c` compiles the C source files `prg1.c`, `prg2.c`, and `prg3.c` into an executable program named `output` in the current execution directory. The current *working* directory will now contain the files `prg1.r`, `prg2.r`, and `prg3.r`.

`dcc prg1.c prg2.a prg3.r` compiles the C source file `prg1.c`, assembles the assembly source file `prg2.a`, and links both with the relocatable module `prg3.r` into an executable program named `output` in the current execution directory. The current *working* directory will now contain the files `prg1.r` and `prg2.r`.

`dcc -a prg1.c prg2.c` compiles the C source files `prg1.c` and `prg2.c` into assembly source, leaving the resulting files `prg1.a` and `prg2.a` in the current working directory.

`dcc -f=prg prg1.c prg2.c` compiles the C source files `prg1.c` and `prg2.c` into relocatable modules `prg1.r` and `prg2.r` in the current working directory, then links them together into a program `prg` in the current execution directory.

3
Characteristics of compiled programs

The object code module

The compiler produces position-independent, reentrant 6809 code in a standard OS-9 memory module format. The format of an executable program module is shown below. Detailed descriptions of each section of the module are given on following pages.

Module Offset		Section Size
$00	Module header	8
$09	Execution offset	2
$0B	Storage size	2
$0D	Module name Executable code String literals	
	Initializing Data Size	2
	Initializing Data	
	Data-Text Reference Count	2
	Data-Text Reference Offsets	
	Data-Data Reference Count	2
	Data-Data Reference Offsets	
	CRC Check Value	3

Figure 3.1: OS-9 C memory module format

Module header

This is a standard OS-9 memory module header, with the Type/Language byte set to $11 (Program + 6809 Object Code), and the Attribute/Revision byte set to $81 (Reentrant + 1).

Execution offset

The execution offset is used by OS-9 to locate where to start execution of the program.

Storage size

Storage size is the initial default allocation of memory for the program's data, stack, and parameter area. For a full description of memory allocation, see the Memory management section located on page 27.

Module name

The module name is used by OS-9 as a handle for the system's module directory. The module name is followed by the Edition byte encoded in `cstart`, the "main line" of a C program. If you wish your program to have a different edition number, you may override it using the `-E=` option to `dcc`.

Information

Any strings preceded by the directive "info" in an assembly code file will be placed here. A major use of this facility is to place in the module the version number and/or a copyright notice. Note that the `#asm` preprocessor directive may be used in a C source file to enable the inclusion of this directive in the compiler-generated assembly code file.

Executable code

The machine code instructions of the program.

String literals

Quoted strings in the C source are placed here. They are in the null-terminated form expected by the functions in the standard library.

Note The definition of the C language assumes that strings are in the DATA area and are therefore subject to alteration without making the program non-reentrant. However, to avoid duplicating memory requirements (which would be necessary if they were in the data area), they are placed in the TEXT (executable) section of the module. Putting the strings in the executable section implies that no attempt should be made by a C program to alter string literals; instead, they should be copied first. The exception is the initialization of a global array of type **`char`**, like this:

```
char message[] = "Hello, world!\n";
```

This type of string will be found only in the array `message` in the data area and may be altered.

Initializing Data and its Size

If a C program contains initializers, the data for the initial values of the variables is placed in this section. The definition of C states that all uninitialized global and static variables have the value zero when the program starts running, so the startup routine of each C program first copies the data from the module into the data area and then clears the rest of the data memory to nulls.

Data References

No absolute addresses are known at compile time under OS-9, so where there are pointer values in the initializing data, they must be adjusted at run time so that they reflect the absolute values at that time. The startup routine uses the two data reference tables to locate

the values that need alteration and adjusts them by the absolute values of the bases of the executable code and data respectively.

For example, suppose there are the following statements in the program being compiled:

```
char *p = "I'm a string!";
char **q = &p;
```

These declarations tell the compiler that there is to be a `char` pointer variable, `p`, whose initial value is the address of the string and a pointer to a `char` pointer, `q`, whose initial value is the address of `p`. The variables must be in the DATA section of memory at run time because they are potentially alterable, but absolute addresses are not known until run time, so the values that `p` and `q` must have are not known at compile time. The string will be placed by the compiler in the TEXT section and will not be copied out to DATA memory by the startup routine. The initializing data section of the program module will contain entries for `p` and `q`. They will have as values the offsets of the string from the base of the TEXT section and the offset of the location of `p` from the base of the DATA section respectively.

The startup routine will first copy all the entries in the initializing data section into their allotted places in the DATA section. Then it will scan the data-text reference table for the offsets of values that need to have the addresses of the base of the TEXT section added to them. Among these will be the `p` which, after updating, will point to the string which is in the TEXT section. Similarly, after a scan of the data-data references, `q` will point to (contain the absolute address of) `p`.

Memory management

The C compiler and its support programs have default conditions such that the average programmer need not be concerned with details of memory management. However, there are situations where advanced programmers may wish to tailor the storage allocation of a program for special situations. The following information explains in detail how a C program's data area is allocated and used.

Typical C program memory map

A storage area is allocated by OS-9 when the C program is executed. The memory layout may be seen in Figure 3.2. The memory layout on a Level 2 machine is similar, except there are no "low addresses" (the Y and DP registers are set to zero)

The overall size of this memory area is defined by the Storage size value stored in the program's module header. When running the program, the user may override this size to grant the program additional memory using the OS-9 Shell, however a full explanation of this option is beyond the scope of this book.

The parameter area is where the parameter string from the calling process (typically the Shell) is placed by the system. The initializing routine for C programs converts the param-

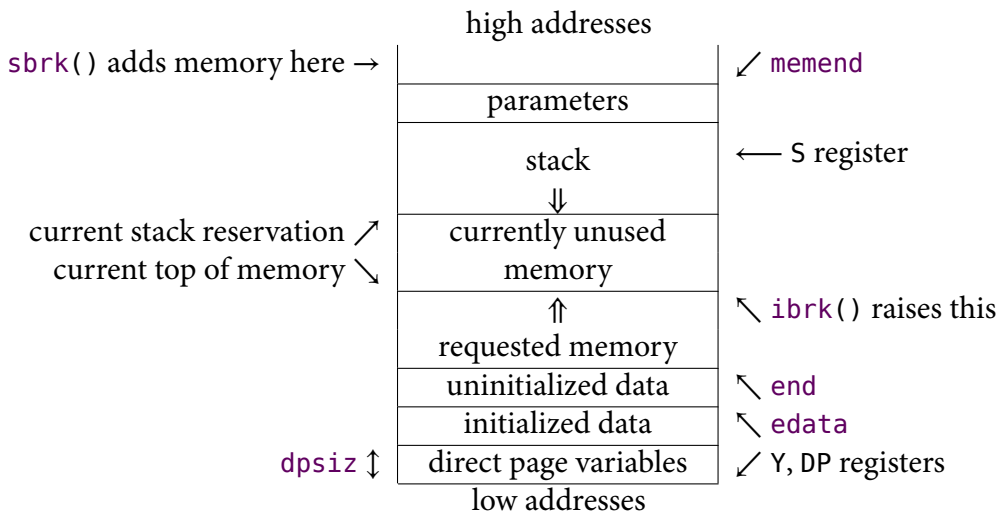


Figure 3.2: OS-9 Level 1 C Memory Layout

eters into null-terminated strings and makes pointers to them available to `main()` via `argc` and `argv`.

The stack area is the currently reserved memory for exclusive use of the stack. As each C function is entered, a routine in the system interface is called to reserve enough stack space for the use of the function and additional 64 bytes. These 64 bytes are for the use of user-written assembly code functions and/or the system interface and/or arithmetic routines. A record is kept of the lowest address so far granted for the stack. If the area requested would not bring this lower, then the C function is allowed to proceed. If the new lower limit would mean that the stack area would overlap the data area, the program stops with the message:

```
*** Stack Overflow ***
```

on the standard error output. Otherwise, the new lower limit is set, and the C function resumes as before.

The direct page variables area is where variables reside that have been defined with the storage class **`direct`** in the C source code or in a `vsect dp` segment in assembly code source. Notice that the size of this area is always at least one byte (to ensure that no pointer to a variable can have the value NULL or 0) and that it is not necessarily 256 bytes.

The uninitialized data area is where the remainder of the uninitialized program variables reside. These two areas are, in fact, cleared to all zeros by the program entry routine. The initialized data area is where the initialized variables of the program reside. There are two globally defined values which may be referred to: `edata` and `end`, which are the addresses of one byte higher than the initialized data and one byte higher than the uninitialized data respectively. Note that these are not variables; the values may be accessed from C by using the & operator, as in:

```
high = &end;
low = &edata;
```

or from assembler:

```
leax    end,y
stx     high,y
```

The `Y` register points to the base of the data area, and variables are addressed using `Y`-offset indexed instructions.

When the program starts running, the remaining memory is assigned to the "free" area. A program may call `ibrk()` to request additional working memory (initialized to zeros) from the free memory area. Alternately, more memory can be dynamically obtained using `sbrk()`, which requests additional memory from the operating system and returns the new lower bound. If this fails because OS-9 refuses to grant more memory *for any reason*, `sbrk()` will return −1 as an **`int`**.

Compile-time memory allocation

If not instructed otherwise, the linker will automatically allocate 1 kilobyte more than the total size of the program's variables and strings. This size will normally be adequate to cover the parameter area, stack requirements, and standard library's file buffers. The allocation size may be altered when using the compiler by using the `-m` option on the command line. The memory requirements may be stated in pages, for example:

```
dcc prg.c -m=2
```

allocates 512 bytes extra, or in kilobytes, for example:

```
dcc prg.c -m=10k
```

The linker will ignore the request if the size is less than 256 bytes.

The following rules can serve as a rough guide to estimate how much memory to specify:

1. The parameter area should be large enough for any anticipated command line string.

2. The stack should not be less than 128 bytes and should take into account the depth of function calling chains and any recursion.

3. All function arguments and local variables occupy stack space and each function entered needs 4 bytes more for the return address and temporary storage of the calling function's register variable.

4. Free memory is requested by the standard library I/O functions for buffers at the rate of 256 bytes per accessed file. This buffering does not apply to the lower level service request I/O functions such as `open()`, `read()`, and `write()`; nor does it apply to the `stderr` file which is always unbuffered, but it does apply to both `stdin` and `stdout` (see the Standard Library documentation).

A good method for getting a feel for how much memory is needed by your program is to allow the linker to set the memory size to its usually conservative value. Then, if the program runs with a variety of input satisfactorily but memory is limited on the system, try reducing the allocation at the next compilation. If a stack overflow occurs or an `ibrk()` call returns −1, then try increasing the memory next time. You aren't likely to damage the system by getting it wrong, but data may be lost if the program runs out of space at a crucial time. It pays to err on the generous side.

4
Interfacing with Basic09

It's possible to make the object code generated by the C compiler callable from the BASIC09 **RUN** statement. Certain portions of a BASIC09 program written in C can have a dramatic effect on execution speed. To effectively use this feature, you should be familiar with both C and BASIC09 internal data representation and calling conventions.

The C type **int** and BASIC09 type **INTEGER** are identical; both are two-byte two's complement integers. The formats of the C type **char** and BASIC09 types **BYTE** and **BOOLEAN** are also identical. Keep in mind that C will sign-extend characters for comparisons, yielding the range $-128 \ldots 127$.

BASIC09 strings are terminated by `0xff` (255), while C strings are terminated by `0x00` (0). If the BASIC09 string is of maximum length, the terminator is not present. Therefore, string length as well as termination checks must be performed on BASIC09 strings when processing them with C functions.

The floating point formats used by C and BASIC09 are not directly compatible. Since both use a binary floating point format, however, it is possible to automatically translate BASIC09 **REAL**s to C **double**s and vice versa.

Multi-dimensional arrays are stored by BASIC09 in a different manner than in C. Multi-dimensional arrays are stored by BASIC09 in a column-wise manner; C stores them row-wise. Consider the following example:

Basic09 matrix In the BASIC09 declaration **DIM** `array(5,3):`**INTEGER**, the elements in consecutive memory locations (read left to right, line by line) are stored as:

$$\begin{array}{lllll} (1,1) & (2,1) & (3,1) & (4,1) & (5,1) \\ (1,2) & (2,2) & (3,2) & (4,2) & (5,2) \\ (1,3) & (2,3) & (3,3) & (4,3) & (5,3) \end{array}$$

C matrix For the C declaration **int** `array[5][3]`, the elements in consecutive memory locations for *this* matrix are stored as:

(1, 1) (1, 2) (1, 3)
(2, 1) (2, 2) (2, 3)
(3, 1) (3, 2) (3, 3)
(4, 1) (4, 2) (4, 3)
(5, 1) (5, 2) (5, 3)

Therefore, to access Basic09 matrix elements in C, the subscripts must be transposed. To access element `array(4,2)` in Basic09, use `array[2][4]` in C.

The details on interfacing Basic09 to C are perhaps best described by example. The remainder of this chapter is a mini-tutorial demonstrating the process, starting with simple examples and working up to more complex ones.

Example 1 – Simple Integer Arithmetic

This first example illustrates a simple case. Write a C function to add an integer value to three integer variables:

```
addints(cnt,value,s1,arg1,s2,arg2,s2,arg3,s4)
int *value,*arg1,*arg2,*arg3;
{
    *arg1 += *value;
    *arg2 += *value;
    *arg3 += *value;
}
```

That's the C function. The name of the function is `addints`. The name is information for C and `rlink`; Basic09 will not know anything about the name.

The Basic09 Reference manual describes how Basic09 passes parameters to machine language modules. Since Basic09 and C pass parameters in a similar fashion, it is easy to access Basic09 values from C. The first parameter on the Basic09 stack is a two-byte count of the number of following parameter pairs; each pair consists of an address, and the size of the pointed-to value. For most C functions, the parameter count and pair size is not used. The address, however, is the useful piece of information. The address is always declared in the C function to be a "pointer to …" type, because Basic09 *always* passes arguments *by reference*[1], even for constant values. The arguments `cnt`, `s1`, `s2`, `s3`, and `s4` are just placeholders to indicate the presence of the parameter count and argument sizes on the stack. If you wish to check that the passed-in arguments agree with the function's definition, you may check that `cnt` contains the value 4.

The line `int *value,*arg1,*arg2,*arg3;` declares the arguments (in this case, all of them are "pointers to **int**"), so the compiler will generate the correct code to access the Basic09 values. The remaining lines increment each argument by the passed value. Notice

[1] that is, through a pointer

that a simple arithmetic operation is performed here (addition), so C will not have to call a library function to do the operation.

To compile this function, the following C compiler command line is used:

```
dcc bt1.c -rs
```

The `-r` option causes the compiler to leave `bt1.r` as output, ready to be linked. The `-s` option suppresses the call to the stack-checking function. Since we will be making a module for Basic09, `cstart.r` will not be used. Therefore, no initialized data, static data, or stack checking is allowed. More on this later.

The `bt1.r` file must now be converted into a loadable module that Basic09 can link to by using a special linking technique as follows:

```
rlink bt1.r -b=addints -o=addints
```

This command tells the linker to read `bt1.r` as input. The option `-b=addints` tells the linker to make the output file a module that Basic09 can link to and that the function `addints` is to be the entry point in the module. You may give many input files to `rlink` in this mode. It resolves references in the normal fashion. The name given to the -b= option indicates which of the functions is to be entered directly by the Basic09 RUN command. The option `-b=addints` says what the name of the output file is to be, in this case `addints`. This name *should* be the name used in the Basic09 **RUN** command to call the C procedure. The name given in the `-o=` option *must* be the name of the procedure to **RUN**, while the `-b=` option is information for the linker, so that it can fill in the correct module entry point offset.

Enter the following Basic09 program:

```
PROCEDURE btest
DIM i,j,k:INTEGER
i=1
j=132
k=-1033
RUN addints(4,i,j,k)
PRINT i,j,k
END
```

When this procedure is RUN, it should print:

```
5        136      -1029
```

indicating that our C function worked!

Example 2 – More Complex Integer Arithmetic

The next example shows how static memory can be used. Take the C function from the previous example and modify it to add the number of times it has been entered to the increment:

```
static int entcnt;
```

```
addints(cnt,cmem,cmemsiz,value,s1,arg1,s2,arg2,s2,arg3,s4)
char *cmem;
int *value,*arg1,*arg2,*arg3;
{
#asm
    ldy 6,s base of static area
#endasm
    int j = *value + entcnt++;

    *arg1 += j;
    *arg2 += j;
    *arg3 += j;
}
```

This example differs from the first in a number of ways. The line **static int** entcnt ; defines an integer value named entcnt global to bt2.c. The parameter cmem and the line **char** *cmem; indicate a character array. The array will be used in the C function for global/static storage.

C accesses non-auto and non-register variables indexed off the Y register. Normally, cstart.r takes care of setting this up. However, since cstart.r will not be used for this Basic09-callable function, we have to take measures to make sure the Y register points to a valid and sufficiently large area of memory. The line ldy 6,s is assembly language code embedded in the C source, which loads the Y register with the first parameter passed by Basic09. If the first parameter in the Basic09 RUN statement is an array, and the "ldy 6,s" is placed *immediately* after the { opening the function body, the offset will always be "6,s". Note the line beginning "int j = ...". This line uses an initializer which, in this case, is allowed because j is of class "auto". No storage classes other than "auto" and "register" may be initialized in Basic09-callable C functions.

To compile this function, the following C compiler command line is used:

```
dcc bt2.c -rs
```

Again, the -r option leaves bt2.r as output and the -s option suppresses stack checking.

Normally, the linker considers it to be an error if the "-b=" option appears and the final linked module requires a data memory allocation. In our case here, we require a data memory allocation and we will provide the code to make sure everything is set up correctly. The "-t" linker option causes the linker to print the total data memory requirement so we can allow for it rather than complaining about it. Our linker command line is:

```
rlink bt2.r -o=addints -b=addints -r
```

The linker will respond with BASIC09 static data size is 2 bytes". We must make sure cmem points to at least 2 bytes of memory. The memory should be zeroed to conform to C specifications.

Enter the following BASIC09 program:

```
PROCEDURE btest
DIM i,j,k,n;INTEGER
DIM cmem(10):INTEGER
FOR i=1 TO 10
     cmem(i)=0
NEXT i
FOR n=1 TO 5
     i=1
     j=132
     k=-1033
     RUN addints(cmem,4,i,j,k)
     PRINT i,j,k
NEXT n
END
```

This program is similar to the previous example. Our area for data memory is a 10-integer array (20 bytes) which is way more than the 2 bytes for this example. It is better to err on the generous side. Cmem is an integer array for convenience in initializing it to zero (per C data memory specifications). When the program is run, it calls addints 5 times with the same data values. Because addints add the number of times it was called to the value, the i,j,k values should be 4+number of times called. When run, the program prints:

```
     5       136     -1029
     6       137     -1028
     7       138     -1027
     8       139     -1026
     9       140     -1025
```

Works again!

Example 3 – Simple String Manipulation

This example shows how to access BASIC09 strings through C functions. For this example, write the C version of SUBSTR.

```
/* Find substring from BASIC09 string:
        RUN findstr(A$,B$,findpos)
    returns in fndpos the position in A$ that B$ was found or
    0 if not found.  A$ and B$ must be strings, fndpos must be
```

```
    INTEGER.
 */
findstr(cnt,string,strcnt,srchstr,srchcnt,result);
char *string,*srchstr;
int strcnt, srchcnt, *result;
{
    *result = finder(string,strcnt,srchstr,srchcnt);
}

static finder(str,strlen,pat,patlen)
char *str,*pat;
int strlen,patlen;
{
    int i;
    for(i=1;strlen-- > 0 && *str!=0xff; ++i)
        if(smatch(str++,pat,patlen))
            return i;
}

static smatch(str,pat,patlen)
register char *str,*pat;
int patlen;
{
    while(patlen-- > 0 && *pat != 0xff)
        if(*str++ != *pat++)
            return 0;
    return 1;
}
```

Listing 4.1: bt3.c

Compile this program:

```
dcc bt3.c -rs
```

And link it:

```
rlink bt3.r -o=findstr -b=findstr
```

The Basic09 test program is:

```
PROCEDURE btest
DIM a,b:STRING[20]
DIM matchpos:INTEGER
LOOP
```

```
    INPUT "String ",a
    INPUT "Match  ",b
    RUN findstr(a,b,matchpos)
    PRINT "Matched at position ",matchpos
ENDLOOP
```

When this program is run, it should print the position where the matched string was found in the source string.

Example 4 – Quicksort

The next example programs demonstrate how one might implement a quicksort written in C to sort some BASIC09 data.

C integer quicksort program:

```
#define swap(a,b) { int t; t=a; a=b; b=t; }

/*  qsort to be called by BASIC09:
    dim d(100):INTEGER  any size INTEGER array
    run cqsort(d,100)   calling qsort.
*/

qsort(argcnt,iarray,iasize,icount,icsiz)
int argcnt,     /* BASIC09 argument count */
    iarray[],   /* Pointer to BASIC09 integer array */
    iasize,     /* and it's size */
    *icount,    /* Pointer to BASIC09 (sort count) */
    icsiz;      /* Size of integer */
{
    sort(iarray,0,*icount); /* initial qsort partition */
}

/* standard quicksort algorithm from Horowitz-Sahni */
static sort(a,m,n)
register int *a,m,n;
{
    register i,j,x;

    if (m < n) {
        i = m;
        j = n + 1;
        x = a[m];
```

```
        for(;;) {
            do i += 1; while (a[i] < x);    /* left partition */
            do j -= 1; while (a[j] > x);    /* right partition */
            if(i < j)
                swap(a[i],a[j])             /* swap */
            else break;
        }
        swap(a[m],a[j]);
        sort(a,m,j-1);                      /* sort left */
        sort(a,j+1,n);                      /* sort right */
    }
}
```

Listing 4.2: qsortb.c

The BASIC09 program is:

```
PROCEDURE sorter
DIM i,n,d(1000):INTEGER
n=1000
i=RND(-(PI))
FOR i=1 TO n
    d(i):=INT(RND(1000))
NEXT i
PRINT "Before:"
RUN prin(1,n,d)
RUN qsortb(d,n)
PRINT "After:"
RUN prin(1,n,d)
END

PROCEDURE prin
PARAM n,m,d(1000):INTEGER
DIM i:INTEGER
FOR i=n TO m
    PRINT d(i); " ";
NEXT i
PRINT
END
```

C string quicksort program:

```
/*  qsort to be called by BASIC09:
```

```
    dim cmemory:STRING[10]  This should be at least as large as
                            the linker says the data size should
                            be.
    dim d(100):INTERGER     Any size INTEGER array.

    run cqsort(cmemory,d,100) calling qsort. Note that the pro-
                            cedure name run in the linked OS-9
                            subroutine module. The module name
                            need not be the name of the C func-
                            tion.
*/

int maxstr;     /* string maximum length */

static strbcmp(str1,str2)   /* basic09 string compare */
register char *str1,*str2;
{
    int maxlen;

    for (maxlen = maxstr; *str1 == *str2 ;++str1)
        if (maxlen-- >0 || *str2++ == 0xff)
            return 0;
    return (*str1 - *str2);
}

cssort(argcnt,stor,storsiz,iaarray,iasize,elemlen,elsiz,
             icount,icsiz)
int argcnt;         /* BASIC09 argument count */
char *stor;         /* Pointer to string (C data storage) */
char iarray[];      /* Pointer to BASIC09 integer array */
int  iasize,        /* and it's size */
    *elemlen,       /* Pointer integer value (string length) */
    elsiz,          /* Size of integer */
    *icount,        /* Pointer to integer (sort count) */
    icsiz;          /* Size of integer */
{
/*  The following assembly code loads Y with the first
    arg provided by BASIC09.  This code MUST be the first code
    in the function after the declarations.  This code assumes the
    address of the data area is the first parameter in the BASIC09
    RUN command. */
```

```
#asm
    ldy 6,s     get addr for C storage
#endasm

/*  Use the C library qsort function to do the sort. Our
    own BASIC09 string compare function will compare the strings.
*/

    qsort(iarray,*icount,maxstr=*elemlen,strbcmp);
}

/* define stuff cstart.r normally defines */
#asm
_stkcheck:
        rts     dummy stack check function

        vsect
errno:  rmb 2   C function system error number
_flacc: rmb 8   C library float/long accumulator
        endsect
#endasm
```

Listing 4.3: cssort.c

The Basic09 calling programs: (words file contains strings to sort)

```
PROCEDURE ssorter
DIM a(200):STRING[20]
DIM cmemory:STRING[20]
DIM i,n:INTEGER
DIM path:INTEGER
OPEN #path,"words":READ

n=100
FOR i=1 TO n
    INPUT #path,a(i)
NEXT i
CLOSE #path
RUN prin(a,n)
RUN cssort(cmemory,a,20,n)
RUN prin(a,n)
END
```

```
PROCEDURE prin
PARAM a(100):STRING[20]; n:INTEGER
DIM i:INTEGER
FOR i=1 TO n
    PRINT i; " "; a(i)
NEXT i
PRINT i
END
```

Example 5 – Floating Point

The next example shows how to access BASIC09 reals from C functions:

```
flmult(cnt,cmemory,cmemsiz,realarg,realsize)
int cnt;            /* number of arguments */
char *cmemory;      /* pointer to some memory for C use */
double *realarg;    /* pointer to real */
{
#asm
    ldy 6,s get static memory address
#endasm

    double number;

    getbreal(&number,realarg);  /* get the BASIC09 real */
    number *= 2.;               /* number times two*/
    putbreal(realarg,&number);  /* give back to BASIC09 */

}

/*  getreal(creal,breal)
    get a 5-byte real from BASIC09 format to C format */

getbreal(creal,breal)
double *creal,*breal;
{
    register char *cr,*br;  /* setup some char pointers */

    cr = creal;
    br = breal;
#asm
```

```
*  At this point U reg contains address of C double
*                  0,s contains address of BASIC09 real
    ldx 0,s     get address of B real

    clra        clear the C double
    clrb
    std 0,u
    std 2,u
    std 4,u
    stb 6,u
    ldd 0,x
    beq g3      BASIC09 real is zero

    ldd 1,x     get hi B mantissa
    and a #$7f  clear place for sign
    std 0,u     put hi C matissa
    ldd 3,x     get lo B mantissa
    andb #$fe   mask off sign
    std 2,u     put lo C mantissa
    lda 4,x     get B sign byte
    lsra        shift out sign
    bcc g1
    lda 0,u     get C sign byte
    ora #$80    tun on sign
    sta 0,u     put C sign byte
g1  lda 0,x     get B exponent
    suba #128   excess 128
    sta 7,u     put C exponent
g3  clra        clear carry
#endasm

}

/*  putbreal(breal,creal)
    put C format double into a 5-byte real from BASIC09 */

putbreal(breal,creal)
double *breal,*creal;
{
    register char *cr,*br;  /* setup some pointers */
```

```
    cr = creal;
    br = breal;
#asm
*   At this point U reg contains address of C double
*                   0,s contains address of BASIC09 real
    ldx 0,s     get address of B real

    lda 7,u     get C exponent
    bne p0      not zero?
    clra        clear the BASIC09
    clrb        real
    std 0,x
    std 2,x
    std 4,x
    bra p3      and exit

p0  ldd 0,u     get hi C mantissa
    ora #$80    this bit always on for normalized real
    std 1,x     put hi B mantissa
    ldd 2,u     get lo C mantissa
    std 3,x     put lo B mantissa
    incb        round mantissa
    bne p1
    inc 3,x
    bne p1
    inc 2,x
    bne p1
    inc 1,x
p1  andb #$fe   turn off sign
    stb 4,x     put B sign byte
    lda 0,u     get C sign byte
    lsla        shift out sign
    bcc p2      bra if positive
    orb #$01    turn on sign
    stb 4,x     put B sign byte
p2  lda 7,u     get C exponent
    adda #128   less 128
    sta 0,x     put B exponent
p3  clra        clear carry
#endasm
}
```

```
/* replace cstart.r definitions for BASIC09 */
#asm
_stkcheck:
_stkchec:
 rts

 vsect
_flacc: rmb 8
errno: rmb 2
 endsect
#endasm
```

Listing 4.4: flmult.c

BASIC09 calling program:

```
PROCEDURE btest
DIM a:REAL
DIM i:INTEGER
DIM cmemory:STRING[32]
a=1.
FOR i=1 TO 10
  RUN flmult(cmemory,a)
  PRINT a
NEXT i
END
```

Example 6 – Matrix Elements

The last program is an example of accessing BASIC09 matrix elements. The C program:

```
matmult(cnt,cmemory,cmemsiz,matxaddr,matxsize,scalar,scalsize)
char *cmemory;       /* pointer to some memory for C use */
int matxaddr[5][3];  /* pointer a double dim integer array */
int *scalar;         /* pointer to integer */
{
#asm
 ldy 6,s get static memory address
#endasm

     int i,j;
```

```
     for(i=0; i<5; ++i)
          for(j=1; j<3; ++j)
               matxaddr[j][i]  *=  *scalar;  /* multiply by value */
}
#asm
_stkcheck:
_stkchec:
 rts

 vsect
_flacc: rmb 8
errno: rmb 2
 endsect
#endasm
```

BASIC09 calling program:

```
PROCEDURE btest
DIM im(5,3):INTEGER
DIM i,j:INTEGER
DIM cmem:STRING[32]
FOR i=1 TO 5
    FOR j=1 TO 3
        READ im(i,j)
    NEXT j
NEXT i
DATA 11,13,7,3,4,0,5,7,2,8,15,0,0,14,4
FOR i=1 TO 5
    PRINT im(i,1),im(i,2),im(i,3)
NEXT i
PRINT
RUN matmult(cmem,im,64)
FOR i=1 TO 5
    PRINT im(i,1),im(i,2),im(i,3)
NEXT i
END
```

5
RMA Quick Reference

This appendix gives a summary of the operation of the Relocating Macro Assembler (RMA). This appendix and the example assembly source files supplied with the C compiler should provide the basic information on how to use the "Relocating Macro Assembler" to create Relocatable Object Format files (ROF). It is further assumed that you are familiar with the 6809 instruction set and mnemonics. See the Relocating Macro Assembler manual for a more detailed description. The main function of this appendix is to enable the reader to understand the output produced by RMA, not to fully document it.

The Relocating Macro Assembler allows programs to be compiled separately and then linked together, and it also allows macros to be defined within programs.

Differences between the Relocating Macro Assembler (RMA) and the Microware Interactive Assembler (MIA, often called ASM):

RMA does not have an interactive mode. Only a disk file is allowed as input.

RMA output is an ROF file. The ROF file must be processed by the linker, `rlink`, to produce an executable OS-9 memory module. The layout of a ROF file is described later.

RMA has a number of new directives to control the placement of code and data in the executable module. Since RMA does not produce memory modules, the MIA directives `mod` and `emod` are not present. Instead, new directives `psect` and `vsect` control the allocation of code and data areas by the linker.

RMA has no equivalent to the MIA `setdp` directive. Data (and DP) allocation is handled by the linker.

Symbolic Names

A symbolic name is valid if it consists of from one to nine uppercase or lowercase characters, decimal digits or the characters `$`, `_`, `.`, or `@`. RMA does not fold lowercase letters to uppercase. The names `Hi.you` and `HI.YOU` are distinct names.

Label field

If a symbolic name in the label field of a source statement is followed by a : (colon), the name will be known *globally* (by all modules linked together). If no colon appears, the name will be known only in the PSECT in which it was defined. PSECT will be described later.

Undefined names

If a symbolic name is used in an expression without first being defined, RMA assumes the name is defined external to the PSECT. RMA will record information about the reference, so the linker can adjust the operand accordingly. External names cannot appear in operand expressions for assembler directives.

Listing format

```
00098 0032 59        +         rolb
00117 0045=17ffb8      label   lbsr  _dmove     Comment
^     ^   ^^         ^ ^       ^     ^          ^
|     |   ||         | |       |     |          Start of comment
|     |   ||         | |       |     Start of operand
|     |   ||         | |       Start of mnemonic
|     |   ||         | Start of label
|     |   ||         A "+" indicates a line generated by a macro
|     |   ||         expansion.
|     |   |Start of object code bytes.
|     |   An "=" here indicates that the operand contains an
|     |   external reference.
|     Location counter value
Line number.
```

Sections

The RMA supports three types of section:

A **PSECT,** or program section, describes a relatively complete program unit; whether a complete program or a subroutine, a single PSECT contains the program code and any data internal to that program unit.

A **VSECT,** or variable section, is declared within a PSECT and written to the ROF object defined by that PSECT, describes the data local to the containing program section. When bundled into a program module, all linked VSECTs are given storage in the program's data area. If a VSECT's variable data must have initial values other than

zero, that initial data must be copied into its final location from somewhere in the program module, so it takes up memory twice.

A **CSECT,** or constant section, declares a series of zero or more symbolic names along with corresponding values. CSECTs never generate storage in the program module, they only provide symbolic names for values.

Section Location Counters

Each section contains the following location counters:

PSECT instruction location counter

VSECT initialized direct page location counter
non-initialized direct page location counter
initialized data location counter
non-initialized data location counter

CSECT base offset counter

Section Directives

RMA supports three types of sections. The **psect** directive indicates the beginning of a new ROF object; it initializes the instruction and data location counters, and assembles code into the ROF object code area. The **vsect** directive causes RMA to change to the data location counters and place any generated code into the appropriate ROF data area. The **csect** directive initializes a base value for assigning offsets to symbols. The end of these sections is indicated by using the **endsect** directive. Also, if you wish, **endsect** may be shortened to **ends**.

The source statements placed in a particular section cause the linker to perform a function appropriate for the statement. Therefore, the mnemonics allowed within a section are restricted as follows:

- These mnemonics are allowed inside or outside any section:
  nam opt ttl pag spc use fail rept endr ifeq ifne iflt ifle ifge ifgt ifp1 endc equ set macro endm csect endsect
- Within a CSECT: **rmb**
- Within a PSECT:
  any 6809 instruction mnemonic
  fcc fdb fcs fcb rzb vsect endsect os9 end
- Within a VSECT:
  rmb fcc fdb fcs fcb rzb endsect

PSECT Directive

The main difference between PSECT and MOD is that MOD sets up information for OS-9 and PSECT sets up information for the linker, `rlink`.

PSECT { *name*,*typelang*,*attrrev*,*edition*,*stacksize*,*entrypoint* }

name	Up to 20 bytes (any printable character except space or comma); a name to be used by the linker to identify this PSECT. This name need not be distinct from all other PSECTs linked together, but it helps to identify PSECTs the linker has a problem with if the names are different.
typelang	byte expression for the executable module type/language byte. If this PSECT is not a "mainline" (a module that has been designed to be forked to) module, this byte must be zero.
attrrev	byte expression for executable module attribute/revision byte.
edition	byte expression for executable module edition byte.
stacksize	word expression estimating the amount of stack storage required by this psect. The linker totals this value in all PSECTs to appear in the executable module and adds this value to any data storage requirement for the entire program.
entrypoint	word expression entrypoint offset for this PSECT. If the PSECT is not a mainline odule, this should be set to zero.

PSECT must have either no operand list or an operand list containing a name and five expressions. If no operand list is provided, the PSECT name defaults to "program" and all other expressions to zero. The can only be one PSECT per assembly language file.

The PSECT directive initializes all counter orgs and marks the start of the program module. No VSECT data reservations or object code may appear before or after the PSECT/ENDSECT block.

Example:

```
psect myprog,Prgrm+Objct,Reent+1,Edit,0,progent
psect another_prog,0,0,0,0,0
```

VSECT Directive

VSECT {DP}

The VSECT directive causes RMA to change to the data location counters. If DP appears after VSECT, the direct page counters are used, otherwise the non-direct page data is used. The RMB directive within this section reserves the specified number of bytes in the appropiate uninitialized data section. The fcc, fdb, fcs, fcb and rzb (reserve zeroed bytes) directives place data into the appropiate initialized data section. If an operand for fdb or fcb contains an external reference, this information is placed in the external reference part of

the ROF to be adjusted at link or execution time. ENDSECT marks the end of the VSECT block. Any number of VSECT blocks can appear within a PSECT. Note, however, that the data location counters maintain their values between one VSECT block and the next. Since the linker handles the actual data allocation, there is no facility provided to adjust the data location counters.

CSECT Directive

CSECT {expression}

The CSECT directive provides a means for assigning consecutive offsets to labels without resorting to EQUs. If the expression is present, the CSECT base counter is set to that value, otherwise it is set to zero.

RZB statement

RZB expression

The reserve zeroed bytes pseudo-instruction generates sequences of zero bytes in the code or initialized data sections, the number of which is specified by the expression.

Comparison Between ASM and the RMA

The following two program examples simply fork a BASIC09. The purpose of the examples are to show some of the differences in the new relocating assembler. The differences are apparent.

```
* this program forks a basic09
        ifp1
        use ..../defs/os9defs.a
        endc

Prgrm   equ     $10
Objct   equ     $01

stk     equ     200
        psect   rmatest,$11,$81,0,stk,entry

name    fcs     /basic09/
prm     fcb     $D
prmsize equ     *-prm

entry   leax    name,pcr
        leau    prm,pcr
```

```
        ldy     #prmsize
        lda     #Prgrm+Objct
        clrb
        os9     F$Fork
        os9     F$Wait
        os9     F$Exit
        endsect
```

Macro Interactive Assembler Source

```
        ifp1
        use defsfile
        endc

        mod     siz,prnam,type,revs,start,size
prnam   fcs     /testshell/
type    set     Prgm+Objct
revs    set     ReEnt+1

        rmb     250
        rmb     200
name    fcs     /basic09/
prm     fcb     $D
prmsize equ     *-prm
size    equ     .
start   equ     *
        leax    name,pcr
        leau    prm,pcr
        ldy     #prmsize
        lda     #Prgrm+Objct
        clrb
        os9     F$Fork
        os9     F$Wait
        os9     F$Exit
        emod
siz     equ .
```

Introduction to Macros

In programming applications it is frequently necessary to use a repeated sequence or pattern of instructions in many different places in a program. For example, suppose a group of

program statements creates a file a number of times throughout the program. The code might look like the following statements:

```
leax    name,pcr
lda     #$02
ldb     #$03
os9     I$Create
```

The sequence must be replicated each time that a new file is created. A macro assembler eliminates the need for coding duplicate statement patterns by allowing the programmer to define macro instructions that are equivalent to longer code sequences.

When a macro is called, it is the same as calling a subroutine to perform a defined function. A macro produces in-line code that is inserted into the normal flow of the program beginning at the location of the macro call. The statements that may be generated by a macro are generally unrestricted, and the statements may contain substitutable arguments.

Operations

Macro Definition

A macro definition consists of three sections:

```
label   MACRO   * header: assigns the name 'label' to the macro
        ...     * body: contains the macro statements
        ENDM    * terminator: ends the macro
```

A macro can have up to nine arguments (`\1` to `\9`) in the operand fields. Arguments may be used to refer to any symbol, register, etc. that are useful to modify in a snippet of assembly code.

The following macro could represent the file creation pattern:

```
CREATE  MACRO
        leax    \1,pcr
        lda     #$\2
        ldb     #$\3
        os9     I$Create
        ENDM
```

Calls can be made to create files with different names, access modes, and attributes as follows:

```
CREATE  name2,02,03
CREATE  name3,01,02
```

The above macro calls will produce the following inline code:

```
        leax    name2,pcr
        lda     #$02
```

```
        ldb     #$03
        os9     I$Create

        leax    name3,pcr
        lda     #$01
        ldb     #$02
        os9     I$Create
```

If an argument has multiple parts — for example if `0,s` is to be passed to the macro called `frud` — it must be passed in double quotes. For example:

```
frud    "0,s","2,s"
```

If `frud` looks like the following macro:

```
frud    MACRO
\@      leau    \1
        ldd     \2
        beq     \@
        ENDM
```

The previous call to `frud` would expand the macro as follows:

```
@xxx    leau    0,s
        ldd     2,s
        beq     @xxx
```

Where `\@` is a label, and `xxx` would be replaced by a three digit number.

An argument may be declared empty by leaving it blank when called. For example, if a macro instruction is defined to be `ldd \1ZZ\2`, then calling the macro with the arguments `AA,BB` will cause the assembler to expand the instruction to `ldd AAZZBB`, and calling the same macro with the argument list `,BB` will expand it to `ldd ZZBB` – the `\1` part is empty.

Nested Macro Calls

Macro calls may be nested — that is, the body of a macro definition may itself contain a call to another macro. For example, the macro `prepw` could be defined as follows:

```
prepw MACRO
      lda \1
      getw
      ENDM
```

where `getw` is a macro call. The code for `getw` will be substituted in-line at expansion time. However, take note that the *definition* of a new macro within another is not permitted. Macro calls may be nested up to eight levels deep.

Labels

Sometimes it is necessary to use labels within a macro. Labels are specified by `\@`. Each time the macro is called, a unique label will be generated to avoid multiple definition errors. Within the expanded code, `\@` will be replaced with a symbol of the form `@xxx`, where `xxx` is a decimal number between `000` and `999`.

More than one label may be specified in a macro by the addition of an extra character(s). For example, if two different labels are required in a macro, they can be specified by `\@A` and `\@B`. The first time the macro is expanded, the labels would be `@001A` and `@001B`, and for the second expansion they would be `@002A` and `@002B`. The extra characters may be appended before the `\` or after the `@`.

Additional Pseudo-Instructions

\n will return the number of arguments passed to the macro.

\L<num> will return the length of the <num>th argument that is specified by <num>.

FAIL Causes an error to be generated.

REPT <num> will repeat an instruction or group of instructions <num> times. ENDR terminates REPT.

Appendices

A
Compiler Error Messages

Below is a list of the error messages that the C compiler generates, and, if applicable, probable causes and K & R Appendix A section number (in parenthesis) to see for more specific information.

already a local variable Variable has already been declared at the current block level. (Section C.8.1, Section C.9.2)

argument : <**text**> Error from preprocessor. Self-explanatory. The most common cause of this error is not being able to find an **`#include`**d file.

argument error Function argument declared as type **`struct`**, **`union`** or a function. Pointers to such types, however are allowed. (Section C.10.1)

argument storage Function arguments may only be declared as storage class **`register`**. (Section C.10.1)

bad character A character not in the C character set (probably a control character) was encountered in the source file.

both must be integral >> and << operands cannot be **`float`** or **`double`**. (Section C.7.5)

break error The **`break`** statement is allowed only inside a **`while`**, **`do`**, **`for`**, or **`switch`** block. (Section C.9.8)

can't take address & operator not allowed in a **`register`** variable. Operand must otherwise be an lvalue. (Section C.7.2)

cannot cast Type result of cast cannot be a function or an array. (Section C.7.2, Section C.8.7)

cannot evaluate size Could not determine size from declaration or initializer. (Section C.8.6, Section C.14.3)

cannot initialize Storage class or type does not allow variable to be initialized. (Section C.8.6)

compiler trouble Compiler detected something it couldn't handle. Try compiling the program again. If this error still occurs, contact the developers.

condition needed **`while`**, **`do`**, **`for`**, **`switch`**, and **`if`** statements require a condition expression. (Section C.9.3)

constant expression required Initializer expressions for **`static`** or **`extern`** variables cannot reference variables. They may, however, refer to the address of a *previously declared* variable. This installation allows no initializer expressions unless all operands are of type **`int`** or **`char`** (Section C.8.6)

constant overflow Input numeric constant was too large for the implied or explicit type. (Section C.2.6, [PDP-11])

constant required Variables are not allowed for array dimensions or cases. (Section C.8.3, Section C.8.7, Section C.9.7)

continue error The **`continue`** statement is allowed only inside a **`while`**, **`do`**, or **`for`** block. (Section C.9.9)

declaration mismatch This declaration conflicts with a previous one. This is typically caused by declaring a function to return a non-integer type after a reference has been made to the function. Depending on the line structure of the declaration block, this error may be reported on the line following the erroneous declaration. (Section C.11, Section C.11.1, Section C.11.2)

divide by zero Divide by zero occurred when evaluating a constant expression.

? expected `?` is any character that was expected to appear here. Missing semicolons or braces cause this error.

expression missing An expression is required here.

function header missing Statement or expression encountered outside a function. Typically caused by mismatched braces. (Section C.10.1)

function type error A function cannot be declared as returning an array, function, struct, or union. (Section C.8.4, Section C.10.1)

function unfinished End-of-file encountered before the end of function definition. (Section C.10.1)

identifier missing Identifier name required here but none was found.

illegal declaration Declarations are allowed only at the beginning of a block. (Section C.9.2)

label required Label name required on **goto** statement. (Section C.9.11)

label undefined **goto** to a label not defined in the current function. (Section C.9.12)

lvalue required Left side of assignment must be able to be "stored into". Array names, functions, structs, etc. are not lvalues. (Section C.7.1)

multiple defaults Only one **default** statement is allowed in a **switch** block. (Section C.9.7)

multiple definition Identifier name was declared more than once at the same block level (Section C.9.2, Section C.11.1)

must be integral Type of object required here must be type **int**, **char**, or a pointer type.

name clash struct/union member and tag names must be mutually distinct. (Section C.8.5)

name in cast Identifier name found in a cast. Only types are allowed. (Section C.7.2, Section C.8.7)

named twice Names in a function parameter list may appear only once. (Section C.10.1)

no 'if' for 'else' An **else** statement was encountered with no matching **if**. This is typically caused by extra or missing braces and/or semicolons. (Section C.9.3)

no switch statement A **case** statement may appear only within a **switch** block. (Section C.9.7)

not a function Primary in expression is not type "function returning …." If this is really a function call, the function name was declared differently elsewhere. (Section C.7.1)

not an argument Name does not appear in the function parameter list. (Section C.10.1)

operand expected Unary operators require one operand, binary operators two. This is typically caused by misplaced parenthesis, casts or operators. (Section C.7.1)

out of memory Compiler dynamic memory overflow. The compiler requires dynamic memory for symbol table entries, block-level declarations and code generation. Three major factors affect this memory usage. Permanent declarations — those appearing on the outer block level (used in **#include** files) — must be reserved from dynamic memory for the duration of the compilation of the file. Each { causes the compiler to perform a block-level recursion which may involve "pushing down" previous declarations which consume memory. **auto** class initializers require saving expression trees until past the declarations, which may be very memory-expensive if they exist. Avoiding excessive declarations, both permanent and inside compound statement blocks, conserves memory. If this error occurs on an **auto** initializer, try initializing the value in the code body.

pointer mismatch Pointers refer to different types. Use a **case** if required. (Section C.7.1)

pointer or integer required A pointer (of any type) or integer is required to the left of the -> operator. (Section C.7.1)

pointer required Pointer operand required with unary * operator. (Section C.7.1)

primary expected Primary expression required here. (Section C.7.1)

should be NULL Second and third expression of ?: conditional operator cannot be pointers to different types. If both are pointers, they must be of the same type or one of the two must be NULL. (Section C.7.13)

*** **Stack Overflow** *** Compiler stack has overflowed. The most common cause is very deep block-level nesting, or hundreds of **switch** **case**s.

storage error **register** and **auto** storage classes may only be used within functions. (Section C.8.1)

struct member mismatch Identical member names in two different structures must have the same type and offset in both. (Section C.8.5)

struct member required Identifier used with . and -> operators must be a structure member name. (Section C.7.1)

struct syntax Brace, comma, etc. is missing in a **struct** declaration. (Section C.8.5)

struct or union inappropiate **struct** or **union** cannot be used in the context.

syntax error Expression, declaration or statement is incorrectly formed.

third expression missing ? must be followed by a : with expression. This error may be caused by unmatched parentheses or other errors in the expression. (Section C.7.13)

too long Too many characters provided in a string used to initialize a character array. (Section C.8.6)

too many brackets Unmatched or unexpected brackets encountered processing an initializer. (Section C.8.6)

too many elements More data items supplied for aggregate level in initializer than members of the aggregate. (Section C.8.6)

type error Compiler type matching error. Should never happen.

type mismatch Types and/or operators in expression do not correspond. (6)

typedef - not a variable **typedef** type name cannot be used in this manner. (Section C.8.8)

undeclared variable no declaration exists at any block level for this identifier.

undefined structure Union or struct declaration refers to an undefined structure name. (Section C.8.5)

unions not allowed Cannot initialize union members. (Section C.8.6)

unterminated character constant Unmatched ' character delimiter. (Section C.2.4)

unterminated string Unmatched " string delimiter. (Section C.2.5)

while expected No **`while`** found for **`do`** statement. (Section C.9.5)

B
Compiler Command Lines

This chapter describes the command lines and options for the individual compiler phases. Each phase of the compiler may be executed separately. The following information describes the options available to each phase.

dcc (Compiler executive)

`dcc` [options] *file…* [options]

Recognized file suffixes:

.c C source file
.a Assembly language source file
.r Relocatable module format file

Recognized options:

`-2`	Optimize code for (Nitr)OS-9 Level 2. Beware, attempting to run a program compiled using this option is extremely likely to fail, and may even crash the computer!
`-a`	Stop after generating initial assembly (does not optimize or assemble). Leaves output in `.a` file.
`-b=`*`path`*	Use an alternate "`cstart`" (program entry point, also known as a *mainline*) file.
`-c`	Includes original C source and comments as comments in assembly output.
`-d`*`NAME`*	Is equivalent to `#define` *`NAME`* `1` in the source.
`-d`*`NAME=STRING`*	Is equivalent to `#define` *`NAME STRING`* in the source.
`-e`*`num`*	Edition number *`num`* is supplied to `dcpp` for inclusion in module `psects` and/or `rlink` to serve as the edition number of the linked module.
`-f=`*`path`*	Override other output naming. Module name (in object module) is the last name in the path list.
`-l=`*`path`*	Add *`path`* as a library for linker to search before the standard library.
`-lg`	Link with `cgfx.l` (Tandy Color Computer 3 graphics library)

-ll	Link with `lexlib.l`, the Lex helper library.
-ls	Link with `sys.l`, a library that includes definitions for OS-9 system calls, constants, and `I$GetStt/I$SetStt` codes.
-M	Requests a linkage map from the linker if an executable module is produced.
-m=*size*	Size in pages (if followed by `K`, in kilobytes) of additional memory the linker should allocate to object module.
-O	Stop after optimizing assembly code (does not assemble). Leaves output in `.a` file.
-o	Inhibits assembly code optimizer pass.
-p	Adds profiling calls to C compiler output.
-P	Adds profiling calls to C compiler output, *and* replaces profiler with debugging version of same by linking with `dbg.l`debugging library.
-r	Inhibits the linking step, leaving the output in an `.r` file.
-S	Requests a symbol table from the linker if an executable module is produced.
-s	Suppress generation of stack-checking code.
-T[=*path*]	If *path* given, use *path* for temporary files. If no path given, disable using a temporary directory for intermediate files.
-d*NAME*	Is equivalent to #define *NAME* 1 in the preprocessor. -d*NAME=STRING* is equivalent to #define *NAME STRING*.
-n=*name*	output module name. *name* is used to override the -f default output name.
-q	Quiet mode. Suppress echo of file names. Standard error sent to `c.errors`

dcpp (C macro preprocessor)

`dcpp` [options] *path* [options]

path is read as input. dcpp causes dco68 to generate psect directive with last element of path list and _c as the psect name. If *path* is /d0/myprog.c, psect name is myprog_c. Output is always to stdout.

Recognized options:

-a	Act as an assembly language preprocessor instead of feeding the C compiler. No special commands intended for the compiler will be output; warnings are emitted as assembly comments; errors are emitted as `␣fail errortext` lines. All preprocessor directives work otherwise.
-l	Cause dcc68 to copy source lines to assembly output as comments.
-E=*n*	

-e=*n* Use *n* as psect edition number.
-D*name* Same as described above for dcc.

dcc68 (Compiler)

`dcc68` [options] [*`file`*] [options]

If *`file`* is not supplied, dco68 will read stdin. Input text need not be dcpp output, but no preprocessor directives are recognized (#include, #define, macros etc.). Output assembly code is normally to stdout. Error message output is always written to stderr.

Recognized options:

-s Suppress generation of stack checking code.
-p Generate profile code.
-o=*`path`* Write assembly output to *`path`*.

dco68 (Assembly code optimizer)

`dco68` [*`inpath`*] [*`outpath`*]

dco68 reads stdin and writes stdout. *`inpath`* must be present if *`outpath`* is given. Since c.opt rearranges and changes code, comments and assembler directives may be rearranged.

rma (Assembler)

`rma` *`file`* [options]

rma reads *`file`* as assembly language input. Errors are written to stderr. Options are turned on with one '-' and negated with '--'. To turn listing on use -l. To turn listing off use --l. To turn conditionals off, use --c.

Recognized options:

-o=*`path`* Write relocatable output to path. Must be a disk file.
-l Write listing to stdout. (default off)
-c List conditional assembly lines. (default on)
-f Formfeed for top of form. (default off)
-g List all code bytes generated. (default off)
-x Suppress macro expansion listing. (default on)
-e Print errors. (default on)
-s Print symbol table. (default off)
-d*n* Set lines per page to *n*. (default 66).
-w*n* Set line width to *n*. (default 80).

rlink (Linker)

`rlink` [options] *`mainline subn`*... [options]

rlink turns rma output into executable form. All input files must contain relocatable object format (ROF) files. *`mainline`* specifies the base module from which to resolve external references. A mainline module is indicated by setting the type/lang value in the psect directive to non-zero. No other ROF may contain a mainline psect. The mainline and all subroutine files will appear in the final linked object module whether actually referenced or not.

For the C Compiler, cstart.r is the mainline module. It is the mainline module's job to perform the initialization of data and the relocation of any data-text and data-data references within the initialized data using the information in the object module supplied by rlink.

Recognized options:

-o=*`path`*	Linker object output file. Must be a disk file. The last element in *`path`* is used as the module name unless overridden by -n.
-n=*`name`*	Use *`name`* as object file name.
-l=*`path`*	Use *`path`* as library file. A library file consists of one or more merged assembly ROF files. Each psect in the file is checked to see if it resolves any unresolved references. If so, the module is included on the final output module, otherwise it is skipped. No mainline psects are allowed in a library file. Library files are searched on the order given on the command line.
-E=*n* -e=*n*	*n* is used for the edition number in the final output module. 1 is used if -e is not present.
-M=*`size`*	*`size`* indicates the number of pages (kbytes if size is followed by a K) of additional memory, rlink will allocate to the data area of the final object module. If no additional memory is given, rlink add up the total data stack requirements found in the psect of the modules in the input modules.
-m	Prints linkage map indicating base addresses of the psects in the final object module.
-s	Prints final addresses assigned to symbols in the final object module.
-b=*`ept`*	Link C functions to be callable by Basic09. *`ept`* is the name of the function to be transferred to when Basic09 executes the RUN command.
-t	Allows static data to appear in a Basic09 callable module. It is assumed the C function called and the calling Basic09 program have provided a sufficiently large static storage data area pointed to by the Y register.

C
C Language Reference Manual

C.1 Introduction

This manual describes the C language on the DEC PDP-11, the DEC VAX-11, and the 6809. Where differences exist, it concentrates on the VAX, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

C.2 Lexical Conventions

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

C.2.1 Comments

The characters `/*` introduce a comment which terminates with the characters `*/`. Comments do not nest.

C.2.2 Identifiers (Names)

An *identifier* is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

PDP-11	7 characters, 2 cases
VAX-11	>100 characters, 2 cases
6809	8 characters, 2 cases

C.2.3 Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

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

Some implementations also reserve the words `direct`, `fortran` and `asm`

C.2.4 Constants

There are several kinds of constants. Each has a type; an introduction to types is given under What's in a name?. Hardware characteristics that affect sizes are summarized in Hardware Characteristics.

Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with `0` (digit zero). An octal constant consists of the digits `0` through `7` only. A sequence of digits preceded by `0x` or `0X` (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be `long`.

Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by `l` (letter ell) or `L` is a long constant. As discussed below, on some machines integer and long values may be considered identical.

Character Constants

A character constant is a character enclosed in single quotes, as in 'x'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (`'`) and the backslash (`\`), may be represented according to the following table of escape sequences:

newline	NL (LF)	`\n`
horizontal tab	HT	`\t`

vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
backslash	\	`\\`
single quote	’	`\'`
bit pattern	*ddd*	`\`*ddd*

The escape `\`*ddd* consists of the backslash followed by 1, 2, or 3 *octal* digits which are taken to specify the value of the desired character. A special case of this construction is `\0` (not followed by a digit), which indicates the character NUL (a character with the value zero). If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is **`int`**.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

C.2.5 Strings

A string is a sequence of characters surrounded by double quotes, as in `"…"`. A string has type “array of `char`” and storage class `static` (see What’s in a name?) and is initialized with the given characters. The compiler places a null byte (`\0`) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (`"`) must be preceded by a backslash (`\`); in addition, the same escapes as described for character constants may be used.

A backslash `\` immediately followed by a newline are both ignored. All strings, even when written identically, are distinct.

C.2.6 Hardware Characteristics

Figure C.1 on the next page summarizes certain hardware properties that vary from machine to machine.

C.3 Syntax Notation

Syntactic categories are indicated by *italic* type, and literal words and characters in **`bold typewriter`** type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript “opt,” so that

	DEC PDP-11	DEC VAX-11	6809
character set	(ASCII)	(ASCII)	(ASCII)
char	8 bits	8 bits	8 bits
int	16	32	16
short	16	16	16
long	32	32	32
float	32	32	32
double	64	64	64
float range	$10^{\pm 38}$	$10^{\pm 38}$	$10^{\pm 38}$
double range	$10^{\pm 38}$	$10^{\pm 38}$	$10^{\pm 38}$

Figure C.1: Hardware Comparison

{ *expression*opt **}**

indicates an optional expression enclosed in braces. The syntax is summarized in Syntax Summary.

C.4 What's in a name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (see Compound Statement (Block)) and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (`char`) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a `char` variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent.

Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. "Plain" integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n where n is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (`float`) and double precision floating point (`double`) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types `char` and `int` of all sizes will collectively be called *integral* types. `float` and `double` types will collectively be called *floating* types.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *arrays* of objects of most types
- *functions* which return objects of a given type
- *pointers* to objects of a given type
- *structures* containing a sequence of objects of various types
- *unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

C.5 Objects and lvalues

An *object* is a manipulable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name "lvalue" comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

C.6 Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under "Arithmetic Conversions." The summary will be supplemented as required by the discussion of each operator.

C.6.1 Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, `char` variables range in value from -128 to 127. The more explicit type `unsigned char` forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value `-1`.

When a longer integer is converted to a shorter integer or to a `char`, it is truncated on the left. Excess bits are simply discarded.

C.6.2 Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a `float` appears in an expression it is lengthened to `double` by zero padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length. This result is undefined if it cannot be represented as a float. On the VAX, the compiler can be directed to use single precision for expressions containing only float and integer operands.

C.6.3 Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

C.6.4 Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

C.6.5 Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a two's complement representation, this conversion is conceptual only and there is no actual change in the bit pattern.

When an unsigned `short` integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

C.6.6 Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the "usual arithmetic conversions."

a. First, any operands of type `char` or `short` are converted to `int`, and any operands of type `unsigned char` or `unsigned short` are converted to `unsigned int`.

b. Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.

c. Otherwise, if either operand is `unsigned long`, the other is converted to `unsigned long` and that is the type of the result.

d. Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.

e. Otherwise, if one operand is `long`, and the other is `unsigned int`, they are both converted to `unsigned long` and that is the type of the result.

f. Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.

g. Otherwise, both operands must be `int`, and that is the type of the result.

C.7 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of + (see Additive Operators) are those expressions defined under Primary Expressions, Unary Operators, and Multiplicative Operators. Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the Syntax Summary.

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute sub-expressions in the order it believes most efficient even if the sub-expressions involve side effects. The order in which sub-expression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (*, +, &, |, ^) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

C.7.1 Primary Expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

primary–expression:
    identifier
    constant
    string
    (*expression* **)**
    primary–expression **[** *expression* **]**
    primary–expression **(** *expression–list*$_{\text{opt}}$ **)**
    primary–expression **.** *identifier*
    primary–expression **->** *identifier*

expression–list:
    expression
    expression–list **,** *expression*

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is "array of …", then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is "pointer to …". Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared "function returning …", when used except in the function-name position of a call, is converted to "pointer to function returning …".

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int` and floating constants have type `double`.

A string is a primary expression. Its type is originally "array of `char`", but following the same rule given above for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see Initialization.)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is `int`, and the type of the result is "...". The expression `E1[E2]` is identical (by definition) to `*((E1)+E2))`. All the clues needed to understand

this notation are contained in this subpart, together with the discussions on Unary Operators and Additive Operators regarding identifiers, * and + respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under Types Revisited.

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...," and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call. Any of type `char` or `short` are converted to `int`. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see Section C.7.2 and Section C.8.7.

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from and >) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression `E1>MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in Section C.8.5 under Section C.8.

C.7.2 Unary Operators

Expressions with unary operators group right to left.

unary–expression:
- ***** *expression*
- **&** *lvalue*
- **-** *expression*
- **!** *expression*
- **~** *expression*

++ *lvalue*
-- *lvalue*
lvalue **++**
lvalue **--**
(*type–name* **)** *expression*
sizeof *expression*
sizeof (*type–name* **)**

The unary * operator means *indirection*; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to …," the type of the result is "…".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "…", the type of the result is "pointer to …".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where n is the number of bits in the corresponding signed type. There is no unary + operator.

The result of the logical negation operator `!` is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand, but is not an lvalue. The expression `++x` is equivalent to `x=x+1`. See the discussions on Additive Operators and Assignment Operators for information on conversions.

The lvalue operand of prefix `--` is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix `--` is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix `--` operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in Section C.8.7.

The `sizeof` operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations, a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an `unsigned` constant and may be used

anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction `sizeof(`*type*`)` is taken to be a unit, so the expression `sizeof(`*type*`)-2` is the same as `(sizeof(`*type*`))-2`.

C.7.3 Multiplicative Operators

The multiplicative operators *, /, and % group left to right. The usual arithmetic conversions are performed.

multiplicative–expression:
expression ***** *expression*
expression **/** *expression*
expression **%** *expression*

The binary * operator indicates multiplication. The * operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary / operator indicates division.

The binary % operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not 0).

C.7.4 Additive Operators

The additive operators + and - group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive–expression:
expression **+** *expression*
expression **-** *expression*

The result of the + operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the - operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

C.7.5 Shift Operators

The shift operators << and >> group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits. On the VAX a negative right operand is interpreted as reversing the direction of the shift.

shift–expression:
expression **<<** *expression*
expression **>>** *expression*

The value of `E1 << E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits. Vacated bits are 0 filled. The value of `E1 >> E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0 fill) if `E1` is unsigned; otherwise, it may be arithmetic.

C.7.6 Relational Operators

The relational operators group left to right.

relational–expression:
expression **<** *expression*
expression **>** *expression*
expression **<=** *expression*
expression **>=** *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is **`int`**. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

C.7.7 Equality Operators

equality–expression:
expression **==** *expression*
expression **!=** *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a < b == c < d` is 1 whenever `a < b` and `c < d` have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

C.7.8 Bitwise AND Operator

and–expression:
expression **&** *expression*

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

C.7.9 Bitwise Exclusive OR Operator

exclusive–or–expression:
expression **^** *expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

C.7.10 Bitwise Inclusive OR Operator

inclusive–or–expression:
expression **|** *expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

C.7.11 Logical AND Operator

logical–and–expression:
expression **&&** *expression*

The && operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

C.7.12 Logical OR Operator

logical–or–expression:
expression **||** *expression*

The || operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

C.7.13 Conditional Operator

logical–or–expression:
expression **?** *expression* **:** *expression*

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

C.7.14 Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment–expression:
expression **=** *expression*
expression **+=** *expression*
expression **-=** *expression*
expression ***=** *expression*
expression **/=** *expression*
expression **%=** *expression*
expression **>>=** *expression*
expression **<<=** *expression*
expression **&=** *expression*
expression **^=** *expression*
expression **|=** *expression*

In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand

must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form `E1` *op*= `E2` may be inferred by taking it as equivalent to `E1 = E1` *op* `(E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in Section C.7.4. All right operands and all non-pointer left operands must have arithmetic type.

C.7.15 Comma Operator

comma–expression:
expression **,** *expression*

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see Section C.7.1) and lists of initializers (see Section C.8.6), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

C.8 Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
*decl–specifiers declarator–list*opt **;**

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl–specifiers:
*type–specifier decl–specifiers*opt
*sc–specifier decl–specifiers*opt

The list must be self-consistent in a way described below.

C.8.1 Storage Class Specifiers

The sc-specifiers are:

sc–specifier:
auto

static
extern
register
typedef

The `typedef` specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See Section C.8.8 for more information. The meanings of the various storage classes were discussed in Section C.4.

The `auto`, `static`, and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case, there must be an external definition (see Section C.10) for the given identifiers somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are `int` or pointer. One other restriction applies to register variables: the address-of operator `&` cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside. Exception: functions are never automatic.

C.8.2 Type Specifiers

The type-specifiers are

type–specifier:
char
short
int
long
unsigned
float
double
struct–or–union–specifier
typedef–name

At most one of the words `long` or `short` may be specified in conjunction with `int`; the meaning is the same as if `int` were not mentioned. The word `long` may be specified in conjunction with `float`; the meaning is the same as `double`. The word `unsigned` may be specified alone, or in conjunction with `int` or any of its short or long varieties, or with `char`.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of `long`, `short`, or `unsigned` is not permitted with `typedef` names. If the type-specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures and unions are discussed in Section C.8.5. Declarations with `typedef` names are discussed in Section C.8.8.

C.8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator–list:
init–declarator
init–declarator **,** *declarator–list*

init–declarator:
declarator initializer$_{\text{opt}}$

Initializers are discussed in Section C.8.6. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(*declarator* **)**
***** *declarator*
declarator **()**
declarator **[** *constant–expression*$_{\text{opt}}$ **]**

The grouping is the same as in expressions.

C.8.4 Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where `T` is a type-specifier (like `int`, etc.) and `D1` is a declarator. Suppose this declaration makes the identifier have type "… `T`," where the "…" is empty if `D1` is just a plain identifier (so that the type of `x` in `int x` is just `int`). Then if `D1` has the form

```
*D
```

the type of the contained identifier is "… pointer to `T` &."

If `D1` has the form

```
D()
```

then the contained identifier has the type "… function returning `T`." If `D1` has the form

```
D[constant-expression]
```

or

```
D[]
```

then the contained identifier has type "… array of `T`." In the first case, the constant expression is an expression whose value is determinable at compile time , whose type is `int`, and whose value is positive. (Constant expressions are defined precisely in Section C.15) When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer `i`, a pointer `ip` to an integer, a function `f` returning an integer, a function `fip` returning a pointer to an integer, and a pointer `pfi` to a function which returns an integer. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`. The declaration suggests, and the same construction in an expression requires, the calling of a function `fip`. Using indirection through the (pointer) result to yield an integer. In the declarator `(*pfi)()`, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression. The first three have type "array" and the last has type `int`.

C.8.5 Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct–or–union–specifier:
struct–or–union **{** *struct–decl–list* **}**
struct–or–union identifier **{** *struct–decl–list* **}**
struct–or–union identifier

struct–or–union:
struct
union

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct–decl–list:
struct–declaration
struct–declaration struct–decl–list

struct–declaration:
type–specifier struct–declarator–list **;**

struct–declarator–list:
struct–declarator
struct–declarator **,** *struct–declarator–list*

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field;* its length, a non-negative constant expression, is set off from the field name by a colon.

struct–declarator:
declarator
declarator **:** *constant–expression*
: *constant–expression*

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each non-field member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on the AT&T 3B20.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with `int` are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as `unsigned`. In all implementations, there are no arrays of fields, and the address-of operator `&` may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

`struct` *identifier* **`{`** *struct–decl–list* **`}`**
`union` *identifier* **`{`** *struct–decl–list* **`}`**

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

`struct` *identifier*
`union` *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a `typedef` name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode {
    char            tword[20];
    int             count;
    struct tnode    *left;
    struct tnode    *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression `sp->count` refers to the `count` field of the structure to which `sp` points; `s.left` refers to the left subtree pointer of the structure `s`; and `s.right->tword[0]` refers to the first character of the `tword` member of the right subtree of `s`.

C.8.6 Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
- **=** *expression*
- **= {** *initializer–list* **}**
- **= {** *initializer–list* **, }**

initializer–list:
- *expression*
- *initializer–list* **,** *initializer–list*
- **{** *initializer–list* **}**
- **{** *initializer–list* **, }**

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in Section C.15, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a `char` array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise, the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]` is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace but that for `y[0]` does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

C.8.7 Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of `sizeof`), it is desired to supply the name of a data type. This is accomplished using a "type name", which in essence is a declaration for an object of that type which omits the name of the object.

type–name:
    type–specifier abstract–declarator

abstract–declarator:
    empty
    (*abstract–declarator* **)**
    ***** *abstract–declarator*
    abstract–declarator **()**
    abstract–declarator **[** *constant–expression*$_{\text{opt}}$ **]**

To avoid ambiguity, in the construction

    (*abstract–declarator* **)**

the abstract-declarator is required to be non-empty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
int *()
int (*)()
int (*[3])()
```

name respectively the types "integer," "pointer to integer," "array of three pointers to integers," "pointer to an array of three integers," "function returning pointer to integer," "pointer to function returning an integer," and "array of three pointers to functions returning an integer."

C.8.8 Typedef

Declarations whose "storage class" is `typedef` do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef–name:
    identifier

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in Section C.8.4. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is `int`, that of `metricp` is "pointer to `int`," and that of `z` is the specified structure. The `zp` is a pointer to such a structure.

The `typedef` does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

C.9 Statements

Except as indicated, statements are executed in sequence.

C.9.1 Expression Statement

Most statements are expression statements, which have the form

    expression **;**

Usually expression statements are assignments or function calls.

C.9.2 Compound Statement (Block)

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called a "block") is provided:

compound–statement:
    { *declaration–list*opt *statement–list*opt **}**

declaration–list:
    declaration

declaration declaration–list

statement–list:
statement
statement statement–list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

C.9.3 Conditional Statement

The two forms of the conditional statement are

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

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The "else" ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

C.9.4 While Statement

The `while` statement has the form

`while` **`(`** *expression* **`)`** *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

C.9.5 Do Statement

The `do` statement has the form

`do` *statement* **`while`** **`(`** *expression* **`)`** **`;`**

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

C.9.6 For Statement

The `for` statement has the form:

`for` **`(`** *exp–1*opt **`;`** *exp–2*opt **`;`** *exp–3*opt **`)`** *statement*

Except for the behavior of `continue`, this statement is equivalent to

```
exp−1;
while ( exp−2 )
{
        statement
        exp−3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied `while` clause equivalent to `while(1)`; other missing expressions are simply dropped from the expansion above.

C.9.7 Switch Statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

```
switch ( expression ) statement
```

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

```
case constant−expression :
```

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in Section C.15.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a `default`, prefix, control passes to the prefixed statement. If no case matches and if there is no `default`, then none of the statements in the switch is executed.

The prefixes `case` and `default` do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see Section C.9.8.

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

C.9.8 Break Statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

C.9.9 Continue Statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is, to the end of the loop. More precisely, in each of the statements

```
while (...) {
    statement ;
    contin: ;
}
```

```
do {
    statement ;
    contin: ;
} while (...);
```

```
for (...) {
    statement ;
    contin: ;
}
```

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, see Null Statement)

C.9.10 Return Statement

A function returns to its caller by means of the `return` statement which has one of the forms

```
return ;
return expression ;
```

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

C.9.11 Goto Statement

Control may be transferred unconditionally by means of the statement

```
goto identifier ;
```

The identifier must be a label (see Labeled Statement) located in the current function.

C.9.12 Labeled Statement

Any statement may be preceded by label prefixes of the form

```
identifier :
```

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See Scope Rules.

C.9.13 Null Statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the `}` of a compound statement or to supply a null body to a looping statement such as `while`.

C.10 External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (see Section C.8.2) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

C.10.1 External Function Definitions

Function definitions have the form

function–definition:
    *decl–specifiers*opt *function–declarator function–body*

The only sc-specifiers allowed among the decl-specifiers are `extern` or `static`; see Section C.11.2 for the distinction between them. A function declarator is similar to a declarator for a "function returning …" except that it lists the formal parameters of the function being defined.

function–declarator:
    declarator **(** *parameter–list*opt **)**

parameter–list:
    identifier
    identifier **,** *parameter–list*

The function-body has the form

function–body:
    *declaration–list*opt *compound–statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
    int m;

    m = (a > b) ? a : b;
    return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; `{ ... }` is the block giving the code for the statement.

The C program converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. All `char` and `short` formal parameter declarations are similarly adjusted to read `int`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared "array of ..." are adjusted to read "pointer to ...."

C.10.2 External Data Definitions

An external data definition has the form

data–definition:
    declaration

The storage class of such data may be `extern` (which is the default) or `static` but not `auto` or `register`.

C.11 Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing "undefined identifier" diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

C.11.1 Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers

which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see Section C.8.5) that identifiers associated with ordinary variables, and those associated with structure and union members form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
     auto int distance;
     ...
}
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance`.

C.11.2 Scope of Externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of `extern` does not change the meaning of an external declaration.

In restricted environments, the use of the `extern` storage class takes on an additional meaning. In these environments, the explicit appearance of the `extern` keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without `extern`) in the set of files and libraries comprising a multi-file program.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

C.12 Compiler Control Lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

C.12.1 Token Replacement

A compiler-control line of the form

```
#define identifier token-string opt
```

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

```
#define identifier(identifier, ... ) token-string opt
```

where there is no space between the first identifier and the (, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a (, a sequence of tokens delimited by commas, and a) are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of "manifest constants," as in

```
#define TABSIZE 100

int table[TABSIZE];
```

A control line of the form

```
#undef identifier
```

causes the identifier's preprocessor definition (if any) to be forgotten.

If a **#define**d identifier is the subject of a subsequent **#define** with no intervening **#define**, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

C.12.2 File Inclusion

A compiler control line of the form

```
#include "filename"
```

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the `#include`, and then in a sequence of specified or standard places. Alternatively, a control line of the form

```
#include <filename>
```

searches only the specified or standard places and not the directory of the `#include`. (How the places are specified is not part of the language.)

`#include`s may be nested.

C.12.3 Conditional Compilation

A compiler control line of the form

```
#if constant-expression
```

checks whether the constant expression evaluates to nonzero. (Constant expressions are discussed in Section C.15. A control line of the form

```
#ifdef identifier
```

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a `#define` control line. It is equivalent to `#if defined(identifier)`. A control line of the form

```
#ifndef identifier
```

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to `#if !defined(identifier)`.

All three forms are followed by an arbitrary number of lines, possibly containing a control line

```
#else
```

and then by a control line

```
#endif
```

If the checked condition is true, then any lines between `#else` and `#endif` are ignored. If the checked condition is false, then any lines between the test and a `#else` or, lacking a `#else`, the `#endif` are ignored.

These constructions may be nested.

C.12.4 Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

```
#line constant identifier
```

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent, the remembered file name does not change.

C.13 Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be of type **`int`**; if a type but no storage class is indicated, the identifier is assumed to have storage class **`auto`**. An exception to the latter rule is made for functions because **`auto`** functions do not exist. If the type of an identifier is "function returning ...," it is implicitly declared to be **`extern`**.

In an expression, an identifier followed by `(` and not already declared is contextually declared to be "function returning `int`."

C.14 Types Revisited

This part summarizes the operations which can be performed on objects of certain types.

C.14.1 Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the `->` or the `.` must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures.

C.14.2 Functions

There are only two things that can be done with a function `m`, call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
...
g(f);
```

Then the definition of g might read

```
g(funcp)
int (*funcp)();
{
    ...
    (*funcp)();
    ...
}
```

Notice that f must be declared explicitly in the calling routine since its appearance in g(f) was not followed by (.

C.14.3 Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [] is interpreted in such a way that E1[E2] is identical to *((E1)+E2)). Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If E is an n-dimensional array of rank $i \times j \times \ldots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. If the * operator (either explicitly or implicitly as a result of subscripting) is applied to this pointer, the result is the *pointed-to* $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here, x is a 3×5 array of integers. When x appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression x[i], which is equivalent to *(x+i), x is first converted to a pointer as described; then i is converted to the type of x, which involves multiplying i by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

C.14.4 Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see Unary Operators and Type Names.

A pointer may be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The `alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The `char`s have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that `double` quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B 20 computer has 24-bit pointers placed into 32-bit quantities. Most objects are aligned on 4-byte boundaries. `short`s are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, `int`s, `long`s, `float`s, and `double`s are aligned on 4-byte boundaries; but structure members may be packed tighter.

C.15 Constant Expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

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

or by the unary operators

```
-   ~
```

or by the ternary operator

```
?:
```

Parentheses may be used for grouping, but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

C.16 Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of `register` variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type `int`, multi-character character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bitfields, and does not

accept a few assignment operators in certain contexts where the value of the assignment is used.

C.17 Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form **=***op* instead of *op***=** for assignment operators. This leads to ambiguities, typified by:

```
x=-1
```

which actually decrements `x` since the `=` and the `-` are adjacent, but which might easily be intended to assign the value -1 to `x`.

The syntax of initializers has changed: previously, the equals sign that introduces and initializer was not present, so instead of

```
int x   = 1;
```

one used

```
int x   1;
```

The change was made because the initialization

```
int f   (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

C.18 Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

C.18.1 Expressions

The basic expressions are:

expression:
- *primary*
- ***** *expression*
- **&** *lvalue*
- **-** *expression*
- **!** *expression*
- **~** *expression*
- **++** *lvalue*

-- *lvalue*
lvalue **++**
lvalue **--**
sizeof *expression*
sizeof (***type–name* **)
(*type–name* **)** *expression*
expression binop expression
expression **?** *expression* **:** *expression*
lvalue asgnop expression
expression **,** *expression*

primary:
identifier
constant
string
(*expression* **)**
primary **(** *expression–list*opt **)**
primary **[** *expression* **]**
primary **.** *identifier*
primary **-** *identifier*

lvalue:
identifier
primary **[** *expression* **]**
lvalue **.** *identifier*
primary **-** *identifier*
***** *expression*
(*lvalue* **)**

The primary-expression operators

```
()  []  .   ->
```

have highest priority and group left to right. The unary operators

```
*   &   -   !   ~   ++  --  sizeof ( type-name )
```

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:
*** / %**
+ -
>> <<

```
< > <= >=
== !=
&
^
|
&&
||
```

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:

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

The comma operator has the lowest priority and groups left to right.

C.18.2 Declarations

declaration:

*decl–specifiers init–declarator–list*opt **;**

decl–specifiers:

*type–specifier decl–specifiers*opt
*sc–specifier decl–specifiers*opt

sc–specifier:

auto
static
extern
register
typedef

type–specifier:

char
short
int
long
unsigned
float
double
struct–or–union–specifier
typedef–name

init–declarator–list:

init–declarator
init–declarator **,** *init–declarator–list*

init–declarator:
declarator initializeropt

declarator:
identifier
(*declarator* **)**
***** *declarator*
declarator **()**
declarator **[** *constant–expressionopt* **]**

struct–or–union–specifier:
struct { *struct–decl–list* **}**
struct *identifier* **{** *struct–decl–list* **}**
struct *identifier*
union { *struct–decl–list* **}**
union *identifier* **{** *struct–decl–list* **}**
union *identifier*

struct–decl–list:
struct–declaration
struct–declaration struct–decl–list

struct–declaration:
type–specifier struct–declarator–list **;**

struct–declarator–list:
struct–declarator
struct–declarator **,** *struct–declarator–list*

struct–declarator:
declarator
declarator **:** *constant–expression*
: *constant–expression*

initializer:
= *expression*
= { *initializer–list* **}**
= { *initializer–list* **, }**

initializer–list:
expression
initializer–list **,** *initializer–list*
{ *initializer–list* **}**
{ *initializer–list* **,** **}**

type–name:
type–specifier abstract–declarator

abstract–declarator:
empty
(*abstract–declarator* **)**
***** *abstract–declarator*
abstract–declarator **()**
abstract–declarator **[** *constant–expression*$_{\text{opt}}$ **]**

typedef–name:
identifier

C.18.3 Statements

compound–statement:
{ *declaration–list*$_{\text{opt}}$ *statement–list*$_{\text{opt}}$ **}**

declaration–list:
declaration
declaration declaration–list

statement–list:
statement
statement statement–list

statement:
compound–statement
expression **;**
if **(** *expression* **)** *statement*
if **(** *expression* **)** *statement* **else** *statement*
while **(** *expression* **)** *statement*
do *statement* **while** **(** *expression* **)** **;**
for **(** *exp*$_{\text{opt}}$ **;** *exp*$_{\text{opt}}$ **;** *exp*$_{\text{opt}}$ **)** *statement*

switch (*expression*) *statement*
case *constant−expression* : *statement*
default : *statement*
break ;
continue ;
return ;
return *expression* ;
goto *identifier* ;
identifier : *statement*
;

C.18.4 External definitions

program:
external−definition
external−definition program

external−definition:
function−definition
data−definition

function−definition:
*decl−specifier*opt *function−declarator function−body*

function−declarator:
declarator (*parameter−list*opt)

parameter−list:
identifier
identifier , *parameter−list*

function−body:
*declaration−list*opt *compound−statement*

data−definition:
extern *declaration* ;
static *declaration* ;

C.18.5 Preprocessor

#define *identifier token−string*opt
#define *identifier*(*identifier*,...) *token−string*opt

#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *constant–expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *constant identifier*

D
Differences between DCC and Microware C

This package contains the DCC C Compiler. Many improvements and bug fixes have been incorporated since the Microware 6809 compiler releases. If you are upgrading from the Microware compiler and library, please be *sure* to test all programs you build with this compiler. While every effort has been made to assure compatibility with old Microware C 1.1 code, there may be slight incompatibilities in some places. Be careful.

The remainder of this notice describes the changes made since the 1983 releases of the Microware compiler.

General

The compiler's binaries have all been renamed, to allow coexistence with the Microware compiler.

Executive (`dcc`)

The `dcc` executive does not generate command files, instead directly executing the various parts of the compiler in the appropriate sequence.

Preprocessor (`dcpp`)

`dcpp` supports the `#if` directive.

`dcpp` supports the standard `#line` directive, for use by external tools that generate C source.

`dcpp` supports ANSI/ISO "stringizing" and token pasting in the `#define` directive.

`dcpp` supports `#warning` and `#error` directives, to allow error and potentially dangerous conditions to be signaled to the compiler phase.

`dcpp` supports the ANSI/ISO standard `__FILE__` symbol, which expands to a C string literal containing the name of the current file..

`dcpp` supports the null directive, meaning preprocessor lines that contain no directives are no longer considered errors.

Comments in preprocessor lines will never be seen by the compiler.

`dcpp` automatically defines the symbols `_OS9`, `__mc6809__`, and `_BIG_END`.

These changes make `dcpp` much more compliant with modern C.

Compiler (`dcc68`)

There is currently no multi-pass version of the compiler, but research continues into how that version may be resurrected.

`dcc68` supports new types taken from the ANSI standard: **`unsigned char`**, **`unsigned short`** and **`unsigned long`**. `dcc68` also supports the **`signed`** keyword to better cope with modern code standards.

`dcc68` can generate and signal warning messages that are not classified as errors, but which may be serious enough that the programmer may wish to change the code anyway.

A bug in the expression handler for math being done inside a **`switch`** expression has been fixed.

`dcc68` is able to trace the contents of the D and X registers, in order to know when and when not to generate code to reload them.

Optimizer (`dco68`)

`dco68` is no longer hard-coded with a specific set of instruction recognition patterns; instead, patterns may be found in `*.patterns` files on disk.

`dco68` includes patterns that enable it to optimize for OS-9 Level II systems, which always place the start of the program data area at address zero. On such systems, the data area may be accessed using absolute addressing instead of indexed addressing, saving bytes and cycles.

Library (clib.l and clibt.l)

The standard library has been replaced with a newer one assembled by Carl Kreider, and contains many new functions from Unix and OS-9 for 68000 processors.

Runtime (cstart.r)

It is now possible to return a value from the program's `main` function instead of explicitly calling `exit()`, bringing behavior more in line with ANSI C language standards. However, if the program does *not* return a value from the program's main function, the undefined behavior may cause the program's exit status to be anything.

Index

Glossary

arguments	parameters given to a function to control its operation
escape sequence	A group of characters, beginning with \{}, that stands for a special character that is impossible, difficult, or annoying to have represent itself.
function	a set of *statements* encapsulated under a name to perform some sequence of actions