An Introduction to Features of the C Language
Gary H. Merrill, SAS Institute Inc., Cary, NC

ABSTRACT

This paper is a brief introduction to the most significant features of the C programming language and is intended as a quick introduction to that language for those who are already familiar with some high-level programming language. It is not intended to be a complete description of C, but rather to give the flavor of the language and serve as a basis for further discussion in the question and answer session.

INTRODUCTION

For someone experienced in one or more programming languages, it is possible to provide a brief but valuable introduction to a new language by describing certain fundamental features of that language. The features of C I focus on in this paper are:

- data types
- operators
- control structures
- program structure.

I will give you a brief and informal glimpse of each of these topics, and then leave time for a question-and-answer session during which we may explore these and related topics in more detail.

ARITHMETIC DATA TYPES

The most familiar data types in C are the arithmetic data types used to define variables which are to represent numbers. The simplest of these are the integral data types:

- char /“character: 1 byte */
- short /“short integer: 2 bytes */
- long /“long integer: 4 bytes */
- int /“integer: either short or long */.

These may be used to represent integral numbers of the appropriate sizes, and each has a corresponding unsigned type that may be used to represent positive integers of the appropriate size:

- unsigned char
- unsigned short
- unsigned long
- unsigned int.

You may wonder what the function of the int type is. On some machines (the PC, for example) this is equivalent to a short, while on other machines (the mainframe) it is equivalent to a long. The reason for the int type in C pertains to efficiency. The idea is that an int should be the "natural" integer size for the machine on which your C compiler has been implemented. And what "natural" means here is that an int should be the same size as the normal data registers on the machine. This is why ints are 16 bits on a PC and 32 bits on a mainframe.

You may see that there is a danger in using the int type in C code that is intended to be portable (compiled and run on more than one machine). A program that runs correctly on a machine where ints are longs will not necessarily run correctly on a machine where ints are smaller. If, for example, you are using an int to keep track of the number of entries in a large database on a mainframe, there may be more entries than can be represented in a short. It would be wiser in such a case to use a long rather than an int. Experienced C programmers who are concerned with the portability of their code will take great care in their use of ints. Typically this type is used where efficiency is an issue (for example, an int might be used as a loop counter) and where the programmer is sure that a long will not be required.

The char type may be used to represent small integral values, but most commonly it is used to represent a character in either ASCII or EBCDIC format. Arrays of chars are used in C to represent strings.

In addition to the integral data types C also offers two arithmetic data types for floating-point numbers:

- float /“floating-point number */
- double /“double precision floating-point number */

C does not provide you with a binary-coded decimal or packed decimal type, though there are add-on libraries for certain C compilers that do offer this kind of functionality.

STRUCTURES

A structure in C is simply a set of things grouped together. For example, in the declaration

```
struct PERSON {
    char lastname[40];
    char firstname[20];
    short age;
};
```

we have defined a structure tag ("PERSON") which stands for a data object containing two character arrays and a short. Clearly we intend an object of this type to represent a person and provide us with the person's name and age. We might then go on to define variables of this type, as in
struct PERSON husband, wife, child1, child2;

and we could then reference the information in such structures by means of the dot operator in C. For example:

    husband.age
    wife.age
    child1.age
    child2.age

each refers to the age field in their respective structures.

Structures are used extensively in C. A structure may contain substructures and pointers (which will be discussed below). It is by means of structures that more complex data structures (such as lists, trees, graphs, tables, and so on) may be represented in C.

A data type closely tied to structures is that of the bit field. Bit fields allow you to save considerable space when all you want to represent is a "flag" or an object that can take on only a small range of values. Suppose, for example, that you had a database in which you needed to keep track of whether a person was male, a female, and an adult or child.

One approach to this would be to define (for each database entry) three variables:

    int male;
    int female;
    int child;

and set each of these to 0 or 1. This is quite wasteful of space -- especially in a large database.

A second alternative would be to define a flags variable:

    int flag;

and treat one bit as meaning "male," one as meaning "female," and one as meaning "child." In fact this approach is frequently taken in C.

A third approach, which combines the readability of the first with the space saving features of the second, is to define three bit fields:

    struct PROPERTIES {
        unsigned int male : 1;
        unsigned int female : 1;
        unsigned int child : 1;
    };

Here we have a structure (bit fields may be defined only within structures) containing three one-bit bit fields. Since these are fields within a structure they may be referenced in the same manner as any other fields. You may also make assignments to these fields rather than using bitwise operations. Of course in this case you could only assign either 1 or 0 to each field. In the case of a wider field larger numbers could be assigned. Bit fields allow you an economical and readable way of representing a variety of properties.

UNIONS

Unions look like structures superficially, but their use is quite different. Suppose you needed a routine (such as a lexical analyzer) that would read input and return different types of values. For example, if the routine read a numeric constant it would return the number corresponding to the constant. This might be either an integral number or a floating-point number. On the other hand, if the string read was not a numeric constant you would want the string itself returned. One way of doing this, of course, would be to use a structure to represent all of the possible return values:

    struct VAL {
        long intvalue;
        double fllvalue;
        char *string;
    };

and set the unused fields to some appropriate value. Even this faces problems. A pointer value of NULL can indicate that string does not have a meaningful value, but then how do you determine which of intvalue or fllvalue contains the intended value? In addition, if only one of these fields is ever used on a given occasion, there is a waste of space.

This is a good application for a union. We could define

    struct VAL {
        char type;
        union VALUE {
            long intvalue;
            double fllvalue;
            char *string;
        } value;
    };

and use the type field of the structure to tell us which field of the union is meaningful. While the compiler must allocate memory for each field in a structure, in the case of a union it must allocate only enough memory for the largest field in the union since all the fields overlap.

POINTERS

Although C is a high-level programming language, it offers much of the utility of a low-level language such as assembly language. One way C does this is through its very general support for pointers. A pointer in C represents the address of an object, and there are very few restrictions on what a pointer may point to.

By using the register keyword in C you may declare a variable to be a register variable. (Only formal parameters or automatic
variables may be so declared.) This serves as advice to the compiler that you intend to use this variable heavily and it would be wise to keep its value in a register rather than moving it to and from memory. Of course you may not have a pointer to a register variable (since registers don't have addresses). And since a bit field may occur part way through an integral data type, bit fields in general do not have addresses. So you cannot have a pointer to a bit field either. Beyond that, you can have a pointer to just about anything you want.

One of the major uses of pointers in C lies in their use with dynamically allocated memory. This allows you to construct complex data structures as you need them, and to free the allocated memory when you no longer need it. For example, the code fragment

```c
struct PERSON *per;
per = (struct PERSON *) malloc(sizeof(struct PERSON));
```

illustrates several interesting features of C. First, a variable is declared as a pointer to a structure. Then memory of a certain size is allocated and the address of this memory is assigned to the pointer variable. The expression

```c
(struct PERSON *)
```

is called a cast and it forces one type to be treated as another. In this case, the address returned from malloc() will be treated as though it were a pointer to a PERSON structure. The size that malloc() is asked to allocate is specified by means of a sizeof expression. In C, "sizeof" is an operator that yields the size of the type of object to which it is applied. It may be applied to variables as well as to type expressions, in which case its value is the size of the type of object of that variable. Once a piece of memory is dynamically allocated in this manner and assigned to a pointer, it may be manipulated.

A typical example of the use of pointers in C would be in the construction of a binary tree out of nodes. You would first define a node structure as

```c
struct NODE {
    struct NODE *left;
    struct NODE *right;
    struct VAL value;
};
```

where left and right are intended to point to the left and right child nodes of the given node. Then each time you need a node in your tree, you allocate a NODE structure and set the appropriate pointer in its parent node to the new structure. Here the value field is assumed to contain some meaningful information for the node.

Another use of pointers in C is in achieving "call by reference." Function calls in C are always "call by value" which means that when a function is called with an argument a copy of that argument is made and passed to the function. However much the function may change the argument it receives, there will be no change in the original argument. (Note that this is unlike PL/I.) However, it is quite easy to achieve call by reference in C by passing the address of the argument whose value you want changed. The function then has access to the value at this address by means of the indirection operator (*). The following code fragment, for example would result in the value of num being incremented:

```c
int num;
/
*/ pass address of num to function */
increment(&num);

...*/ Where increment is defined as */
increment();
int *i; /* pointer to int */
{
  /* increment object pointed to */
i++;
}
```

The & operator is the address-of operator and yields the address of the object to which it is applied. And in general the expression "p" means "the object pointed to by p."

**ARRAYS**

C offers arrays of either one dimension or multiple dimensions. Arrays are indexed from 0, rather than from 1 as in some other languages. Thus, if myarray is an array of characters:

```c
char myarray[80];
```

then myarray[0] is the first character in this array and myarray[79] is the last. Character arrays are rather special in C since they are used to represent strings (there is no separate string data type in C). By convention, a string is a character array ending with the "null character" (\'\0\' or \'\0\'). While this is only a convention, it is one followed by all C programmers, and a number of standard library functions in C assume this convention. These are such functions as:

```c
strcpy "copy one string to another */
strlen "return the length of a string */
```

In addition to arrays of characters, you may have arrays of virtually anything else you like in C: shorts, longs, structures, pointers, unions. The only exception to this is that you may not have an array of functions (though you may have an array of function pointers).
OPERATORS IN C

C provides the usual arithmetic operators for addition, subtraction, multiplication, and division. There is no "power" operator, and this is regarded by some as a serious flaw when C is to be used for numerical analysis. However, operations involving powers commonly are implemented as library functions and these in turn may be implemented directly in the compiler as in-line functions.

Since C allows bit-level access to data, it is only reasonable that it permits bit-oriented operations such as:

> right shift
< left shift
& bitwise "and"
| bitwise "or"
^ bitwise "exclusive or"
~ complement.

The operands of each of these must be integer valued, of course.

Also available are the usual logical operators of:

&& logical "and"
|| logical "or"
! logical negation

and a rather peculiar ternary operator — the conditional operator:

?:

which appears in expressions of the form:

condition ? expr1 : expr2

Such an expression will have either the value of expr1 or expr2, depending upon whether or not condition is true. The two expressions that serve as the second and third operands must be of compatible types, and a conditional expression may appear wherever any "normal" expressions may. It frequently is a handy way of abbreviating an if construction involving a conditional assignment.

The familiar relational operators are available in C, though their forms may strike the novice as a bit odd at times:

== equality
!= inequality
> greater than
< less than
>= greater than or equal
<= less than or equal.

One point of continuing confusion and error (even to experienced C programmers) is the fact that the assignment operator is a single equals sign (=) while the relational operator for equality is a double equals sign.

C provides a variety of assignment operators, but most of these simply allow you to write complex expressions more economically. The normal assignment operator in C is of course

= 

And the other assignment operators are:

+= -= *= /= %= >>= <<= &= |= ^= &~

In the first case,

x += y;

means the same as

x = x + y;

and the other cases are similar.

C also supports the pre-increment and post-increment operators: += and +=. For example, the expression

x++; 

means the same as

x = x + 1;

while

x--;

means the same as

x = x - 1;

If one of these operators occurs before an expression, then the operation is applied to the expression before its value is used. Thus, for example, in the fragment

e if ( ++x == y )
{} 
...
}

the value of x is incremented first and it is then this (incremented) value that is compared to y.
Similarly, if the operator occurs after an expression, then the value of the expression is used prior to the incrementation. So in

```c
if (x++ == y)
    {
        ... 
    }
```

x is first compared to y, and only then is x incremented.

There are several other operators in C that bear mentioning briefly. First, there is the modulus operator (%) which yields the integral remainder of a division. There are of course the dot operator, mentioned above, that is used in referencing union and structure members, and the corresponding arrow operator (->) used to reference such members when dealing with a pointer to a structure or union. Finally, there is the somewhat mysterious and frequently misused comma operator (,) which may be used to construct complex expressions you wish to be evaluated in a particular order. Its use is too arcane for further discussion here.

CONTROL STRUCTURES IN C

C supports the following common control structures:

```c
while (condition)
    {
        /* statements */
    }
```

```c
for (init; condition; reinit)
    {
        /* statements */
    }
```

```c
do
    {
        /* statements */
    } while (condition);
```

In both while and for loops the condition is tested before the body of the loop is executed. However, in a do loop the body of the loop is executed prior to the testing of the condition.

In a C for loop the statement that forms the init condition is executed before the loop is entered, and the reinit statement is executed at the end of the loop before the condition is checked for the next iteration. In addition to these high-level control constructs C allows for labels and goto statements, though the use of these is quite infrequent in C programs.

A common control structure encountered in C programs is the switch statement (similar to select or case constructs in other programming languages). The general form of a switch statement is:

```c
switch (integral_valued_expression)
    {
        case i:
            /* statements */
            break;
        ... 
        case j:
            /* statements */
            break;
        default:
            /* statements */
            break;
    }
```

The expression being tested in the switch must be an integer-valued expression, and those in the case statements must be integer-valued constant expressions. If a break statement is not present, control will "drop through" to the following statement (which at times can be useful), and if a default statement is missing then control drops through to the statement following the body of the switch statement. When a default label is present, the statements following it are executed if the value of the test expression does not match any of those in the case statements.

Of course C has a version of the venerable if construction in the form:

```c
if (condition)
    statement;
```

or

```c
if (condition)
    {
        /* statements */
    }
```

The statement comprising the body of the if construct is executed provided that the condition evaluates to non-zero.

C PROGRAM STRUCTURE

I will not dwell on the details of the structure of a C program or the role of the compiler and linker in constructing such a program. For details of this nature, see my paper "Introduction to the C Programming Language," which appeared in the proceedings of The Fifteenth Annual SAS Users Group International Conference.

Typically a C program is composed of a number of (usually small) modules. Each module contains definitions of data and functions. A module may contain data definitions and no function definitions, or it may contain function definitions and no data definitions. The modules are compiled separately and linked together to form the C program.
OTHER TOPICS

The topics I have discussed here are sufficient to give you an overview of the major features of the C language, but there are many more topics that have been omitted. I have said nothing of the role of the C preprocessor or of its language, nor have I mentioned function prototypes and their use. For some discussion of these topics see my paper mentioned above.

Other topics worthy of note include the following:

- the C library
- constants and constant expressions (numeric, hex, octal, string, character)
- the ANSI Standard for C
- input and output in C
- C functions and return values
- declarations versus definitions.

To pursue these topics and get started programming in C, you should acquire a copy of the SAS/C Compiler, Student Edition. This book contains a full description of the C language, a complete tutorial for learning C, numerous examples integrated with the text, and software for practicing C programming on your PC.