Chapter 4: An Introduction to Classes

The most fundamental principles of OO design:

> **A class protects its members.**
> **A class takes care of itself ... and takes care of its own emergencies**
> **What you SHOULD do with classes is a small subset of what you CAN do.**

4.1 Class Basics

A `class` is used in C++ in the same way that a `struct` is used in C to form a compound data type. In C++, both may contain function members as well as data members; the functions are used to manipulate the data members and form the interface between the data structure and the rest of the program. There is only one differences between a `struct` and a `class` in C++:

- All `struct` members are public (unprotected) while `class` members default to private (fully protected). Privacy allows a class to encapsulate or "hide" members. This ability is the foundation of the power of object-oriented languages.

The rules for class construction are very flexible; function and data members can be declared with three different protection levels, and the protections can be breeched by using a `friend` declaration. Wise use of classes makes a project easier to build and easier to maintain. But wise use implies a highly disciplined style and strict adherence to basic design rules.

Public and private parts. The keywords `public`, `protected`, and `private` are used to declare the protection level of the class members. Both data and function members can be declared to have any level of protection. However, data members should almost always be private, and most class functions are public. We refer to the collection of all public members as the *class interface*. The `protected` level is only used with polymorphic classes.

The Form of a Class Declaration Figure 4.1 illustrates the general form of a class declaration, normally found in a `.hpp` file. Figure 4.1 shows a skeleton of the corresponding class implementation, normally kept in a `.cpp` file. The implementation (.cpp) supplies full definitions for class functions that were only prototyped within the class declaration (.hpp). The comments, below, are a collection of basic facts about class structure and usage.

4.1.1 Data members.

Like a `struct`, a class normally has two or more data members that represent parts of some real-world object. Also like a `struct`, a data member is used by writing an object name followed by a dot and the member name. Data members are normally declared to be private members because privacy protects them from being corrupted, accidentally, by other parts of the program. Taken together, the data members define the *state* of the object.

- Please declare data members at the top of the class. Although they may be declared anywhere within the class, your program is much easier for me to read if you declare data members before declaring the functions that use them.
- In managing objects, it is important to maintain a consistent and meaningful state at all times. To achieve this, All forms of assignment to a class object or to any of its parts should be controlled and validated by class functions. As much as possible, we want to avoid letting a function outside the class assign values to individual class members one at a time. For this reason, you must avoid defining "set" functions in your classes.

```
//-----------------------------------------------------------------
// Documentation for author, date, nature and purpose of class.
//-----------------------------------------------------------------
 #fndef MINE
 #define MINE
 #include "tools.hpp"
 class Mine {
   private:  // ---------------------------------------------------
       Put all data members here, following the format:
       TypeName  variableName;             // Comment on purpose of the data member

       Put private function prototypes here and definitions in the .cpp file.
       Or put the entire private inline function definitions here.

   public:   // ---------------------------------------------------
     Mine (param list){                    // constructor
        initialization actions
     }
     ~Mine() {  ...  }                     // destructor
     ostream& print ( ostream& s );        // print function, defined in .cpp file
     Put other interface function prototypes and definitions here.

   friend class  and friend function declarations, if any;
};
inline ostream& operator<<(ostream& st, Mine& m){ return m.print(st); }
#endif
```

Figure 4.1: The anatomy of a class declaration.

```
// Class name, file name and date.
#include "mine.hpp"
// --------------------------------------------------------------
ostream&                                   // Every class should implement print().
Mine::print ( ostream& s ) {
   Body of print function.
}

// --------------------------------------------------------------
// Documentation for interface function A.
returnType                                 // Put return type on a separate line.
Mine :: funA(param list) {                 // Remember to write the class name.
   Body of function A.
}

// --------------------------------------------------------------
// Documentation for interface function B.
returnType                                 // Documentation for return value.
Mine :: funB(param list) {
   Body of function B.
}
```

Figure 4.2: Implementation of the class “Mine”.

- Read only-access to a private data member can be provided by a “get” function that returns a copy of the member’s value. In the example class, we can provide read-only access to the data member named `name1`, by writing this public function member: `const type1 get_name1(){ return name1; }`

  In this function, the `const` is needed only when `type1` is a pointer, array, or address; for ordinary numbers it can be omitted.

- Some OO books illustrate a coding in which each private data member has a corresponding public function "set" function to allow any part of the program to assign a new value to the data member at any time. Do not imitate this style. Public "set" functions are rarely needed and should not normally be defined. Instead, the external function should pass a set of related data values into the class and permit class functions to validate them and store them in its own data members if they make sense.

4.1.2 Functions

Operators are functions. In C and Java, functions and operators are two different kinds of things. In C++, they are not different. All operators are functions, and can be written in either traditional infix operator notation or traditional function call notation, with a parenthesized argument list. Thus, we can add `a` to `b` in two ways:

```
c = a + b;
c = operator +(a, b);
```

Functions and methods In C, a function has a name and exactly one definition. In C++, a function has a name and one or more definitions. Each definition is called a *method* of the function. For example, the `close()` function has a method for input streams and a method for output streams. The two methods perform the same function on different types of objects.

Similarly, `operator+` is predefined on types `int` and `double`, and you may add methods to this function for numeric types you create yourself, such as `complex`. Part of the attraction of OO languages is that they support generic programming: the same function can be defined for many types, with slightly different meanings, and the appropriate meaning will be used each time the function is called.

At compile time[1], the C++ system must select which method to use to execute each function call. Selection is made by looking for a method that is appropriate for the type of the argument in each function call. This process is called *dispatching* the call.

Class functions, related functions, and non-class functions. Functions can be defined globally or as class members[2]. The main function is always defined globally. With few exceptions, the only other global functions should be those that interact with predefined C or C++ libraries or with the operating system. Common examples of global functions include those used in combination with `qsort()` and extensions to the C++ output operator[3]. The `operator >>` extensions form a special case. Each new method is directly associated with a class, but cannot be inside the class because `operator >>` is a global function.

Functions defined or prototyped within a class declaration are called *class functions*. A class function can freely read from or assign to the private class members, and the class functions are the *only* functions that *should* accesses class data members directly. If a class function is intended to be called from the outside world, it should be a public member. If the function's purpose is an internal task, it should be a private member. Taken together, the public functions of a class are called the *class interface*. They define the services provided by the class for a client program and the ways a client can access private members of the class.

Each function method has a short name and a full name. For example, suppose `Point` is a class, and the `plot()` function has a method in that class. The full name of the method is `Point::plot`; the short name is just `plot`. We use the short name whenever the context makes clear which class we are talking about. When there is no context, we use the full name.

Calling functions. A class function can be called either with an object or with a pointer to an object. Both kinds of calls are common and useful in C++. To call a class function with an object, write the name of the object followed by a dot followed by the function name and an appropriate list of arguments. To call a class function with a pointer to an object, write the name of the pointer followed by an `->` followed by the function name and an appropriate list of arguments. Using the `Point` class again, we could create Points and call the plot() function two ways:

[1]Polymorphic functions are dispatched at run time.
[2]Almost all C++ functions are class functions.
[3]These cases will be explained in a few weeks.

```
Point p1(1, 0);               // Allocate and initialize a point on the stack.
Point * pp = new Point(0, 1);  // Dynamically create an initialized point.
p1.plot( );
pp->plot();
```

In both cases, the declared type of `p1` or `p2` supplies context for the function call, so we know we are calling the `plot` function in the `Point` class.

Calls on non-class functions are exactly the same in C and in C++. Compare lines 17 (C) and 1011 (C++) in the code examples at the end of this chapter.

Typical Function Members Almost every class should have at least three public member functions: a constructor, a destructor, and a `print` function.

- A constructor function initializes a newly created class object. More than one constructor can be defined for a class as long as each has a different *signature*[4]. The name of the constructor is the same as the name of the class.
- A destructor function is used to the free storage occupied by a class object when that object dies. Any parts of the object that were dynamically allocated must be explicitly freed. If an object is created by a declaration, its destructor is automatically called when the object goes out of scope. In an object is created by `new`, the destructor must be invoked explicitly by calling `delete`. The name of the destructor is a tilde followed by the name of the class.
- A `print()` function defines the class object's image, that is, how should look on the screen or on paper. This function normally has a stream parameter so that the image can be sent to the screen, a file, etc. Call this function simply "print()", with no word added to say the type of the thing it is printing, and define a method in every class you write. Format your class object so that it will be readable and look good in lists of objects.

4.2 Inline Functions

Inline and out-of-line function translation. C++ permits the programmer to choose whether each function will be compiled out-of-line (the normal way) or inline. An out-of-line function is compiled once per application. No matter how many times the function is called, its code exists only once.

When a call on an out-of-line function is translated, the compiler generates code to build a stack frame, copy the function arguments into it, then jump to the the function. At load time, the linker connects the jump instruction to the actual memory location where the single copy of the function code exists.

At run time, when the function call is executed, a stack frame is built, the argument values are copied into it, and control is transferred from the caller to the function code. After the last line of the function, control and a return value are passed back to the caller and the function's stack frame is deallocated (popped off the system run-time stack). This process is done efficiently but still takes non-zero time, memory space for parameter storage, and space for the code that performs the calling sequence.

Inline functions are not macros. Inline expansion is like macro expansion. However, an inline function is more than and less than a macro, and is used differently:

- The number of arguments in both a function call and a macro call must match the number of parameters declared in the function or macro definition. The types of the arguments in the function call must match (or be convertible to) the parameter types in the definition. However, this is not true of macros, where parameter types are not declared and arguments are text strings, not typed objects.
- Macros are expanded during the very first stage of translation, and the expansion process is like the search-and-replace process in a text editor. Because of this, a macro can be defined and used as a shorthand notation for any kind of frequently-used text string, even one with unbalanced parentheses or quotes. Functions cannot be used this way. Functions are compiled at a later stage of translation.

[4] The signature of a function is a list of the types of its parameters.

When a call on an inline function is compiled, the entire body of the function is copied into the object code of the caller, like a macro, with the argument values replacing references to the parameters. If the same function is called many times, many copies of its code will be made. For a one-line function, the compiled code is probably shorter than the code required to implement an ordinary function call. Short functions should be inline because it always saves time and can even save space if there are only a few machine instructions in the function's definition. However, if the function is not short and it is called more than once, inline expansion will lead to a longer object file.

Usage. The main reason for using inline definitions is efficiency. Inline expansion of a very short function will save both time and space at run time.

The main reason for using out-of-line definitions is readability. It is helpful to be able to see an entire class declaration at once. If a class has some long functions, we put them in a separate file so that the rest of the class will fit on one page or one computer screen. For this reason, any function that is longer than two or three lines is generally given only a prototype within the class declaration. In contrast, when a function is only one or two lines long, it is easier to read when it is inside the class definition; there is no advantage in breaking it into two parts.

The inline keyword. A function defined fully within the class declaration is automatically inline. In addition, a function prototype can be declared to be `inline` even if it is defined remotely. To do this, simply write the qualifier `inline` before the return type in the function definition. This qualifier is advice to the compiler, not a requirement. A compiler may ignore the `inline` qualifier if it judges that a function is too long or too complex to be inline.

Summary of inline rules and concepts.

- An inline function is expanded like a macro, in place of the function call. No stack frame is built and there is not jump-to-subroutine.
- Functions defined in a class (between the class name and the closing bracket) default to inline.
- Non-class functions and class functions defined after the class may be compiled as inline functions if the keyword "inline" is written before the function return type of the prototype within the class.
- The compiler treats "inline" as advice, not as orders; if a particular compiler thinks it is inappropriate in a particular situation, the code will be compile out-of-line anyway.
- In order to compile, the full definition of an inline function must be available to the compiler when every call to the function is compiled.
- Make extensions of operator `<<` and operator `>>` inline or put them in the `.cpp` file of the related class.

4.2.1 Code Files and Header Files

Using .cpp and .hpp files The purpose of a header file is to provide information about the class interface for inclusion by other modules that use the class. The `.hpp` file, therefore contains the class declaration and related `#include`, `#define`, and `typdef` statements. Executable code, other than definitions of inline functions, does *not* belong in a `.hpp` file.

In-class and remote definitions. A class function can be fully defined within the class declaration or just prototyped there and defined later. Although there are no common words for these placement options, I call them *in-class* and *remote*. Being in-class or remote has no connection to privacy: both public and private functions can be defined both ways.

Remote inline functions cause an organization problem. In order for inline expansion to be possible, the actual definition of an inline function must be available to the compiler whenever the function is called. The only reasonable way to do this is to put the full definition of each inline function in the `.hpp` file, after the end of the class declaration, along with related inline non-class functions, such as extensions of the input and output operators. Then, wherever the class is visible to the compiler, the related inline definition will also be visible.

The definitions of non-inline remote class functions, if any, are written in a `.cpp` file. Each `.cpp` file will be compiled separately and later linked with other modules and with the library functions to form an executable program. This job is done by the *system linker*, usually called from your IDE. Non-inline functions that are related to a class, but not part of it, can also be placed in its .cpp file.

Friends. Friend classes will be explained soon; they are used to build data structures (such as linked lists) that require two or more mutually-dependant class declarations. Friend functions are legal but not really useful; I discourage their use. Friend declarations are written within the class declaration in the .hpp file, either at the very top or the very bottom of the class.

4.3 Declaration, Implementation, and Application of a Stack Class

In this section, we compare C and C++ implementations of a program to analyze a text file and determine whether its bracketing symbols are correctly nested and balanced. A stack class is used to check the nesting. Two application classes are also defined, Token (to represent one opening bracket) and Brackets (the boss class of the application). For each part of the program, the C version is presented first (with line numbers starting at 1), followed by the C++ version (with line numbers starting at 1000). The program notes compare the versions.

4.3.1 The Input and Output (banners have been deleted).

Each set of output on the right was produced by both versions of the Brackets program after processing the input on the left. This program ends when the first bracketing error is discovered.

Contents of input file:	Output produced:

`(<>){}[[]]`	`Welcome to the bracket checker!`
	`Checking file 'text2'`
	`The brackets in this file are properly nested and matched.`
	
	`Normal termination.`

`(<>){}[[`	`Welcome to the bracket checker!`
	`Checking file 'text4'`
	`Created stack brackets`
	
	`Mismatch at end of file:  Too many left brackets`
	`The stack brackets contains:  Bottom   [ [   Top`
	
	`Error exit; press '.'  and 'Enter' to continue`

`(< This is some text`
`>)`
` Some more text <<`
`>>)`
`{}`
	`Welcome to the bracket checker!`
	`Checking file 'text5'`
	`Mismatch on line 4:  Closing bracket has wrong type`
	`The stack 'bracket stack' contains:  Bottom   [   Top`
	
	`The current mismatching bracket is ')'`

Incorrect file name	`Welcome to the bracket checker!`
	`Checking file 'text'`
	`can't open file 'text' for reading`
	
	`Press '.'  and 'Enter' to continue`

No file name supplied.	`Welcome to the bracket checker!`
	`usage:  brackets file`
	
	`Press '.'  and 'Enter' to continue`

4.3.2 The main function: main.c and main.cpp

```
// ============================================================================
// Project: Bracket-matching example of stack usage               File: main.c
// Author:  Michael and Alice Fischer                   Copyright: January 2009
// ============================================================================
#include "tools.h"
#include "token.h"
#include "stack.h"

void analyze( stream in );
void mismatch( string msg, int lineno, Stack* stkp, Token* tokp );

//---------------------------------------------------------------------------
int main( int argc, char* argv[] )
{
    banner();
    say("Welcome to the bracket checker!");

    if (argc!=2) fatal( "usage: %s file", argv[0] );
    say( "Checking file '%s'", argv[1] );

    stream instream = fopen( argv[1], "r" );
    if (instream==NULL) fatal( "can't open file '%s' for reading", argv[1] );

    analyze( instream );
    fclose( instream );
    bye();
}

//---------------------------------------------------------------------------
void analyze( stream in )
{
    char ch;
    Token curtok;
    Token toptok;
    Stack stk;
    int lineno = 1;             // Number of lines of text in the  input file.

    init( &stk, "bracket stack" );

    for(;;){                    // Read and process each character in the file.
        ch = getc(in);
        if (feof( in )) break;
        if (ch == '\n') lineno ++;
        curtok = classify(ch);
        if (curtok.typ == BKT_NONE) continue;   // skip non-bracket characters
        switch (curtok.sense) {
            case SENSE_LEFT:
                push(&stk, curtok);
                break;

            case SENSE_RIGHT:
                if (empty(&stk))
                    mismatch("Too many right brackets", lineno, &stk, &curtok);
                toptok = peek(&stk);
                if (toptok.typ != curtok.typ)
                    mismatch("Closing bracket has wrong type", lineno, &stk, &curtok);
                pop(&stk);
                break;
        }
    }

    if (!empty( &stk )) mismatch("Too many left brackets", lineno, &stk, NULL);
    else printf("The brackets in this file are properly nested and matched.\n");
    recycle( &stk );
}
```

```

//---------------------------------------------------------------------------
void mismatch(string msg, int lineno, Stack* stkp, Token* tokp )
{
    if (tokp == NULL)  printf("Mismatch at end of file: %s\n", msg);
    else  printf("Mismatch on line %d: %s\n", lineno, msg);

    print( stkp ); printf( "\n\n" );        // print stack contents
    if (tokp != NULL)                       // print current token, if any
        printf("The current mismatching bracket is '%c'\n\n", tokp->ch );

    exit(1);                                // abort further processing
}
```

Notes on both versions of the main program:

- Every program must have a function named `main`. Unlike Java, `main()` is not inside a class. Like C, the proper prototype of main is one of these two lines. (Copy exactly, please. Do not use anything else.)

```
int main( void );
int main( int argc, char* argv[] );
```

- Although both versions were written on the same day by the same people, the C++ version is more modular: main.c contains the functionality of both main.cpp and brackets.cpp. Thus, the include files, prototypes, and constant definitions for the two purposes are mushed together in main.c.
- The C++ version separates the actions of main (deal with the command-line arguments, initialize and start up the application) from the actions of the application itself (read a file and analyze the brackets within it). This is a cleaner design.
- The functions `banner`, `say`, `fatal()`, and `bye` are defined in the tools library. Note that some of these functions use C-style formats, and that the output produced that way mixes freely with C++-style output.
- Both versions call banner and print a greeting comment. This is the minimum that a program should do for its human user. (Lines 15–16 and 1010–1011)
- Both versions test for a legal number of command-line arguments and provide user feedback whether or not the number is correct. (Lines 18–19 and 1013–1014) Note the form of the usage error comment and imitate it when you are using command-line arguments.
- Both versions use the command-line argument to open and input file and check for success of the opening process. (Lines 21–22 and 1016–1017)

```
1000 //============================================================================
1001 // Project: Bracket-matching example of stack usage              File: main.cpp
1002 // Author:  Michael and Alice Fischer                   Copyright: January 2009
1003 // ===========================================================================
1004 #include "tools.hpp"
1005 #include "brackets.hpp"
1006
1007 //---------------------------------------------------------------------------
1008 int main(int argc, char* argv[])
1009 {
1010     banner();
1011     say("Welcome to the bracket checker!");
1012
1013     if (argc!=2) fatal("usage: %s file", argv[0]);
1014     say("Checking file '%s'", argv[1]);
1015
1016     ifstream in( argv[1] );
1017     if (! in ) fatal("can't open file '%s' for reading", argv[1]);
1018
1019     Brackets b;             // Declare and initialize the application class.
1020     b.analyze( in );        // Execute the primary application function.
1021     in.close();
1022     bye();
1023 }
```

- Line 24 calls the primary application function and supplies the open input stream as an argument. Line 1019 calls new to create and initialize an object of the Brackets class. Line 1020 calls the primary application function with the stream as an argument. The C++ brackets-object, `b`, was allocated by declaration instead of by calling `new`, so it will be deallocated automatically when control leaves the function on line 1023. We can do the same job by calling `new`, as in `Java`, like this:

```
Brackets* b = new Brackets();     // Create and initialize the application class.
b->analyze( in );
delete b;                         // If you create with new, you must later call delete.
```

- The last two lines close the input file and print a termination message. (Lines 25–26 and 1021–22)
- Note that, at this point, the C++ program is five lines shorter than the C program that does the same thing in the same way with the same code formatting.

4.3.3 The Brackets class.

The `C++` implementation has a class named Brackets that corresponds to the two functions at the end of `main.c`. The code in Brackets.cpp is like the two functions at the end of main.c, and the code in Brackets.hpp is largely extra. At this point, the `C++` version is 21 lines longer than the `C` version. So why make a Brackets class?

1. It is better design. The functionality of handling command line arguments and files is completely separated from the work of analyzing a text. Similarly, the central data parts of the class are separated from local temporary variables.
2. Each class gives you a constructor function, where all the initializations can be written, and a destructor function for writing calls on `delete`. You are unlikely to forget to initialize or free your storage.

```
1024  // ==========================================================================
1025  // Name: Bracket-matching example of stack usage             File: brackets.hpp
1026  // ==========================================================================
1027  #ifndef BRACKETS_H
1028  #define BRACKETS_H
1029
1030  #include "tools.hpp"
1031  #include "token.hpp"
1032  #include "stack.hpp"
1033
1034  class Brackets {
1035    private:
1036      Stack* stk;
1037      Token toptok;
1038      int lineno;
1039
1040    public:
1041      Brackets() {
1042          stk = new Stack( "brackets" );
1043          lineno = 1;
1044      }
1045      ~Brackets(){ delete stk; }
1046
1047      void analyze( istream& in);   // Check bracket nesting and matching in file.
1048      void mismatch( cstring msg, Token tok, bool eofile );      // Handle errors.
1049  };
1050  #endif
```

Notes on the Brackets header file. File headers have been shortened from here on, to conserve space on the page.

- The first two lines of each header file and the last line (1027, 1028, and 1050), are conditional compilation commands. Their purpose is to ensure that no header file gets included twice in any compilation step. Note the keywords and the symbol and copy it.

- Next come the `#include` commands (Lines 1030–32) for classes and libraries that will be needed by functions defined in this class. Put them here, not in the .cpp file. Note: the file `tools.hpp` includes all the necessary standard header files.
- The destructor (Line 1045) is responsible for freeing all dynamic memory allocated by the constructor and/or by other class functions.
- The constructor (Lines 1041–1044) allocates the necessary dynamic space and initializes the two relevant data members. The third data member, `toptok`, will be used later as a way for the —pg analyze() function to communicate with the `mismatch()` function.
- This constructor and destructor are both short, so they are defined inline. The other two functions are long and contain control structures, so they are declared here (Lines 1047–48) and defined in the .cpp file.

Notes on Brackets.cpp and the corresponding C functions.

- The .cpp file for a class should `#include` the corresponding header file and nothing else.
- Please note the line of dashes before each function. This is a huge visual aide. Do it. Good style also dictates that you add comments to explain the purpose of the function. I omit that to conserve space on the page, and because these notes are provided.
- Note that the return type of functions in the .cpp file are written, with the class name, on the line above the function name. As the term goes on, return types and class names will get more and more complex. Writing the function name at the left margin improves program readability.
- The definitions of `analyze` and `mismatch` belong to the Brackets class, but they are not *inside* the class (between the word class and the closing curly brace). Unlike Java, a C++ compiler does not look at your file names, and has no way to know that these functions belong to your Brackets class. Therefore, the full name of each function (i.e. Brackets::analyze) must be given in the .cpp file.
- The argument to the `analyze` function (Line 1057) is a reference to an open stream. Streams are always passed by reference in C++.
- The C program has a call on `init` (Line 38) that is not needed in C++ because class constructor functions are called automatically when an object is declared.
- An infinite `for` loop with an `if..break` is used here to process the input file because it is the simplest control form for this purpose. It is not valid to test for end of file until after the input statement, so you normally do not want to start an input loop by writing: `while (!in.eof()) ...`
- We use the `get` function (line 1062) to read the input. This is a generic function; what gets read is determined by the type of its argument. In this case, `ch` is a `char`, so the next keystroke in the file will be read and stored in `ch` (`get()` does not skip whitespace).
- We count the newlines (Lines 43, 1064) so that we can give intelligent error comments.
- Line 1065 declares a local temporary variable named `curtok` and calls the `Token` constructor with the input character to initialize it. This object is created on the stack inside the for loop. It will be deleted when control reaches the end of the loop on line 1082. Every time we go around this loop, a new Token is created, initialized, used, and discarded. This is efficient, convenient and provides maximal locality.
- We create Tokens so that we can store the input along with its two classifications: the side (left or right) and the type of bracket (paren, angle, brace, square). The task of figuring out the proper classification is delegated to the Token class because the Token class is the expert on everything having to do with Tokens.
- In the C version, we do not have the automatic initialization action of a class constructor, so we have to call the classify function explicitly. (Line 44)
- When we get to line 45 or 1066, the token has been classified and we can tell whether it is of interest (brackets) or not (most characters). If it is of no interest, we `continue` at the bottom of the loop (line 1082), deallocate the Token, and repeat.
- The switch (Lines 46–59 and 1068..1081) stacks opening brackets for later matching, and attempts to match closing brackets. The second case is complex because it must test for two error conditions.

- Line 48 pushes the new Token onto the stack `stk` (declared on line 35). Note that we write `stk` inside the parentheses here, and before the parentheses in the C++ version. We use call-by-address here because the function will modify the stack. Line 1070 pushes the new Token onto the stack named `stk` which is a member of the Brackets class. We could write this line as: `this->stk->push(curtok);`, which is longer and has exactly the same meaning.
- Lines 62–63 and 1083–86 handle normal termination and another error condition. To keep the `analyze` function as short as possible, most of the work of error handling is factored out into the `mismatch` function. It prints a comment, the current token (if any) and the stack contents.
- Because there is no destructor function, the C version must free dynamic storage explicitly (Line 64).

```
1051  // ============================================================================
1052  // Name: Bracket-matching example of stack usage           File: brackets.cpp
1053  // ============================================================================
1054  #include "brackets.hpp"
1055  //-----------------------------------------------------------------------------
1056  void Brackets::
1057  analyze( istream& in)
1058  {
1059      char ch;
1060
1061      for (;;) {                  // Read and process the file.
1062          in.get(ch);             // This does not skip leading whitespace.
1063          if ( in.eof() ) break;
1064          if (ch == '\n') lineno ++;
1065          Token curtok( ch );
1066          if (curtok.getType() == BKT_NONE) continue; // skip non-bracket characters
1067
1068          switch (curtok.getSense()) {
1069              case SENSE_LEFT:
1070                  stk->push(curtok);
1071                  break;
1072
1073              case SENSE_RIGHT:
1074                  if (stk->empty())
1075                      mismatch("Too many right brackets", curtok, false);
1076                  toptok = stk->peek();
1077                  if (toptok.getType() != curtok.getType())
1078                      mismatch("Closing bracket has wrong type", curtok, false);
1079                  stk->pop();
1080                  break;
1081          }
1082      }
1083      if ( stk->empty())
1084          cout <<"The brackets in this file are properly nested and matched.\n";
1085      else
1086          mismatch("Too many left brackets", toptok, true);
1087  }
1088
1089  //-----------------------------------------------------------------------------
1090  void Brackets::
1091  mismatch( cstring msg, Token tok, bool eofile )
1092  {
1093      if (eofile) cout <<"\nMismatch at end of file: " <<msg <<endl;
1094      else        cout <<"\nMismatch on line " <<lineno <<" : " <<msg <<endl;
1095
1096      stk->print( cout );         // print stack contents
1097      if (!eofile)                // print current token, if any
1098          cout <<"The current mismatching bracket is " << tok;
1099
1100      fatal("\n");                // Call  exit.
1101  }
```

4.3.4 Class Declaration: token.h and token.hpp

```
79   // ==========================================================================
80   // Project: Bracket-matching example of stack usage               File: token.h
81   // ==========================================================================
82   #ifndef TOKEN_H
83   #define TOKEN_H
84
85   #include "tools.h"
86
87   typedef enum {BKT_SQ, BKT_RND, BKT_CURLY, BKT_ANGLE, BKT_NONE} Bracket_type;
88   typedef enum {SENSE_LEFT, SENSE_RIGHT} Token_sense;
89
90   typedef struct {
91       Bracket_type typ;
92       Token_sense sense;
93       char ch;
94   } Token;
95
96   Token classify( char ch );
97   #endif // TOKEN_H
```

```
1102 // ==========================================================================
1103 // Project: Bracket-matching example of stack usage             File: token.hpp
1104 // ==========================================================================
1105 #ifndef TOKEN_HPP
1106 #define TOKEN_HPP
1107
1108 #include "tools.hpp"
1109
1110 enum BracketType {BKT_SQ, BKT_RND, BKT_CURLY, BKT_ANGLE, BKT_NONE};
1111 enum TokenSense {SENSE_LEFT, SENSE_RIGHT};
1112
1113 class Token {
1114 private:
1115     BracketType type;
1116     TokenSense sense;
1117     char ch;
1118     void classify( char ch );
1119
1120 public:
1121     Token( char ch );
1122     Token(){}
1123     ~Token(){}
1124     ostream&    print( ostream& out) { return out << ch; }
1125     BracketType getType()           { return type; }
1126     TokenSense  getSense()          { return sense; }
1127 };
1128
1129 inline ostream& operator<<( ostream& out, Token t ) { return t.print( out ); }
1130 #endif // TOKEN_HPP
```

Notes on the header files for the token class.

- Both header files start and end with the conditional compilation directives that protect against multiple inclusion. The second time your code attempts to include the same header file, the symbol (Line 84 or 1106) will already be defined., and the `#ifndef` on line 83 or 1105 will fail. In that case, everything up to the matching `#endif` will be skipped (not included).
- The enum definitions for the two languages are the same except that we do not need typedef in C++ because enum types are first-class types.
- The structure members in C are exactly like the class data members in C++, but the class members are private and the structure members are public.

- The `classify` function is public in C and private in C++ because, in OO languages, it is possible to encapsulate class members that are not useful to client classes. (Definition of encapsulate: separate the interface from everything else in the class, and keep everything else private.)
- The C `classify` function returns a Token value but the C++ version does not. The C++ version does its work by storing information in the data members of the current object. When called by the constructor, `classify` will initialize part of the Token under construction.
- The C++ class has six functions that are not present in the C version. There are two constructors: the normal one (line 1121) and a do-nothing default constructor (line 1122) that allows us to create uninitialized Token variables. Such a variables become useful when they receive assignments.
- The destructor (line 1123) is a do-nothing function because this class does not ever allocate dynamic memory. It is good style, but not necessary, to write this line of code. If omitted, the compiler will supply it automatically.
- The other three functions (lines 1124–1126) that have no C analog are accessor functions, sometimes called "get functions". Although it is traditional to use the word "get" as the first part of the name, that is not necessary. The entire purpose of a get function is to provide read-only access to a private class data member.
- Associated with the Token class, but not part of it, is an extension of `operator<<`. All it does is call the class print function and return the `ostream&` result. By implementing this operator and the underlying print function, we are able to use Token as the base type for the generic implementation of Stack.

Notes on token.c and token.cpp.

- The C and C++ versions of the code file start out identically, with a single `#include` statement.
- Both contain definitions of the `classify` function. In addition, the C++ file contains the definition of a constructor.
- The name of the parameter in the constructor is the same as the name of the data member it will initialize. We distinguish between the two same-name objects by writing `this->` in front of the name of the class member. You could skip the `this->` if you gave the parameter a different name.
- A class takes care of itself. The constructor's responsibility is to initialize all part of the new object consistently. Thus, it *must* call the classify function. It would be improper to expect some other part of the program to initialize a Token object.
- The job of the `classify` function is to sort out whether a token is a bracket, and if so, what kind.
- The first thing inside the `classify` function is the definition of the kinds of brackets this program is looking for. These definitions are const to prevent assignment to the variable and static so that they will be allocated and initialized only once, at load time, not every time the function is called. (This is more efficient.)
- Lines 107 and 108 of the C version are not necessary in C++ because of the way constructors work, and because C++ does not need a return value to convey its result.
- Lines 110 and 1145 search the constant brackets string for a character matching the input character. The result is NULL (for failure) or a pointer to the place the input was found in the string. If found, the subscript of the match can be calculated by subtracting the address of the beginning of the const array (Lines 116, 1151). This is far easier than using a switch to process the input character. Learn how to use it.
- If the input is a bracket, the next step is to decide whether it is left- or right-handed. The left-handed brackets are all at even-subscript positions in the string; right-handed are at odd positions. So computing the position mod 2 tells us whether it is left or right. Lines 117 and 1152 use a conditional operator to store the answer.
- In C, enum constants are names for integers. Our constant string and the enumeration symbols for bracket type were carefully written in the same order: two chars in the string for each symbol in the enum. Therefore, the token-type of the new token can be computed by dividing the string position by 2. (Line 118).

- In C++, enumeration symbols are *not* the same as integers. We can use the same calculation as we do in C, but then we must cast the result to the enum type. Being a more modern language, C++ is much more careful about type identity.

```
// ===========================================================================
// Project: Bracket-matching example of stack usage                File: token.c
// ===========================================================================
#include "token.h"

Token classify( char ch )
{
    static const string brackets = "[](){}<>";
    Token tok;
    tok.ch = ch;

    char* p = strchr( brackets, ch );
    if (p==NULL) {
        tok.typ = BKT_NONE;
        tok.sense = SENSE_LEFT;          // arbitrary value
    }
    else {
        int pos = p-brackets;            // pointer difference gives subscript.
        tok.sense = (pos%2 == 0) ? SENSE_LEFT : SENSE_RIGHT;
        tok.typ = (pos/2);               // integer arithmetic, with truncation.
    }
    return tok;
}
```

```
// ==========================================================================
// Name: Bracket-matching example of stack usage            File: token.cpp
// ==========================================================================
#include "token.hpp"
//---------------------------------------------------------------------------
Token::Token( char ch ){
    this->ch = ch;
    classify( ch );
}

//---------------------------------------------------------------------------
void Token::
classify( char ch )
{
    static const cstring brackets = "[](){}<>";
    char* p = strchr( brackets, ch );
    if (p==NULL) {
        type = BKT_NONE;
        sense = SENSE_LEFT;             // arbitrary value
    }
    else {
        int pos = p-brackets;           // pointer difference gives subscript.
        sense = (pos % 2 == 0) ? SENSE_LEFT : SENSE_RIGHT;
        type = (BracketType)(pos/2);    // integer arithmetic, with truncation.
    }
}

```

4.3.5 The Stack Class

To the greatest extent possible, this class was created as a general-purpose, reusable stack class, allowing any type of objects can be stacked. This is done by using an abstract name, `T`, for the base type of the stack. Then a typedef is used (Lines 131 and 1167) to map `T` onto a real type such as `char` or (in this case) `Token`.

The C implementation falls sort of the goal (fully generic programming) in one way: C does not provide any way to do generic output. For that reason, one line of the `print` function in stack.c will need to be changed if the typedef at the top of stack.h is changed. This is corrected in the C++ version by using `operator<<` and defining it for the class named in the typedef.

This stack class “grows”, when needed, to accommodate any number of data items pushed onto it.

```
122 // ==========================================================================
123 // Project: Bracket-matching example of stack usage                File: stack.h
124 // ==========================================================================
125 #ifndef STACK_H
126 #define STACK_H
127
128 #include "tools.h"
129 #include "token.h"
130
131 #define INIT_DEPTH 10            // initial stack size
132 typedef Token T;
133 /*-------------------------------------------- Type definition for stack of T */
134 typedef struct {
135     int max;        /* Number of slots in stack.  */
136     int top;        /* Stack cursor.              */
137     T* s;           /* Beginning of stack.        */
138     string name;    /* Internal label, used to make output clearer. */
139 } Stack;
140
141 /*-------------------------------------------------------------- Prototypes */
142 void init   ( Stack* St, string label );
143 void recycle( Stack* St );
144 void print  ( Stack* St );
145 void push   ( Stack* St, T c );
146 T    pop    ( Stack* St );
147 T    peek   ( Stack* St );
148 bool empty  ( Stack* St );
149 int  size   ( Stack* St );
150 #endif // STACK_H
```

Class declarations: stack.h and stack.hpp

- The initial size of the stack defined on lines 131 and 1167. The actual initial size matters little, since the stack size will double each time it becomes full.
- Lines 132 and 1168 define the base type of the stack as Token.
- Like any growable data structure, this stack needs three data members: the current allocation length, the current fill-level, and a pointer to a dynamically allocated array to store the data. In addition, we have a name, largely for debugging purposes. In the C++ version, all these things are private.
- The `init` and `recycle` functions in the C version take the place of the constructor and destructor in C++. All other functions are the same, except that the first parameter, the stack itself, does not need to be listed in the C++ functions.
- All of the `C` code is in stack.c but much of the `C++` code is written as inline functions in stack.hpp. Thus, the `C++` version will be more efficient.

Class implementation: stack.c, stack.hpp inline, and stack.cpp

- Compare the constructor (Lines 1180–85) to the `init` function (Lines 156–161). The C++ version has less clutter because we do not need to write `St->` every time we do anything.
- Compare the destructor (Line 1186) to the recycle function (Line 164). Both free the dynamic memory and print a trace comment[5].

[5] We will thoroughly discuss the `delete []` statement later.

```
1158 // ==========================================================================
1159 // Name: Bracket-matching example of stack usage               File: stack.hpp
1160 // ==========================================================================
1161 #ifndef STACK_HPP
1162 #define STACK_HPP
1163
1164 #include "tools.hpp"
1165 #include "token.hpp"
1166
1167 #define INIT_DEPTH 16   // initial stack size
1168 typedef Token T;
1169
1170 //-------------------------------- Type definition for stack of base type T
1171 class Stack {
1172 private:
1173     int max;        // Number of slots in stack.
1174     int top;        // Stack cursor.
1175     T* s;           // Beginning of stack.
1176     string name;    // Internal label, used to make output clearer.
1177
1178 public:
1179     //------------------------------------------------------- Constructors
1180     Stack( cstring name ){
1181         s = new T[ max=INIT_DEPTH ];
1182         top = 0;
1183         this->name = name;
1184     }
1185
1186     ~Stack(){ delete[] s;  cout <<"Freeing stack " <<name <<endl; }
1187
1188     //------------------------------------------------------- Prototypes
1189     void print( ostream& out );
1190     void push   ( T c );
1191     //------------------------------------------------------- Inline functions
1192     T    pop    ( ){ return s[--top]; }     // Pop top item and return it.
1193     T    peek   ( ){ return s[top-1]; }     // Return top item without popping it.
1194     bool empty  ( ){ return top == 0; }
1195     int  size   ( ){ return top; }          // Number of items on the stack.
1196 };
1197 #endif
```

- Four other functions are inline in C++: `pop, peek, empty, and size`. They are all written on one line, with a minimum of fuss. (Lines 1192–95 and 190–93)
- Stack::top always stores the subscript of the first empty stack slot. Therefore, it must be decremented before the subscript operation when popping. Note the difference between the code for `pop` (line 1192 and 190), which changes the stack, and the code for `peek` (line 1193 and 191), which does not change the stack.
- The `size` function is an accessor. We chose `not` to use the word "get" as part of the name of the function.
- The use of type `bool`. In C, the result of a comparison is an `int`; in C++ it is a `bool` On lines 192 and 1194, we use the operator `==`. The result is a `bool` value which we return as the result of the function. Inexperienced programmers are likely to write clumsy code like one of these fragments:

```
Clumsy:       top == 0 ? true :  false;

Worse:        if (top == 0) return true;
              else return false;

Even worse:   top == 0 ?  1 : 0;  // 1 and 0 are type integer in C++, not bool !
```

These are clumsy because they involve unnecessary operations. We don't need a conditional operator or an `if` to turn `true` into `true` because the result of the comparison is exactly the same as the result of the conditional. Line 23 shows the mature way to write this kind of test.

```
1199 // ==========================================================================
1200 // Name: Bracket-matching example of stack usage              File: stack.cpp
1201 // ==========================================================================
1202 #include "stack.hpp"
1203 //------------------------------------------------------------------------
1204 void Stack::
1205 print( ostream& out )  {
1206     T* p=s;                         // Scanner & end pointer for data
1207     T* pend = s+top;
1208     out << "The stack " <<name << " contains: Bottom~~ ";
1209     for ( ; p < pend; ++p)  cout <<' ' << *p;
1210     out << " ~~Top" <<endl;
1211 }
1212
1213 //--------------------------------------------------------------------------
1214 void Stack::
1215 push( T c ) {
1216     if (top == max) {       // If stack is full, allocate more space.
1217         say( "-Doubling stack length-" );
1218         T* temp = s;                         // grab old array.
1219         s = new T[max*=2];                   // make bigger one,
1220         memcpy( s, temp, top*sizeof(T) );    // copy data to it.
1221         delete temp;                         // free old array.
1222     }
1223     s[top++] = c;           // Store data in array, prepare for next push.
1224 }
```

- The print functions are the same, line by line, except for the actual output commands, which are language-appropriate. Line 185 prints a part of the Token structure and must be changed if the base type of the stack is changed. In contrast, line 1222 uses the generic `operator <<` to print. By doing so, we are able to implement a truly generic stack. This works for any type so long as the `operator <<` is defined for that type (line 1129).
- The push functions implement a data structure that grows to accommodate as many Tokens as needed. This implements the basic OO commandment: *A class takes care of its own emergencies.*
- The basic strategy is:
  1. When the stack is created, the data array must be allocated to some non-zero size.
  2. Before storing another object in the stack, test whether there is room for it.
  3. If not, double the size of the array and copy the information from the original array into the new space.
  4. Now store the new item.
- It might seem that this strategy produces a very inefficient data structure. That would be true if we increased the array size by 1 each time. But we don't, we double it. By doubling, we are assured that the total time used to for all the copy operations is always less than the current length of the stack. In other words, the algorithms work in linear time.
- Let us look at the details of this code:
  1. Lines 1216 and 180 test whether the stack is full.
  2. Lines 1217 and 181 display trace comments to aid debugging.
  3. Line 1218 is needed because C++ does not support a `reallocate` function, so the reallocation and copying must be coded by hand. We need a temporary pointer to hang onto the original copy of the data array.
  4. Lines 1219 and 182 update the stack structure to double its allocation length.
  5. Lines 1219 and 183 allocate a double-length array. Line 183 also copies the data from the old array to the new array and frees the old array.

6. Line 1220 copies the data to the new array. The first argument to memcpy is the target array, the second is the source, the third is the number of bytes to copy. In this case, we want to copy all the data in the original array. These two lines are unnecessary in C because `realloc` does the job.
7. The final line (1223 and 183) pushes the new data into the array, which is now guaranteed to be long enough.

```
151 // ==========================================================================
152 // Project: Bracket-matching example of stack usage               File: stack.c
153 // ==========================================================================
154 #include "stack.h"
155 // -------------------------------------------------- the constructor
156 void init( Stack* St, string label )  {
157     St->s = malloc( INIT_DEPTH * sizeof(T) );
158     St->max = INIT_DEPTH;
159     St->top = 0;
160     St->name = label;
161 }
162
163 // --------------------------------------------------- the destructor
164 void recycle(Stack* St){ free(St->s);  say( "\tdeleting %s", St->name); }
165
166 // -------------------------------------------------------------------
167 // ***** the printf statement must be modified for base type != Token
168 void print( Stack* St )         // Print contents of stack, formatted.
169 {   T* p = St->s;               // Scanner & end pointer for data.
170     T* pend = p + St->top;
171
172     printf( "The stack '%s' contains: Bottom~~ ", St->name );
173     for ( ;  p < pend;  ++p)  printf( " %c", p->ch );
174     printf( " ~~Top" );
175 }
176
177 // ---------------------------------------------- the Stack functions
178 void push( Stack* St, T c )
179 {
180     if (St->top == St->max) {   // If stack is full, allocate more space.
181         say( "-Doubling stack length-" );
182         St->max*=2;
183         St->s = realloc( St->s, St->max * sizeof(char) );
184     }
185     St->s[St->top++] = c;       // Store data in array, prepare for next push.
186 }
187
188 // -------------------------------------------------------------------
189 T    pop( Stack* St )   { return St->s[-- St->top]; }
190 T    peek( Stack* St )  { return St->s[St->top-1]; }
191 bool empty( Stack* St ) { return ( St->top == 0 ); }
192 int  size( Stack* St )  { return St->top; }
```

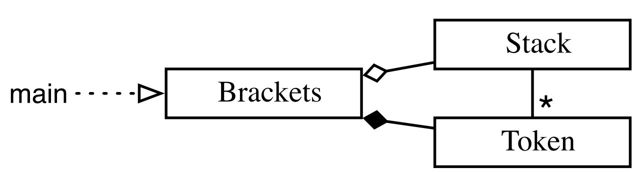

Figure 4.3: UML class diagram for the Brackets progam.