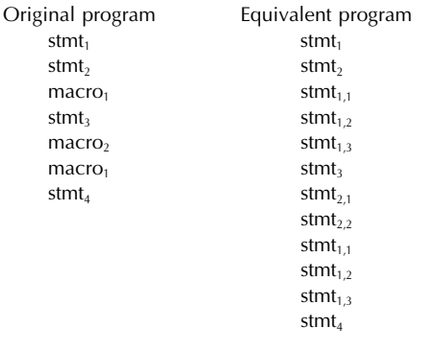

Figure 1. Illustration of macro expansion.

MACROS

A *macro,* or *macroinstruction,* is a computer programming language construct that defines an abbreviation for a longer sequence of statements or instructions. Macros can be used to simplify programming and reduce programmer errors, to improve the clarity of code by hiding underlying complexity, or to extend the syntax of the programming language itself. They are a common feature found in many environments.

A macro definition consists of a name or *prototype* and a *body* composed of the programming language statements for which the prototype serves as an abbreviation. A *macro call* is an occurrence of a macro's name in the text of the program. At some point prior to the actual processing of the program text (often called the *preprocessing* phase, to distinguish it from the actual compilation or interpretation phase), macro calls are *expanded* (i.e., they are replaced with the body of the macro definition, called the *expansion* of the macro). Thus, for every program that contains macros, there exists an equivalent program containing no macros; this is simply the text of the original program with all the macros expanded and is the result of the preprocessing phase.

Figure 1 illustrates the process of macro expansion. The original program on the left contains three macro calls, including two calls to the same macro. Note in particular that these two calls to the same macro result in different copies of the statements making up the body. Macros are based upon the concept of literal inclusion of the text in the body of the definition, and this is the main difference between macros and subroutines. Although used for a similar purpose, subroutines are based on the concept of multiple calls to the same shared code. This contrasts with the situation of multiple macro calls, which result in multiple copies of the text of the definition, one for each call.

A program using macros may take up more space than an equivalent program using subroutines. However, in order for multiple subroutine calls to share common code, certain additional code is required to coordinate these calls and handle branching to the subroutine code, saving and restoring of registers and other machine state, and returning to the main code. This code is additional overhead at runtime; in some cases, this overhead can dwarf the time required to execute the actual code in the subroutine. Thus, the choice between macros and subroutines often represents a tradeoff between the size of the eventual code and the speed at which the code executes.

The basic mechanism already described can be made more useful by the addition of a number of extensions. First among these is the addition of formal parameters to the macro prototype; these may be referenced in the body of the macro either by name or by position. Actual parameters are supplied with the macro call and replace occurrences of the formal parameters in the macro expansion.

A second extension is to allow macros to be nested (i.e., to allow the body of a macro to contain macro calls itself). Nested macro calls are recursively expanded during preprocessing. Care must be exercised to ensure that this does not result in an infinite recursion, and so this extension is often combined with conditional expansion of macros. It is useful in this context to allow evaluation of arbitrary expressions during the preprocessing phase as well.

A third extension is to allow the creation of unique identifiers for use as labels or variables. This is necessary because the expanded code of separate macro calls is nearly identical, and for some purposes (e.g., assembly language statement labels) unique identifiers are required.

MACROS IN DIFFERENT PROGRAMMING LANGUAGES

Macros and similar constructs are found across the spectrum of programming languages and can even be found in application software such as word processors. Although some languages do not include a macro facility as part of the language definition (e.g., FORTRAN, Java), a stand-alone preprocessor can be used to obtain the benefits of macros with any language.

The first macro processors accompanied the first assembly languages. Because the instructions provided by a machine language are generally very low level, macros were extremely

useful for allowing the programmer to work at a higher level. For example, a common operation is to save or restore all the general-purpose registers; when no single machine instruction is available to do this, a macro can be defined that expands to the necessary sequence of instructions.

The C programming language includes a macro facility as a part of the "C preprocessor," which also provides other facilities such as file inclusion and conditional compilation. In C, macros are typically used to provide a single point of definition for literal constant values (e.g., array sizes) and as a replacement for functions in cases where in-line code is more efficient than a subroutine. For example, the Standard C Library contains a function `getc` that reads a single character of input. This "function" is actually a macro because the overhead of a subroutine call outweighs the time required to process a single character.

The C preprocessor also includes advanced capabilities for *stringitizing* macros (enclosing the result of the macro expansion in double quotes so that it is treated as a string literal by the compilation phase) and for *token pasting* (combining adjacent lexical tokens, one of which is typically the result of a macro expansion, to form a single token). In addition, a number of predefined macros can expand to the current source file name, the current date, and so on.

The UNIX operating system includes a stand-alone macro processor called `m4` that can be used to provide macro capabilities with any programming language. The `m4` macro processor includes many powerful features, including conditional expansion of macros, that allow recursive macros to be written. The fact that `m4` is not tied to any one language can be a disadvantage; for example, it does not understand (and will try to expand macros within) the structure of C language comments.

The C++ programming language inherits all the macro facilities of the C language preprocessor and adds two new facilities as well. These facilities, *in-line functions* and *templates*, are not macros in the strict sense; however, they are based on the same concept of textual substitution. Furthermore, they are not a part of the preprocessor, as are the C language macro facilities but are part of the C++ language definition itself.

In-line functions are meant for situations in which the overhead of a subroutine call would exceed the amount of work accomplished by the subroutine itself (e.g., the `getc` macro discussed earlier). Macros attack this problem by *inlining* the body of the subroutine, avoiding run-time overhead at the expense of increased code space. However, macros do not always provide the same semantics as a function call, and this can lead to a number of pitfalls for the unwary programmer (see details later in this article). In-line functions provide the same benefits as macros by in-lining the body of the subroutine (in most cases; the code may not be in-lined in complex cases such as recursive functions), while avoiding their pitfalls by providing precisely the same semantics as a normal subroutine call.

Templates are used to allow the specification of a family of C++ classes or functions, parameterized by type. During compilation, templates are *instantiated* in a process similar to macro expansion to create the required classes and functions according to the actual types used in the program. Templates are an important mechanism for supporting generic programming; a common application is the construction of container data structures, such as stacks and queues, which are indifferent to the type of data that they contain.

The Lisp programming language includes a powerful macro facility. In Lisp, the process of macro expansion occurs not at the textural level as in most other languages but at the expression level. A Lisp macro is actually an expression that is evaluated (this corresponds to the expansion process) to produce a second Lisp expression (the expansion). This allows for very flexible macro expansion because the expansion of the body of a macro can be controlled using any of the programming language constructs in the Lisp language. Macros in Lisp can become quite complex and often involve special "quoting" operators in order to provide very fine control over the expansion process.

Many modern office productivity applications contain some kind of macro facility; similar to macros in programming languages, a macro in these applications is a shorthand for a longer sequence of commands. Generally a user has the ability to "record" a macro, during which the application stores the sequence of commands given by the user. Later the user can "play back" (analogous to macro call expansion in programming languages) the macro, and the entire sequence of commands will be executed.

MACRO PROCESSORS

To translate macros, one can use preprocessors or embed the macro translation into the interpreter. Parameters that occur in a macro can be referenced positionally or by name. Named parameters are more convenient in instances where there are a large number of formal parameters, some of which may get default values.

Languages such as C have a separate preprocessor to handle macros. The macro preprocessor works in a fashion similar to a translator, with three important phases. The first phase consists of reading the macro definitions; the second phase consists of storing these definitions; and the last phase consists of expanding macros occurring in the program text.

Factors that need to be considered include computing the position of formal parameters (if they are referred to positionally) as well as substituting actual parameter values in macro expansions. The macro preprocessor also must maintain a symbol table containing the macro prototypes. If recursive or nested macro calls are permitted, extra care must be taken in the macro preprocessor.

The macro preprocessor is capable of detecting a number of errors. These include errors in the macro definition (e.g., multiple definitions of the same macro), as well as in the macro expansion (e.g., calling a macro with the wrong number of arguments).

The operation of the macro preprocessor can consist of either one or two passes. In a two-pass preprocessor, macro definitions are read and accumulated in the symbol table during the first pass, and macro expansion takes place during the second pass. Figures 2 and 3 give a pseudo-code description of a two-pass macro processor.

Implementation Details

A macro name table is implemented similarly to a symbol table in an assembler or a compiler. Hash table algorithms are used to insert and find entries in Macro name tables.

```
Read a line from the input;
while (end of file is not encountered)
{
        if (line contains a macro name)
        {
                Write the macro name in the macro name table;
                Prepare the formal argument array list;
                Set Macro_definition_phase = True;
        }
        else if (Macro_definition_phase==True)
        {
                Enter line in the macro definition table after
                substituting position numbers for formal parameters.
                if (end of macro definition is encountered)
                {
                        Set Macro_definition_phase = False;
                }
        }
        else
        {
                Write line back to the output;
        }
Read a line from the input;
}
```

Figure 2. Pseudo-code description of pass one of a two-pass macro processor.

To implement recursive macro calls, actual parameters are pushed on to a stack. The actual parameters are substituted for formal parameters after reading lines from the Macro definition table. When the end of a current macro definition is encountered, the actual parameter stack gets popped.

One-Pass Macro Processor

The two-pass macro processor described earlier makes the functionality of the processor explicit. As already mentioned, a two-pass macro processor cannot handle macro definitions inside a macro call. Also, for a two-pass macro processor, it is unnecessary for a macro to be defined before a macro is called (or used).

The steps involved in a single-pass processor are the same as a two-pass processor, namely, reading, storing the macro definitions, preparing both the formal and actual parameters, expanding the macros, and writing to the output. A single-pass algorithm also maintains information about whether a macro is being defined or expanded. Unless a macro is defined inside a macro call (this case is rare among programs), the state of the single-pass processor is either a definition phase or an expansion phase. If a macro is defined inside a macro expansion (macro call), the algorithm substitutes for actual parameters and enters the definition in the macro definition table. The macro name is also entered in the macro name table.

In a single-pass algorithm, a macro must be defined before it can be used. However, by maintaining a chain of macro calls that call yet-to-be-defined macros, a single-pass algorithm expands macros when they become defined.

APPLICATIONS

Search Problems

Search problems are an important class of problems. To obtain a solution to a search problem, we often look at the entire

```
Read a line from the input;
while (end of file is not encountered)
{
        if (line contains a macro name)
        {
                if (macro name appears in the macro name table)
                {
                        Set Macro_expansion_phase = True;
                        Prepare the Actual Parameter List;
                }
                else
                {
                        Error ''Macro Not Yet Defined'';
                        exit;
                }
        }
        else if (Macro_expansion_phase == True)
        {
                Read line from the macro definition table;
                Substitute Actual Parameters for Positions;
                if (end of macro is encountered)
                        Set Macro_expansion_phase = False;
                else
                        Write line to output;
        }
        else
                Write line to output;
if (Macro_expansion_phase == True)

        Read line from Macro definition table;
else
        Read line from the input;
}
```

Figure 3. Pseudo-code description of pass two of a two-pass macro processor.

solution space. There are many different methods of searching this solution space (e.g., local neighborhood search, gradient methods, and linear programming).

A tree is implicitly traversed while searching the solution space (e.g., a binary search tree when one performs a binary search in an ordered collection of entries). Macros can be used to speed up such searches by expanding the statements at compile time and effectively doing the recursion during the macro expansion rather than during the execution of the program. Fletcher (1) describes a backtracking algorithm using macros that solves a tiling problem involving polyominoes. Bitner and Reingold (2) show how to use macros to solve a large number of combinatorial problems. Such recursive uses of macros require a macro preprocessor capable of conditional macro expansion and cannot be accomplished in languages such as C and C++.

Assert Macros

Macros are often used to make assertions about what the programmer expects to be true during the execution of a program. Using such macros makes it easier to track down errors as well as to understand the program. Rosenblum (3) suggests the following guidelines in using assertions:

1. Assert explicit programmer errors.
2. Assert public API functions.

3. Assert assumptions.
4. Assert reasonable limits.
5. Assert unimplemented and untested code.
6. Assert classes.

This functionality is provided as a macro for the following reason: as a macro every assertion will result in a distinct bit of code, making it possible to refer, via the special macros built into the C preprocessor, to the name of the file and line number where the assertion occurs in the program text. If and when an assertion fails, this information can be printed out, making it easier for the programmer to track down the source of the problem. This would be impossible if assertions were implemented as subroutines.

Include Facility

The C preprocessor's "include" facility is similar to the use of macros. It allows one file to be textually included in another; usually this is used for data and macro definition statements. Many common system routines are accessed via include files, such as ⟨`stdio.h`⟩, ⟨`math.h`⟩, and ⟨`stdlib.h`⟩.

Block Structure

Coplien (4) describes how macros can be used to add the features of a "block-structured" language to C++ by using macros. This is an example of using macros to extend the syntax of a language.

Text Formatting Languages

Text formatting languages such as LaTeX and AMSTeX are macro packages written on top of TeX. This makes using document formatting languages much easier. Publishers have their own styles, and they use macro statements to facilitate the style. Many drawing packages (e.g., `idraw`) use macros as an intermediate language in their storage of figures.

Scripts

Many programming languages use scripts and macros interchangeably. Some modern programming languages (e.g., Visual Basic and Tcl/Tk) and many spreadsheet programs use macros extensively. The advantage of using macros for scripts is the ability to cut and paste statements to construct a program without knowledge of the details. Even a computer virus has been written using macros (5).

PITFALLS OF MACRO USE

Even though macros are convenient and powerful, their use in programming languages can be dangerous if care is not exercised. The following are a few of the pitfalls that can result from the naive use of macros.

- Hard to understand code. Just as macros can be used to make code more clearly understood, when misused they can make code harder to understand. This is especially the case when macros are deeply nested, making it difficult to understand what the macro expansion will ultimately look like.
- Hidden use of registers, declaration of variables, etc. Another danger of macros is that they can hide the use and/or declarations of variables and other resources. A novice programmer can experience difficulties when the code surrounding the macro conflicts with the code in the expanded body.
- Confusion over the results of expanded code. This problem is also a result of the interactions between the body of the expanded macro and the surrounding code. For example, a macro may be expanded within an expression, and the result of the evaluation of the macro body may depend on the rules of operator precedence. It is for this reason that macros in the C programming language are commonly surrounded by parentheses in order to make explicit the order of evaluation that is expected.
- Expressions evaluated twice. Expressions can and often are given as actual parameters in a macro call. If the corresponding formal parameter appears more than once in the body of the macro, the expression will be evaluated multiple times in the expansion. This is problematic for expressions that have side effects.
- Type mismatches. It is impossible to check that the types of actual parameters in macro calls are correct, because such checking depends upon the context in which these parameters appear in the expansion. This results in errors being detected only after macro expansion, which can make tracking down the source of the error difficult.
- Confusing scope effects. The expanded macro code can have confusing interactions with regard to the scope of variables. For example, a naive macro containing two or more C language statements will not have the expected effect if it is used in an if-then-else construct.
- Tool problems. Because the programmer sees one thing (the macro call) and the language processing tools see another (the expanded code), macros can lead to problems with various tools. This is especially common with debuggers, which typically are unable to single-step or otherwise diagnose the code inside of a macro definition. Macros usually must be debugged separately from the code in which they appear by expanding the macro call and examining the result by hand.

CONCLUSION

Macros are a common feature in most programming languages. The key to understanding them is the idea of macro expansion; a macro call is replaced with the text in the macro's definition, possibly with parameter replacement and recursive expansion of nested macro calls. The use of macros requires a modicum of care, and a number of pitfalls must be avoided. However, used properly, macros are a useful and powerful tool.

BIBLIOGRAPHY

1. J. G. Fletcher, A program to solve the pentamino problem by the recursive use of macros, *Commun. ACM,* **8**: 621–623, 1965.
2. J. R. Bitner and E. M. Reingold, Backtrack programming techniques, *Commun. ACM,* **18**: 651–656, 1975.

3. B. D. Rosenblum, Improve your programming with asserts, *Dr. Dobb's J.,* **22** (12): 60–63, Dec. 1997.
4. J. O. Coplien, *Advanced C++ Programming Styles and Idioms,* Reading, MA: Addison-Wesley, 1992.
5. J. O. Kephar et al., Fighting computer viruses, *Sci. Amer.,* **277** (5): 88–93, Nov. 1997.

M. S. KRISHNAMOORTHY
JOHN D. VALOIS
Rensselaer Polytechnic Institute

MAGLEV. See MAGNETIC LEVITATION.