C++ > C + OOP

(A PERSONAL VIEW OF TEACHING INTRODUCTORY
PROGRAMMING USING C/C++)

Venu Dasigi
Department of Computer Science and Information Technology
Sacred Heart University
5151 Park Avenue
Fairfield, CT 06432-1000
dasigi@shu.sacredheart.edu

ABSTRACT
Many Computer Science departments are moving toward C or C++ as the
programming language used in teaching the introductory computer science sequence.
Some departments and many students believe that an initial introduction to C is a good
idea since it allows for a smooth transition later into C++. In this paper, some of the
problems in using C as an introductory programming language are briefly reviewed, and
some specific issues and solutions that are possible when C++ is adopted are pointed
out. It is concluded that while both C and C++ have their disadvantages as
introductory languages, the latter would be the better choice between the two.

1. INTRODUCTION
Since the advent of the Pascal programming language in the late 1960s, it has been widely
used as an introductory programming language in teaching computer science. Pascal was invented
primarily for teaching the best of structured programming practices, in a simple language. Several
other modern languages came about in the following decades, some incorporating the evolving
paradigms for reliable software development. Most notable among them were Modula-2, Ada, C
and C++. (This excludes the very mathematically oriented functional programming paradigm, on
which languages like LISP, Scheme, ML, etc. are based, and the logic programming paradigm, on
which PROLOG is based.) Modula-2 is a direct descendent of Pascal, with limited means of
encapsulating data types based on pointers and packaging related procedures together. Ada has
extensive abstraction mechanisms for data (packages) as well as modules (generics), and supports
all this with its encapsulation mechanism. C, a relatively older language, dating back to the early
1970s, was apparently motivated by concerns of conciseness and efficiency of code, closely tied to
the underlying machine. Its popularity grew with that of the UNIX operating system, with which
it is closely associated, both in terms of the history of development as well as in its interaction
through system calls. The limitations of C led to the evolution of C++, which attempted not only
to incorporate object-oriented programming principles into the otherwise procedural language, but
to address the limitations of and be backward compatible with C, as well.

With such a variety of choices available today to teach the introductory computer science
sequence, including functional programming languages, different universities have gone in different
directions, with almost all the major languages having a following. The most common choices seem
to be those mentioned before, namely, Pascal, Ada, C and C++, with an increasing trend toward C/C++. The reasons for this trend appear to be that much of the software development in the industry is based on C, and it is an important skill for students to market themselves. C has, however, faced considerable well-deserved criticism in the literature, not in the least as a candidate high level language to introduce computer programming [Mateti, 84]. Some studies indicate that there are no pronounced differences in student performance caused by a switch from a traditional language (e.g., Pascal / Modula-2) to a language like C++ [Hitz and Hudec, 95]. Some believe that the pragmatic considerations of using an "industry" language such as C/C++ might even motivate students better. In this paper, I quickly review some of the problems in using C as an introductory programming language, suggest specific solutions to some of the problems, and also indicate how the choice of C++ might help in addressing some of the issues. The phrase "introductory computer science sequence" and its variations used in this paper refer to the first few courses taken by computer science majors, that is, those courses that incorporate the basics of structured programming and data structures.

2. USE OF C AS THE INTRODUCTORY PROGRAMMING LANGUAGE

Kerninghan and Ritchie's classic book on the C programming language points out in [Kerninghan and Ritchie, 78]: C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages.

In pointing out that C is not a big language, they perhaps refer to the fact that C has only a small number (fewer than thirty of them in the original definition) of reserved words, and that it is designed for and encourages a terse programming style. They acknowledge that C is a relatively low level language, indicating perhaps that several constructs of the language are closely tied to the underlying machine architecture (e.g., pre-/post- auto incrementation and decrementation, bit shifting operators, etc.). Indeed, C imposes few restrictions on the programmer by being quite permissive about mixing types, not providing for bounds checking, etc. A consequence of some of these design decisions, for correctly written programs, is improved efficiency and good exploitation of machine architecture. An attestation to the extreme view of C as a glorified assembly language are programs that are part of the folklore, which look like a sequence of garbage characters, but after successful compilation, can be run to produce interesting Christmas rhymes [Phillipps]!

It is often said that some of the above features and the lack of complete type checking in C are both its strength and weakness. Those who like C enjoy the flexibility C provides, along with the fast compilation and efficient machine code. Those who dislike C critique these same features for their lack of safety features, and for how easy C makes it for a student / novice programmer to write incorrect programs. I have come across many such students, some of whom have been "liberated" by C after semesters of struggling with strongly typed languages and their fussy compilers, who often wrote programs that "compiled without a hitch", but not infrequently caused them trouble at run time. (To put things in perspective, most of these students had not learned C in a systematic way from an instructor, but were required to pick it up on their own, based on their past experience with the more strongly typed languages.) I also remember a few students who were exposed to a strongly typed language for the first time, after having known no language other than C for years, and were pleasantly surprised by how their programs ran correctly once they had satisfied the compiler.

I also taught the C++ programming language and its object-oriented features to students who had programmed for some time in C and considered themselves very proficient in the language. The typical student in this group expected that (s)he would have a very easy time in the C++ course, but at the end invariably wondered how "steep their learning curve" ended up being. Much
of this phenomenon spoke for the intricacies of the object-oriented paradigm as an almost new way of thinking, but some of the wonderment often related to the non-object-oriented features of C++. That made it somewhat more secure than C. In the next section, several specific features of C that spell trouble for first-time programmers are pointed out, and some solutions are indicated, both within C itself, as well as using C++ as a better version of C. Others also address similar concerns and describe their experience in using C as the introductory language, but do not focus on as many specific issues as done here [Roberts, 93].

3. SOME SPECIFIC CONCERNS FOR INTRODUCTORY PROGRAMMING

There are a number of specific issues that come up in using C or C++ in teaching introductory programming. They may be broadly categorized into concerns related to types and type checking, pointers, lack of reference parameters and a variety of miscellaneous concerns. Each is dealt with in turn.

3.1. Types and Type Checking

C is not a strongly typed language. For instance, the following program, which freely mixes the distinct types char and int, compiles without a problem and produces the output as shown, when tested with both the unix cc compiler as well as a gnu C compiler. (Since the variables here serve no deeper purpose, single letter names have been chosen.)

```c
#include <stdio.h>
main()
{
    char c, d, e, b;
    int i, j, k;
    float f;
    c = 'a';
    i = c;
    d = i+1;
    j = c+2;
    e = j;
    k = i*i;
    b = k;
    f = e;
    printf("%c %c %c %c %d %d %f\n", c, d, e, b, i, j, k);
}
/* produces the output:
 a b c A 97 99 9409 99.000000
 */
```

Given the permissivity of C in freely mixing types, it is easily possible that unintended typos might compile correctly and produce unintended and surprising output, as happens above, to some extent. Since it is generally observed that certain typos and other casual errors often manifest themselves as type mismatches, lack of type checking in an introductory programming language is a major handicap. How can this issue be addressed in an introductory programming course? There is no easy answer to this question. It would be doubly more important on the part of the professor to emphasize the importance of correct use of types. It should also be emphasized that the friendliness of the compiler in forgiving (or at least overlooking) many kinds of type mismatches is only superficial, since not only do type mismatches result in incorrect output, frequently the output
would be nonsensical. Some responsibility that is normally assumed by the compiler gets shifted to the professor when C (or even C++) is used in introductory programming.

3.2. Pointers and Dynamic Allocation

A variation on the theme of lack of type checking is the interchangeability of pointers and array variables in many contexts. Kernaghan and Ritchie point out that any operation that can be performed using array subscripting (e.g., a[5]) can also be done using pointer arithmetic (e.g., *(a+5)), and that the latter version would in general be faster [Kernaghan and Ritchie, 78]. They also caution that the pointer version would also be harder, especially for "the uninitiated". The argument about speed is not very convincing, both because a good compiler should be able to generate machine code that is equally fast in both cases, especially given that the responsibility of figuring the size of the array elements is left to the compiler even when pointer arithmetic is used, but also because the speed gain, if any, is marginal. The concern about the difficulty in understanding or designing the pointer version is perhaps more to the point, because it encourages the programmer to think from the perspective of the machine, which is admittedly a hard task, and which is generally what high level languages are designed to free the programmer from.

Thus, I believe that when C/C++ is used as the language of choice in an introductory programming language course, any discussion of the close connection between pointers and arrays should be viewed as beyond the scope of the course, and even detrimental to the course objective, and so should be left out of the syllabus. Arrays should be introduced as in any other high level language, with no mention of how they relate to pointers. Granted, something like *(a += 5) would not be easy to do when only arrays and no pointers are used, but that would be a topic for discussion in the context of pointers in C/C++.

All primitives of C for dynamic allocation require one to specify the number of bytes to be allocated. Further, the primitives return a generic pointer (void *), which requires appropriate casting before being dereferenced. These are details that are easily handled by the compiler, yet the programmer must worry about two specific details. The pattern is easy enough, but students get into trouble by forgetting one or the other of these details in a statement whose purpose is otherwise straightforward enough:

```c
nodeptr1 = (node *) malloc(sizeof(node));
nodeptr2 = (node *) malloc(num*sizeof(node));
    /* version 1 for allocating arrays */
nodeptr2 = (node *) calloc(num, sizeof(node));
    /* version 2 for allocating arrays */
```

A good way to abstract away from this detail is to define a macro for this purpose, since this is a pattern used often enough. Something like:

```c
#define new(type) (type *)malloc(sizeof(type))
#define newarray(type,num) (type *)malloc((num)*sizeof(type))
    /* version 1 for allocating arrays */
#define newarray(type,num) (type *)calloc(num,sizeof(type))
    /* version 2 for allocating arrays */
nodeptr1 = new(node);
nodeptr2 = newarray(node, m+n);
```

This method is very effective, and the macros could be made part of a generic header file the students would use throughout the course. In the context of a first programming course, the new and newarray macros may be introduced as primitives, with the caveat that they would not work unless the generic header file is included. For the newarray method, the calloc version is preferable.
to the other version, because the latter fails to work correctly when the parentheses around num in the macro definition are forgotten, and an expression such as m+n in the last example is passed as the actual parameter corresponding to num.

3.3. Reference Parameters

One of the most striking problems with C as a language for introducing programming is its lack of reference parameters. All parameters are passed by value, but to quote from [Kernighan and Ritchie, 78], "fortunately, there is a way to obtain the desired effect" by simulating parameter passing by reference using pointers. This puts a substantial load on the novice programmer, and even practicing C programmers (who must have known no language that allows reference parameters) have been known to stumble when the underlying subtleties are brought to the fore, as in the following example to delete the node containing the smallest element from a binary search tree:

typedef struct node *nodeptr;
struct node
{
    int element;
    nodeptr left, right;
};

deleteMin (node **tree)
{
    node *temp;
    if (!*tree)
        printf ("Given tree is empty!\n");
    else if (!(*tree)->left)
    {
        temp = (*tree);
        *tree = (*tree)->right;
        free(temp);
    }
    else
        deleteMin (&(*tree)->left);
};

It can easily be seen that this code can be very hard to understand because of the absence of reference parameters in C. Simulation of reference parameters using pointers is in itself not easy for many first time programming students to understand, much less to implement. Implementation involves not only passing a pointer to the parameter intended to be changeable, but dereferencing it each time it appears in the body of the function, as well. And then if the programmer forgets to pass the address of the actual parameter in a call to such a function, things will not work right. In such a situation, all that many C compilers do is to issue a warning (that a pointer and an integer are being illegally combined)! At the core of this confusion for most students is a lack of appreciation of the difference between changing the value of a pointer (which is what the above deletemin function attempts to do) and changing the value that a pointer points to (which is typically what happens in simulating reference parameters through pointers). Fischer and Grodzinsky call this simulation mechanism call by pointer and discuss it in some detail [Fischer and Grodzinsky, 93].

What can be done about this problem if C were to be used in an introductory programming course? Not much, unfortunately, other than to instruct the students in the nuances of simulating
reference parameter passing using pointers. It does not appear that an easy macro-based solution can be arrived at for this problem. This weakness of C makes a strong case in favor of C++, if career, market or political considerations dictate that either C or C++ be used to introduce basic programming.

A specific manifestation of the problem of reference parameters being simulated by pointers is the requirement in C that only references be passed as actual parameters to scanf and its variations. This requirement is particularly annoying in that when the programmer forgets the ampersand before the variable name, it generally results in the wrong behavior or even a segmentation fault. Roberts suggests the development of a simplified I/O library to address this problem [Roberts, 93]. He acknowledges that such an approach amounts to teaching a non-standard local version of C, but points out, rightly so in my opinion, that the emphasis in introductory courses ought to be on teaching programming, not a language. 3.4. Other Miscellaneous Issues

There are a number of other issues that create difficulties for inexperienced programmers and students. All of the following problems show up in both C and C++:

- case sensitivity
- use of the semicolon before the else in an if statement
- need for endsw in a switch statement
- for loops that are not really counting/indexed loops
- use of 0 in three different senses: the integer 0, the Boolean false and the null pointer

There is not much that can be done about case sensitivity, although this is a source of constant headache in introductory programming classes, it was a deliberate design decision in the languages. Adopting a simple, mnemonic and consistent naming convention, e.g., all lower-case characters unless multi-word variable names are involved (e.g., ValueSought), would alleviate the difficulties to a large extent. As experienced C programmers know, the semicolon must be present before an else if it follows a simple statement, but must be absent if it follows a compound statement (that is a set of statements enclosed by { }). The need for endsw often leads to unexpected errors when forgotten. There is not much an instructor can do about these last two issues, other than to explicitly caution the students about them.

The for loop in C/C++ is really a different form of the while loop, and is not really the traditional indexed/counting loop that it is in most other languages. This is misleading to those who expect the for loop to perform some repeated task for several evenly spaced values of an index variable, and is detrimental to readability. However, the instructor could make a prescription to the student that the construct be used only for counting purposes as in:

```c
for (i=1; i<=n; i++) {...};
```

or

```c
for (i=n; i> =1; i=3) {...};
```

but not for something like

```c
for (p=head; (p & & (p->data != ValueSought)); p=p->next) {
```

because clearly the last example corresponds to a traditional while loop.

As for the use of the value 0 in multiple senses, to be understood from context, the instructor would do well to include something like the following in the generic header file for the class, and require the students to stick to an appropriate, consistent use of the identifiers:

```c
#define FALSE 0
#define NULL 0
```

Actually C++ allows a better way to do the above, using typed constants, and that naturally leads the instructor to consider C++ as a significant alternative to C, even when the object-oriented features of C++ are viewed as beyond the scope of the course.
4. USE OF C++ AS THE LANGUAGE OF CHOICE

Although many colleges and universities seem to use C to teach introductory programming, few departments have opted for C++. The main reason for this decision seems to be that the major strength of C++ is its object-oriented features, which are generally beyond the scope of the first course or two in programming. I believe that C++, sans its object-oriented features (essentially, classes and encapsulation, polymorphism and inheritance), still has a number of attractive features that make it possible to view it as a better C from the standpoint of teaching introductory programming. The discussion in the previous section is intended to set the premise to this point. Below, the features of C++ that make it substantially more suitable for teaching introductory computer science, compared to C, are examined. The discussion roughly parallels the issues raised in the previous section.

According to Bjarne Stroustrup, the developer of C++, the type rules of C++ are a result of the C experience. However, C++ is not significantly stricter than C in this respect because of the backward compatibility goal in the design of C++. Newer features of C++, e.g., the new function and classes, however, appear to be a consequence of the acknowledgement of the significance of type checking. To quote from [Stroustrup, 94], "I saw, and still see, type checking as a practical tool rather than a goal in itself. It is essential to realize that eliminating every type violation in a program doesn’t imply that the resulting program is correct or even that [it] cannot crash because an object was used in a way that was inconsistent with its definition. For example, a stray electric pulse may cause a critical memory bit to change its value in a way that is impossible according to the language definition. Equating type insecurities with program crashes and program crashes with catastrophic failures such as airplane crashes, telephone system breakdowns, and nuclear power station meltdowns is irresponsible and misleading..."

Actually, there is an anecdote that lends credence to the view of those people who, according to Stroustrup make "irresponsible and misleading" statements [Tropp, 84]. This story is about how, in the early days of FORTRAN, the permissive lexicalization rules and default typing rules of FORTRAN caused a statement (with a typo in it: the period key hit instead of the neighboring comma key) such as:

```
DO 3 I = 1.3
```

to be interpreted as:

```
DO3 I = 1.3
```

and consequently caused a space mission to be aborted. Our concern here, anyway, is teaching introductory programming, a context where unusual typos and syntax errors can be expected to be common. (To be fair, Stroustrup perhaps did not have this context in mind.) Thus, although it cannot be guaranteed that a program containing no type violations is correct, it certainly increases the likelihood of its being close to correct and decreases the frustration normally caused by program crashes that are more likely to result from type violations. The program of Section 3.1 compiled equally well (with minor modifications to conform to C++) under a gnu C++ compiler. Thus C++ is not significantly more helpful in terms of type checking, when mostly C compatible features are used. C++ has been criticized on a number of other counts, as well [Joyner, 92].

As for the use of pointers, C++ is consistent with C in the way it treats the relationship between pointers and arrays, and I believe it is best to avoid any references to this close relationship between pointers and arrays in introductory courses. C++ does offer a better solution with respect to dynamic allocation, namely, a built-in, type sensitive new function. Thus, in

```c
int *i; float *f;
i = new int; f = new float[10]; // compiles fine
i = new float; f = new int; // won't compile
```
The second line is legal, but the last line does not compile. As can be seen, the built-in new is overloaded to dynamically allocate an array of the specified number of elements of the given type.

In contrast to C, C++ not only has reference parameters, much as in languages like Pascal, but it supports the constant reference parameter mode, as well, when arbitrarily large structures are to be passed to functions, yet not taking up too much space on the program stack.

All of the miscellaneous issues of Section 3.4 apply to C++ as well as C. However, the very last issue, which points to the need to distinguish the different senses of the literal 0 to improve readability, lends itself to a slightly better, type sensitive solution, e.g.,
\[
\begin{align*}
\text{const int FALSE} & = 0; \\
\text{const Node *} \text{NULL} & = 0;
\end{align*}
\]

In general, most constant declarations that are achieved in C through the use of \#define may be done in C++ through typed constant declaration. Another related point here is the use of \#define for defining function macros. C++ provides the inline function feature to do the same thing, to a much better end. Consider the two versions below:

\[
\begin{align*}
\text{\#define sqmod(i,j)} & \quad i*i \% j \quad /\ast \text{the C version} \ast/ \\
\text{inline int sqmod (int i, int j)} & \quad \{ \text{return i*i \% j;} \} // \text{C++ version}
\end{align*}
\]

The second version is clearly superior to the first one, because it works better for all the following uses, where the first version does not work correctly:

\[
\begin{align*}
\text{int x, y, z; float f, g, h;}
\text{z} & = \text{sqmod(x+y, z-1);} \\
\text{z} & = \text{sqmod(x+++, y);} \\
\text{h} & = \text{sqmod(f, g)};
\end{align*}
\]

In the first use above, the \#define version can be made to work after enclosing each occurrence of the identifiers i and j in parentheses in the definition, but for the next two cases, there is no easy solution in C. The C++ version handles the second use correctly by establishing a parameter correspondence, and issues a warning about type mismatch between the actual and formal parameters in the third case (although raising an error would be better).

C++ offers some other advantages over C, such as function and operator overloading, which is a version of polymorphism that can be exploited to advantage even in non-object-oriented contexts. C++ also allows functions to return references, a feature that takes operator overloading to its true logical conclusion, e.g., by making it possible to define overloaded operators such as the subscripting operator [], the input operator >>, etc., that can be used as lvalues.

5. CONCLUSION

I conclude that neither C nor C++ is a very appropriate choice as a language to teach introductory programming, in spite of their individual strengths in other contexts. If pragmatic considerations dictate the adoption of either C or C++ for introducing programming in a curriculum, C++ would certainly be a better choice. It also has the advantage that when it is time to introduce data structures, data abstraction concepts can be readily introduced through C++ classes, opening the doors into object-oriented programming. Also, I tried to point out a few solutions to the common weaknesses of C/C++ when used to teach basic programming.

ACKNOWLEDGEMENTS

I thank two of the anonymous reviewers for their valuable comments.
REFERENCES


