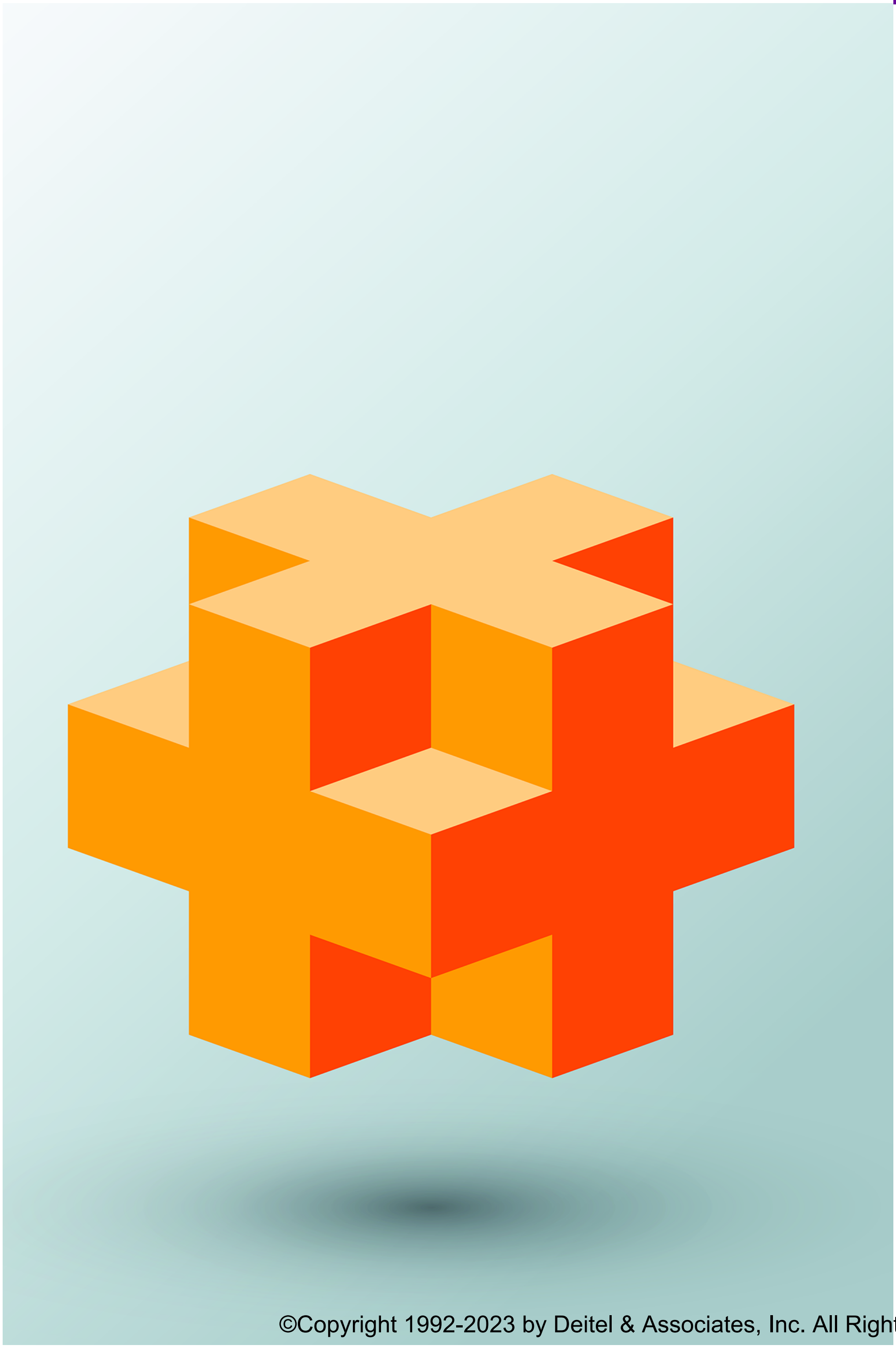
C
Preprocessor



Objectives

In this appendix, you'll:

- Understand `#include` in the context of developing large programs.
- Understand include guards for ensuring a header is included only once per translation unit.
- Use `#define` to create macros and macros with arguments.
- Understand conditional compilation.
- Display error messages during conditional compilation.
- Use assertions to test if the values of expressions are correct.

Outline

C.1 Introduction

This appendix discusses preprocessor directives in more depth. We provide it primarily for programmers handling legacy C++ code. C++20 modules (Chapter 16) and other Modern C++ techniques such as `constexpr` and templates are preferred to using the preprocessor directives shown here.

Some preprocessor actions are:

- including headers into C++ source-code files,
- defining **symbolic constants** and **macros**,
- **conditionally compiling** source code, and
- **conditionally executing preprocessing directives.**

All preprocessing directives begin with `#`, and only whitespace characters may appear before a preprocessing directive on a line. Preprocessing directives are not C++ statements, so they do not end in a semicolon (`;`). They are processed fully for a translation unit before it is compiled. Placing a semicolon at the end of a preprocessing directive can lead to various errors.

Err

C.2 #include Preprocessing Directive

The **`#include` preprocessing directive** has been used throughout this text. It includes the text contents of a specified file in place of the directive. With C++20 modules, you can now `import` many headers as header units. Some compilers also enable you to `import` a modular version of the entire standard library or specific portions of it. As we discussed in Chapter 16, this can significantly reduce compile times and translation unit sizes.

The two forms of the `#include` directive are

```
#include <filename>
#include "filename"
```

The difference is the location the preprocessor searches for the included file. For angle brackets (< and >), the preprocessor searches implementation-dependent predesignated folders and folders you add to the preprocessor's search path. For quotes, the preprocessor searches first in the same directory as the file being preprocessed, then in the same folders as files contained in angle brackets. Quotes typically are used to include programmer-defined header files.

C.3 #define Preprocessing Directive: Symbolic Constants

The **#define preprocessing directive** creates

- **symbolic constants**—constants represented as symbols—and
- macros (Section C.4)—function-like operations defined as symbols.

The C++ standard refers to both symbolic constants and macros as macros. The `#define` preprocessing directive format is

```
#define  identifier  replacement-text
```

The preprocessor replaces all subsequent occurrences of *identifier* (except those in string literals) in the file with *replacement-text* before the program is compiled. After encountering

```
#define PI 3.14159
```

the preprocessor replaces all subsequent occurrences `PI` with `3.14159`.

Everything to the right of the symbolic constant name replaces the symbolic constant. For example, if you were to accidentally include an `=`, as in

```
#define PI = 3.14159
```

the preprocessor to replace every occurrence of `PI` with `"= 3.14159"`. Replacements like this cause many subtle logic and syntax errors. Redefining a symbolic constant with a new value is also an error. Err

The C++ Core Guidelines say not to use the preprocessor for text manipulations like `PI` shown above. They indicate that "macros are a major source of bugs" and "don't obey the usual scope and type rules."[1,2] Instead, you should prefer `const` and `constexpr` variables, which have a specific data type and are visible by name to a debugger. Once a symbolic constant is replaced with its replacement text, only the replacement text is visible to a debugger. CG Err

C.4 #define Preprocessing Directive: Macros

This section is included for the benefit of C++ programmers who will need to work with C legacy code. Rather than macros, Modern C++ programs should use templates and functions.

You can define function-like macros in `#define` preprocessing directives. As with a symbolic constant, the preprocessor replaces a ***macro-identifier*** with its ***replacement-text*** before the translation unit is compiled. A macro without arguments is processed like a symbolic constant. In a macro with arguments, the preprocessor substitutes the arguments in the *replacement-text*, then expands the macro—that is, the *replacement-text* replaces the macro-identifier and argument list in the program. **There is no data type checking for macro arguments—a macro is used simply for text substitution.**

1. C++ Core Guidelines, "ES.30: Don't use macros for program text manipulation." Accessed April 4, 2023. `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-macros`.
2. C++ Core Guidelines, "Don't use macros for constants or 'functions'." Accessed April 4, 2023. `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-macros2`.

Macro for a Circle's Area

Consider the following macro definition with one argument for the area of a circle:

```
#define CIRCLE_AREA(x) (PI * (x) * (x))
```

In the macro call `CIRCLE_AREA(y)`, the preprocessor substitutes `y`'s value for `x` in the replacement text, inserts the symbolic constant `PI`'s value (defined previously) and expands the macro in the program. For example, the statement

```
area = CIRCLE_AREA(4);
```

expands `CIRCLE_AREA(4)`, as in

```
area = (3.14159 * (4) * (4));
```

Because the expression consists only of constants, the compiler can evaluate the expression and the result will be assigned to `area` at runtime. The parentheses around each `x` in the replacement text and the entire expression force the proper order of evaluation when the macro argument is an expression. For example, the statement

```
area = CIRCLE_AREA(c + 2);
```

expands `CIRCLE_AREA(c + 2)`, as in

```
area = (3.14159 * (c + 2) * (c + 2));
```

This evaluates correctly because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expansion becomes

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly as

```
area = (3.14159 * c) + (2 * c) + 2;
```

Err because of the rules of operator precedence. Forgetting to enclose macro arguments in parentheses in the replacement text is an error. Like symbolic constants, macros are error-
CG prone, and the C++ Core Guidelines say to avoid them.[3]

Function for a Circle's Area

Macro `CIRCLE_AREA` should be defined as a function, as in

```
constexpr double circleArea(double x) {return 3.14159 * x * x;}
```

If you require support for multiple data types, define `circleArea` as a function template
Perf instead. Though there's overhead associated with a function call, modern C++ compilers often can perform optimizations to eliminate that overhead, and `constexpr` functions, in particular, can be completely evaluated at compile-time if their arguments are compile-time constants.

Macro for a Rectangle's Area

The following is a macro definition with two arguments for the area of a rectangle:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

3. C++ Core Guidelines, "Don't use macros for constants or 'functions'." Accessed April 4, 2023. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-macros2.

Wherever `RECTANGLE_AREA( a, b )` appears in the program, the values of `a` and `b` are substituted in the macro replacement text, and the macro is expanded in place of the macro name. For example, the statement

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

is expanded to

```
rectArea = ((a + 4) * (b + 7));
```

The value of the expression is evaluated and assigned to variable `rectArea`.

Defining Multiline Macros

The replacement text for a macro or symbolic constant usually consists of any text to the right of the identifier and its argument list in the `#define` directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, you must place a backslash (\) at the end of each line of the macro except the last. These indicate that the replacement text continues on the next line.

Undefining Macros

Symbolic constants and macros can be discarded using the **`#undef` preprocessing directive**. Directive `#undef` "undefines" a symbolic constant or macro name. The scope of a symbolic constant or macro is from its definition until it is either undefined with `#undef` or the end of the file is reached. Once undefined, a name can be redefined with `#define`.

Avoid Expressions with Side Effects When Calling Macros

Note that expressions with side effects (e.g., variable values are modified) should not be passed to a macro because **macro arguments might be evaluated more than once**. Macros can accidentally replace identifiers that were not intended to be a use of the macro but just happened to be spelled the same. This can lead to exceptionally mysterious compilation and syntax errors. On the other hand, if you define the same identifier more than once in C++ code, you'll get a compilation error. Err

C.5 Conditional Compilation

Conditional compilation enables you to control the execution of preprocessing directives and the compilation of program code. Each of the conditional preprocessing directives evaluates a constant integer expression that will determine whether the code will be compiled. Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessing directives because these are all determined by the compiler and preprocessing happens before compilation.

The conditional preprocessor construct is much like the `if` selection structure. Consider the following preprocessor code:

```
#ifndef NULL
   #define NULL 0
#endif
```

which determines whether the symbolic constant `NULL` is already defined. The expression `#ifndef NULL` includes the code up to `#endif` if `NULL` is not defined and skips the code if `NULL` is defined. Every **`#if`** construct must end with **`#endif`**. The directives **`#ifdef`** and

`#ifndef` are shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`. A multiple-part conditional preprocessor construct may be tested using the `#elif` (the equivalent of `else if` in an `if` statement) and the `#else` (the equivalent of `else` in an `if` statement) directives.

"Commenting Out" Large Blocks of Code

During program development, programmers often find it helpful to "comment out" large portions of code to prevent it from being compiled. If the code contains `/*` and `*/` multi-line comments, `/*` and `*/` cannot be used because they cannot be nested. Instead, you can use the following preprocessor construct

```
#if 0
   code prevented from compiling
#endif:
```

To enable the code to be compiled, simply replace the value `0` in the preceding construct with the value `1`.

Conditional Compilation in Debugging

Conditional compilation is commonly used as a debugging aid. Output statements are often used to print variable values and confirm the flow of control. These output statements can be enclosed in conditional preprocessing directives so that the statements are compiled only until the debugging process is completed. For example,

```
#ifdef DEBUG
   cerr << "Variable x = " << x << "\n";
#endif
```

causes the `cerr` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined before directive `#ifdef DEBUG`. This symbolic constant is normally set by a command-line compiler or by settings in the IDE (e.g., Visual Studio) and not by an explicit `#define` definition. When debugging is completed, the `#define` directive is removed from the source file, and the output statements inserted for debugging purposes are ignored during compilation. In larger programs, it might be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.

Err ⊗ Inserting conditionally compiled output statements for debugging purposes in locations where C++ currently expects a single statement can lead to syntax errors and logic errors. In this case, the conditionally compiled statement should be enclosed in a compound statement. Thus, when the program is compiled with debugging statements, the flow of control of the program is not altered.

Include Guards

In our headers, we use

```
#pragma once
```

to ensure that their contents are included into a given translation unit only once. Though this directive is nonstandard, it's widely supported by popular C++ compilers, including all the compilers we use in this book.

The standard way to prevent multiple inclusion is with an **#include guard**, which consists of conditional compilation and `#define` directives, as in:

```
#ifndef HEADER_NAME
#define HEADER_NAME
   ...
#endif
```

where `HEADER_NAME` is a symbolic constant that, by convention, uses the header name in uppercase with the period replaced by an underscore.

The `#include` guard prevents the code between `#ifndef` and `#endif` from being `#include`d if `HEADER_NAME` has been defined. When a header containing an `#include` guard is `#include`d the first time, the identifier `HEADER_NAME` is not yet defined. In this case, the `#define` directive defines `HEADER_NAME`, and the preprocessor includes the header's contents in the translation unit. If the header is `#include`d again, `HEADER_NAME` already would be defined, so any code between `#ifndef` and `#endif` would be ignored.

Attempts to include a header multiple times (inadvertently) often occur in large programs with many headers that, in turn, include other headers. This could lead to compilation errors if the same definition appears more than once in a preprocessed file. Chapter 16 discussed how C++20 modules help prevent such problems. Err Mod

C.6 #error and #pragma Preprocessing Directives

The **#error directive**

```
#error tokens
```

prints an implementation-dependent message including the *tokens* specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```
#error 1 - Out of range error
```

contains six tokens. In one popular C++ compiler, for example, when an `#error` directive is processed, the tokens in the directive are displayed as an error message, preprocessing stops, and the program does not compile.

The **#pragma directive**

```
#pragma tokens
```

causes an implementation-defined action. A `pragma` not recognized by the implementation is ignored. A particular C++ compiler, for example, might recognize `pragma`s that enable you to take advantage of that compiler's specific capabilities.

C.7 Operators # and

The `#` and `##` preprocessor operators are available in C++ and ANSI/ISO C. The `#` operator causes a replacement-text token to be converted to a string surrounded by quotes. Consider the following macro definition:

```
#define HELLO(x) cout << "Hello, " #x << "\n";
```

When `HELLO(John)` appears in a program file, it is expanded to

```
cout << "Hello, " "John" << "\n";
```

The string "John" replaces #x in the replacement text. Strings separated by white space are concatenated during preprocessing, so the above statement is equivalent to

```
cout << "Hello, John" << "\n";
```

The # operator must be used in a macro with arguments because the operand of # refers to a macro argument.

The ## operator concatenates two tokens. Consider the following macro definition:

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

When TOKENCONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, TOKENCONCAT(O, K) is replaced by OK in the program. The ## operator must have two operands.

C.8 Predefined Symbolic Constants

The following table shows several **predefined symbolic constants**. The identifiers for all but __cplusplus begin and end with two underscores. These identifiers and preprocessor operator defined (Section C.5) cannot be used in #define or #undef directives. For the complete list, see https://en.cppreference.com/w/cpp/preprocessor/replace.

Symbolic constant	Description
__LINE__	The line number of the current source-code line (an integer constant).
__FILE__	The presumed name of the source file (a string).
__DATE__	The date the source file is compiled (a string of the form "Mmm dd yyyy" such as "Aug 19 2002").
__STDC__	Indicates whether the program conforms to the ANSI/ISO C standard. Contains value 1 if there is full conformance and is undefined otherwise.
__TIME__	The time the source file is compiled (a string literal of the form "hh:mm:ss").
__cplusplus	Contains the value 199711L (until C++11), 201103L (C++11), 201402L (C++14), 201703L (C++17) or 202002L (C++20)

C.9 Assertions

The **assert macro**—defined in the **<cassert>** header file—tests the value of an expression. If the value of the expression is 0 (false), then assert prints an error message and calls function **abort** (of the general utilities library—<cstdlib>) to terminate program execution. This is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable x should never be larger than 10 in a program. An assertion may be used to test the value of x and print an error message if the value of x is incorrect. The statement would be

```
assert(x <= 10);
```

If x is greater than 10, an error message containing the line number and file name is displayed, and the program terminates. You would then concentrate on this area of the code to find the error. If the symbolic constant NDEBUG is defined, subsequent assertions will be ignored. Thus, when assertions are no longer needed (i.e., when debugging is complete), we insert the line

```
#define NDEBUG
```

in the program file rather than deleting each assertion manually. As with the `DEBUG` symbolic constant, `NDEBUG` is often set by compiler command-line options or through a setting in the IDE.