Systems Programming

Chapter 4

Macro Processors

1

Macro (Instruction)

- a macro represents a commonly used group of statements in the source programming language
  - simply a notational convenience for the programmer
- expanding the macros: the macro processor replaces each macro instruction with the corresponding group of source language statements
  - Ex: On SIC/XE, it is necessary to save the contents of all registers before calling a subprogram and restore them on return:
    - corresponding instructions can be made two macros: LOADREGS and SAVEREGS.

2

- The functions of a macro processor essentially involve the substitution of one group of characters of lines for another.
  - Except in a few specialized cases, the macro performs no analysis of the text it handles
    - looks at the form, not the meaning of statements
- Most common use of macro processor is in assembler language programming, but macro processors can be used with high-level programming languages, O/S command languages, etc.

Macro processors usually work in one pass and not directly related to the machine architecture.

3

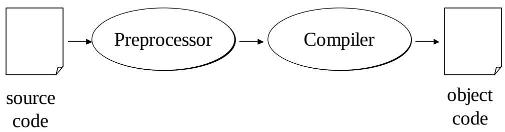
Conditional Macro Expansion

- Most macro processors can modify the sequence of statements generated for a macro expension, depending on the arguments supplied in the macro invocation
  - great power and flexibility
- Implementation is easy : macro processor maintains a symbol table that contains the current values of all macro variables processed. This table is used to look up the current value of a macro variable whenever it is required.

4

ANSI C Macro Language

- Definitions and invocations of macros are handled by a preprocessor
  - not integrated with the rest of the compiler



5

#include preprocessor directive

- causes a copy of a specified file to be included in place of the directive
- Two versions:
  - #include "filename"

    preprocessor searches in the same directory as the file being compiled for the file to be included
  - #include <filename>

    used for standard library header files, the search normally performed through pre-designated directories

6

#define preprocessor directive

- creates symbolic constants and macros

```
#define PI   3.14159
```

identifier   replacement-text

replaces all subsequent occurrences of the symbolic constant PI with the numeric constant 3.14159.

7

#define preprocessor directive

- A <u>macro</u> is an operation defined in a #define preprocessor directive
  - without arguments – processed like a symbolic constant
  - with arguments – arguments are substituted in the replacement text, then the macro is expanded

```
#define  ABSDIFF(X,Y)   ( (X) > (Y) ? (X) – (Y) : (Y) – (X) )

ABSDIFF(I+1, J-5)
```

Common programming error: Forgetting to enclose macro arguments in parentheses in the replacement text.

8

Conditional Compilation

- enables the programmer to control the execution of preprocessor directives and the compilation of program code
  - Each of the conditional preprocessor directives evaluates a constant integer expression

9

Uses of Conditonal Compilation

- To make sure that a macro is defined at least and at most once.

```
#ifndef NULL   or #if !defined(NULL)
#define NULL 0
#endif
```

- to control the inclusion of debugging statements

```
#define DEBUG 1
#if DEBUG == 1   or #ifdef DEBUG
   code prevented from compiling
#endif
```

10

#error and #pragma

#error tokens

- prints an implementation dependent message including tokens specified in the directive

```
#error 1 - out of range error
```

#pragma tokens

- causes an implementation defined action
- A pragma not recognized by an implementation is ignored
  - Borland c++ recognizes several pragmas that enable the programmer to take full advantage of the Borland's compiler

11

Predefined Symbolic Constants

- __LINE__ line number of current source code line
- __FILE__ presumed name of source file
- __DATE__ compilation date as Mmm dd yyyy
- __TIME__ compilation time as hh:mm:ss
- __STDC__ int constant 1, to indicate that implementation is ANSI compliant

12

Assertions

- assert macro is defined in assert.h
  - tests the value of an expression
  - if value of expr is 0, becomes false
- prints an error message and calls abort function of stdlib.h to terminate program execution

```
assert (R != 0);
x = y / R;
```

- if symbolic constant NDEBUF is defined subsequent asserts will be ignored.
  - Use #define NDEBUG when assert is no longer needed

13

and

- # stringizing operator
  - argument substitution is performed in the usual way, but the resulting string is enclosed in quotes

```
#define DISPLAY(EXPR) printf(#EXPR ″= %d\n″,EXPR)
   vs.
#define DISPLAY(EXPR) printf(#EXPR ″= %d\n″,EXPR)
```

TRY: `DISPLAY(I*J+1)`

- ## concats two tokens

```
#define TOKENCONCAT( x, y)    x##y
```

TRY: TOKENCONCAT(O, K)

14

MACRO PROCESSOR DESIGN OPTIONS

- Recursive macro extension
- General-purpose macro processors
- Macro processing within language translators
  - Line-by-line macro processor
  - Integrated macro processor

15

Recursive Macro Expansion

- Invocation of one macro by another
- It is not difficult if the macro processor is being written in a programming language that allows recursive calls
  - macro processor recursively processes the macros until all are resolved.

Try:

`DISPLAY(ABSDIFF(3,8))`

16

General-Purpose Macro Processors

- not dependent on any particular programming language, but can be used with a variety of different languages
  - Advantages:
    - programmer does not need to learn about a different macro facility for each compiler or assembler language
    - costs involved in producing a different macro processor for each language is not needed

17

General-Purpose macro processors are not common

- large number of details that must be dealt with in a real programming language
    - a special-purpose macro processor can have these details built into its logic and structure
    - a general-purpose facility on the other hand, must provide some way for a user to define the specific set of rules to be followed.

18

- Implementation problems related to the differences among langauges
  - There are several situations in which normal macro parameter substitution should not occur
    - e.g., different comment styles: /* */ or //
  - Grouping statements in languages highly differ
    - e.g., { }, begin end
  - Tokens and rules for forming tokens differ
    - e.g., = and :=
  - syntax used for macro definitions and macro invocation statements should be similar to language to make it more readable and writeable.

19

Macro Processing within Language Translators

- The macro processors that we have discussed so far are preprocessors.
- A line-by-line macro processor: combines macro processing functions with the language translator itself.
  - macro processor reads the source program statements and performs all of its functions as previously described
  - However, the output lines are passed to the language translator as they are generated

20

- Advantages:
  + avoids making an extra pass over the source program
    + more efficient – some of the data structures can be combined
    + makes it easier to give diagnostic messages related to the source statement containing the error

21

- Integrated macro processor: instead of passing information macro-processor and translator, they are combined as one unit.
  - can potentially make use of any information about the source program that is extracted by the language translator
    - special rules of the language are handled by the translator
      - ex: If macro involved substituting for the variable name I in the FORTRAN statement DO 100 I = 1

22

Disadvantages of integrated and line-by-line macro processors

- must be specially designed and written to work with a particular implementation of an assembler or compiler
- the costs of macro processor development must be added to the cost of the language translator
- the assembler or compiler will be considerably larger and more complex

23