**Pre-Processor Directives**

The pre-processing stage of the compilation of a C++ module resolves all directives in the source code and prepares a translation unit for the compiler proper.  This resolution includes inserting the contents of all specified header files and replacing and/or expanding all macros.

This chapter describes the pre-processor directives, lists those available to the programmer and describes each of these directives in detail.

Syntax

The pre-processor converts directives that are embedded in the original source code.  Each directive begins with # and occupies at least one full line in the source file.  The directive may include leading whitespace and may consist solely of the character #.  We call such an empty directive the *null directive*.  The null directive has no effect.

The pre-processor directives include:

* #define - defines a macro
* #include - inserts a file
* #ifdef - brackets conditional implementation
* #ifndef - brackets conditional implementation
* #endif - ends conditional implementation
* #error - sends an error message to stdout
* #pragma - passes a directive to the compiler or linker

To view output from the pre-processing stage, we specify the pre-process-only flag on the command-line.  This output is sometime helpful in debugging source code that contains improperly constructed directives.  The compiler flags are:

* cl /P \*.cpp
* g++ -E \*.cpp

Macros

A *macro* defines the substitution of a text string for a symbolic name.  We use macros to set constant values and platform-dependent values.  A macro may include textual parameters.

We specify the replacement text in the macro's definition and insert its symbolic name wherever the replacement text should appear in the source code.  Two directives identify macro definitions:

* #define - define the replacement text for a symbolic name
* #undef - cancel all replacement text for a symbolic name

The pre-processor recognizes categories types of macros:

* object-like macros
* function-like macros
* pre-defined macros

Object-Like Macros

An object-like macro defines an unmodifiable replacement text.  The definition of an object-like macro takes the form

|  |
| --- |
| #define SYMBOLIC\_NAME Replacement\_text |

**SYMBOLIC\_NAME** is the placeholder throughout the original (unprocessed) source file.  By convention, we capitalize this symbolic name.  This all-capitals convention highlights the macros throughout the source code.  An underscore character - \_ - between adjacent words within the symbolic name enhances its readability.  The symbolic name is not a C++ name as defined in the introductory chapters and does not refer to any C++ entity.

The symbolic name precedes the replacement text and whitespace separates them.  The replacement text terminates with the end-of-line character and may include any number of whitespace characters other than end-of-line.  The pre-processor replaces comments within the replacement text with a single blank space.  The replacement text itself is optional.

The pre-processor replaces every occurence of the symbolic name with the replacement text throughout the original source code, except within string literals, character constants, comments, and #include file names.  For example, the pre-processor converts

|  |
| --- |
| #define PI 3.141592654  double r = 2.35;  double area = PI \* r \* r;  double volume = (4.0/3.0) \* PI \* r \* r \* r;  std::cout << "PIr^2 is " << area << std::endl;  std::cout << "(4/3)PIr^3 is " << volume << std::endl; |

into

|  |
| --- |
| double r = 2.35;  double area = 3.141592654 \* r \* r;  double volume = (4.0/3.0) \* 3.141592654 \* r \* r \* r;  std::cout << "PIr^2 is " << area << std::endl;  std::cout << "(4/3)PIr^3 is " << volume << std::endl; |

A macro definition differs from the definition of an initialized variable in two ways.

* there is no assignment operator
* there is no terminating semi-colon

Note that the PI's enclosed within the literal strings have remained unaltered.

Continuation

A macro definition may extend over several lines.  The backslash character - \ - immediately followed by the end-of-line character identifies a continuation onto the next line.  For example,

|  |
| --- |
| #define PI 3.141\  592654 |

is the same as

|  |
| --- |
| #define PI 3.141592654 |

Note that there is no whitespace between 3.141 and the backslash character and no whitespace before 592654, ensuring a contiguous string of digits.

Function-Like Macros

A function-like macro is a macro with one or more parameters in its definition.  A function-like macro is similar in form , but not identical, to a C++ function definition.

Single-Parameter Macros

A function-like macro with a single parameter takes the form

|  |
| --- |
| #define SYMBOLIC\_NAME(P) Replacement text including one or more P's |

There is no whitespace between the symbolic name and the opening parenthesis.  The parentheses enclose the parameter's name (P).  The pair of parentheses may include whitespace.  The parameter serves as a placeholder for itself in the replacement text.  This placeholder may appear several times in the replacement text.

Each reference to a function-like macro in the original source code takes the form of a function call.  The pre-processor substitutes the argument in the macro for each appearance of the placeholder in the replacement text.  For example,

|  |
| --- |
| // Function-Like Macros  // fnMacros.cpp  #include <iostream>  #define PI 3.14  #define AREA(R) PI \* R \* R // function-like macro  int main() {  double area = AREA(2.35);  std::cout << "Area is " << area << std::endl;  } |

The pre-processor substitutes 3.14 for PI and 2.35 for R in two separate locations

|  |
| --- |
| // Function-Like Macros  // fnMacros.i  // iostream declarations here  int main() {  double area = 3.14 \* 2.35 \* 2.35;  std::cout << "Area is " << area << std::endl;  } |

If any whitespace is present between the symbolic name and the opening left parenthesis, the pre-processor interprets the left parenthesis as the first character of the replacement text.  For example, the pre-processor converts the following code

|  |
| --- |
| // Function-Like Macros  // fnMacrosError.cpp  #include <iostream>  #define PI 3.14  #define AREA (R) PI \* R \* R // WHITESPACE BEFORE (  int main() {  double area = AREA(2.35);  std::cout << "Area is " << area << std::endl;  } |

into

|  |
| --- |
| // Function-Like Macros  // fnMacrosError.cpp  // iostream declarations here  int main() {  double area = (R) 3.14 \* R \* R(2.35) /\* <- CAUSES COMPILER ERROR \*/;  std::cout << "Area is " << area << std::endl;  } |

Multi-Parameter Macros

A function-like macro with several parameters takes the form

|  |
| --- |
| #define SYMBOLIC\_NAME(R1, R2, ... , Rn) Replacement text  including R1 R2 ... Rn |

We list the placeholder names within the parentheses and separate them by commas, and possibly by whitespace (for readability).  The placeholder names must be unique.  The ellipsis stands for the list of placeholders between R2 and Rn.  The replacement text holds each placeholder slot separately.

The pre-processor substitutes the arguments in the macro call(s) for the placeholder names in the replacement text.  For example, the pre-processor converts the following source code

|  |
| --- |
| // Function-Like Macros  // fnMacrosTriangle.cpp  #include <iostream>  #define TRIANGLE\_AREA(A, B) 0.5 \* B \* A  int main() {  double area = TRIANGLE\_AREA(2.5, 4);  std::cout << "Area is " << area << std::endl;  } |

into

|  |
| --- |
| // Function-Like Macros  // fnMacrosTriangle.i  // iostream declaractions here  int main() {  double area = 0.5 \* 4 \* 2.5;  std::cout << "Area is " << area << std::endl;  } |

Note that the order of the arguments in the macro call matters.

Efficiency and Flexibility

Function-like macros provide efficient and flexible solutions.  They avoid the overhead of function calls and do not impose type constraints on placeholder names.

Consider the repeated calculation of a product of two numbers using a macro

|  |  |
| --- | --- |
| // Macro Efficiency  // effMacro.cpp  #include <iostream>  #include <ctime>  #define PRODUCT(X, Y) (X)\*(Y)  #define NITER 1000000000  int main () {  double x;  int i;  std::clock\_t c0, c1;  x = 1;  c0 = std::clock();  for (i = 0; i < NITER; i++)  x = PRODUCT(x, 1.0000000001);  c1 = std::clock();  std::cout << "Process time is " <<  (double)(c1-c0)/CLOCKS\_PER\_SEC  << " secs" << std::endl;  std::cout << "Value of x is " << x <<  std::endl;  } | Process time is 3.095 secs  Value of x is 1.10517 |

The equivalent type-safe C++ function solution to this same problem is

|  |  |
| --- | --- |
| // Macro Efficiency  // effFunction.cpp  #include <iostream>  #include <ctime>  double product(double x, double y) {  return x \* y;  }  #define NITER 1000000000  int main () {  double x;  int i;  std::clock\_t c0, c1;  x = 1;  c0 = std::clock();  for (i = 0; i < NITER; i++)  x = product(x, 1.0000000001);  c1 = clock();  std::cout << "Process time is " <<  (double)(c1-c0)/CLOCKS\_PER\_SEC  << " secs" << std::endl;  std::cout << "Value of x is " << x <<  std::endl;  } | Process time is 6.673 secs  Value of x is 1.10517 |

On a Windows platform the function alternative takes twice the time to execute as the function-style macro.  Both executables are about the same size (156Kb).

Note that macros do not provide the type-checking facilities of a language like C++ and that there are other ways to achieve efficiency than through function-like macros.

Precedence Errors

Macros are a potential source of unexpected errors.  A common error is to treat the arguments in a macro call as expressions rather than as simple text.  Consider the following definitions

|  |
| --- |
| // Macros A Potential Error  // macrosPotError.cpp  #include <iostream>  #define PI 3.14  #define SQUARE(x) x \* x /\* OK \*/  int main() {  std::cout << "PI(2.35)^2 is " << PI \* SQUARE(2.35) << std::endl;  } |

which the pre-processor converts into

|  |
| --- |
| // Macros A Potential Error  // macrosPotError.i  // iostream declarations here  int main() {  std::cout << "PI(2.35)^2 is " << 3.14 \* 2.35 \* 2.35 << std::endl; /\* OK \*/  } |

If we introduce an expression as an argument in the macro call, the pre-processor expands the macro

|  |
| --- |
| // Macros Precedence Error  // macrosPreError.cpp  #include <iostream>  #define PI 3.14  #define SQUARE(X) X \* X /\* ??? \*/  int main() {  std::cout << "PI(2.35 + 1)^2 is " << PI \* SQUARE(2.35 + 1) << std::endl;  } |

into

|  |
| --- |
| // Macros Precedence Error  // macrosPreError.i  // iostream declarations here  int main() {  std::cout << "PI(2.35 + 1)^2 is " << 3.14 \* 2.35 + 1 \* 2.35 + 1 << std::endl;  } |

That is, the second argument in the std::cout << statement reduces to

|  |
| --- |
| std::cout << "PI(2.35 + 1)^2 is " << 3.14 \* 2.35 + 1 \* 2.35 + 1 << std::endl;  std::cout << "PI(2.35 + 1)^2 is " << 3.14 \* 2.35 + 3.35 + 1 << std::endl;  std::cout << "PI(2.35 + 1)^2 is " << 3.14 \* 2.35 + 4.35 << std::endl;  std::cout << "PI(2.35 + 1)^2 is " << 7.379 + 4.35 << std::endl;  std::cout << "PI(2.35 + 1)^2 is " << 11.729 << std::endl; |

But we expect 11.2225, which is 3.35 \* 3.35.

To avoid such precedence errors, we wrap each placeholder in the definition's replacement text within parentheses.  For example,

|  |
| --- |
| // Macros Avoiding Precedence Errors  // macrosPrecedence.cpp  #include <iostream>  #define PI 3.14  #define SQUARE(X) ((X) \* (X)) /\* NOTE THE PARENTHESES \*/  int main() {  std::cout << "PI(2.35 + 1)^2 is " << PI \* SQUARE(2.35 + 1) << std::endl;  } |

The pre-processor converts this source code into

|  |
| --- |
| // Macros Avoiding Precedence Errors  // macrosPrecedence.i  // iostream declaractions here  int main() {  std::cout << "PI(2.35 + 1)^2 is " << 3.14 \* ((2.35 + 1) \* (2.35 + 1))  << std::endl;  } |

which is what we expect.

Side-Effects

Macro definitions can generate side effects.  Because the pre-processor substitutes text without evaluating expressions, it generates repeated evaluations that might have been intended as single evaluations.  Consider the post-fix operator

|  |
| --- |
| // Macros Side-Effects  // macrosSideEffects.cpp  #include <iostream>  #define PI 3.14  #define SQUARE(X) ((X) \* (X))  int main() {  double r = 2.35;  std::cout << "PI(r++)^2 is " << PI \* SQUARE(r++) << std::endl;  std::cout << "r is now " << r << std::endl;  } |

The pre-processor converts this original source code into

|  |
| --- |
| // Macros Side-Effects  // macrosSideEffects.i  // iostream declaractions here  int main() {  double r = 2.35;  std::cout << "PI(r++)^2 is " << 3.14 \* ((r++) \* (r++)) << std::endl;  std::cout << "r is now " << r << std::endl;  } |

and displays a value of 4.35 for r.

To avoid side-effects, we need to review each macro call and replace any argument that is an expression with its evaluated result.

#undef

Naming conflicts amongst macro definitions may arise if a source file includes several header files.  The pre-processor rejects redefinitions of a symbolic name, unless those redefinitions are identical to the original definition.  For example, the pre-processor accepts the following code

|  |
| --- |
| #define PI 3.14  #define PI 3.14 // OK  #define AREA(R) PI \* R \* R  double r = 2.35;  double area = AREA(r); |

as well as the following redefinition

|  |
| --- |
| #define PI 3.14  #define PI /\* the math constant \*/ 3.14 // OK  #define AREA(R) PI \* R \* R  double r = 2.35;  double area = AREA(r); |

The amount of whitespace before the replacement text need not be identical, but the amount and the positioning of the whitespace within the replacement text must be identical.

On the other hand, the pre-processor rejects the following code

|  |
| --- |
| #define PI 3.14  #define PI 3.14159 /\* ERROR Replacement text differs from current text \*/  #define AREA(R) PI \* R \* R  double area = AREA(2.35); |

To avoid naming conflicts, we can limit the region of the source code over which the macro applies.  Before redefining a macro, we can cancel its current definition, using the directive

|  |
| --- |
| #undef SYMBOLIC\_NAME |

For example,

|  |
| --- |
| #define PI 3.14  #undef PI  #define PI 3.14159 // OK  #define AREA(R) PI \* R \* R  double area = AREA(2.35); |

If the original macro definition has no replacement text, the pre-processor ignores any request to undefine that definition.

|  |
| --- |
| #define PI  #undef PI  #define PI 3.14159 // OK  #define AREA(r) PI \* r \* r  double area = AREA(2.35); |

Pre-Defined Macros

The pre-processor predefines certain symbolic names itself, which we cannot alter.  These names include

* \_\_FILE\_\_ name of the current source file expressed as a string literal,
* \_\_DATE\_\_ calendar date of the pre-processing expressed as a string literal (Mmm dd yyyy),
* \_\_TIME\_\_ time of the pre-processing expressed as a string literal (hh:mm:ss),

Note the double underscore prefix and the double underscore suffix in each symbolic name.

For example, consider the following program

|  |
| --- |
| // Predefined Macros  // predefined.cpp  #include <iostream>  int main() {  std::cout << "The name of the source file is " << \_\_FILE\_\_ << std::endl;  std::cout << "The date of its pre-processing is " << \_\_DATE\_\_ << std::endl;  std::cout << "The time of its pre-processing is " << \_\_TIME\_\_ << std::endl;  } |

The executable version produces the following output

|  |
| --- |
| The name of the source file is predefined.cpp  The date of its pre-processing is May 16 2014  The time of its pre-processing is 14:17:48 |

Good Design Practice

The stricter syntactic rules of a typed programming language do not apply to macro definitions.  For type-safety we always prefer language constructs over macro definitions.  We use macro definitions for interpreting platform-specific features.

Include Directive

The #include directive inserts a file into the original source code.  The pre-processor inserts the contents of the named file in place of the directive regardless of the contents.  The directive takes the form

|  |
| --- |
| #include FILE\_TO\_BE\_INSERTED |

FILE\_TO\_BE\_INSERTED is one of

* a filename enclosed within < and > characters
* a filename enclosed within " and " characters
* a symbolic name

If the enclosing characters are < and >, the pre-processor searches the system directory first.  If the enclosing characters are " and ", the pre-processor searches the current directory first.  For example,

|  |
| --- |
| #include <iostream> // searches for iostream in the system directory  #include "console.h" // searches for console.h in the current directory  #define FILENAME "bye.h" // defines the symbolic name FILENAME  #include FILENAME // searches for bye.h in the current directory |

Some, but not all, pre-processors, on failing to find a named file in the preferred directory, search the alternate directory.

Good Design Practice

We should only use the #include directive to insert header files.

Conditional Directives

Conditional directives insert source code based on satisfaction of a condition.  Typical uses for conditional directives include:

* bypassing class or struct definitions
* inserting optional debugging code
* inserting platform-specific code

There are two types of conditional directives:

* logical
* definitional

Logical Directives

Logical directives insert source code based on satisfaction of a pre-processor condition.  They take the form

|  |
| --- |
| #if CONDITION  group of statements  #elif CONDITION  group of statements  #else  group of statements  #endif |

CONDITION is a relational or logical expression (either simple or compound) that evaluates to a constant arithmetic value: non-zero or 0.  The expression is subject to macro substitution prior to evaluation.

A logical directive may contain any number of #elif CONDITION directives.  The #elif CONDITION and #else branches are optional.  The #endif directive terminates the directive and must be present.

The pre-processor searches each branch of a logical construct for the first satisfied condition and includes all of the source code for the first branch that satisfies its condition.  The pre-processor excludes all source code within all other branches and does not complete any macro substitutions within those branches.

For example, the pre-processor reduces

|  |
| --- |
| #define CASE\_A 0  #define CASE\_B 1  #define CASE\_C 2  #define CASE CASE\_C  #if CASE == CASE\_A  #include "case\_a.h"  #elif CASE == CASE\_B  #include "case\_b.h"  #elif CASE == CASE\_C  #include "case\_c.h"  #endif |

to

|  |
| --- |
| #include "case\_c.h" |

In the result, the pre-processor only inserts the "case\_c.h" header file in constructing the translation unit for the compiler proper.

Definitional Directives

A definitional directive checks for the existence of a symbolic name and takes either of two forms:

* definitional construct
* functional directive

Definitional Construct

A definitional construct takes either of the following forms

|  |  |
| --- | --- |
| #ifdef SYMBOLIC\_NAME  group of statements  #endif | #ifndef SYMBOLIC\_NAME  group of statements  #endif |

#ifdef inserts the group of statements if SYMBOLIC\_NAME has been defined; #ifndef inserts the group of statements if SYMBOLIC\_NAME has not been defined.

For example, we enclose each class definition within a definitional construct to avoid redefinitions within the same translation unit

|  |
| --- |
| #ifndef SHAPE\_H  #define SHAPE\_H  // A Shape  // Shape.h  class Shape {  public:  virtual double volume() const;  };  #endif |

On the first request to include this header file, the pre-processor defines SHAPE\_H and includes the class definition.  On every subsequent request, the pre-processor finds that SHAPE\_H has been defined and skips the insertion.

We may use a definitional construct to encapsulate debugging code embedded within the production source code.  We define a macro (DEBUG) near the beginning of the source code and surround each snippet of debugging code with a definitional construct:

|  |
| --- |
| #define DEBUG  int main() {  // production instructions  }  #ifdef DEBUG  // debugging statements  #endif  // production instructions |

Near the beginning of the source file, we comment out the macro to exclude the debugging statements from the translation unit:

|  |
| --- |
| // #define DEBUG |

If we comment out this directive, DEBUG is not defined and the pre-processor skips all debugging statements throughout the source code.  If we uncomment the #define directive, DEBUG is defined and the pre-processor inserts the debugging statements in constructing the translation unit.

Functional Directive

A functional directive supports a logical construct:

|  |
| --- |
| #if defined(SYMBOLIC\_NAME)  group of statements  #endif |

A call to defined returns true if SYMBOLIC\_NAME has been defined, false otherwise.

We can build compound logical expressions from symbolic names using this functional syntax

|  |
| --- |
| #if defined(DEBUG) && defined(VERBOSE)  group of statements  #elif defined(DEBUG)  group of statements  #endif |