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 What is OO?

What Is a Class? A class is the description of a data structure and its associated behaviors. The behaviors include functionality, restrictions on visibility, and relationships with other classes. There are several kinds of classes:

- A *data class* models the state and behavior of a real-world object.
- A *controller class* uses inputs to control a process.
- A *viewer* class presents the state of a computation in a way that can be understood by the user.
- An *abstract class* characterizes some functionality without implementing it.
- A *business logic* class implements the policies of an organization.
- An *polymorphic class* is a class that has more than one variant, or subclass.

A program, or application, is constructed of a main function and many classes that, together, model some real-world process (a game, a calculation, data collection and processing, etc.).

What is an object? An object is an instance of a class, that is a structure that has all the data members and functionality that have been declared by the class definition. In C++ every instance of the same class has the same parts.

In some languages, a distinction is made between objects (instances of classes) and primitive values (integers, characters, etc.). This is not true in C++; a program can use a primitive value in the same contexts in which objects are used.

The behavior of a class is defined by its functions. In C++, a function can have more than one definition; each definition is called a *method*. In this text, we will use the term `function` to refer to a collection of methods with the same function name and the word `method` to refer to an individual method definition.

This chapter explains how state and behavior are represented within classes and introduces the concept of data encapsulation.

What are class relationships? A program is built from interacting classes. What kinds of interaction can be defined in C++?

- A class A can *compose* another class, B. This means that an object of type B is one of the parts of an A object.
- A class A can be *associated with* another object, B. This means that an A is connected by a pointer to a B object.
- A class B can be *derived from* class A. This means that B has all the parts of A plus more parts of its own. We say that A is the *base class* and B is a *derived class*, or *subclass* that *inherits* members from A. Typically, A will have more than one subclass.

  Instances of class B have *two* types, A and B, and can be used in any context where either type A or type B is accepted. The base class is called `polymorphic` because there are two or more different kinds of objects that belong to the type

- A class A can give *friendship* to B. This means that A permits B to manipulate its private parts. Friend classes will be explained fully in a future chapter. At this time, it is enough to know that they are used to build data structures (such as linked lists) that require two or more mutually-dependant class declarations.
- A object of class A can be *cast* to an object of class B.

During this course, all of these relationships will be introduced and used. A diagram that shows all of an application's classes and class relationships is called a *UML class diagram*. UML diagrams are an important way to understand the overall structure of a complex application.

What is a Template? A template is a partially-abstract class declaration. By itself, a template is not compilable code. Typically, templates are defined to represent data structures (often called collections) such as stacks and queues. They are abstract because the type of the data to be stored in the collection is not declared as part of the template. Instead, a type-variable name is used in the code.

To use a template, a program *instantiates* it by specifying the type of data that will be stored in the collection. The compiler then replaces the type-variable name by the instantiating type name to produce concrete, compilable code.

Templates are important at two levels. First, the templates supplied by the C++ standard template library implement most of the data structures and many of the standard algorithms that programmers need. Using STL templates can save any programmer (especially beginners) time and hassle. However, we are not limited to the templates provided by STL. This book covers the techniques needed to make new templates.

4.2 Code Files and Header Files

A basic C++ class definition starts with the word `class` and the name of the class. Following that are declarations for the members of the class, both data declarations and method prototypes, enclosed in braces. The class definition is stored in a header `.hpp` file. The actual code for the methods might or might not be inside the class definition. Short method definitions are there, but longer ones are in a separate implementation `.cpp` file. Most C++ classes have both a header `.hpp` file and an implementation `.cpp` file. Very simple classes are sometimes defined entirely in the header file.

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. Variable declarations and executable code, other than definitions of inline methods, do *not* belong in a `.hpp` file.

In-class and remote definitions. A class method 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 methods can be defined both ways.

The definitions of non-inline remote class methods, 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 methods to form an executable program. This job is done by the *system linker*, usually called from your IDE. Non-inline methods that are related to a class, but not part of it, can also be placed in its .cpp file.

4.3 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 method members as well as data members; the methods 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++:

- Members of a `struct` default to public (unprotected) visibility, 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.
- In both a `struct` and a `class`, the default visibility can be overridden by explicitly declaring `public`, `protected`, or `private`.

The rules for class construction are very flexible: method 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 method members can be declared to have any level of protection. However, data members should almost always be private, and most class methods 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 header (`.hpp`) file and the parts of a typical class declaration.

```
//-------------------------------------------------------------------
// 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.

4.3.1 Data members.

Like a `struct`, a class normally has two or more data members that represent parts of some real-world object. 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.

- 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 methods that use them.

- Taken together, the data members define the *state* of the object.
- Read only-access to a private data member can be provided by a "get" method that returns a copy of the member's value. For example, we can provide read-only access to the data member named `name1`, by writing this public method: `const char* getName1(){ return name1; }` In this method, the `const` is needed only when `type1` is a pointer, array, or address; for ordinary numbers it can be omitted.
- 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 member methods. 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" methods in your classes.
- Some OO books illustrate a coding in which each private data member has a corresponding public "set" method 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" methods 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 member methods to validate them and store them in its own data members if they make sense.

The Form of a Class Implementation Figure 4.2 shows the skeleton of the corresponding `.cpp` file. It supplies full definitions for member methods that were only prototyped within the class declaration (.hpp). Note that every implementation file starts with an `#include` for the corresponding header file.

```
// 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".

4.3.2 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.

Member methods, related methods, and non-member methods. Methods 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 *member functions*. A member method can freely read from or assign to the private class members, and the member methods are the *only* functions that *should* accesses class data members directly. If a member method is intended to be called from the outside world, it should be a public member. If the method's purpose is an internal task, it should be a private member. Taken together, the public methods 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 member 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 member 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 member 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:

```
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 `pp` supplies context for the function call, so we know we are calling the `plot` function in the `Point` class.

Static member functions. An object or pointer to an object is needed to call an ordinary member function. However, sometimes it is useful to be able to call a class function before the first object of that class is created. This can be done by making the method `static`. A static method can only use static data[4], that is, data that was allocated and initialized at program-load time. To call a static member function, we use the class name:

```
bool dataOK = Point::validate( xInput, yInput );
```

Global functions. Calls on non-member 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.

[1]Polymorphic functions are dispatched at run time.
[2]Almost all C++ methods are member functions.
[3]These cases will be explained in a few weeks.
[4]A static data member is called a *class variable* because there is only one copy of it per class. All instances of the class share that same static variable.

4.3.3 Typical Class Methods

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

- A constructor method initializes a newly created class object. More than one constructor can be defined for a class as long as each has a different *signature*[5]. The name of the constructor is the same as the name of the class.
- A destructor method 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 (∼`Point`).
- A `print()` method defines the class object's image, that is, how should look on the screen or on paper. Print functions normally have a stream parameter so that the image can be sent to the screen, a file, etc. Name this function simply "print()", with no word added to say the type of the thing it is printing, and define a method for it in every class you write. In your print() method, format your class object so that it will be readable and look good in lists of objects

4.4 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.

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 save time but it may lead to a longer object file.

[5]The signature of a method is a list of the types of its parameters.

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 reasons for *not* using an inline function are:

- A recursive function cannot be expanded inline.
- Code size: Long functions called repeatedly should not be inline.
- 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 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 and is fully defined in the `cpp` file.

The inline keyword. A function defined fully within the class declaration is automatically inline and the keyword `inline` is not used. In addition, you can declare a function prototype to be `inline`, and provide the rest of the function definition in the `.hpp` file after the brace that closes the class definition.

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.
- Methods defined fully within a class (between the class name and the closing bracket) default to inline.
- 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 and put them at the end of the `.hpp` file. Normally, these functions are implemented by calling the public `print()` method within the class.
- Methods for global functions can be inline.

4.5 Declaration, Implementation, and Application of a Stack Class

In this section, we present a C++ implementations of a program to analyze a text file and determine whether its bracketing symbols are correctly nested and balanced. Three classes are used:

- Stack, a container class, a tool used to check the nesting.
- Token, a data class, to represent one opening bracket, and a controller class.
- Brackets, a controller class, implements the logic of the application.

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

Each set of output on the right was produced by 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:

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`

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

4.5.2 The main function, from file main.cpp

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

//--------------------------------------------------------------------------
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]);

    ifstream in( argv[1] );
    if (! in ) fatal("can’t open file ’%s’ for reading", argv[1]);

    Brackets b;             // Declare and initialize the application class.
    b.analyze( in );        // Execute the primary application function.
    in.close();
    bye();
}
```

Notes on 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[] );
```

- The actions of main (deal with the command-line arguments, initialize and start up the application) are separated from the actions of the application itself (read a file and analyze the brackets within it). Separation of functionality is one of the basic design goals in OO programming.

- We include the header file for the tools library. 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++
- We call banner and print a greeting comment (lines 11–12). This is the minimum that a program should do for its human user.
- line 14 tests for a legal number of command-line arguments. It provides user feedback and aborts execution if the number is not correct. Note the form of the usage error comment and imitate it when you are using command-line arguments.
- We use the command-line argument to open and input file and check for success of the opening process (lines 17–18).
- The header file for the brackets module is included on line 6. This module is the main application controller. On line 20, we instantiate a Brackets object, and on line 21, we call its primary function with the open input 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 24.
- Alternately, we could 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 (22 and 23) close the input file and print a termination message. Neither is strictly necessary. Both are good practice.

4.5.3 The Brackets class.

The Brackets class contains the code that analyzes the input text to find mismatched brackets. Why are we using three modules (main, Brackets, Token) for this simple program?

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, this class is completely separated from the Token class that describes the kinds of brackets we are looking for. By separating the functionality into layers, we minimize overall complexity and allow the programmer or reader to focus on one part of the program at a time.
2. Each class gives you a constructor function, where all the initializations can be written, and a destructor function for writing calls on `delete`. This makes it easy to initialize or free your storage, and you are unlikely to forget those tasks.

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 (28, 29, and 51), 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 31–33) 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 constructor (lines 42–45) allocates the necessary dynamic space and initializes the two relevant data members. The third data member, `toptok`, is uninitialized; it will be used later as a way for the `analyze()` function to communicate with the `mismatch()` function.
- The destructor (line 46) is responsible for freeing all dynamic memory allocated by the constructor and/or by other class functions.

- 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 just declared here (lines 48–49) and defined in the .cpp file.

```
25  // ==========================================================================
26  // Name: Bracket-matching example of stack usage            File: brackets.hpp
27  // ==========================================================================
28  #ifndef BRACKETS_H
29  #define BRACKETS_H
30
31  #include "tools.hpp"
32  #include "token.hpp"
33  #include "stack.hpp"
34
35  class Brackets {
36    private:
37      Stack* stk;
38      Token toptok;
39      int lineno;
40
41    public:
42      Brackets() {
43          stk = new Stack( "brackets" );
44          lineno = 1;
45      }
46      ~Brackets(){ delete stk; }
47
48      void analyze( istream& in);   // Check bracket nesting and matching in file.
49      void mismatch( const char* msg, Token tok, bool eofile );      // Handle errors.
50  };
51  #endif
```

Notes on Brackets.cpp.

- 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 types of functions in this class 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 on a second line 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 (lines 48 and 58) is a reference to an open istream. Streams are always passed by reference in `C++`.
- An infinite `for` loop (line 62) with an `if..break` (line 63) 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 62) 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 (line 65) so that we can give intelligent error comments.
- line 66 declares a local temporary variable named `curtok` and calls the `Token` constructor with the input character to initialize it. This object is created inside the for loop. It will be deallocated automatically

when control reaches the end of the loop on line 82. 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.

```
 52  // ==========================================================================
 53  // Name: Bracket-matching example of stack usage             File: brackets.cpp
 54  // ==========================================================================
 55  #include "brackets.hpp"
 56  //---------------------------------------------------------------------------
 57  void Brackets::
 58  analyze( istream& in)
 59  {
 60      char ch;
 61
 62      for (;;) {                  // Read and process the file.
 63          in.get(ch);             // This does not skip leading whitespace.
 64          if ( in.eof() ) break;
 65          if (ch == '\n') lineno ++;
 66          Token curtok( ch );
 67          if (curtok.getType() == BKT_NONE) continue; // skip non-bracket characters
 68
 69          switch (curtok.getSense()) {
 70              case SENSE_LEFT:
 71                  stk->push(curtok);
 72                  break;
 73
 74              case SENSE_RIGHT:
 75                  if (stk->empty())
 76                      mismatch("Too many right brackets", curtok, false);
 77                  toptok = stk->peek();
 78                  if (toptok.getType() != curtok.getType())
 79                      mismatch("Closing bracket has wrong type", curtok, false);
 80                  stk->pop();
 81                  break;
 82          }
 83      }
 84      if ( stk->empty())
 85          cout <<"The brackets in this file are properly nested and matched.\n";
 86      else
 87          mismatch("Too many left brackets", toptok, true);
 88  }
 89
 90  //---------------------------------------------------------------------------
 91  void Brackets::
 92  mismatch( const char* msg, Token tok, bool eofile )
 93  {
 94      if (eofile) cout <<"\nMismatch at end of file: " <<msg <<endl;
 95      else        cout <<"\nMismatch on line " <<lineno <<" : " <<msg <<endl;
 96
 97      stk->print( cout );         // print stack contents
 98      if (!eofile)                // print current token, if any
 99          cout <<"The current mismatching bracket is " << tok;
100
101      fatal("\n");                // Call  exit.
102  }
```

- When we get to line 67, the token has been classified and we can test 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 82), deallocate the Token, and repeat.

- The switch (lines 69. . . 82) puts opening brackets onto a stack data structure, to be saved for later matching. With closing brackets, it attempts to match the input to the bracket on top of the stack. The second case is complex because it must test for two error conditions.
- Line 71 pushes the new Token onto the stack named `stk` declared as a data member on line 37. We could write this line as: `this->stk->push(curtok);`, which is longer and has exactly the same meaning.
- Lines 84. . . 87 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.

4.5.4 The Token Class

Class Declaration: token.hpp

```
103  // ==========================================================================
104  // Project: Bracket-matching example of stack usage             File: token.hpp
105  // ==========================================================================
106  #ifndef TOKEN_HPP
107  #define TOKEN_HPP
108
109  #include "tools.hpp"
110
111  enum BracketType {BKT_SQ, BKT_RND, BKT_CURLY, BKT_ANGLE, BKT_NONE};
112  enum TokenSense {SENSE_LEFT, SENSE_RIGHT};
113
114  class Token {
115  private:
116      BracketType type;
117      TokenSense sense;
118      char ch;
119      void classify( char ch );
120
121  public:
122      Token( char ch );
123      Token(){}
124      ~Token(){}
125      ostream&    print( ostream& out) { return out << ch; }
126      BracketType getType()            { return type; }
127      TokenSense  getSense()           { return sense; }
128  };
129
130  inline ostream& operator<<( ostream& out, Token t ) { return t.print( out ); }
131  #endif // TOKEN_HPP
```

Notes on the Token header file.

- A header files starts and ends 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 107) will already be defined, and the `#ifndef` on line 104 will fail. In that case, everything up to the matching `#endif` will be skipped (not included).
- We do not need typedef in C++ enum definitions.
- As always, the class data members are private.
- The `classify` function is private because it is not intended for use by client classes. In C++, it is possible to encapsulate class members that are only used locally.
- The `classify` does its work by storing information in the data members of the current Token object. When called by the constructor, `classify` will initialize part of the Token under construction.
- This class has two constructors: the normal one (line 122) and a do-nothing default constructor (line 123) that allows us to create uninitialized Token variables. Such a variables become useful when they receive assignments.

- The destructor (line 124) 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 destructor (line 125) defines how the class object will appear when displayed on the screen or written to a file. It returns an `ostream&`result so that it is easy to use in the definition of the output operator.
- The last two functions (lines 126–127) in the class 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 output Tokens with `<<`. This makes it possible to use Token as the base type for the implementation of Stack.

Class Implementation: token.cpp

```
// ============================================================================
// 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 char* 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.
    }
}

```

Notes on token.cpp.

- As usual, the file starts with a single `#include` statement for the module's own header file.
- All class functions except `classify` and `print` were defined inline. In this file, we to define only have only 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. This is a common practice. 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 (line 146) is the definition of the kinds of brackets this program is looking for. The definition is made `const` to prevent assignment to the variable and static so that the variable will be allocated and initialized only once, at load time, not every time the function is called. (This is more efficient for both time and space.)
- Line 147 searches the constant brackets string for a character matching the input character. The result is either a pointer to the place the input was found in the string, or it is NULL (indicating failure). This is far easier than using a switch to process the input character. Learn how to use it.
- If the search fails, that fact is recorded inside the Token object by setting the member variable named type to `BKT_NONE`.
- If found, the subscript of the match can be calculated by subtracting the address of the beginning of the const array from the match-pointer (line 153).
- 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. line 154 use a conditional operator to store the answer.
- Our constant string and the enumeration symbols for bracket -ype were carefully written in the same order: two chars in the string for each symbol in the enum. Dividing the string position by 2 therefore gives us the position of the enum constant in the enum definitions.
- In C++, enumeration symbols are *not* the same as integers. After calculating the position, as above, we must cast the result to the enum type. Being a modern language, C++ is careful about type identity.

4.5.5 The Stack Class

To the greatest extent possible, this class was created as a general-purpose, reusable stack class, allowing any type of objects to be stacked. This is done by using an abstract name, `T`, for the base type of the stack. Then a typedef is used (line 169) to map `T` onto a real type such as `char` or (in this case) `Token`.

C++ provides the `<<` operator which has methods for all of the built-in types. It also permits you to add methods for types you define yourself. In this module, we use `operator<<` to print the contents of the stack and depend on the programmer to define a method for `operator<<` for any type of data he wants to store in the stack.

This stack class "grows", when needed, to accommodate any number of data items pushed onto it. For all practical purposes, it has no size limitations.

Notes on the Stack class declarations: stack.hpp

- The initial size of the stack defined on line 169. The actual initial size matters little, since the stack size will double in size each time it becomes full.
- line 169 defines 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 that stores the data. In addition, we have a name, largely for debugging purposes. All these things are private.
- Member functions include the usual constructor, destructor, and print functions. The constructor initializes the object as an empty stack of the default capacity.
- The constructor allocated a dynamic array, so the destructor must deallocate it. In C++, the keyword `delete` is applied to pointers, and frees the dynamic memory attached to the pointer. This destructor also prints a trace comment that can be very useful during debugging.
- If that memory is an array, using `delete[]` also frees any second-level memory that is part of objects attached to the array. The run-time system calls the destructor that is defined for the base type of array, and it does this for every element in the array.[6].

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

```
159  // ============================================================================
160  // Name: Bracket-matching example of stack usage                 File: stack.hpp
161  // ============================================================================
162  #ifndef STACK_HPP
163  #define STACK_HPP
164
165  #include "tools.hpp"
166  #include "token.hpp"
167
168  #define INIT_DEPTH 16   // initial stack size
169  typedef Token T;
170
171  //-------------------------------- Type definition for stack of base type T
172  class Stack {
173  private:
174      int max;        // Number of slots in stack.
175      int top;        // Stack cursor.
176      T* s;           // Beginning of stack.
177      string name;    // Internal label, used to make output clearer.
178
179  public:
180      //------------------------------------------------------- Constructors
181      Stack( const char* name ){
182          s = new T[ max=INIT_DEPTH ];
183          top = 0;
184          this->name = name;
185      }
186
187      ~Stack(){ delete[] s;  cout <<"Freeing stack " <<name <<endl; }
188
189      //------------------------------------------------------- Prototypes
190      void print( ostream& out );
191      void push   ( T c );
192      //------------------------------------------------------- Inline functions
193      T    pop    ( ){ return s[--top]; }     // Pop top item and return it.
194      T    peek   ( ){ return s[top-1]; }     // Return top item without popping it.
195      bool empty  ( ){ return top == 0; }
196      int  size   ( ){ return top; }          // Number of items on the stack.
197  };
198  #endif
```

- This class also has definitions for the usual stack functions: `push`, `pop`, `peek`, `empty`, and `size`. Four of these are defined inline because they are so very short (lines 193–196).
- 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 193), which changes the stack, and the code for `peek` (line 194), which does not change the stack. `Pop` decrements the stack pointer, but `peek` does not.
- The `size` function is an accessor. We chose `not` to use the word "get" as part of the name of the function.
- On line 195, we test whether the top of the stack is at subscript 0. If so, the stack is empty.

  The result of an `==` comparison is a `bool` value. We simply return it as the result of the `empty()` function. Inexperienced programmers are likely to write clumsy code like one of these fragments:

  ```
  Clumsy:        return 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 195 shows the mature way to write this kind of test.

Class implementation: stack.cpp. The class `.cpp` file contains definitions of functions that are too long to be inline. Note that we do not need to write “`this.`’ every time we use a class member.

- The print function is logically simple. It prints a header, uses a loop to print the stack contents, and prints a brief footer. Note that a print function should *never* modify the contents of the object it is printing.

```
199  // ==========================================================================
200  // Name: Bracket-matching example of stack usage              File: stack.cpp
201  // ==========================================================================
202  #include "stack.hpp"
203  //--------------------------------------------------------------------
204  void Stack::
205  print( ostream& out )  {
206      T* p=s;                         // Scanner & end pointer for data
207      T* pend = s+top;
208      out << "The stack " <<name << " contains: Bottom~~ ";
209      for ( ; p < pend; ++p)  cout <<’ ’ << *p;
210      out << " ~~Top" <<endl;
211  }
212
213  //----------------------------------------------------------------------------
214  void Stack::
215  push( T c ) {
216      if (top == max) {       // If stack is full, allocate more space.
217          say( "-Doubling stack length-" );
218          T* temp = s;                            // grab old array.
219          s = new T[max*=2];                      // make bigger one,
220          memcpy( s, temp, top*sizeof(T) );       // copy data to it.
221          delete temp;                            // free old array.
222      }
223      s[top++] = c;           // Store data in array, prepare for next push.
224  }
```

- The `push` function is too long to be inline because it must ensure that there is enough memory in the array to store the next Token. It implements a data structure that grows to accommodate as many Tokens as needed. This implements a 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 algorithm works in linear time.
- Let us look at the details of this code:
  1. Line 216 tests whether the stack is full. If so, it must grow before doing the push operation.
  2. Line 217 prints a trace comments to aid debugging.
  3. Line 218 makes a copy of the pointer to the stack’s array. We need a temporary pointer to hang onto the original copy of the data array long enough allocate a new, longer array and move the data.
  4. Line 219 doubles the max size of the array and simultaneously allocates a new array that is twice as big as the original.
  5. Line 220 copies all of the bytes from the old array to the new array. [7]

[7] The method used here to move the data is an old non-OO `C` function that works at a lower level than the object model. The `C++-11` standard provides an OO way to move the data from one array to another. A full explanation will come later.

6. Line 221 frees the old array.
7. The final line (223) pushes the new data into the array, which is now guaranteed to be long enough.

- This implements a very useful data structure that I call a flex-array. It is the basis for the more complex `vector` type in C++'s template library.

A Collection of Utility Functions: tools.hpp

This file is included for reference.

```
// modified September 2011
// file: tools.hpp -----------------------------------------------------------
// header file for the tools library.
// ---------------------------------------------------------------------------
// Local definitions and portability code.
// Please enter your own system, name, class, and printer stream name.
// ---------------------------------------------------------------------------
#pragma once
#define NAME    "Alice Fischer"
#define CLASS   "CS 620"

#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <vector>
#include <string>
#include <algorithm>
#include <limits>

#include <cstdio>      // for NULL
#include <cstdlib>     // for malloc() and calloc()
#include <cstring>
#include <cmath>
#include <ctime>
#include <cctype>      // for isspace() and isdigit()
#include <cstdarg>     // for functions with variable # of arguments

using namespace std;

// ---------------------------------------------------------------------------
// Macros for debugging.
// ---------------------------------------------------------------------------
#define DUMPp(p) "\n" <<"    " #p " @ " <<&p <<"    value = " <<p <<"    " #p " --> " <<*p
#define DUMPv(k) "\n" <<"    " #k " @ " <<&k <<"    value = " <<k

// ---------------------------------------------------------------------------
// I/O Extension. ------------------------------------------------------------
// ---------------------------------------------------------------------------
istream& cleanline( istream& is );     // discards remainder of line
istream& flush( istream& is );         // Use: cin >> x >> flush;
ostream& general( ostream& os );       // Use: cout <<fixed <<x <<general <<y;

// ---------------------------------------------------------------------------
// Routine screen and process management.-------------------------------------
// ---------------------------------------------------------------------------
void     fbanner( ostream& fout );
void     banner();
void     bye( void );
void     hold( void );
void     say( const char* format, ... );

// ---------------------------------------------------------------------------
// Error handling. ------------------------------------------------------------
// ---------------------------------------------------------------------------
void    fatal( const char* format, ... );

// ---------------------------------------------------------------------------
// time and date. -------------------------------------------------------------
// ---------------------------------------------------------------------------
void   when( char* date, char* hour );
char*  today( char* date );
char*  oclock( char* hour );
```

4.6 A Stack Template

The typedef trick for achieving abstraction (lines 169, 176, 193, 194) was invented for C and can also be used in C++. However, C++ has a more convenient and more powerful tool for achieving abstraction: the class template. The creation of class templates is covered in Chapter 13. Much simpler than template creation is template use. Following is the Stack class from the Brackets program, rewritten as a template, and the header file for the Brackets class that uses the template.

Class Declaration: token.hpp

```
// ============================================================================
// Name: Bracket-matching example of stack usage               File: stack.hpp
// ============================================================================
#ifndef STACK_HPP
#define STACK_HPP

#include "tools.hpp"
#include "token.hpp"

#define INIT_DEPTH 16   // initial stack size

//--------------------------------- Type definition for stack of base type T
template <class T> class Stack {
private:
    int max;            // Number of slots in stack.
    int top;            // Stack cursor.
    T* s;               // Beginning of stack.
    const char* name;   // Internal label, used to make output clearer.

public:
    //------------------------------------------------------- Constructors
    Stack( const char* name ){
        s = new T[ max=INIT_DEPTH ];
        top = 0;
        this->name = name;
    }

    ~Stack(){ delete[] s;  cout <<"Freeing stack " <<name <<endl; }

    //------------------------------------------------------- Prototypes
    void print( ostream& out );
    void push   ( T c );
    //------------------------------------------------------- Inline functions
    T    pop    ( ){ return s[--top]; }     // Pop top item and return it.
    T    peek   ( ){ return s[top-1]; }     // Return top item without popping it.
    bool empty  ( ){ return top == 0; }
    int  size   ( ){ return top; }          // Number of items on the stack.
};

```

```
339  //-----------------------------------------------------------------------
340  template <class T>
341  void Stack<T>::
342  print( ostream& out )  {
343      T* p=s;                          // Scanner & end pointer for data
344      T* pend = s+top;
345      out << "The stack " <<name << " contains: Bottom~~ ";
346      for ( ; p < pend; ++p)  cout <<' ' << *p;
347      out << " ~~Top" <<endl;
348  }
349
350  //-----------------------------------------------------------------------------
351  template <class T>
352  void Stack<T>::
353  push( T c ) {
354      if (top == max) {       // If stack is full, allocate more space.
355          say( "-Doubling stack length-" );
356          T* temp = s;                            // grab old array.
357          s = new T[max*=2];                      // make bigger one,
358          memcpy( s, temp, top*sizeof(T) );       // copy data to it.
359          delete temp;                            // free old array.
360      }
361      s[top++] = c;           // Store data in array, prepare for next push.
362  }
363
364  #endif
```

Notes on the Stack Template file.

- The typedef on line 169 has been replaced by a template declaration preceding the class name (line 312). `template <class T> class Stack`
- Other than that, nothing within the class declaration has changed.
- However, the function definitions that were in the .cpp file in the non-template version are now at the end of the .hpp file. This is because all parts of a template must be available at compile time to any class that uses the template.
- The template declaration precedes the definition of each class function (lines 340, 351).

Using the Template: Brackets.hpp

Why use Templates? The C++ standard template library contains debugged code for a variety of containers (data structures). Template instantiation is easy, makes these powerful data structures available to you.

- To use the template, a client class must instantiate it and supply a defined type name in the angle brackets. (It does not need to name a class. It could name any predefined type.) We instantiate the template on line 377: `Stack<Token>* stk;` (Compare to line 37).
- Line 43 was also modified by adding the angle brackets and the template base type.
- These are the only ways that the program was modified to use a template instead of the typedef trick.

```
// ==========================================================================
// Name: Bracket-matching example of stack usage            File: brackets.hpp
// ==========================================================================
#ifndef BRACKETS_H
#define BRACKETS_H

#include "tools.hpp"
#include "token.hpp"
#include "stack.hpp"

class Brackets {
  private:
    Stack<Token>* stk;
    Token toptok;
    int lineno;

  public:
    Brackets() {
        stk = new Stack<Token>( "brackets" );
        lineno = 1;
    }
    ~Brackets(){ delete stk; }

    void analyze( istream& in);   // Check bracket nesting and matching in file.
    void mismatch( const char* msg, Token tok, bool eofile );      // Handle errors.
};
#endif
```