TIGHTER, NEATER, SAFER C AND C++

Boyko Bantchev

Institute of Mathematics and Informatics
boykobb@gmail.com

***Abstract**: Constructs are presented for alternative — tighter, neater and arguably less vulnerable — expression of frequently occurring patterns in C and C++ programming. We find them useful in several ways, both in teaching and in participating in programming competitions. Working programmers can also benefit from such or similar constructs.*

***Keywords**: program structure, preprocessing, C++ template metaprogramming*

Introduction

Programmers often feel the need to adapt their programming language to particular needs. Adaptation, when it is possible, may take different forms, e.g., creating libraries, esp. of higher-order procedures, or any sort of metaprogramming — the choice depends on the actual scope of capabilities provided by the language.

Here we consider several instances of bending C and C++ to a more rational use. In some of them 'rational' means achieving straightforward and concise expression through removing apparent redundancy. In others, constructs are introduced that provide more suitable and versatile repetitive statements than those built in the language.

All proposed constructs have been motivated by the author's practice of teaching programming to high and higher school students. Their implementation makes use of the C/C++ preprocessor, as well as the operator overloading and template programming facilities of C++.

For convenience of reference, complete definitions of all constructs under discussion are given in an Appendix at the end of the paper.

Printing values and names

Programming invariably involves debugging, and a major part of debugging is tracing the values of all sorts of data entities. Value tracing can be automatized when dedicated debugging software is used, but a debugger may not be available, or, for a number of reasons, not used even if present. The programmer then has to manually insert print statements in the program.

Print statements are of course also used for displaying the results of programs that communicate with the user through the standard output.

In a typical situation of teaching programming to novices, the user-program communication is indeed through the standard input and output channels. Print

statements are used to display values and thus assure oneself that the program behaves as intended. For example, a number is printed which is supposed to be the result of adding some numbers, or a sequence of numbers is printed to confirm that they are in some specific order. And in case they are not, additional values need to be printed in order to identify the exact source of erroneous behaviour.

A teacher would want to achieve his goals regarding any particular program example without having to bring attention to details of the language that are insignificant to the example. In addition, simple and frequently used actions should be expressed concisely. How does this apply to printing values? Consider printing the values of two variables in C, a character and an integer:

```
printf("%c %i\n",letter,num)
```

Powerful though `printf` is in the hands of an experienced programmer, to a novice it is a nuisance. The concept of formatting strings with format specifiers and value replacement, along with some other details, constitute a significant memory and work load to the uninitiated, and yet are irrelevant to the algorithmic and other aspects of the particular program.

It is somewhat better in C++, where formatting, let alone type-specific one, can be left to a convenient default:

```
cout << letter << " " << num << endl
```

but some unnecessary detail — four instances of `<<`, explicit output of a space character — is still present.

The problem aggravates when the names of the variables also have to be printed rather than just the values (in order to make the printed information more useful). Now we need, in addition, to specify each name twice — as a string and as a variable — and to separate names from values. The above example can be rendered, in C and C++ respectively, as:

```
printf("letter:%c num:%i\n",letter,num)
```

and

```
cout << "letter:" << letter << " num:" << num << endl
```

Either of these is already rather verbose and is difficult to get correct. Clearly, with more or longer names the verbosity of such a command increases drastically, and so does the difficulty of writing and even reading it. When the same or similar commands have to be placed at several or more locations, the task becomes really daunting.

The problem can be solved by introducing simple macros. In C++, we define `S` to print a value of an expression and a space after it, and `D` to display a variable name followed by the respective value and a space. The macro `N` prints a newline character. Thus, the above examples reduce to

```
S(letter) S(num) N
```

and

```
D(letter) D(num) N
```

In C, we are forced to define specific macros for each datatype — `SC`, `SI`, `SF`, and `SS`, and similarly `DC`, `DI`, `DF`, and `DS`, but otherwise there is no redundancy, e.g.

```
SC(letter) SI(num) N
```

and

```
DC(letter) DI(num) N
```

In C++, we also define `DD`, which lists the contents of a collection, such as a vector, a list, etc., and then calls `N`. This is useful mostly for saving the user the necessity to define the required iteration variable and for making that variable local.

The above macros are useful not only to novices and their teachers but also to experienced programmers, as the time and attention needed to add and edit data tracing commands to a program is very substantially reduced. A notable case where time is a particularly precious resource and routine activities should require as little attention as possible is participating in a programming competition. As the macros are small and easy to reproduce, it is a good tactic to ensure they are available by typing them at, or even before the beginning of a competition.

Using macros to define `D` and the similar C macros is essential, as the pre-processor has a means (the # unary operator) to obtain a string from a name, which cannot be done at a language level. Having defined all the necessary commands as macros has the further advantage that these commands can be made ineffective by replacing their definitions with empty ones, which can be done with a single pre-processor command. Thus a debugging mode can be switched on and off with a minimum program change. This can even be done separately at different places.

Stack operations on sequences

The stack data structure is one of the most intensively used in programming. As sequential data structures, such as vectors, lists, or strings, are often used as stacks, it is natural to ask how convenient this is.

The basic stack operations (apart from creation, checking for emptiness and other similar), are adding and removing a value, conventionally referred to as pushing and popping. With respect to them, from practical programming point of view, we find it reasonable to postulate the following properties desirable:

- atomicity — each operation should be expressed as a single action;
- conciseness — each operation should be denoted concisely;
- evaluability — each operation should produce a value;

- composability — this is closely related to the above and means the ability to perform several operations of the same kind in a row;
- uniformity — each operation should have the same denotation and the same operational properties for all kinds of sequences.

It is easy to observe that these properties do not hold in C++. For example, the push method (named `push_back`) for vectors, linked lists, deques and strings is not particularly concise, does not produce value, and therefore also is not composable. The pop operation is not atomic. There is, for example, a `back` method that produces the topmost stack value but does not remove it from the stack. That method is also not composable. Removing the top value off the stack can be done e.g. by `pop_back`, but the latter does not produce the removed value (or any other value).

The C++'s stack container adapter is identical in the said respects (apart from `push_back`, `back` and `pop_back` being called `push`, `top` and `pop`, respectively).

To overcome these deficiencies, it is useful to implement push and pop operations on sequences generically and ensuring the above stated properties.

We have chosen to implement push as `<<`, and pop in two variants, `>>` and `~`, each returning a different kind of value. `>>`, with its left argument a sequence container and right argument a variable (i.e., a reference), returns the container, as does `<<`. Thus both `<<` and `>>` admit composition. `~` is a unary operator on a container, returning the value that is being removed from the stack.

As an example, if `s` is a sequence, then

```
s << 3 << a << b << 10 << 15
```

adds several values to its end,

```
s >> x >> y
```

assigns the two last added values — 15 to `x` and 10 to `y`, and

```
while (!s.empty()) S(~s); N;
```

pops up and prints each remaining value, leaving `s` empty.

The C struct names problem

In the C language, a compound, heterogeneous type is defined according to the pattern

```
struct S { ... };
```

where the user-selected identifier `S` is the name of the new type. However, to designate the type for whatever purpose, e.g. in declaring variables, `struct` or `union` fields, function parameters and results, etc., `S` alone does not work; the phrase `struct S` has to be used instead. This includes the very frequently oc-

curring case when the definition of that same structure `S` needs to be self-referential, because it contains pointers to `S`. For many C programmers, having to type two words to designate a single notion is a sign of redundance which they would rather avoid. As this part of the language is syntactically anomalous, it is also inconvenient to teach and a source of errors for novice programmers.

The problem does not occur in C++, where any of `struct S` and `S` can be used to refer to the same datatype. Ideally, we would like to achieve the same or as close as possible in C, and with minimum coding and language distortion.

There are several ways to partially solve the problem which one can find in textbooks on C or on data structures — they all resort to C's `typedef` instruction which serves to define aliases to type expressions. These partial solutions are analyzed in [1]. Our solution combines the use of `typedef` and C's preprocessor, and for a number of reasons we believe that it is the optimal possible one (for this, again see [1]).

The solution consists of the following definition:

```
#define struct(n) typedef struct n n; struct n
```

where the word `struct` becomes also the name of a macro. Now `struct` types can be defined using that macro, as e.g. in

```
struct(S) { ... S* ptr; ... };
```

and variables of such types can be defined as we wished:

```
S x, y, *p;
```

This new use of the word `struct` does not prohibit its traditional use as a keyword; effectively `S` and `struct S` are the same everywhere in the program.

Loop enhancement

In programming language design, loops have always been a notoriously difficult subject. Several decades ago, the computing literature was flooded by hundreds of attempts to find better if not 'best' answers to the question of what control structures should a language provide at an intraprocedural level. Most of the research was dedicated to loops, and one particularly comprehensive paper is [2].

This research wave does not seem to have been very influential to the design of the widely used programming languages, as most of the then observed problems remain. C and C++ are not exceptions.

Here we propose a set of macros that can be used to both teach and practice designing and writing procedural code with a more elaborate structure than raw C and C++ permit. The use of the standard preprocessor poses certain limitations and prevents introducing really adequate syntax, but avoids the need for using other software along with the compiler. If more ambitious macrodefinitions are to be attempted, a more accomplished tool may be needed, such as m4 [3].

The most general loop construct in C and C++ — the `for` statement — allows repetition to be structured into four different parts; let us call them prologue, condition, body, and renewal. Each of the prologue, condition and renewal can only be an expression, which is probably justified for the condition but not for the other two. The prologue also permits a single variable to be defined which is local to the loop. This too seems a rather artificial limitation: two or more local variables may be needed.

On the other hand, there are at least several specific patterns of repetition occurring frequently enough to deserve simpler, dedicated loop constructs.

Exiting from within the loop is something that C (and C++) is half-equipped with: its `break` statement ensures leaving the loop, but, paradoxically, it is never a body statement — instead, it is nested at least one level deeper than the body itself. The same holds of `continue`.

It sometimes proves useful if an optional finalization part — an epilogue — can be attached to a loop. If present, the epilogue is the last point where the local variables of the loop are in use.

The above notes inform the functionality of our loop enhancing macros, which can be summarized as follows:

- introducing specific forms of loops;
- arbitrary number of local variables (both for the general loop and for most of the specific ones);
- arbitrary content within the prologue and renewal parts;
- providing loop epilogue;
- conditionally exiting the loop or the body's current execution from within the body, possibly with a specific (pre)epilogue.

Finally, let us introduce the macros themselves by use of several examples.

The following code fragment reverses a vector. The variables `i` and `j` walk the vector from both ends. Because we want to avoid mutual influence between the loop and the rest of the program, `i` and `j` are locals of an enclosing block.

```
{
int i,j;
for (i=0,j=a.size()-1; i<j; ++i,--j)
  swap(a[i],a[j]);
DD(a)
}
```

One unwanted effect here is that there are necessarily two levels of nesting, the enclosing block and the loop itself. Another is that `i` and `j` are not initialized where they are defined. Surely, we could have written

```
int i=0,j=a.size()-1;
for (; i<j; ++i,++j) ...
```

instead, but thus the prologue of the loop is entirely outside it.

The following code is in fact equivalent to the above, but the `loop` and `fin` macros hide the enclosing block and ensure that the prologue is in place:

```
loop (def(int,i=0,j=a.size()-1),
      i<j,
      swap(a[i],a[j]),
      (++i,--j))
fin(DD(a))
```

In general, `loop` expects three or four arguments, namely prologue, condition, body and removal, where only condition must be an expression, and the other three are statements or sequences of such.

`fin`, when its argument is non-empty, provides an epilogue.

The `def` macro is useful when two or more variables of the same type must be defined — such a definition cannot be written directly, as the commas in it would confuse the loop macro.

The `rep` macro implements a loop with a given repetition count. If additional arguments are given, they are a prologue. Thus, the code

```
rep (3,char c='a')
  rep (5) S(c++); fin (N)
fin (D(c); N)
```

makes use of a local variable `c` whose value is advanced in both loops. The above fragment outputs

```
a b c d e
f g h i j
k l m n o
c:p
```

The macro `fori(t,i,a,b)` represents an 'arithmetic' loop — one that runs a variable `i` of type `t` through the range [`a`,`b`). The following example

```
fori (int,i,0,7) D(i);
fin (N)
```

outputs `i:0 i:1 i:2 i:3 i:4 i:5 i:6`.

The macro `fora(a,i,n,...)` walks an array `a` or a part of it using `i` as an index variable. If given, `n` defines an upper bound of the index range to run through.

If n is omitted, the entire array is walked. Possible additional arguments define a prologue. Thus, e.g.

```
fora (s,k,,int s[] = {0,1,2,3,4})
  S((char)('A'+s[k]));
fin (N)
```

outputs A B C D E.

The macros exitif and contif are meant as a replacement of break and continue. In fact, they are precisely break and continue, each within an if and possibly preceded by arbitrary statements. By hiding the nesting of break and continue, these macros improve the program readability: unlike break and continue, exitif and contif can be immediate members of a loop body.

References

1. Bantchev B. C struct names with no verbosity. http://www.math.bas.bg/bantchev/misc/c-struct-names.html.
2. Knuth D.E. Structured programming with goto statements. ACM Computing Surveys, Vol. 6 (1974), No. 4, pp. 261-301.
3. m4 — macro processor. POSIX.1-2008, 2013 Edition, Base Specifications. http://opengroup.org/onlinepubs/9699919799/utilities/m4.html.

Appendix: complete definitions

```
// general definitions (C & C++)
#define asize(a) (sizeof(a)/sizeof(*(a)))
#define def(t,a,...) t a,__VA_ARGS__;

// struct definition
#define struct(n) typedef struct n n; struct n

// displaying values, C++ style
#define S(x) (cout << (x) << ' ')
#define D(x) (cout << #x":" << x << ' ')
#define N (cout << endl)
#define DD(c) {for (const auto& x: (c)) S(x); N;}



// displaying values, C style
#define SC(x) (printf("%c ",(x)))
#define SS(x) (printf("%s ",(x)))
#define SI(x) (printf("%i ",(x)))
#define SF(x) (printf("%f ",(x)))
#define DC(x) (printf(#x":%c ",x))
```

```
#define DS(x) (printf(#x":%s ",x))
#define DI(x) (printf(#x":%i ",x))
#define DF(x) (printf(#x":%f ",x))
#define N (putchar('\n'))

// iteration (C & C++)
#define rep(k,...) {__VA_ARGS__; for (int _=k; _>0; --_) {
#define fori(d,i,f,t) {d i; for (i=f; i!=t; ++i) {
#define fora(a,i,n,...) { \
        __VA_ARGS__; int _ = #n[0]=='\0' ? asize(a) : n+0; \
        for(int i=0; i<_; ++i) {
#define loop(a,b,c,...) {a; while(b) {c;__VA_ARGS__;
#define exitif(c,...) if (c) {__VA_ARGS__; break;}
#define contif(c,...) if (c) {__VA_ARGS__; continue;}
#define fin(a) }a;}

// stack operations on sequences (C++)
template <typename C, typename I>
C& operator<<(C& c, I x) {
  c.push_back(x);
  return c;
}

template <typename C, typename I>
C& operator>>(C& c, I& x) {
  x = c.back();
  c.pop_back();
  return c;
}

template <template <typename,typename> class C, typename I>
I operator~(C<I,allocator<I>>& c) {
  I x = c.back();
  c.pop_back();
  return x;
}
```

ПО-СТЕГНАТ, ПОДРЕДЕН И БЕЗОПАСЕН С И С++

Бойко Банчев

***Резюме**: Представят се конструкции за алтернативно – по-стегнато, по-подредено и, надяваме се, по-малко уязвимо за грешки – изразяване на често възникващи схеми на програми на C и C++. Смятаме, че те са полезни по няколко начина при обучение и в състезания по програмиране. Такива или подобни конструкции могат да бъдат от полза и на професионалните програмисти.*