Clarifying 'C'

Rick Homkes
Assistant Professor
Computer Technology
Purdue University
2300 South Washington Street
Kokomo, IN 46904-9003
317-455-9242

John Minor Ross
Associate Professor of
Data Processing
Indiana University Kokomo
2300 South Washington Street
Kokomo, IN 46904-9003
317-455-9213

Abstract

In the classroom of today, students learning C often have experience in one or more languages such as BASIC, COBOL or Pascal. With the possible exception of students with Assembler skills, many of these individuals stumble when making the change to C. This review of the teaching and learning processes surrounding C highlights areas where students either face misunderstandings or will predictably under-use new features.

Clarifying 'C'

Some readers may believe that Clarifying 'C' is an oxymoron. One (Pascal) colleague, upon hearing this title, said "that's like clarifying mud." Another computing pundit went so far as to say that "one of the biggest obstacles to the future of computing is C. C is the last attempt of the high priesthood to control the computing business (Pournelle, 1990)."

However, in spite of its detractors, the rise of C is no longer arguable. The marketplace has decided that the strengths of C outweigh the weaknesses. Referred to by Peter Norton as "industrial strength programming," C often provides programmers in college with their first chance to work on the edge. Norton (1987) summed up this way: *While it may be fair and accurate to throw bricks at C and bouquets at the likes of Pascal and Modula II in the name of programming design features, to do so would be to miss a very important point: professional tools work to a different standard and serve different needs than amateur tools. If C is a "dirty" language that allows programmers to do lots of tricky and powerful things, that's why its become the language of choice for deep professional programming.*

This is a guide to avoiding some of the potential pitfalls awaiting those new to the C language based upon the experiences of two C instructors. The primary intent of the review is to provide advice for the reader, someone currently teaching or getting ready to use C, on how to improve productivity by reducing 'C'onfusion.

C Syntax-related Confusion

When learning any new language (C, Ada, French, Chinese or whatever), there are new ways to say the same things. Certain ways are better than others. Certain ways are simply wrong.

Not surprisingly, there are many causes of learner confusion. Some confusion is attributable to chance but other syntax-related problems can be foretold by the instructor and,

thereby, avoided by the student. At least some of the confusion found among new C users can be mitigated through emphasis on how and when to apply various C syntax components and through following traditional good coding techniques.

Useful topic areas where extra emphasis in teaching can return dividends are described next. Sample code fragments are included because "seeing working code always helps (Plauger, 1992)."

'for' versus 'while' loops. There are times in C when it is easier and clearer to use a 'for' instead of a 'while' loop. This is because the 'for' controls are all shown at the top of the code. The advice to give students is if they need to initialize variables before entering a loop or if they need to step variables at the end of a loop then they should likely use a 'for' loop. The next two examples produce identical results.

```
/* Example A */
x = 10;
y = 20;
while (x) {
        printf("\n z=%d", x + y);
        --x;
        --y;
}

/* Example B */
for (x = 10, y = 20; x; x--, y--) {
        printf("\n z=%d", x + y);
}
```

Some loops may be coded so they fit on a single line. The loop then controls a NUL statement. If this is done, the student should use the {} around the semicolon to emphasize the NUL statement.

```
for (x = 10; x; printf("\n x = ", x--)) {;}
```

Multiple exits. In the rush to learn C syntax, the possibility exists for new C programmers to back-slide on structured programming issues. For example, functions with multiple return()s, multiple exit()s, or a mixture of both often show up on early student assignments. While this may not seem like trouble in a small routine it can lead to serious problems in debugging and supporting real code. Students need to be shown how multiple return code (Example C) can be avoided (Example D).

```
/* Example C */
int FuncABC(int this) {
        int x;
        if (this < 0) {
                return (-1);
        } else {
                if (this == 0) {
                        return (1);
                }
        }
        x = FuncXYZ(this);
        return (x);
}
/* Example D */
int FuncABC(int this) {
        int rcode = 1;
        if (this < 0) {
                rcode = -1;
        } else {
                if (this != 0) {
                        rcode = FuncXYZ(this);
                }
        }
        return (rcode);
}
```

Furthermore, the instructor can show students situations where a return() in one function is used to pass through the returned value from another function. The second example in the next section illustrates the code compression which is possible.

Ternary operator (? :). Students new to C commonly hate the ternary operator. Experienced C programmers hate to do without it. Essentially an if/then/else, the ternary operator allows powerful, clearly written code to often fit on one line. For example, the following line sets the exitsw variable to TRUE (1) if the operator enters an 'X' (or 'x') or FALSE (0) if any other key was pressed.

```
exitsw = toupper(getch()) == 'X' ? TRUE : FALSE;
```

Another example is a substitute for the multiple-return code shown in the prior section (FuncXYZ returns an int). Note the use of nested ternary decisions. Here ten lines of code shown earlier is replaced by one -- with no variables (x or rcode) defined.

```
int FuncABC(int this) {
    return ( this > 0 ? FuncXYZ(this) : (this ? -1 : 1) );
}
```

Buffered versus unbuffered input. Beginning C students soon have several methods of character input for their programs. However, this is a mixed blessing in that it often results in confusion over the delimiter for each method. Therefore an early discussion of a buffer is useful.

By explaining that some functions are separated from the original source of the input by a temporary storage area called a buffer, students understand better that functions that use a buffer, such as those accessing a file, have a delimiter of \n. Examples of those functions are getchar(), getc(), and scanf(). Other functions, such as getch() and getche(), bypass the buffer and directly receive input from the keyboard device. These functions use \r as the delimiter.

File "r" before "w". Since an fopen() in write mode will destroy an existing file of the same name without warning, it is a dangerous first move. Students should be told to avoid opening "new" files in write mode until they have opened them in read mode and have gotten an error (confirming that the file does not currently exist). An fopen() in read mode that does not return an error value should be followed by an fclose() and a message to the user of the program asking for advice on how to proceed (replace, exit, etc.).

= versus == operators. A classic mistake of new and old C programmers alike, is to use the = assignment operator when he or she meant to use the == test for equality. A debugging hint for the student is that this mistake usually results in a TRUE decision while also changing the value of the left operand (it is FALSE only if the right operand was zero). Another debugging hint is to use the editor to find all occurrences of a "=" and check if it was used appropriately.

Defines versus declare. In their 1978 foundation book on the C language, Kernighan and Ritchie stress: *it is important to distinguish between the declaration of an external variable and its definition. A declaration announces the properties of an external variable (its type, size, etc.); a definition also causes storage to be allocated.*

Unfortunately, many instructors (and texts) blur this distinction. The confusion is further complicated if a student who is unclear on these terms then tries to consider what a prototype (sometimes called a forward declaration!) is and how it fits in with the first two terms.

General Areas of Confusion

Storing Integers. Students coming into a C class usually have an understanding of positive binary numbers and binary arithmetic. However, the concept of negative binary numbers may remain unclear. A class discussion of two's complement notation for the storing of integer values may prove useful. In this notation an "on" high order bit is the "most negative value." Turning other bits "on" chips away at this negative value until a -1 is reached. Four representative values (using 8 bit storage) are:

```
0000 0001 =   1      1000 0000 = -128
0111 1111 = 127      1111 1111 =   -1
```

An example to demonstrate this is a routine to cube a number.

```
int cube(int a) {
        return(a * a * a);
}
```

Students run a program with this function for increasing sample values and record the results. When the students enter 32 and receive an answer of -32,768 a pause results, followed by the inevitable question: "How can this machine multiply three positive numbers and end up with a negative number?"

The instructor needs to explain that as the result gets larger, the first "on" bit migrates towards the left of the binary number. Finally the left most bit is turned on. At this point the number is interpreted as having a starting value of $-(2^{15})$ or $-32{,}768_{10}$. Input of numbers slightly larger than 32 result in a less negative number as bits other than the left most are turned on. However, an input of 41 results in positive 3385. This is the result of the left most "on" bit migrating left off the storage area and into "bit heaven" -- without an overflow error in C. The resultant number is back to positive, but it is still incorrect.

Modulus operator (%). The mod operator in C (and other languages) can make many programming tasks easier. It can prevent such troublesome occurrences as screen overflow (scrolling) and processing beyond the end of an array. (Reminder: mod division returns the remainder of a division operation and discards any whole number results. The result returned is an integer remainder between zero and the divisor minus 1. Examples: 6 % 3 returns 0; 7 % 3 returns 1.)

Consider a routine to print the next three entries in an array (char txt[20]). The value of x is incremented without regard to the maximum size of the array so that the array is cycled through as if it were a seamless loop (i.e., array element 19 is followed by 0 and 1). Using mod finds the next three characters without multiple "if" statements to reset x if it gets too big.

```
printf("\n%c %c %c",
        txt[x % 20], txt[(x+1) % 20], txt[(x+2) % 20] );
```

When x is 19, the characters printed will come from the desired array locations since 19 % 20 returns a remainder of 19, 20 % 20 has a 0 remainder, and 21 % 20 has a remainder of 1.

Page/Screen Breaks. Mod can also eliminate page or screen line counters from programs. To check for a page break, a brand new C programmer writes code based on traditional approaches using both line and page counters:

```
if (lines = = 55) {
        page = page + 1;
        PageBreak(page);
        lines = 0;
} else {
        lines = lines +1;
}
```

Using the ternary operator (without mod yet), the prior seven lines of code may be replaced with a single line (lines is incremented after the test and page is incremented before it is sent to the PageBreak function):

```
lines+ + = = 55 ? PageBreak(+ +page), lines = 0 : NUL;
```

Putting mod to work with the ternary operator uses a logical not (!) that is TRUE whenever mod returns 0 (lines is evenly divisible 55). Here the PageBreak function is passed the page number by dividing the lines by the page size. During program execution, lines is not reset to zero (watch out for integer overflow at 32,768 plus as mentioned earlier).

```
! (lines % 55) ? PageBreak(lines+ + / 55) : lines+ +;
```

Arguments. The organization of a C program into independent functions allows great freedom in writing code. It is not necessary to remember the hierarchy of function ownership. Instead, any function can be invoked (called) from any other function. Even the required function main() can invoke itself or be invoked from another function (though such recursion is usually a bad idea).

These independent functions often require independent, or local, variables. This leads to

the passing of variable values from the calling to the called function. Some texts refer to these values are parameters. However, a better term is argument. Argument is defined as "the independent variable of a function (American, 1985)."

An argument can be further differentiated as either a formal argument declared in a function's prototype and header or as an actual argument used in a function's call. It is important for students to note that an actual argument can be a numeric constant or alphabetic literal, but the formal arguments must be variables. Luckily, the value returned by a function seems to have no other name than that of a returned value.

Local variables. Local variables are created each time a function is invoked. Their scope (where their values are accessible) is within the function and their life (how long their values are accessible) is for the time that the function is active (unless they are defined as static types).

Students typically accept the previous statements without much question -- but also without total understanding. This is demonstrated by the receipt each semester of programs with a type of sideways recursion. In these programs main() calls a function to perform a task, but when the task is complete the function calls main() instead of simply returning to main().

While this does show that the concept of reusability of functions has gotten across, it is still bad code. Pointing out the error here always generates this response: "But it works." True, for small test data sets. However, the program will run out of memory for production runs because of the stack overflow caused by this hidden recursion. In order to demonstrate this problem the following function to calculate a factorial number (e.g. 3! = 3 * 2 * 1 = 6) is used in a laboratory test. The function is recursive so that it calls itself to execute as many times as there are factors in the expression.

```
int fact(int a) {
        return( (a = = 1) ? 1 : fact(a-1) * a);
}
```

Students run a program with this function and record the two problems that result. The first is a numeric overflow of the integer number with an initial value of 7. The second is a stack overflow when too many versions of the same function are resident in memory at the same time when the initial input value is 123.

Arrays. The introduction of arrays to new programmers often can be referred to as a 'blank stare generator.' The instructor starts talking about 'dense lists' of characters or numbers and students respond with a blank stare. However, C makes this first discussion easier because of the handling of strings. By making a string nothing more that a group of consecutive characters, it is much easier to visualize the string in memory as a simple array. Each character of the string or character array is stored in one byte of memory and the entire array is ended with the special character \0 (ASCII zero). It can be visualized as a line of text running from left to right.

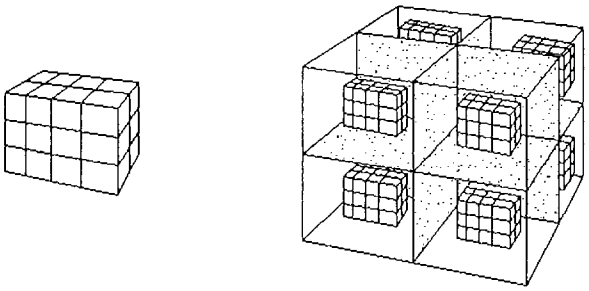
It is important to visualize the one dimensional array as a row instead of a column in order to move on to the two and n - dimensional arrays. This is because of the storage method used when a multidimensional array is kept in memory. This method is referred to as 'row major.' Thus a two dimensional numeric array can be defined and initialized as:

```
int mdarray[3][3] = {
        { 0, 1, 2},
        { 3, 4, 5},
        { 6, 7, 8}
};
```

The nine members of this two dimensional array are stored in consecutive bytes of memory in row order (i.e. 012345678). Adding a third

dimension to the array adds layers to the table. An array declared as [2][3][4] has two pages of a three row by four column table. Thus twenty-four storage locations are defined. The order of storage is column within row within page and the visual reference is a two by three by four cube.

Four and more dimensional arrays. Many minds start to bend when a fourth dimension is added to an array. While the declaration is simple, [2][2][3][4], the visual reference of the shape of this 'object' is not simple. However, it can be pictured simply as a row of cubes. Likewise, five dimensions can be pictured as a table of cubes and six dimensions can be pictured as a cube of cubes. The arrays [2][3][4] and [2][2][2][2][3][4] are shown below.



This relating of four or more dimensions back to a three dimensional shape can obviously be continued until the programmer gets tired of the ever expanding picture. Data storage is still linear with the last dimension (column) being the most minor for storage purposes.

Truth tables. A truth table of logical values is seen by students in many forms. These range from the sparse tables seen by students in a Logic Circuits class to the verbose tables seen by students in a Systems Analysis class. A few problems are constant, however. In creating a table, students often miss possible variations of the table, do not produce the table in ascending order, and do not understand the direct relationship of zero (FALSE) and non-zero (TRUE). The following function is given to students as a laboratory exercise to help in the understanding of a truth table.

```
void truthtable(void) {
  int a, b, c;
  printf("a\tb\tc\t(a||b)&&c\n");
  printf("_____\n");
    for (a=0; a<=1; a++) {
      for (b=0; b<=1; b++) {
        for (c=0; c<=1; c++) {
          printf("%d\t%d\t%d\t%d", a, b, c, (a||b)&&c);
          printf("\t %s\n", ((a||b)&&c) ?
                    "TRUE" : "FALSE");
        }
      }
    }
} /* end truthtable */
```

It produces a truth table with all eight possible combinations in ascending order using ((a||b)&&c) to display either a "TRUE" or "FALSE." This will not end all confusion, but it does give students a routine to modify to try other logical expressions.

Formatting errors. Those just learning C should resist the pressure to get something running and then make it look better. "The price you pay in finicky typing and editing is repaid many times over [Plauger]." Code that looks bad will be much harder to debug and support. This includes trouble-compounders like: poor variable and function naming style, lack of consistent indentation, and lack of internal documentation.

While avoiding creating bugs sounds like a platitude, it is probably the most effective form of debugging. The best method of bug avoidance is to know what is expected of your program and to develop clearly written specifications before writing the code. In addition, writing a comment block for each function -- that describes its input, output, and assumptions -- will not only prevent bugs, but will make them much easier to find, not only now, but three years in the future (Baldwin, 1991).

After learning the foundations of the language, the programmer should move up from the level zero error detection level they probably started with -- even if it takes longer to satisfy the

compiler that they really know what they are doing. Simply put, the programmer's motto should be 'Plan Your Program and Program Your Plan.'

Hints and Conclusion

As with all classes, it is important to know your audience. A considerable number of "grad" students take beginning C classes. These students already have a degree(s) and careers but want to know C. The instructor needs to be careful not to focus on the interests of these higher-level students. This approach quickly alienates and loses the undergraduates the course was designed to serve. If willing, the "grads" can serve as mentors for some of the traditional students.

Picking C software is somewhat more difficult as both Microsoft and Borland move to blend C and C++ product lines together. Using a C++ product in a C class would allow the use of new comment formats (//... instead of /*...*/) and I/O functions (cin/cout instead of scanf/printf) albeit at the expense of backwards compatibility. Perhaps more useful, will be the possibility of having a two-class sequence, ANSI C followed by C++ for object-oriented programming, where the students would use the same compiler and editor for both.

Finally, the C instructor should explain advantages and disadvantages of various approaches, showing more than one right way to accomplish a task. Code from the book can be modified on-the-run if a computer projection system is in use. Another way to show multiple approaches to problems where each student fully understands the specifications is to post student programming assignments (two source code listings are turned in -- one is marked during grading and returned to the student, the other is available for taping to the board). Such "good examples" may be demonstrated by the author in class if a computer projection system is available.

References

The American Heritage Dictionary. (1985). Boston: Houghton Mifflin Company.

Baldwin, W. Debugging In C -- An Overview. C Users Journal, 9(10). (Computer Select, October 1992, #74755)

Pournelle, J. The BYTE Summit: Obstacles to Overcome. BYTE, September 1990, 281.

Kernighan, B. W., & Ritchie, D. M. (1978). The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc., 76.

Norton, P. PC Magazine, February 10, 1987, 75-76.

Plauger, P. Bugs. C Users Journal, 10(9). (Computer Select, October 1992, #2055)

Figure drawn by MET student Paul Pepka.

EIGHT ENOUGH REFERENCES From Page 44

6. References

[1] R. L. Kruse, Data Structures & Program Design, Prentice Hall, Englewood Cliffs, New Jersey, 1987, 586 pages.

[2] R. Sosic and J. Gu, "A Polynomial Time Algorithm for the N-Queens Problem", SIGART Bulletin, Vol. 1, No. 3, October 1990, 7-11.

[3] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall, Englewood Cliffs, New Jersey, 1976, 366 pages.