Exploring C++

Alice E. Fischer

University of New Haven

January 2009
(Revised to June 23, 2010)

Contents

Chapter 1: Preamble

1.1 Commandments

A nationally-known expert has said that C++ is a coding monster that forces us to use a disciplined style in order to tame it. This leads to a fundamental rule for skilled C++ programming:

Can is not the same as should.

The style guidelines below are motivated by years of experience writing and debugging C and C++ . None of them are arbitrary. I expect you to read them, understand them, and follow them.

Commandment 1. Use C++ , not C for all work in this course. The biggest difference is how one does input and output.

Commandment 2. The use of global variables for any purpose will not be tolerated. If your code contains a global variable declaration, your work will be handed back ungraded.

Commandment 3. Test every line of code you write. It is your job to prove to me that your entire program works. If I get a program without a test plan and output, I will assume that it does not compile. A program with a partial or inadequate test plan will be assumed to be buggy.

1.2 Style

If you want to be a professional, learn to make your work look professional. Start now. Read these guidelines and follow them. Your willingness, diligence, and ability to write clean code in my prescribed style will determine my willingness and ability to write a strong job recommendation for you.

1.2.1 General Rules

1. Learn how to include and use a local library, how to create, organize, and compile a multi-module program, and how to use the C preprocessor for conditional compilation.
2. Wherever possible, use **symbolic names** in your code. For example, use quoted characters instead of numeric ASCII codes.
3. Use only the **standard language**. Do not use proprietary extensions and language additions such as `clrscr()` and `conio.h`. Also, do not use system commands such as `pause`. Learn how to do these things within the standard language.
4. **Whitespace**. In general, put spaces around your operators and either before or after each parenthesis. Do not write a whole line of code with no spaces!
5. **Comments**. Use // and keep comments short. Do not repeat the obvious.
6. **Blank lines**. Use blank lines to group your code into "paragraphs" of related statements. Put a blank line after each group. DO NOT put a blank line between every line of code. Do not use more than one blank line. Do not put a blank line before an opening or closing brace. Do not separate the first line of a function from its body.
7. **Simplicity** is good; complexity is bad. If there are two ways to do something, the more straightforward or simpler way is preferred.

8. **Locality** is good; permitting one part of the program to depend on a distant part is bad. For example, initialize variables immediately before the loop that uses them, not somewhere else in the program.

9. Avoid writing **useless code**. For example, don't initialize variables that don't need initialization. Do not leave blocks of code in your program that you have commented out. Remove them before you send me the code.

10. Avoid writing the **same code** twice. For example, don't call strlen(s) twice on the same string. Don't write the same thing in both clauses of an if statement. Don't write two functions that output the same thing to two different streams. This way, when you correct an error once, it is corrected. You don't need to ask how many times the same error occurs.

11. File system **path names** are not portable. If you must write one in your program, put it in a `#define` at the top of main so that it can be found easily and changed easily.

1.2.2 Naming

1. Please **do not use** `i`, `I`, `l`, or `O` as a variable name because `i` and `l` look like 1 and `O` looks like 0. Use `j`, `k`, `m`, or `n` instead.

2. **Long, jointed names and short, meaningless names** make code equally hard to read. Try for moderate-length names.

3. **Local variables and class members** should have short names because they are always used in a restricted context. Global objects should have longer names because they can be used from distant parts of the program.

4. Names need to be **different enough** to be easily distinguished. The first part of a name is most important, the last letter next, and the middle is the least important. Any two names in your program should differ at the beginning or the end, not just in the middle.

5. Do not use the same name for two purposes (a class and a variable, for example). Do not use two names that differ only by an '`s`' on the end of one. Do not use two names that differ only by the case (for example `Object` and `object`).

6. I should be **able to pronounce** every variable name in English, and no two names should have the same pronunciation.

7. Learn to use the names of the various **zero-constants** appropriately. *NULL* is a pointer to location 0, `'\0'` is the null character, *""* is the null string, *false* is a bool value, and 0 and 0.0 are numbers. Use the constant that is correct for the current context.

8. Use names written entirely in UPPER CASE for `#defined` **constants** but not for other things. This is a long-honored `C` custom.

9. To be consistent with `Java` usage, the name of a class should start with an Upper case letter and use camel-case after that: `MyClassName`. Variable and function names should always start with a lower case letter: `myVar, myFunction()`.

1.2.3 Usage

1. **Use the `C++` language as it is intended**. Learn to use ++ (instead of +1) and if...break instead of complex loop structures with flags.

2. Please do not misuse the **conditional operator** to convert a *true* or *false* value to a 1 or a 0:

```
write this    (a < b)
instead of    (a < b) ?  1 :  0
```

3. Use the `->` **operator** when you are using a pointer to a struct. Example:

   ```
   write this    p->next->data
   instead of    *((*p).next).data
   ```

   Using * and . for this purpose leads to code that fails the simplicity test: it is hard to write, hard to debug, and hard to read.

4. Use pointers for sequential **array access** and subscripts for random access.

5. C has **subscripts**... use them. They are easy to write, easy to debug, and easy to read. Use pointer arithmetic rarely, if at all. Example: suppose that `ar` is an array of MAX ints and `p` is a pointer to an array of ints.

   ```
   write this                                ar[n]    or    p[n]
   instead of                                *(ar+n)  or    *(p+n)
   sometimes you should write this, however  int* ar_end = ar + MAX;
   ```

   The last line in this example is the ordinary way to set a pointer to the end of the array

1.2.4 Indentation

1. Never let **comments** interrupt the flow of the code-indentation.

2. Break up a long line of code into parts; **do not let it wrap around** to the left edge of the paper on your listings.

3. The **indentation style** preferred by experts is this:

   ```
   while ( k < num_items) {
       cout << age[k];
       k++;
   }
   ```

   This style is space-efficient and helps avoid certain kinds of common errors. Note that the lines within the loop are **indented a modest amount**: more than 1 or 2 spaces and less than 8 spaces. The opening bracket is on the end of the line that starts with the keyword and the closing bracket is directly below the first letter of the keyword.

4. Put the indentation into the file when you type the code originally. If you indent one line, most text editors will indent the next line similarly.

5. The other two generally approved styles are shown below. Please **adopt one of these three styles and use it consistently**. Randomly indented code is unprofessional and inappropriate for a graduate student.

Brackets aligned on the left:

```
while ( k < num_items)
{
    cout << age[k];
    k++;
}
```

Brackets aligned with the indented code:

```
while ( k < num_items)
    {
    cout << age[k];
    k++;
    }
```

1.2.5 Function Definitions

1. If a function is simple and fits entirely on **one line**, write it that way. Example:

   ```
   bool square_sum( double x, double y ) { return x*x + y*y; }
   ```

2. Otherwise, each function should **start with a whole-line comment** that forms a visual divider. If the function is nontrivial, a comment describing its purpose is often helpful. If there are preconditions for the function, state them here.

```
// ------------------------------------------------------------------
// If needed, put description of function here. ----------------------
void
print( Stack* St )             // Print contents of stack, formatted.
{
    char* p = St->s;           // Scanner and end pointer for data.
    char* pend = p + St->top;

    printf( "The stack %s contains: -[", St->name );
    for ( ;  p < pend;  ++p)  printf( " %c", *p );
    printf( " ]>" );
}
```

3. Write the **return type** on a line by itself immediately below the comment block then write the name of the function at the beginning of the next line, as in the above sample. Why? As we progress through this language, return types become more and more complex. It is greatly helpful to be able to find the function name easily.

4. In general, try to stick to **one return statement per function**. The only exception is when a second return statement substantially simplifies the code.

5. In a well-designed program **all function definitions are short**. Brief comments at the top of the function are often enough to explain the operation of the code. If the function code has identifiable sections or phases, there is a good chance that each phase should be broken out into separate functions. When a function cannot be broken up, each phase should be introduced by a whole-line comment.

1.2.6 Types, type definitions and struct

1. As this course progresses, a clear concept of the type of things is going to become essential. Regardless of examples you see in other contexts, when writing the name of a pointer or reference type, write **the asterisk or ampersand as part of the type name**, not as part of the object name. Example:

```
cell* make_cslist( cell& item );    // write it this way
cell *make_cslist( cell &item );    // not this way.
```

2. Use an *enum* **declaration** to give symbolic names to error codes. The name of the error then becomes adequate documentation for the line that uses it. Suppose you want to define codes for various different kinds of errors. Write it this way in C++ :

```
enum error_type { size_OK, Too_small, Too_large };
```

Declare and initialize an error_type variable like this:

```
error_type ercode = size_OK;
```

3. When using **structured types**, use a class declaration (in C++). Please DO NOT use the `struct` declaration or the `struct` keyword this term.

1.2.7 Using the Tools Library

All programs for this course must be done as projects and all must use the `banner()`, `fatal()`, and `bye()` functions from the tools library.

1. Establish a directory named cs427 on your hard disk for all the work in this course.
2. From the course website, please download a small library called "tools". It has two files: a source code file `tools.cpp` and a header file `tools.hpp`. On some browsers, you must download the code by using your mouse to copy the code and paste it into a new, empty, file. (Otherwise, the browser may insert HTML tags into the code file.) The ideal place to put your two tools files is at the top level of your cs427 directory. Alternatively, you can copy them directly from the Zoo directory `/c/cs427/code/tools`.
3. Within the cs427 directory, create a separate subdirectory for each programming assignment plus a subdirectory called "`submit`". Put your source code files, input file, and output files into this directory. If you are using an IDE (integrated development environment) such as Eclipse or Xcode, create a project file using the IDE's menu system, and copy the tools files into the project.
4. Before submitting, copy the files to be submitted into `submit`, make sure they meet the requirements of the assignment, and submit them from there according to the submission procedures for the course.
5. To use the tools, put `#include tools.hpp` in your main program and in any `.hpp` files you write yourself. Do not include `tools.c` anywhere – that job is done by `tools.hpp`, but be sure to copy it into your project so that it will get compiled and linked in with your code.
6. Various useful functions and macros are included in the tools library; please look at tools.hpp and learn about them. You will need to use *banner()*, *bye()*, *fatal()*, *flush()*, *DUMPp*, and *DUMPv*. You may also need *say()*, *today()*, *oclock()*,
7. Additional functions may be added during the term, so please check before each new assignment to see if the master copy has changed.
8. Look at the first several lines of `tools.hpp`; note that it includes all the useful `C` library headers and several of the `C++` headers. When you include the tools, you do not need to include these standard header files.
9. Personalization. Before you can use the tools, you must put your own name in the `#define NAME` line of the file "tools.hpp" in place of the dummy name "Ima Goetting Closeur".
10. Start each program with a call on *banner()* or *fbanner()*. This will label the top of your output with your name, etc. End each program with a call on *bye()*, which prints a "Normal termination" message.
11. If you need to abort execution, use *fatal()*. This prints a message and calls *exit()*. Do not use *assert()* because it does not give adequate user feedback. The syntax for *fatal()* is exactly like *printf()*. Example:

```
f_in = fopen( "data.in", "r" );
if (f_in == NULL) fatal( "Input file cannot be opened." );
```

Chapter 2: Issues and Overview

2.1 Why did C need a ++?

For application modeling. Shakespeare once explained it for houses, but it should be true of programs as well:

> **When we mean to build, we first survey the plot then draw the model.**
> . . . Shakespeare, King Henry IV.

2.1.1 Design Goals for C

C was designed to write Unix. C is sometimes called a "low level" language. It was created by Dennis Ritchie so that he and Kenneth Thompson could write a new operating system that they named Unix. The new language was designed to control the machine hardware (clock, registers, memory, devices) and implement input and output conversion. Thus, it was essential for C to be able to work efficiently and easily at a low level.

Ritchie and Thompson worked with small, slow machines, so they put great emphasis on creating an simple language that could be easily compiled into efficient object code. There is a direct and transparent relationship between C source code and the machine code produced by a C compiler.

Because C is a simple language, a C compiler can be much simpler than a compiler for C++ or Java. As a result, a good C compiler produces simple error comments tied to specific lines of code. Compilers for full-featured modern languages such as C++ and Java are the opposite: error comments can be hopelessly wordy and also vague. Often, they do not correctly pinpoint the erroneous line.

Ritchie never imagined that his language would leave their lab and become a dominant force in the world and the ancestor of three powerful modern languages, C++, C#, and Java. Thus, he did not worry about readability, portability, and reusability. Because of that, readability is only achieved in C by using self-discipline and adhering to strict rules of style. However, because of the clean design, C became the most portable and reusable language of its time.

In 1978, Brian Kernighan and Dennis Ritchie published "The C Programming Language", which served as the only language specification for eleven years. During that time, C and Unix became popular and widespread, and different implementations had subtle and troublesome differences. The ANSI C standard (1989) addressed this by providing a clear definition of the syntax and the meaning of C. The result was a low-level language that provides unlimited opportunity for expressing algorithms and excellent support for modular program construction. However, it provides little or no support for expressing higher-level abstractions. We can write many different efficient programs for implementing a queue in C, but we cannot express the abstraction "queue" in a clear, simple, coherent manner.

2.1.2 C++ Extends C.

C++ is an extension and adaptation of C. The designer, Bjarne. Stroustrup, originally implemented C++ as a set of macros that were translated by a preprocessor into ordinary C code. His intent was to retain efficiency and transparency and simultaneously improve the ability of the language to model abstractions. The full C language remains as a subset of C++ in the sense that anything that was legal in C is still legal in C++ (although some things have a slightly different meaning). In addition, many things that were considered "errors" in C are now legal and meaningful in C++.

Readability. C++ was no more readable than C, because C was retained as the basic vehicle for coding in C++ and is a proper subset of C++ . However, an application program, as a whole, may be much more readable in C++ than in C because of the new support for application modeling.

Portability. A portable program can be "brought up" on different kinds of computers and produce uniform results across platforms. By definition, if a language is fully portable, it does not exploit the special features of any hardware platform or operating system. It cannot rely on any particular bit-level representation of any object, operation, or device; therefore, it cannot manipulate such things. A compromise between portability and flexibility is important for real systems.

A program in C or C++ can be very portable if the programmer designs it with portability in mind and follows strict guidelines about segregating sections of code that are not portable. Skillful use of the preprocessor and conditional compilation can compensate for differences in hardware and in the system environment. However, programs written by naive programmers are usually not portable because C's most basic type, `int`, is partially undefined. Programs written for the 4-byte integer model often malfunction when compiled under 2-byte compilers and vice versa. C++ does nothing to improve this situation.

Reusability. Code that is filled with details about a particular application is not very reusable. In C, the typedef and #define commands do provide a little bit of support for creating generic code that can be tailored to a particular situation as a last step before compilation. The C libraries even include two generic functions (`quicksort()` and `bsearch()` that can be used to process an array of any base type. C++ provides much broader support for this technique and provides new type definition and type conversion facilities that make generic code easier to write.

Teamwork potential. C++ supports highly modular design and implementation and reusable components. This is ideal for team projects. The most skilled members of the group can design the project and implement any non-routine portions. Lesser-skilled programmers can implement the routine modules using the expert's classes, classes from the standard template library, and proprietary class libraries. All these people can work simultaneously, guided by defined class interfaces, to produce a complete application.

2.1.3 Modeling.

The problems of modeling are the same in C and C++. In both cases the questions are, what data objects do you need to define, how should each object work, and how do they relate to each other? A good C programmer would put the code for each type of object or activity in a different file and could use type modifiers `extern` and `static` to control visibility. A poor C programmer, however, would throw it all into one file, producing an unreadable and incomprehensible mess. Skill and style made a huge difference. In contrast, C++ provides classes for modeling objects and several ways to declare or define the relationship of one class to others.

What is a model? A *model of an object* is a list of the relevant facts about that object in some language. A *low level* model is a description of a particular implementation of the object, that specifies the number of parts in the object, the type of each part, and the position of each part in relation to other parts. C supports only low level models.

C++ also supports *high-level* or *abstract* models, which specify the functional properties of an object without specifying a particular representation. This high-level model must be backed up by specific low-level definitions for each abstraction before a program can be translated. However, depending on the translator used and the low-level definitions supplied, the actual number and arrangement of bytes of storage that will be used to represent the object may vary from translator to translator.

A high level model of a *process* or *function* specifies the pre- and post-conditions without specifying exactly how to get from one to the other. A low level model of a *function* is a sequence of program definitions, declarations, and statements that can be performed on objects from specific data types. This sequence must start with objects that meet the specified pre-conditions and end by achieving the post-conditions.

High level process models are not supported by the C language but do form an important element of project documentation. In contrast, C++ provides class hierarchies and virtual functions which allow the programmer to build high-level models of functionality, and later implement them.

Explicit vs. Implicit Representation. Information expressed explicitly in a program may be used by the language translator. For example, the type qualifier `const` is used liberally in well-written C++ applications.

This permits the compiler to verify that the variable is never modified directly and is never passed to a function that might modify it.

A language that permits explicit communication of information must have a translator that can identify, store, organize, and utilize that information. For example, if a language permits programmers to define types and relationships among types, the translator needs to implement type tables (where type descriptions are stored), new allocation methods that use these programmer-defined descriptions, and more elaborate rules for type checking, type conversions, and type errors. This is one of the reasons why C++ translators are bigger and slower than C translators. The greater the variety and sophistication of the information that is declared, the more effort it is to translate it into low-level code that carries out the intent of the declarations.

Semantic Intent A data object (variable, record, array, etc.) in a program has some intended meaning that can be known only if the programmer communicates or declares it. A programmer can try to choose a meaningful name and can supplement the code by adding comments that explain the intent, but those mechanisms communicate only to humans, not to compilers. The primary mechanism for expressing intent in most languages is the data type of an object. Some languages support more explicit declaration of intent than others. For example, C uses type `int` to represent many kinds of objects with very different semantics (numbers, truth values, characters, and bit masks). C++ is more discriminating; truth values are represented by a semantically distinct type, `bool`, and programmer-defined enumerations also create distinct types.

A program has *semantic validity* if it faithfully carries out the programmer's semantic intent. A language is badly designed to the extent that it lets the programmer write code that has no reasonable valid interpretation. A well-designed language will identify such code as an error. A strong type checking mechanism can help a programmer write a semantically valid (meaningful) program. Before applying a function to a data object, both C and C++ translators test whether the call is meaningful, that is, the function is defined for objects of the given type. An attempt to apply a function to a data object of the wrong type is identified as a semantic error at compile time.

However, the type system and the rules for translating function calls are much more complex in C++ than in C for the reasons discussed in Section 2.3. For these reasons and others, achieving the first error-free compile is more difficult in C++, but the compiled code is more likely to run correctly.

2.2 Object Oriented Principles.

Classes. The term "object-oriented" has become popular, and "object-oriented" analysis, design, and implementation has been put forward as a solution to several problems that plague the software industry. OO analysis is a set of formal methods for analyzing and structuring an application; the result of an OO analysis is an OO design. OO programs are built out of a collection of modules, often called *classes* that contain both functions and data. The most fundamental design goal in these classes is that a class should take care of itself.

The way a language is used is more important in OO design than which language is used. C++ was designed to support OO programming; it is a convenient and powerful vehicle for implementing an OO design. However, with somewhat more effort, that same OO design could also be implemented in C.[1] Similarly, a non-OO program can be written in C++.

Three principles central to object-oriented programming are locality, coherent representation, and generic functions.

Locality and Encapsulation. The effects of an action or a declaration can be global (affecting all parts of a program) or local (affecting only nearby parts). The further the effects of an action reach in time (elapsed during execution) or in space (measured in pages of code), the more complex and harder it is to debug a program. The further an action has influence, the harder it is to remember relevant details, and the more subtle errors seem to creep into the code.

A well-designed language supports and encourages locality. All modern languages permit functions to have local variables and minimize the need for global variables. C goes farther than many language by supporting static local variables that have the lifetime of a global object but only local visibility. OO languages go further

[1]The insertion sort code example is an OO design implemented in C that illustrates locality, coherent representation, and reusable generic code.

still by introducing "private" fields in structures (class objects) that cannot be seen or changed by functions outside the class. We say that the private members are *encapsulated* within the class.

Coherent vs. Diffuse Representation. A representation is *coherent* if an external entity (object, idea, or process) is represented by a single symbol in the program (a name or a pointer) so that it may be referenced and manipulated as a unit. A representation is *diffuse* if various parts of the representation are known by different names, and no one name or symbol applies to the whole.

Coherence is the most important way in which object-oriented languages differ from older languages. In Pascal, for example, a programmer can declare a new data type and write a set of functions to operate on objects of that type. Taken together, the data objects and functions implement the programmer's model. However, Pascal does not provide a way to group the data and functions into a coherent package, or declare that they form a meaningful module. C is a little better in this respect because separately-compiled code modules allow the programmer to group related things together and keep both functions and data objects private within the module. In an OO language, however, the class mechanisms do this and more, and provide a convenient syntax for declaring classes and class relationships.

Generic code. A big leap forward in representational power was the introduction of generic code. A generic function is one like "+" whose meaning depends on the type of the operands. Floating-point "+" and integer "+" carry out the same conceptual operation on two different representations of numbers. If we wish to define the same operation (such as "print") on five data types, C forces us to introduce five different function names. C++ lets us use one name to refer to several methods which, taken together, comprise a function. (In C++ terminology, a single name is "overloaded" by giving it several meanings.) The translator decides which definition to use for each function call based on the types of the arguments in that call. This makes it much easier to build libraries of functions that can be used and reused in a variety of situations.

OO Drawbacks. Unfortunately, a language with OO capabilities is complex. The OO extensions in C++ make it considerably more complicated than C. It is a massive and complex language. To become an "expert" requires a much higher level of understanding in C++ than in C, and C is difficult compared to Pascal or FORTRAN. The innate complexity of the language is reflected in its translators; C++ compilers are slow and give very confusing error comments.

The ease with which one can write legal but meaningless code is a hallmark characteristic of C. The C programmer can write all sorts of senseless but legal things (such as a<b<c). C++ has a better-developed system of types and type checking, which improves the situation somewhat. However C++ also provides powerful tools, such as the ability to add new definitions to old operators, that can easily be overused or misused. A good C++ programmer designs and writes code in a strictly disciplined style, following design guidelines that have been evolved from experience over the years. Learning these guidelines and how to apply them is more important than learning the syntax of C++, if the goal is to produce high-quality, debugged, programs.

2.3 Important Differences

- **Comments.** Comments can begin with // and end with newline. Please use only this kind of comment to write C++ code. Then the old kind can be used to /* comment out */ sections of text during debugging.

- **Executable declarations.** Declarations can be mixed in with the code. This makes it possible to print greeting messages before processing the declarations. Why is this useful? Because C++ declarations can trigger file processing and dynamic memory allocation and it is VERY helpful to precede each major declaration with a message that will let the programmer track the progress of the program.

  When you put a declaration in a loop, the object will be allocated, initialized, and deallocated every time around the loop. This is unnecessary and inefficient for most variables but it can be useful when you need a strictly temporary object of a class type.

  Declarations must not be written inside one `case` of a `switch` statement unless you open a new block (using curly braces) surrounding the code for the case.

- **Type identity.** In C, the type system is based on the way values are *represented*, not on what they *mean*. For example, pointers, integers, truth values, and characters are all represented by bitstrings of various lengths. Because they are represented identically, an `int` variable can be used where a `char` value is wanted, and vice versa. Truth values and integers are even more closely associated: there is no distinction at all. Because of this, one of the most powerful tools for ensuring correctness is compromised, and expressions that should cause type errors are accepted. (Example: `k < m < n`.)

  In C++, as in all modern languages, type identity is based on `meaning`, not representation. Thus, truth values and integers form distinct types. To allow backwards compatibility, automatic type coercion rules have been added. However, new programs should be written in ways that do not depend on the old features of C or the presence of automatic type conversions.

- **Type bool.** In standard C++ , type `bool`, whose values are named `false` and `true`, is not the same type as type `int`. It is defined as a separate type along with type conversions between `bool` and `int`. The conversions will be used automatically by the compiler when a programmer uses type `int` in a context that calls for type `bool`. However, good style demands that a C++ programmer use type `bool`, not `int` and the constants true (not 1) and false (not 0), for true/false situations.

- **Enumerated types.** Enumerated type declarations were one of the last additions to the C language prior to standardization. In older versions of the language, `#define` statements were used to introduce symbolic codes, and a program might start with dozens of `#define` statements. They were tedious to read and write and gave no clue about which symbols might be related to each other. Enumerations were added to the language to provide a concise way to define sets of related symbols. For example, a program might contain error codes and category codes, all used to classify the input. Using `#define`, there is no good way to distinguish one kind of code from the other. By using two enumerations, you can give a name to each set and easily show which codes belong to it.

  In C, enumeration symbols are implemented as integers, not as a distinct, identifiable type. Because of this, the compiler does not generate type errors when they are used interchangeably, and many C programmers make no distinction. In contrast, in C++, an enumeration forms s distinct type that is not the same as type `int` or any other enumeration. Compliers *will* generate error and warning comments when they are used inappropriately.

- **Type conversions.** C provides type cast operators that work from any numeric type (`double, float, int, unsigned, char`) to any other numeric type, and from a pointer of any base type to a pointer of any other base type. These casts are used automatically whenever necessary:

  - When an argument type fails to match a parameter type.
  - When the expression in a return statement does not match the declared return type.
  - When the types of the left and right sides of an assignment do not match.
  - When two operands of a binary operator are different numeric types.

  These rules are the same in C++, but, in addition, the programmer can define new type casts and conversions for new classes. These operations can be applied explicitly (like a type cast) and will also be used by the compiler, as described above, to coerce types of arguments, return values, assignments, and operands.

- **Assignment.** First, the operator = is predefined for all types except arrays. Second, the behavior of the assignment operator has changed: the C++ version returns the address of the location that received the stored value. (The C version returns the value that was stored.) This makes no difference in normal usage. However, the result of some complex, nested assignments might be different.

- **Operators can be extended to work with new types.** For example, the numeric operators `+, *, -, and /` are predefined for pairs of integers, pairs of floating point numbers, and mixed pairs. Now suppose you have defined a new numeric type, Complex, and wish to define arithmetic on Complex numbers. You can define an additional method for `+` to add pairs of Complex numbers, and you can define conversion functions that convert Complex numbers to other numeric types.

- **Reference parameters and return values.** C supports only call-by-value for simple objects and structures, and only call-by-reference for arrays. When a pointer is passed as an argument in C, we can refer to it as "call-by-pointer", which is an abbreviation for "call by passing a pointer by value". This permits the function to change a value in the caller's territory. However, it is not the same as call-by-reference, which is supported by C++ in addition to all the parameter passing mechanisms in C.

  In call-by-reference, an address (not a pointer) is passed to the function and the parameter name becomes an alias for the caller's variable. Like a pointer parameter, a reference parameter permits a function to change the value stored in the caller's variable. Unlike a pointer parameter, the reference parameter cannot be made to refer to a different location. Also, unlike a pointer, a * does not need to be used when the variable is used within the function. General restrictions on the use of references are:

  - A reference parameter can be passed by value as an argument to another function, but it cannot be passed by reference.
  - Sometimes reference parameters are used to avoid the time and space necessary to `copy` an entire argument value. In that case, if it is undesirable for the function to be able to change the caller's variable, the parameter should be declared to be a `const &` type.
  - Arrays are automatically passed by reference in both C and C++. You must not (and do not need to) write the ampersand in the parameter declaration.
  - You can't set a pointer variable to point at a reference.
  - You can't create a reference to a part of a bitfield structure.

- **Protection.** Structures, as in C are still available in C++, but a C++ `struct` may have function members as well as data members. In addition, C++ provides *classes*, which are structures with added encapsulation capabilities. Learning to use encapsulation wisely will be a major part of this course.

- **Function methods.** Any function can have more than one definition, as long as the list of parameter types, (the *signature*) of every definition is different. The individual definitions are called *methods* of the function. In the common terminology, such a function is called *overloaded*. I prefer not to use this term so broadly. In this course, I will distinguish among *extending*, *overriding*, and *overloading* a function.

- **I/O.** C and C++ I/O are completely different, but a program can use both kinds of I/O statements in almost any mixture. The C and C++ I/O systems both have advantages and disadvantages. For example, simple output is easier in C++ but output in columns is easier in C. Since one purpose of this course is to learn C++, please use only C++ output in the C++ programs you write for this course.

- **Using the C++ libraries.** To use one of the standard libraries in C, we write an `#include` statement of the form `#include <stdio.h>` or `#include <math.h>`. A statement of the same form can be used in C++ . For example, the standard input/output library can be used by writing `#include <iostream.h>`. However, this form of the include statement is "old fashioned", and should not be used in new programs. Instead, you should write two lines that do the same thing:

  ```
  #include <iostream>
  using namespace std;
  ```

  The new kind of include statement still tells the preprocessor to include the headers for the iostream library, but it does not give the precise name of the file that contains those headers. It is left to the compiler to map the abstract name "iostream" onto whatever local file actually contains those headers. The second line brings the function names from the `iostream` library into your own context. Without this declaration, you would have to write `iostream::` in front of every function or constant name defined by the library.

2.4 Generic Insertion Sort

This is an object-oriented program written in a non-OO language, C. It embodies a variety of OO design, coding, and style principles, and also a few old C techniques. Notes follow the code. Use this example as a general guide for doing your own work. The most important themes are:

- Highly modular code with a streamlined main function.
- Reusable generic code instantiated using typedef and #define.
- The use of pointers, cursors, and sentinels to process an array.
- OO concept: The data pack, along with its setup, input, and print functions forms a class (or would, if written in C++). A data pack is an array packaged together with the information needed to use and manage it: the number of slots in the array and the number of slots that are currently filled with valid data. This implementation of a data pack includes a sort function, as well as the basic functions for construction, input and output from the pack. An addition that could be useful is a status code that indicates whether the data is sorted, and if so, in what order.
- OO design principle: A class should take care of itself. The functions that belong to a class operate on the class's data objects. No other functions should manipulate these objects.

A type declaration and the main program are given first, followed by detailed notes, keyed to the line numbers in the code. A call chart for the program is given in Figure 2.1. In the chart, white boxes surround functions defined in this application, light gray boxes denote functions in the standard libraries and dark gray denotes the tools library.

2.4.1 Main Program

1. A program that is built in modules has a pair of files for each module: a header file and a code file. The code file starts with a command to include the corresponding header file. The main program starts with commands to include one or more module headers. In this case, `main()` uses the Datapack module, so we include the header for that module.
2. The main program consists entirely of calls on the DataPack functions. All the work on the values in the `dataPack` is done by those modules.
3. The `setup()` function constructs a fully formed, legal, empty, DataPack. Then `readData()` fills it with data, `sortData()` sorts the data, and `printData()` shows us the data before and after sorting.

```
 1  /*  -----------------------------------------------------------------------
 2  //  Main program for Sorting DataPacks.                             2_main.c
 3  //  Created by Alice Fischer on Mon Dec 22 2003.
 4  //  ---------------------------------------------------------------------*/
 5  #include "pack.h"
 6
 7  /*  ---------------------------------------------------------------------*/
 8  int main( void )
 9  {
10      DataPack theData;
11      banner();
12      say( "Construct pack, read data" );     setup( &theData );
13      say( "\n%d data items read:",           theData.n );
14      say( "\nUnsorted data:" );              printData( &theData, stdout );
15      say( "\nBeginning to sort." );          sortData( &theData );
16      say( "\nSorted results:" );             printData( &theData, stdout );
17      bye();
18      return 0;
19  }
```

2.4.2 The DataPack Header File

1. By changing the statements on lines 26. . . 28, we can change the program to work with any basic C data type instead of with float. We need only to change the formats and define `BT` (which stands for base type) to be the type of data that will be stored in the array and sorted. For example, these lines would be used for type int:

```
#define IN_FMT "%i"
#define OUT_FMT "%i\n"
typedef int BT;
```

2. Line 24 includes definitions from the local library, `tools`, that we will be using throughout the term. Note the use of quotes, not angle brackets. Note that we include the header file, not the code file, because we are doing a multi-module compile and separate link operation.

   ```
   #include "tools.h"
   ```

3. The `dataPack` type makes a coherent grouping of *all* the parts that are needed to manage an array of data. In C++, it would be a class, not a structure.
4. The code of `main()` (lines 11. . . 17) is simply an outline of the processing steps with some user feedback added. All work is delegated to functions. A call chart (Figure 2.1) shows the overall dynamic structure of a program and is an important form of documentation. The arrows indicate which functions are called by each other function. Frequently, the standard I/O functions are omitted from such charts.

```
20  /*  ---------------------------------------------------------------------
21  //  Header file for all DataPack programs.                          pack.h
22  //  Created by Alice Fischer on Mon Dec 22 2003.
23  */
24  #include "tools.h"
25  /* --------------------------------- Instantiations for generic parts  */
26  #define IN_FMT "%g"
27  #define OUT_FMT "%.7g\n"
28  typedef float BT ;
29  #define LENGTH 20
30
31  /* -------------------------------------------- Generic type definition  */
32  typedef struct {
33      int n;          /* Allocation length. */
34      int max;        /* Data length. */
35      BT* store;      /* Dynamic data array. */
36  } DataPack;
37
38  /* -------------------------------------------------------- Prototypes  */
39  void setup( DataPack* pData );
40  void printData( DataPack* pData, stream outstream );
41  void sortData( DataPack* pData );
```

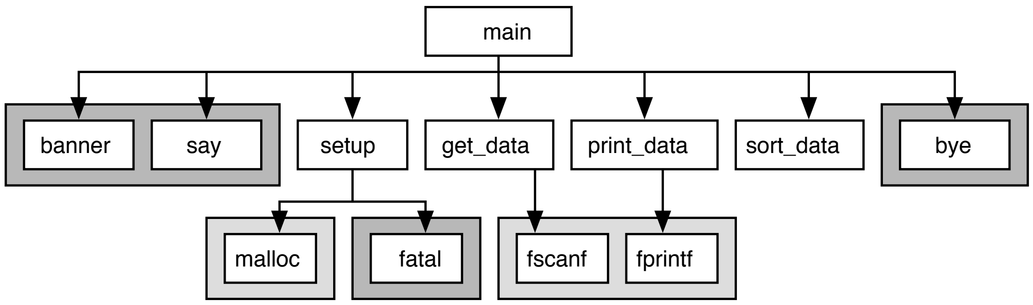


Figure 2.1: Call hierarchy chart for insertion sort.

2.4.3 The Data Pack Functions

Constructing the data structure. The `setup()` function (lines 52. . . 61) is called from line 12 of `main()`.

1. Every data structure should have a function to build and initialize its parts. In C++, these are called constructors. At the end of construction, the object should be completely formed and internally consistent.
2. In C, this function must be called explicitly, while in a C++ program, the appropriate constructor function is called automatically whenever an object is allocated.
3. We allocate necessary storage on line 57 and check for successful allocation on the next two lines. In C++, this kind of error checking is not necessary. Note the call on fatal() (line 59) to handle an unrecoverable error.
4. Lines 55. . . 57 initialize all the fields of the object so that they are consistent with each other and with the amount of storage allocated.

```
42  // ------------------------------------------------------------------------------
43  //  Code file for all DataPack programs.                            pack.c
44  //  Created by Alice Fischer on Mon Dec 22 2003.
45
46  #include "pack.h"
47  static void readData( DataPack* pData );   // A private function
48
49  //---------------------------------------------------------------------
50  // Allocate memory for data.  Errors are fatal.
51
52  void
53  setup( DataPack* pData )
54  {
55      pData->n = 0;                              // Currently empty.
56      pData->max = LENGTH;                       // Space for LENGTH items.
57      pData->store = malloc(LENGTH*sizeof(BT));
58      if (pData->store == NULL)
59          fatal( " Error: not enough memory for %i BTs\n", LENGTH );
60      readData( pData );
61  }
62
63  //------------------------------------------------------------------------------
64  // Use sequential access pattern to store data from infile into data pack.
65
66  static void
67  readData( DataPack* pData )
68  {
69      char filename[80];              // For name of input file
70      stream infile;
71      BT* cursor, * end;
72      int status;
73
74      printf( "\nEnter name of data file to be searched: " );
75      scanf( "%79s", filename );
76      infile = fopen ( filename, "r" );
77      if (! infile)  fatal( "    Error: couldn't open input %s\n", filename );
78
79      cursor = pData->store;          // Scanning pointer, set to start of array.
80      end = pData->store + pData->max;    // An off-board sentinel
81
82      for( ; cursor<end; ++ cursor ) {
83          status = fscanf( infile, IN_FMT, cursor);
84          if( status != 1 ) break;    // Quit for bad data or for end of file.
85      }
86      pData->n = cursor-pData->store; // Actual # of items read.
87  }
88
```

Filling the DataPack. Every data structure needs one or more functions for entering data into it.

1. Ideally, these functions should not only provide access to the data structure; they should also be the *only* way to enter data into it. In C++, private data members allow us to restrict access in this way. However, in C, good programming style and lack of knowledge of the implementation are the only ways that unwanted access can be prevented.
2. Upon entry to `readData()` (lines 66. . . 87), the information in the data pack is used to establish the pointers that will be used for sequential array processing. At no time is reference made to global variables or constants; the data structure is self sufficient.
3. On line 79, the cursor is initialized to the beginning of the array and will always point at the first unfilled slot. Line 80 establishes the end pointer, an off-board sentinel. It points to the first location that is not in the array. This is legal and sound practice that conforms to the C standard. When `cursor==end`, the array is full.

4. On line 83, there is a call on fscanf() that uses the format string `IN_FMT` defined at the top of the program. By changing the define statements and the typedef, we can change the type of data that we are sorting to any type for which `<=` is defined.

   Note that the return value from fscanf() (line 83) is stored and checked. When the format for `fscanf()` calls for reading one item, the return value of 1 indicates success. A smaller value indicates that a read error or an end-of-file caused the program to stop reading the data file before a data element was read. In a production program, more elaborate error handling would be needed.

5. It is important to maintain internal consistency within an object. Before returning, the data pack is updated (line 86) to reflect the number of items actually stored in it. We use pointer subtraction to calculate this number; the address of the head of the array is subtracted from the cursor. This tells us the number of array slots (not the number of bytes) between them.

```
 89   // Continuation of code file for  DataPack programs.                      pack.c
 90   // -----------------------------------------------------------------------------
 91   // Print array values, one per line, to the selected stream.
 92
 93   void
 94   printData( DataPack* pData, stream outstream )
 95   {
 96       for( int k=0; k < pData->n; ++k)
 97           fprintf( outstream, OUT_FMT, pData->store[k] );
 98   }
 99
100   // -----------------------------------------------------------------------------
101   // Generic insertion sort using a DataPack.
102   // Sort n values starting at pData->store by an insertion sort algorithm.
103
104   void
105   sortData( DataPack* pData )
106   {
107       BT* const end = pData->store + pData->n;    // Off-board sentinel.
108       BT* pass;                           // First unsorted item; begin pass here.
109       BT  newcomer;                       // Data value being inserted.
110       BT* hole;                           // Array slot containing no data.
111
112       for ( pass=pData->store+1; pass<end; ++pass ) {
113           // Pick up next item and insert into sorted portion of array.
114           newcomer = *pass;
115           for ( hole=pass; hole>pData->store; --hole ) {
116               if ( *(hole-1) <= newcomer ) break;   // Insertion slot is found.
117               *hole = *(hole-1);                    // Move item back one slot.
118           }
119           *hole = newcomer;
120       }
121   }
```

Printing the data. Some sort of output function is usually necessary in an application. It is also essential during debugging, and therefore is one of the first functions written for a new data structure.

1. The `printData()` function (lines 93. . . 98) is simpler than `readData()` because we know at the outset how much data is in the array and that does not change, and because there is no need to check for errors or end-of-file. This makes a simple `for` loop (line 96) and subscripts (line 97) an attractive alternative to a cursor for scanning through the array values.

2. Lines 89. . . 91 are a loop that calls `fprintf()` using the format string defined at the top of the program.

Sorting by insertion. The `sortData()` function (lines 104. . . 121) implements insertion sort, the most efficient of the simple sorts (and reasonable to use in practice for small arrays, perhaps up to size 8 or so). This is a typical double-loop implementation of the insertion sort algorithm. The implementation uses pointers, not subscripts, which slightly increases efficiency.

1. Each declaration has a comment that explains the usage of the variable. Note the use of brief, imaginative, meaningful variable names. This makes a huge difference in the readability of the code. In contrast, code is much harder to understand when it is written using identifiers like `i` and `j` for loop indices.
2. As in the previous functions, the pointer named **`end`** points at the first memory location that is not in the array being sorted.
3. At all times, the array is divided into two portions: a sorted portion (smaller subscripts) and an unsorted portion (larger subscripts). Initially, the sorted part of the array contains one element, so the sorting loop (lines 112. . . 120) starts with the element at data[1].
4. The algorithm makes $N - 1$ passes over the data to sort N items. On each pass through the outer loop, it selects and copies one item (the **`newcomer`**, line 114) from the beginning of the unsorted portion of the array, leaving a **`hole`** that does not contain significant data.
5. The inner loop (lines 115. . . 118) searches the sorted portion of the array for the correct slot for the newcomer. The **`for`** loop is used to guard the boundaries of the array. Within the loop body, an **`if...break`**, line 116, is used to leave the loop when the correct insertion slot is located.
6. To find that position, the loop scans backwards through the array, comparing the **`newcomer`** to each element in turn (line 116). If the **`newcomer`** is smaller, the other element is moved into the **`hole`** (line 117) and the **`hole`** pointer is decremented by the loop (line 118).
7. If the **`newcomer`** is not smaller, we break out of the inner loop and copy the **`newcomer`** into the **`hole`** (line 119).

Chapter 3: C++ I/O for the C Programmer

How to learn C++:

> **Try to put into practice what you already know, and in so doing you will in good time discover the hidden things which you now inquire about.**
> — Henry Van Dyke, American clergyman, educator, and author.

3.1 Familiar Things in a New Language

In C, `scanf()` and `printf()` are only defined for the built-in types and the programmer must supply a format field specifier for each variable read or printed. If the field specifier does not agree with the type of the variable, garbage will result.

In contrast, C++ supports a generic I/O facility called "C++ stream I/O". Input and output conversion are controlled by the declared type of the I/O variables: you write the same thing to output a double or a string as you write for an integer, but the results are different. This makes casual input and output easier. However, controlling field width, justification, and output precision is a pain in C++. The commands to do these jobs are wordy, non-intuitive, and cannot be combined on the same line with ordinary output commands. You can always use C formatted I/O in C++; you may prefer to do so if you want easy format control. Both systems can be, and often are, used in the same program. However, in this class, please use only C++ I/O.

This section is for people who know the stdio library in C and want convert their knowledge to C++ (or vice-versa). Several I/O tasks are listed; for each the C solution is given first, followed by the C++ solution. Code fragments are given to illustrate each item. Boring but accurate short programs that use the commands in context are also supplied.

3.2 Include Files

C: `#include <stdio.h>`
C++:

- `#include <iostream>` for interactive I/O.
- `#include <iomanip>` for format control.
- `#include <fstream>` for file I/O.
- `#include <sstream>` for strings that emulate streams.
- `using namespace std;` to bring the names of included library facilities into your working namespace.

3.3 Streams and Files

A stream is an object created by a program to allow it to access a file, socket, or some other source or destination for data.

Predeclared streams:

- C: `stdin, stdout, stderr`
- C++: `cin, cout, cerr, clog`
- The C++ streams `cin` and `cout` share buffers with the corresponding C streams. The streams `stderr` and `cerr` are unbuffered. The new stream, `clog` is used for logging transactions.

Stream handling. When a stream is opened, a data structure is created that contains the stream buffer, several status flags, and all the other information necessary to manage the stream.

- C:
```
typedef FILE* stream;              /* Or include tools.h. */
stream fin, fout;
fout = fopen( "myfile.out", "w" );  /* Open stream for writing. */
fin = fopen( "myfile.in", "r" );    /* Open stream for reading. */
if (fin == NULL)                   /* Test for unsuccessful open. */
if (feof( fin ))                   /* Test for end of file. */
```

- C++: Stream classes are built into C++; a `typedef` is not needed. There are several stream classes that form a class hierarchy whose root is the class `ios`. This class defines flags and functions that are common to all stream classes. Below that are the two classes whose names are used most often: `istream` for input and `ostream` for output. The predefined stream `cin` is an `istream`; `cout`, `cerr`, and `clog` are `ostreams`. These are all sequential streams–data is either read or written in strict sequential order. In contrast, the `iostreams` permit random-access input and output, such as a database would require. Below all three of these main stream classes are file-based streams and strings that emulate streams.

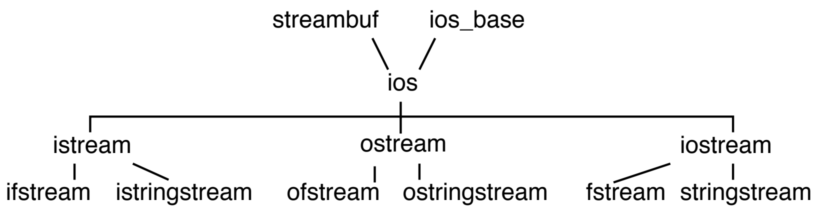


Figure 3.1: The stream type hierarchy.

Each class in the stream hierarchy is a variety of the class above it. When you call a function that is defined for a class high in the tree, you can use an argument of any class that is below that. For example, if `fin` is an `ifstream` (which is used for an input file) you can call any function defined for class `istream` or class `ios` with `fin` as an argument.

```
ofstream fout ( "myfile.out" );  // Open stream for writing.
ifstream fin ( "myfile.in" );    // Open stream for reading.
fin.open( "myfile.in" );         // Alternate way to open a stream.
fin.close();                     // Close a stream--not usually necessary.
if (!fin) fatal(...);            // Test for unsuccessful open.
if (fin.eof()) break;            // Test for end of file.
if (fin.good()) ...              // Test for successful read operation.
if (fin.fail()) ...              // Test for hardware or conversion error.
```

The stream declaration and open statement are combined in one line. The second declaration shown above is supposed to work, and works on my system. If the file does not exist, it is not created, and the `if` statement on the fifth line is used to handle the error. According to one student, though, this does not work properly in Visual C++ 4.1. If the file does not exist, this command creates it. To avoid creation of an empty file, he must open the file with the flag `ios::nocreate`, thus:

```
ifstream in ( "parts.in", ios::nocreate | ios::in );   // For Visual C++ ?
if (!in) fatal( "Could not open file parts.in" );
```

Closing streams. In both languages, streams are closed automatically when the program terminates. To close a stream prior to the end of the program:

- In C: `fclose( fin );`
- In C++: `fin.close();`

3.3.1 Common Stream Errors, Misconceptions, and Problems

Stream ties: `cout` is tied to `cin`, that is, whenever the cout buffer is non-empty and input is called for on `cin, cout` is automatically flushed. This is also true in C: `stdout` is tied to `stdin`. Both languages were designed this way so that you could display an input prompt without adding a newline to flush the buffer, and permit the input to be on the same line as the prompt.

Flushing streams. The standard `fflush()` function in C applies only to output streams. Some students have the belief that you can also flush a C input stream; this might be true in some nonstandard implementation, but it is not true according to the C standard.

In C++, `flush` is technically a manipulator, not a function, but it also applies only to `ostreams`. The `tools.cpp` library contains a definition of flush as a manipulator for `istreams`. This allows the programmer to leave the input buffer in a predictable state: empty. It is primarily useful for handling errors during interactive input.

```
// -----------------------------------------------------------------
// Flush cin buffer as in cin >>x >>flush >>y; or cin >> flush;
istream&
flush( istream& is ) { return is.seekg( 0, ios::end ); }
```

End-of-file and error processing. In both C and C++, the end-of-file flag does not come on until you attempt to read data beyond the end of a file. The last line of data can be read fully and correctly without turning on the end-of-file flag. Therefore, an end-of-file test should be made between reading the data and attempting to process it. The cleanest, most reliable, simplest way to do this is by using an `if...break` statement to leave the input loop. The short program at the end of Section 3.6 shows how to test for hardware and format errors as well as eof.

3.4 Input

The following variables will be used throughout this section and the next:

```
char   c1;                 int    k1, k2;
short  n1, n2;             long   m1, m2;
float  x1, x2;             double y1, y2;
char*  word2 = "Polly";    char w1[80], w2[80], w3[80];
float* px1= &x1, *px2= &x2;
```

Numeric input. Basic operations are shown in the table below for both C and C++. Note how simple basic input is when using the C++ stream operators `>>` and `<<`. All of the operations in this table skip leading whitespace. It is not necessary and not helpful to do your own number conversion using `atoi` or `strtol`. Learn to use the stream library as it was intended!

The table below shows the normal way to read numbers. This works properly if the input is correct. However, it will cause problems if anything other than a number is in the input stream when a number is expected. If a non-numeric character is found while attempting a numeric read, the input stream will signal an error. A bullet-proof program tests the stream status flag using good() or fail() to detect such errors. To recover, use `clear()`. An example of read-error detection and proper error handling is shown at the bottom of the eof_demo program.

Type	In C	In C++
`int`	`scanf("%i%d", &k1, &k2);`	`cin >> k1 >> k2;`
`long`	`scanf("%li%ld", &m1, &m2);`	`cin >> m1 >> m2;`
`short`	`scanf("%hi%hd", &n1, &n2);`	`cin >> n1 >> n2;`
`float`	`scanf("%f%g ", &x1, &x2);`	`cin >> x1 >> x2;`
`double`	`scanf("%lf%lg", &y1, &y2);`	`cin >> y1 >> y2;`

Single character input. Stream operator: `>>`. Function: `get(char_var)`.

Operation	In C	In C++
Read next keystroke	`scanf("%c", &c1);`	`cin.get(c1);`
Skip leading whitespace	`scanf(" %c", &c1);`	`cin >> c1;`

String input. Functions: `ignore(n)`, `getline( buf, limit )`, `get( buf, limit, terminator )`, and `getline( buf, limit, terminator )`. Manipulator: `ws`.

Operation	In C	In C++
Skip leading whitespace, stop reading at next ws, no limit on string length: DO NOT DO THIS.	`scanf("%s", w1);`	`cin >> w1;`
Read up to first comma:	`scanf("%79[^,]", w1);`	`cin.get(w1, 80, ',');`
Read to and remove comma:	`scanf("%79[^,],", w1);`	`cin.getline(w1, 80, ',');`
Read line including `\n`:	`fgets(fin, w1, 79);`	`fin.getline(w1, 80);`
Read line excluding `\n`:	`scanf("%79[^\n]", w1);`	`cin.get(w1, 80);`
... remove the newline	`(void)getchar();`	`cin.ignore(1);`
... or		`cin >> ws;`
Allocate space for string	`malloc(1+strlen(w1));`	
... after `get()`		`new char[1+fin.gcount()];`
... after `getline()`		`new char[fin.gcount()];`

Note that the operations that use `>>` can be chained (combined in one statement) but those based on `get()` and `getline()` cannot. A single call on one of these functions is a complete statement

Using `get()` and `getline()`. Two input functions, `get()` and `getline()` are defined for class `istream` and all its derived classes. The difference is that `getline()` removes the delimiter from the stream and `get()` does not. Therefore, after using `get()` to read part of a line (up to a specified terminating character), you must remove that character from the input stream. The easiest way is to use `ignore(1)`.

After any read operation, the stream function named `gcount()` contains the number of characters actually read and removed from the stream. Saving this information (line 36 of the demonstration program at the end of this chapter) is useful when your program dynamically allocates storage for the input string, as in lines 40 and 41. The value returned by `gcount()` after using `getline()` will be one larger than the result after calling `get()` to read the same data.

Whitespace. When you are using `>>`, leading whitespace is automatically skipped. However, before reading anything with `get()` or `getline()`, whitespace must be skipped unless you *want* the whitespace in the input. Use `ws` for this purpose, not `ignore()`. This skips any whitespace that may (or may not) be in the input stream between the end of the previous input operation and the first visible keystroke on the current line. Usually, this is only one space, one tab, or one newline at the end of a prior input line. However, it could be more than one keystroke. By removing the invisible material using `ws` you are also able to remove any other invisible stuff that might be there.

3.5 Output

The C++ output stream, `cout`, was carefully defined to be compatible with the C stream, `stdout`: they write to the same output buffer. If you alternate calls on these two streams, the output will appear alternately on your screen. However, unless there is a very good reason, it is better style to stick to one set of output commands in any given program.

Simple types, pointers, and strings. The table shows alternative output format specifiers for several types. It uses the stream operator: `<<` and the stream manipulators `hex, dec`, and `endl`. Note that the string `"\n"`, the character `'\n'`, and the manipulator `endl` can all be used to end a line of output. For screen output they are equivalent. However, for file output there is one difference: the manipulator flushes the output stream; the character and string do not.

Type	Lang.	Function call.
Numeric:	C	`printf("c1= %c k1= %i n1= %hd m1= %ld x1= %g y1= %g\n", c1, k1, n1, m1, x1, y1);`
Numeric:	C++	`cout <<"c1=" <<c1 <<" k1=" <<k1 <<" n1=" <<n1 <<" m1=" <<m1 <<" x1=" <<x1 <<" y1=" <<y1 <<"\n";`
`char[]` or	C	`printf("%s...%s %s\n", word, w2, w3);`
`char*`	C++	`cout <<word <<"..." <<w2 <<" " <<w3 <<'\n';`
pointer in	C	`printf( "%p \n", px1 );`
hexadecimal	C++	`cout <<px1 <<endl;`
`ints` in	C	`printf( "%x %x \n", k1, k2 );`
hexadecimal	C++	`cout <<hex <<k1 <<' ' <<k2 <<dec <<endl;`

Output in hexadecimal notation. Manipulators are used to change the setting of a C++ output stream. When created, all streams default to output in base 10, but this can be changed by "sending" the manipulator `hex` to the stream. Once the stream is put into hex mode it stays in hex until changed back by the `dec` manipulator.

Field width, fill, and justification. This table shows how to use the formatting functions `setw()`, `setfill()` and `setf()`. You must specify field width in C++ separately for every field. However, the justification setting and the fill character stay set until changed. Changing the justification requires a separate function call; `setf()` cannot be used as part of a series of `<<` operations.

Style	How To Do It
In C, 12 columns, default justification (right).	`printf("%12i %12d\n", k1, k2);`
In C++, 12 cols, default justification (right).	`cout <<setw(12) <<k1 <<" " <<k2;`
In C, k1 in 12 cols, k2 in default width.	`printf("%12i %d\n", k1, k2);`
In C++, k1 12 cols (. fill), k2 default width	`cout <<setw(12) <<setfill('.') <<k1 <<k2 <<endl;`
In C, two variables, 12 columns, left justified.	`printf("%-12i%-12d\n", k1, k2);`
In C++, twice, 12 columns, left justified.	`cout <<left <<setw(12) <<k1 <<setw(12) <<k2 <<endl;`
In C++, 12 columns, right justified, -fill.	`cout <<right <<setw(12) <<setfill('-') <<k1 <<endl;`

Floating point style and precision. This table shows how to control precision and notation, which can be fixed point, exponential, or flexible . All of these settings remain until changed by a subsequent call.

Style		HowTo Do It
Default notation & precision (6)	C	`printf( "%g %g\n", y1, y2 );`
	C++	`cout <<y1 <<' ' <<y2 <<endl;`
Change to precision=4	C	`printf( "%.4g %.4g\n", y1, y2 );`
	C++	`cout << setprecision(4) <<y1 <<' ' <<y2 <<endl;`
Fixed point, no decimal places	C	`printf( "%.0f \n", y1 );`
	C++	`cout <<fixed <<setprecision(0) <<y1 <<endl;`
Scientific notation, default precision	C	`printf( "%e \n", y1 );`
	C++	`cout <<scientific <<y1 <<endl;`
Scientific, 4 significant digits	C	`printf( "%.4e \n", y1 );`
	C++	`cout <<scientific << setprecision(4) <<y1 <<endl;`
Return to default %g format	C++	`cout.setf( 0, ios::floatfield );`

The old notation. Older C++ compilers may not support the manipulators `fixed, scientific, right, and left`. If your compiler gives errors on these manipulators, you may need to use the older notation shown below.

Right justification:	`fout.setf(ios::right, ios::adjustfield);`
Left justification:	`fout.setf(ios::left, ios::adjustfield);`
Fixed point notation.	`fout.setf(ios::fixed, ios::floatfield);`
Scientific notation.	`fout.setf(ios::scientific, ios::floatfield);`

3.6 I/O and File Handling Demonstration

In this section, we give a program that uses many of the stream, input, and output facilities in C++. It should serve as a model for you in writing correct and robust file-handling programs. The main programis given first, followed by an input file , the corresponding output, and a header/code file pair that defines a structure named `Part` and its associated functions. Program notes are given after each file.

The main program.

```
 1  #include "Part.hpp"
 2
 3  //-------------------------------------------------------------------
 4  int main( void )
 5  {
 6      Part inventory[N];
 7      ifstream instr( "parts.in" );
 8      if ( !instr )  fatal( "Cannot open parts file %s", "parts.in" );
 9
10      cerr << "\nReading inventory from parts input file.\n";
11      int n = get_parts( instr, inventory );               // Bad style!
12      instr.close();
13
14      cerr <<n <<" parts read successfully.\n\n";
15      print_parts( cout, inventory, n );
16      bye();
17  }
18
```

- Line 7 declares and opens the input stream; line 12 closes it. It is not necessary to close files explicitly; program termination will trigger closure. However, when you finish using a file during an early phase of program execution, it is a good practice to close it.

- This is a typical main program for C++: it delegates almost all activity to functions and provides output before every major step so that the user can monitor progress and diagnose malfunction.
- An integer variable is declared in the middle of the code on line 11. This is legal in C++, but is not really a good style to follow. This variable should probably be declared before line 7.
- Note that there is no return statement at the end of the main function. This is legal in C++ and follows the style used by some experts. Use the return statement at the end of `main` if your compiler gives warnings about omitting it. (Mine does not give an error comment.)

Input. In the input file, each data set should start with a part description, terminated by a comma and followed by two integers. The program should not depend on the number of spaces before or after each numeric field. The last line of the file should end in a newline character.

```
claw hammer,    57 3 9.99
claw hammer, 3 5  10.885
long nosed pliers, 57 15 2.34
roofing nails: 1 lb, 3 173 1.55
roofing nails: 1 lb, 57 85 1.59
```

Output. Using the given input file, the output looks like this:

```
Reading inventory from parts input file.
5 parts read successfully.

claw hammer.............57     3    9.99
claw hammer.............3      5   10.89
long nosed pliers.......57    15    2.34
roofing nails: 1 lb.....3    173    1.55
roofing nails: 1 lb.....57    85    1.59

Normal termination.
```

The header file. `part.hpp`.

```
24  /* ---------------------------------------------------------------------
25  //  Header file for hardware store parts.                        Part.hpp
26  //  Created by Alice Fischer on Mon Dec 29 2003.
27  */
28  #include "tools.hpp"
29  #define BUFLENGTH 100   // Maximum length of the name of a part
30  #define N 1000          // Maximum number of parts in the inventory
31
32  struct Part{
33          char* part_name;
34          int store_code, quantity;
35          float price;
36  };
37
38  int get_parts( ifstream& fin, Part* data );
39  void print_parts( ostream& fout, Part* inventory, int n );
```

- The file `tools.hpp` includes all of the standard C and C++ header files that you are likely to need. If you include the tools source code file or the tools header file, you do not need to include the standard libraries. It also contains the line `using namespace std;`.
- Two constants are defined here because they are required by one of the two `Part` functions. The main program also uses `N`.
- In C++, you can use `struct` (line 32) without a typedef and without writing the keyword `struct` every time you use the type name. Note the syntax used in lines 6, 38, 39, etc.
- Note the ampersands in the second and third prototypes. They indicate that the stream parameters are passed by reference (not by value or by pointer). Most input and output functions take a stream parameter, and it is always a reference parameter.

The implementation file, `part.cpp`**.**

```
24  /*  ----------------------------------------------------------------------
25  //  Implementation file for hardware store parts.               Part.cpp
26  //  Created by Alice Fischer on Mon Dec 29 2003.
27  */
28  #include "Part.hpp"
29  //----------------------------------------------------------------------
30  int
31  get_parts( ifstream& fin, Part* data ) {
32      char buf[BUFLENGTH];
33      int len;            // Length of input string.
34      Part* p = data;     // Cursor to traverse data array.
35      Part* pend = p+N;   // Off-board pointer to end of array.
36
37      while (p<pend) {
38          fin >> ws;
39          if (fin.eof()) break;
40          fin.getline( buf, BUFLENGTH, ',' );
41          len = fin.gcount();
42          fin >> p->store_code >> p->quantity >> p->price;
43
44          if (fin.good()) {   // All required data items were read.
45              p->part_name = new char[len];
46              strcpy( p->part_name, buf );
47              ++p;            // Position cursor for next input.
48          }
49          else {
50              fin.clear();
51              cerr <<"Error reading line " <<(p-data+1) <<":\n"
52                   <<"    Before error: " <<buf <<" "
53                   <<p->store_code <<"  " <<p->quantity <<endl;
54              fin.getline( buf, BUFLENGTH ); // Skip rest of defective line.
55              cerr <<"    After error: " <<buf <<endl;
56          }
57      }
58      return p - data;  // Number of data lines read correctly and stored.
59  }
60
61  //------------------------------------------------------------------
62  void
63  print_parts( ostream& fout, Part* inventory, int n ){
64      Part* p = inventory;
65      Part* pend = inventory+n;
66
67      for ( ; p<pend; ++p) {
68          fout <<left  <<setw(25) <<setfill('.') <<p->part_name;
69          fout         <<setw(3)  <<setfill(' ') <<p->store_code;
70          fout <<right <<setw(5)                 <<p->quantity;
71          fout <<fixed <<setw(8)  <<setprecision(2) <<p->price <<endl;
72      }
73  }
```

Input: `get_parts`**.** This function attempts to read the input file no matter what errors it may contain. If an error is discovered while reading a data set, the entire data set is skipped and the input stream is flushed up to the next newline character. The faulty input is reported.

- We use pointers to process the data array. In C++, as in C, pointers are simpler and more efficient to use than subscripts for sequential array processing. This function uses the normal pointer paradigm for an input function:
  - Line 34 sets a local cursor, `p`, to the head of the data array and line 35 sets a stationary pointer, `pend`, to the first address that is not part of the array.

  - The processing loop (lines 37. . . 57) processes each array slot sequentially and increments the cursor (line 47) if good data is received.
  - Processing continues until the array becomes full (line 37, `while (p<pend)`) or until the end of the input file is reached. (The test is on line 39).

- Line 38 removes leading whitespace from the input stream. This is necessary before using `get()` or `getline()` in order to eliminate the newline character that terminated the previous line of input. If no whitespace is present in the stream, no harm is done. The I/O operator does remove leading whitespace, but we cannot use `>>` to read the part name because it stops reading at the first space and parts are often given names with multiple words.

- Line 39 breaks out of the input loop when end of file is encountered. This is the appropriate time to test for end of file because (a) we know there is no data, good or bad, to be processed and (b) the act of reading in all the whitespace will turn on the eof flag. This works whether or not there is a newline character on the last line of input, and whether or not the file is empty. DO NOT put your end-of-file test inside the `while` test on line 32.

- Line 40 reads characters from the input stream into the array named `buf`. Reading stops at the first comma, which is read but not stored in the input array. Instead, a null terminator is stored in the array. If a comma is found, `fin.good()` will be true. If no comma is found in the first `BUFLENGTH-1` characters, the read operation will stop and `fin.fail()` will be true.

- Line 41 sets `len` to the actual number of characters that were read and removed from the input stream. This number should be stored before doing any more input operations. After using `getline()`, it is the correct array length to use for allocating storage for the input string. (After calling `get()`, you need to add one to get the allocation length.) Lines 45 and 46 allocate storage, attach it to the `Part` array, and copy the input into it.

- Line 42 reads and stores three numbers. Whitespace before each number is automatically skipped.

- Line 44 tests whether all preceding input requests were successful. If so, we know that we have all four of the data fields needed to create a new part. Allocation was deferred until the input line was validated to avoid a problem known as a "memory leak". We want to be sure that all dynamically allocated storage is attached to the data array, so that it can be located and deallocated properly at a later time.

- Lines 49. . . 56 handle errors. Control could come here because of a hardware error or because one of the data fields is missing. In either case, error recovery has two steps:
  - Line 45 calls `fin.clear()` to reset the error flags in the stream data structure. This must be done before normal operation can resume.
  - Line 49 clears the rest of the current line out of the input stream, in an effort to resynchronize the program and data. This code assumes that each data line ends in a newline character, so skipping to the next newline will position the file at the beginning of a data item.
  - In a real application, it can be important to know what data has been processed and what has been skipped because of errors. Lines 51. . . 53 and 55 print all data that has not been processed on the `cerr` stream. This gives the user the information necessary to correct any errors.

- At any time, we can use pointer subtraction to compute the number of items that have been correctly stored in the array; simply subtract the pointer to the head of the array from the cursor. We return this value to the main program for use in all further processing of the data. There is no need to maintain a separate counter.

Output: `print_parts`. The input file was not formatted in neat columns. To make a neat table, we must be able to specify, for each column, a width that does not depend on the data printed in that column. Having done so, we must also specify whether the data will be printed at the left side or the right side of the column, and what padding character will be used to fill the space. In C, we can control field width and justification (but not the fill character) by using an appropriate format. This lets us use a single `printf()` statement to print an entire line of the table (with the space character used as filler).

C++ I/O does not use formats, but we can accomplish the same goals with stream manipulators. However, we must use a series of operations that that control each aspect of the format independently. The output loop, below, illustrates the use of the formatting controls.

- The paradigm for an array output function is simpler than an array input function: we know that `n` data items must be printed, so the print loop is executed exactly `n` times. No validity checking or eof tests are necessary.
- To make the code layout easier to understand, this function formats and prints one output value per line of code.
- To print data in neat columns, the width of each column must be set individually. Lines 66, 67, and 70, and 71 use `setw()` to control the width of the four columns.
- Right justification is the default. This is appropriate for most numbers but is unusual for alphabetic information. We set the stream to left justification (line 68) before printing the part name. In this example, the first column of numbers (`store_code`) is also printed using left justification, simply to demonstrate what this looks like. The stream is changed back to right justification for the third and fourth columns.
- The default fill character is a space, but once this is changed, it stays changed until the fill character is reset to a space. For example, line 66 sets the fill character to `'.'`, so dot fill is used for the first column. Line 67 resets the filler to `' '`, so spaces are used to fill the other three columns.
- The price is a floating point number. Since the default formatting (%g with six digits of precision) is not acceptable, we must supply both the style (fixed point) and the precision (2 decimal places) that we need. Right justification and space as a fill character remain set from lines 70 and 69, but the field width must be supplied for every field.

Bad files. If the error and end-of-file tests are made in the right order, it does not matter whether the file ends in a newline character or not. (This will be covered in detail in the next section.) The output without a final newline is the same as was shown, above, for a well formed file. The result of running the program on an empty file is:

```
Reading inventory from parts input file.
0 parts read successfully.
```

Bad data. Suppose we introduce an error into the file by deleting the price on line three. The output becomes:

```
Reading inventory from parts input file.
Error reading line 3:
    Before error: long nosed pliers 57  15
    After error: roofing nails: 1 lb, 3 173 1.55
3 parts read successfully.

claw hammer..............57     3    9.99
claw hammer..............3      5   10.89
roofing nails: 1 lb......57    85    1.59
```

The actual error was a missing float on the third line of input. The error is discovered when there is a type mismatch between the expectation (a number) and the `'r'` on the beginning of the fourth line. Because of the type mismatch, nothing is read in for the price, and the value in memory is set to 0. The faulty data is printed. Then the error recovery effort starts and wipes out the data on the line where the error was discovered. Details of the operation of the error flags are covered in the next section.

3.7 End of File and Error Handling

3.7.1 Using the command line.

Command-line arguments allow the programmer to select a data file at the time the command is given to run the program. This is especially useful when you wish to test the program with two different input files, as in the next program example. There are two ways to use file names as command line arguments:

1. Run the program from a command shell and type the file names after the name of the executable program.
2. In an integrated development environment (IDE), find a menu item that pops up a window that controls execution and enter the file names in the space provided. The location of these windows varies from one IDE to another, but the principle is always the same.

The program in this section illustrates how to use these arguments.

In main(): Command-line arguments and file handling.

1. Line 136 declares that the program expects to receive arguments from the operating system. If you have never seen command-line arguments, refer to Chapter 22 of *Applied C*, the text used in our C classes.
2. Line 140 lets us know that the user should supply two file names on the command line. Lines 142 and 144 pick up these arguments and use them to open two input streams. It is a good idea to open all files at the beginning of the program, especially if a human user will be entering data. There is nothing more frustrating that working for a while and *then* discovering that the program cannot proceed because of a file error.
3. Lines 143 and 145 test both streams to be sure they are properly open.

```
130  //------------------------------------------------------------------------------
131  // C++ demonstration program for end of file and error handling.
132  // A. Fischer, April 2001                                           file: main.cpp
133  #include "tools.hpp"
134  #include "funcs.hpp"
135
136  int main( int argc, char* argv[] )
137  {
138      banner();
139      // Command line must give the name of the program followed by 2 file names.
140      if (argc < 3)  fatal( "Usage: eof_demo textfile numberfile" );
141
142      ifstream alpha_stream( argv[1] );
143      if (! alpha_stream)  fatal( "Cannot open text file %s", argv[1] );
144      ifstream num_stream( argv[2] );
145      if (! num_stream)  fatal( "Cannot open numeric file %s", argv[2] );
146
147      cout <<"\nIOstream flags are printed after each read in order:\n"
148           <<"goodbit : eofbit : failbit : badbit :\n";
149
150      use_get( alpha_stream );
151
152      alpha_stream.close();                   // Close file.
153      ifstream new_stream ( argv[1] );    // Re-open the file to use it again.
154      if (! new_stream)  fatal( "Cannot open text file %s", argv[1] );
155
156      use_getline( new_stream );
157      use_nums( num_stream );
158  }
```

4. Line 152 closes the input stream so that we can open it again and start all over using `getline()` instead of `get()`.
5. Line 153 reopens the stream using an additional parameter that seems to be necessary in Visual C++ but is optional according to the standard and in other C++ compilers that I use.

6. This program calls three functions that illustrate end-of-file and error handling with three different kinds of read operations. The numeric input file is shown in the next section with the function that processes it.
7. End-of-file handling interacts with error handling and with the way the data file ends. Two text files were used to test this program: one with and the other without a newline character on the end of the last line. The two text files and the output generated from each are shown after the code for the two functions.

3.7.2 Reading Lines of Text

It is possible to check for errors after reading from an input stream. If this is not done, and a read error or format incompatibility occurs, both the input stream and the input variable may be left containing garbage. After detecting an input error, you must clear the bad character or characters out of the stream and reset the stream's error flags. The program in this section shows how to detect read errors and handle end-of-file and other stream exception conditions.

A function that uses `get()` to read lines of text. There is only one difference in the operation of `get()` and `getline()`: the first does not remove the terminating character from the input stream, the second does. However, this small difference affects end-of-file and error handling. This pair of examples is provided to show the differences and to provide guidance about handling them.

```
159 //==============================================================================
160 // The get function leaves the trailing \n in the input stream.
161 // A. Fischer, April 2001                                            file: get.cpp
162 //------------------------------------------------------------------------------
163 #include "tools.hpp"
164
165 void use_get( istream& instr )
166 {
167     cout <<"\nUsing get to read entire lines.\n";
168     char buf[80];
169     while (instr.good() ){
170         instr >> ws;                 // Without this line, it is an infinite loop.
171         instr.get( buf, 80 );
172         cout <<instr.rdstate() <<" = ";
173         cout <<instr.good() <<":" <<instr.eof() <<":" <<instr.fail() <<":"
174              <<instr.bad() <<": ";
175         if (!instr.fail() && !instr.bad()) cout << buf << endl;
176     }
177     if ( instr.eof() ) cout << "-----------------------------------\n" ;
178     else cout << "Low-level failure; stream corrupted.\n" ;
179 }
```

Notes on the use_get() function.

1. Line 169 demonstrates the use of an eof and error test as the `while` condition. The `good()` function tests for both eof and errors, so it is safer to use than the more common `while (!instr.eof())`.
2. Line 171 demonstrates the use of `get()`. The error flags are printed immediately after the read operation and before the line is echoed. The output shows that three read operations were performed, altogether. The third read went past the end of the file and caused the stream's eof flag to be set.
3. The good() function returns true (which is printed as 1) when none of the three exception flags are turned on. The `fail()` function returns a true result if no good data was processed on the prior operation. The `bad()` function returns true if a fatal low level IO error occurred, and the eof() function returns true when and end-or-file condition has occurred. These status bits remain set until explicitly cleared.

   As shown in the output, `eof()` can be true when good data was read, and also when no good data was read. (`fail()` is true).

4. On Line 172, we print a summary of the error conditions by writing `cout << instr.rdstate()`. The state value that is printed is the sum of the values of the stream flags that are set: eofbit=1, failbit=2, and badbit=4. Thus, when failbit and badbit are both turned on, the value of the state variable is 2 + 4 = 6.
5. Line 174 tests for the presence of good data. The test will be false when the read operation fails to bring in new data for any reason. We test it before printing or processing the data because we do not want to process old garbage from the input array. The fail flag was turned on after the fourth read operation because there was nothing left in the stream.
6. Using data or printing it without checking for input errors is taking a risk. A responsible programmer does not permit garbage input to cause garbage output. If `fail()` is true, the contents of the input variable are unreliable. If they are used after a failed get, the contents of the last correct input operation may be processed again. (This is a common problem.)
7. When using `get()` or `getline()`, the input variable may or may not be cleared to the empty string when `fail()` is true. My system does clear it, but I can find no mention of this in my reference books. It may be up to the compiler designer whether it is cleared or continues to hold the data from the last good read operation. Until this is clearly documented in reliable reference books, programmers should not depend on having the old garbage cleared out.

A function that uses getline() to read lines of text.

```
180  //===========================================================================
181  // The getline function removes the trailing \n from the stream and discards it.
182  // A. Fischer, April 2001                                         file: getline.cpp
183  //---------------------------------------------------------------------------
184  #include "tools.hpp"
185
186  void use_getline( istream& instr )
187  {
188      cout <<"\nUsing getline to read entire lines.\n";
189      char buf[80];
190      while (instr.good()) {
191          instr.getline( buf, 80 );
192          cout <<instr.rdstate() <<" = ";
193          cout << instr.good() <<":" << instr.eof() <<":" << instr.fail() <<":"
194               << instr.bad() <<": ";
195          if (!instr.fail()) cout << buf << endl;
196      }
197      cout << "-----------------------------------\n";
198  }
```

Notes on the use_getline() function.

1. Lines 186. . . 198 demonstrate the use of `getline()`. Looking at the output, you can see that the third line *was* processed, whether or not it ended in a newline character. However, the outputs are not identical. The eofbit is turned on after the third line is read if it does not end in a newline, and after the fourth read if it does. This makes it very important to read, test for eof, and process the data in the right order.
2. When using `getline()`, good data and end-of-file can happen at the same time, as shown by the last line of output on the left. Therefore, we must make the test for good data first, process good data (if it exists) second, and make the eof test third. Combining the two tests in one statement, anywhere in the loop will cause errors under some conditions.
3. In this example, we read the input, print the flags, then test whether we *have* good input before printing it. In the `use_get()` function, we tested for both kinds of errors. A less-bullet-proof alternative would be to take a risk, and not test for low-level I/O system errors. Since these are rare, we usually do not have a problem when we omit this test. Code that implements this scheme would be:

```
while(instr.good()) {
    instr.getline( buf, 80 );
    if (! instr.fail()) { Process the new data here. }
}
```

Input files for the EOF and error demo.

The file eofDemo.in:

```
First line.
Second line.
Third line, without a newline on the end.
```

The file eofDemo2.in:

```
First line.
Second line.
Third line, with a newline on the end.
```

Output from use_get(). Compare the results shown here on files without (left) and with (right) a newline character on the end of the file. In both cases, the data was read and processed correctly. This is only possible when the error indicators are checked in the "safe" order, that is, the `good()` function was called *before* checking for `eof()`. This allows us to capture the good data from the last line. If the outcome flags are checked in the wrong order, the contents of the last line may be lost. The IOstream flags are printed after each read in this order: goodbit : eofbit : failbit : badbit.

Output from use_get() reading eofDemo.in.

```
Using get to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:1:0:0: Third line, no newline on end.
-----------------------------------
```

Output from use_get() with eofDemo2.in.

```
Using get to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
0 = 1:0:0:0: Third line, with newline on end.
6 = 0:1:1:0: -----------------------------------
```

Output from use_getline() using eofDemo.in.

```
Using getline to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:1:0:0: Third line, no newline on end.
-----------------------------------
```

Output from use_getline() with eofDemo2.in.

```
Using getline to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
0 = 1:0:0:0: Third line, with newline on end.
6 = 0:1:1:0: -----------------------------------
```

3.7.3 EOF and error handling with numeric input.

Reading numeric input introduces the possibility of non-fatal errors and the need to know how to recover from them. Such errors occur during numeric input when the type of the variable receiving the data is incompatible with the nature of the input data. For example, if we attempt to read alphabetic characters into a numeric variable.

```
200  //===========================================================================
201  // Handle conflicts between input data type and expected data type like this.
202  // A. Fischer, April 2001                                     file: getnum.cpp
203  //---------------------------------------------------------------------------
204  #include "tools.hpp"
205
206  void use_nums( istream& instr )
207  {
208      cout <<"\nReading numbers.\n";
209      int number;
210      for(;;) {
211          instr >> number;
212          cout <<instr.good() <<":" <<instr.eof()<<":"<<instr.fail()<<":"<<instr.bad()<<": ";
213          if (instr.good()) cout << number << endl;
214          else if (instr.eof() ) break;
215          else if (instr.fail()) {       // Without these three lines
216             instr.clear();              // an alphabetic character in the input
217             instr.ignore(1);            // stream causes an infinite loop.
218          }
219          else if (instr.bad())          // Abort after an unrecoverable stream error.
220              fatal( "Bad error while reading input stream." );
221      }
222      cout << "-----------------------------------\n" ;
223  }
```

1. Lines 200. . . 221 show how to deal with random numeric input errors. The file errDemo.in has three fully correct lines surrounding one line that has erroneous letters on it.

The file errDemo.in:

```
1
278
45abc
6
```

Output from get_nums() using errDemo.in:

```
Reading numbers.
1:0:0:0: 1
1:0:0:0: 278
1:0:0:0: 45
0:0:1:0: 0:0:1:0: 0:0:1:0: 1:0:0:0: 6
0:1:1:0: ---------------------------------
```

2. The first three numbers are read correctly, and the "abc" is left in the stream, unread, after the third read. When the program tries to read the fourth number, a conversion error occurs because 'a' is not a base-10 digit. The failbit is turned on but the eofbit is not, so control passes through line 214 to line 215, which detects the conversion error.
3. Lines 216 and 217 recover from this error. First, the stream's error flags are cleared, putting the stream back into the `good` state. Then a single character is read and discarded. Control goes back to line 210, the top of the read loop, and we try again to read the fourth number.
4. We cleared only one keystroke from the stream, leaving the "bc". So another read error occurs. In fact, we go through the error detection and recovery process three times, once for each non-numeric input character. On the fourth recovery attempt, a number is read, converted, and stored in the variable `number`, and the program goes on to normal completion. Compare the input file to the output: the "abc" just "disappeared", but you can see there were three "failed" attempts to read the 6.
5. Because of the complex logic required to detect and recover from format errors, we cannot use the eof test as part of a while loop. The only reasonable way to handle this logic is to use a blank `for(;;)` loop and write the required tests as `if` statements in the body of the loop. Note the use of the `if...break`; please learn to write loops this way.

3.8 Assorted short notes.

Reading hexadecimal data. The line `45abc` is a legitimate hexadecimal number, but not a correct base-10 number. Using the same program, we could read the same file correctly if we wrote line 211 as:

`instr >> hex >> number;` instead of `instr >> number;`

In this case, all input numbers would be interpreted as hexadecimal and the characters "abcdef" would be legitimate digits. The output shown below is still given in base-10 because we did not write `cout<<hex` on line 199. The results are:

```
Reading numbers.
1:0:0:0: 1
1:0:0:0: 632
1:0:0:0: 285372
1:0:0:0: 6
0:1:2:0: ---------------------------------
```

Appending to a file. A C++ output stream can be opened in the declaration. When this is done, the previous contents of that file are discarded. To open a stream in append mode, you must use the explicit `open` function, thus:

```
ofstream fout;
fout.open( "mycollection.out",  ios::out | ios::app );
```

The mode `ios::out` is the default for an ofstream and is used if no mode is specified explicitly. Here, we are specifying append mode, "app", so we also be write the "out" explicitly.

Reading and writing binary files. The `open` function can also be used with binary input and output files:

```
ifstream bin;
ofstream bout;
bin.open( "pixels.in", ios::in | ios::binary );
bout.open( "pixels.out",  ios::out | ios::binary );
```

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`	`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 ’)’`
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 );
}
```

```
66
67  //-----------------------------------------------------------------------------
68  void mismatch(string msg, int lineno, Stack* stkp, Token* tokp )
69  {
70      if (tokp == NULL)  printf("Mismatch at end of file: %s\n", msg);
71      else  printf("Mismatch on line %d: %s\n", lineno, msg);
72
73      print( stkp ); printf( "\n\n" );        // print stack contents
74      if (tokp != NULL)                       // print current token, if any
75          printf("The current mismatching bracket is '%c'\n\n", tokp->ch );
76
77      exit(1);                                // abort further processing
78  }
```

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.

```
  99  // ==========================================================================
 100  // Project: Bracket-matching example of stack usage                File: token.c
 101  // ==========================================================================
 102  #include "token.h"
 103
 104  Token classify( char ch )
 105  {
 106      static const string brackets = "[](){}<>";
 107      Token tok;
 108      tok.ch = ch;
 109
 110      char* p = strchr( brackets, ch );
 111      if (p==NULL) {
 112          tok.typ = BKT_NONE;
 113          tok.sense = SENSE_LEFT;          // arbitrary value
 114      }
 115      else {
 116          int pos = p-brackets;            // pointer difference gives subscript.
 117          tok.sense = (pos%2 == 0) ? SENSE_LEFT : SENSE_RIGHT;
 118          tok.typ = (pos/2);               // integer arithmetic, with truncation.
 119      }
 120      return tok;
 121  }
```

```
1131  // =========================================================================
1132  // Name: Bracket-matching example of stack usage            File: token.cpp
1133  // =========================================================================
1134  #include "token.hpp"
1135  //-------------------------------------------------------------------------
1136  Token::Token( char ch ){
1137      this->ch = ch;
1138      classify( ch );
1139  }
1140
1141  //-------------------------------------------------------------------------
1142  void Token::
1143  classify( char ch )
1144  {
1145      static const cstring brackets = "[](){}<>";
1146      char* p = strchr( brackets, ch );
1147      if (p==NULL) {
1148          type = BKT_NONE;
1149          sense = SENSE_LEFT;             // arbitrary value
1150      }
1151      else {
1152          int pos = p-brackets;           // pointer difference gives subscript.
1153          sense = (pos % 2 == 0) ? SENSE_LEFT : SENSE_RIGHT;
1154          type = (BracketType)(pos/2);    // integer arithmetic, with truncation.
1155      }
1156  }
1157
```

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; }
```

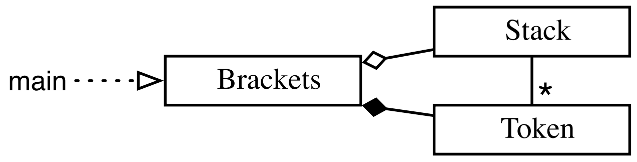


Figure 4.3: UML class diagram for the Brackets progam.

Chapter 5: Functions and Parameter Passing

In this chapter, we examine the difference between function calls in C and C++ and the resulting difference in the way functions are defined in the two languages. Finally, we illustrate the semantics of the three methods of passing a parameter in C++ and the three kinds of function return values. You should use the demo program and its output as reference material when you are uncertain about how to use the various parameter and return types.

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it.
— Samuel Johnson, English, author of the Dictionary of the English Language.

5.1 Function Calls

The implied argument. In C++, every call to a class function has an "implied argument", which is the object whose name is written before the function name. The rest of the arguments are written inside the parentheses that follow the function name. Additional explicit arguments are written inside parentheses, as in C. Compare this C++ code fragment to the C code fragment below:

```
if ( in.eof() )...
stk->push(curtok);
```

The implied argument is called `this`. In contrast, in C, all arguments are written inside the parentheses. In the stack.c example, the end-of-file test and the call on `push()` are written like this:

```
if (feof( in ))...
push(&stk, curtok);
```

Number of parameters. Function calls in C++ tend to be shorter and easier to write than in C. In C, functions often have several parameters, and functions with no parameters are rare. In the `stack.c` example, note that the call on `push()` has two arguments in C and one in C++. The difference is the stack object itself, which must be passed by address in C and is the *implied argument* in C++.

Functions with empty or very short parameter lists are common in C++. First, the implied argument itself is not writtten as part of the argument list. Also, classes are used to group together related data, and that data is available for use by class functions. We often eliminate parameters by using class data members to store information that is created by one class function and used by others.

Code organization. When writing in a procedural language, many programmers think sequentially: do this first, this second, and so on. Function definitions often reflect this pattern of thought Thus, in C, functions tend to be organized by the sequence in which things should be done.

In an object-oriented language, the code is not arranged sequentially. Thus, in C++, functions are part of the class of the objects they work on. Control passes constantly into and out of classes; consecutive actions are likely to be in separate functions, in separate classes with a part-whole relationship. One object passes control to the class of one of its components, which does part of the required action and passes on the rest of the responsibillity to one of its own components.

The number and length of functions. There are many more functions in a C++ program than in a C program and C++ functions tend to be very short. Each C++ function operates on a class object. The actions it performs directly should all be related to that object and to the purpose of that class within the application. Actions related to class components are *delegated* (passed on) to the components to execute. Thus, long functions with nested control structures are unusual, while one-line functions and one-line loops are common. Inline expansion is used to improve the efficiency of this scheme.

Non-class functions. Functions can be defined inside or outside classes in C++ . For those defined outside a class, the syntax for definitions and calls is the same as in C. Non-class functions in C++ are used primarily to modularize the main program, to extend the input and output operators, and to permit use of certain C library functions.

5.2 Parameter Passing

C supports two ways to pass a parameter to a function: call-by-value and call-by-pointer. C++ supports a third parameter-passing mechanism: call-by-reference. The purpose of this section is to demonstrate how the three parameter-passing mechanisms work and to help you understand which to use, when, and why.

5.2.1 Three Odd Functions

These three functions do not compute anything sensible or useful; their purpose is to illustrate how the three parameter-passing mechanisms are similar and/or different. Each function computes the average of its two parameters, then prints the parameter values and the average. Finally, both parameters are incremented. The computation works the same way in all versions. However, the increment operator does not work uniformly: sometimes a pointer is incremented, sometimes an integer; sometimes the change is local, sometimes not.

Call by value. The argument values are copied into the parameter storage locations in the function's stack frame. There is no way for the function to change values in main's stack frame.

```
#include "odds.hpp"                          // Includes necessary libraries.

void odd1( int aa, int bb ){                 // Make copies of the argument values.
    int ansr = (aa + bb) / 2;                // See diagram 1.
    cout <<"\nIn odd1, average of " <<aa <<", " <<bb <<" = " <<ansr <<endl;
    ++aa;  ++bb;                             // Increment the two local integers.
}
```

Call by pointer. The arguments must be addresses of integer variables in the caller's stack frame. These addresses are stored in the parameter locations in the function's stack frame. They permit the function to modify its caller's memory locations.

```
#include "odds.hpp"                          // Includes necessary libraries.

void odd2( int* aa, int* bb ){               // Call by address or pointer.
    int ansr = (*aa + *bb) / 2;              // See diagram 2.
    cout << "\nIn odd2, average of " <<*aa <<", " <<*bb <<" = " <<ansr <<endl;
    ++aa;                                    // increment the local pointer
    ++*bb;                                   // increment main's integer indirectly
}
```

Call by reference. The arguments must be integer variables in the caller's stack frame. The addresses of these variables are stored in the parameter locations in the function's stack frame, permitting the function to modify its caller's memory locations.

```
#include "odds.hpp"                          // Includes necessary libraries.

void odd4( int& aa, int& bb ){               // Call by reference
    int ansr = (aa + bb) / 2;                // See diagram 3.
    cout << "\nIn odd4, average of " <<aa <<", " <<bb <<" = " <<ansr <<endl;
    ++aa;  ++bb;                             // increment two integers in main.
}
```

5.2.2 Calling The Odd Functions

This main program calls the three odd functions with arguments selected from the array named `ar`. It prints the whole array before and after each call so that you can see any changes that take place. Each call (and its results) is explained in the paragraphs that follow and stack diagrams are given to help you understand how the system works.

```
//-----------------------------------------------------------------------
// Calling the odd average functions.                       file: oddM.cpp
//-----------------------------------------------------------------------
#include "odds.hpp"                // Contains the three odd functions.

void print( int* ap, int n ) {      //  Print the n values in an array.
    for( int k=0; k<n; ++k )  cout << "    [" << k << "] = " << ap[k] ;
    cout << endl;
}
//-----------------------------------------------------------------------
int main( void )
{
    int  ar[6] = {11, 22, 33, 44, 55, 66};
    int* ip = ar;       // Set a pointer to beginning of an array.
    int* iq = &ar[2];   // Set a pointer to 3rd item of array.

    cout << "\nInitially, ar is: -----------------------------------\n";
    print( ar, 6 );
    odd1( ar[0], *iq );     print( ar, 6 );     // See Figure 5.1.
    odd2( ip, iq );         print( ar, 6 );     // See Figure 5.2.
    odd2( &ar[1], &ar[4] ); print( ar, 6 );     // See Figure 5.3.
    odd4( ar[2], *(iq+2) ); print( ar, 6 );     // See Figure 5.4.
}
```

The output.

```
Initially, ar is: -----------------------------------
    [0] = 11    [1] = 22    [2] = 33    [3] = 44    [4] = 55    [5] = 66

In odd1, average of 11, 33 = 22
    [0] = 11    [1] = 22    [2] = 33    [3] = 44    [4] = 55    [5] = 66

In odd2, average of 11, 33 = 22
    [0] = 11    [1] = 22    [2] = 34    [3] = 44    [4] = 55    [5] = 66

In odd2, average of 22, 55 = 38
    [0] = 11    [1] = 22    [2] = 34    [3] = 44    [4] = 56    [5] = 66

In odd4, average of 34, 56 = 45
    [0] = 11    [1] = 22    [2] = 35    [3] = 44    [4] = 57    [5] = 66
```

5.2.3 Parameter Passing Mechanisms

In these diagrams, a round-cornered box represents a *stack frame* or *activation record* on the system's run-time stack. A stack frame is the data structure that is built by the run-time system to implement a function call. Within the box, the heavy dot-dashed line divides the local variables (above it) from the parameters (if any, below it). The leftmost column gives the name of the function and the return value, if any. The middle two columns supply the type, name, and address of each object allocated for the function. The rightmost column lists the current value, which is a literal number, a pointer (arrow with one stem), or a binding (arrow with a double stem).

Call by value is like making a Xerox copy. In C, there is only one way to pass a non-array argument to a function: call by value (Figure 5.1). In call by value, the value of the argument (in the caller) is shallowly copied into the parameter variable within the function's stack frame. This parameter passing method isolates the function from the caller; all references to this parameter are strictly local. . . the function can neither see nor change the caller's variables.

If the function increments the parameter variable or assigns a new value to it, the change is purely local and does not affect variables belonging to the caller. In this example, `aa` is a call-by-value parameter, so executing `++aa`, changes its value locally but does not affect `main's` variable `ar[0]`. After the function returns, `ar[0]` still contains `11`.

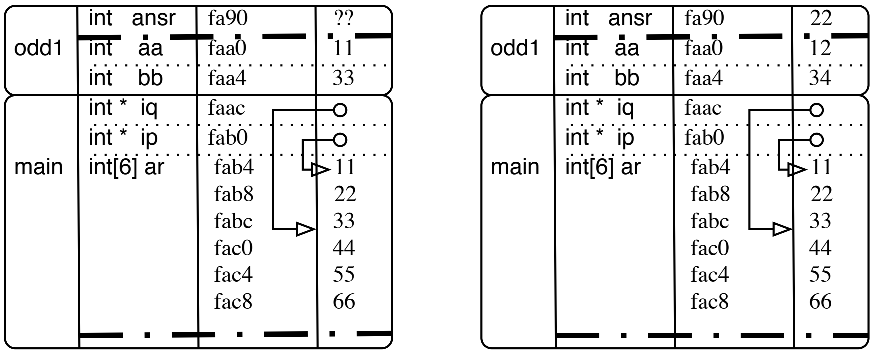


Figure 5.1: Execution of `odd1` after computing `ansr` (left) and after increments (right).

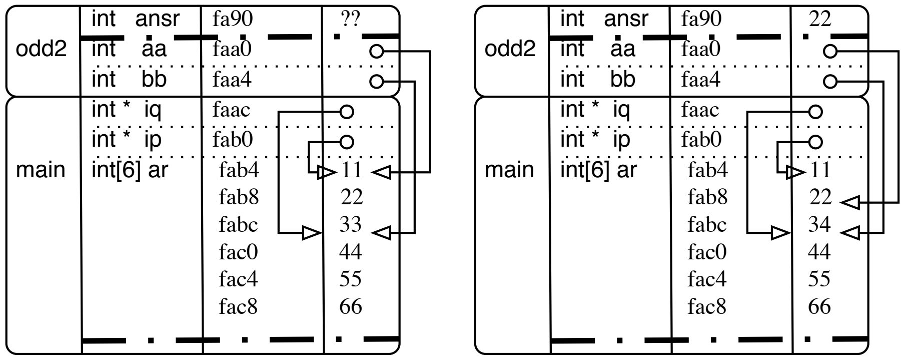


Figure 5.2: Execution of `odd2` after computing `ansr` (left) and after increments (right).

Call by pointer or address. Even in a call-by-value language, an argument that is an address or the value of a pointer can be used with a pointer parameter to give the function access to the underlying variable in the stack frame of the caller. The parameter can be used two ways within the function. By using the parameter name wiithout a `*`, the address stored locally in the parameter variable can be used or changed. By using the parameter name with a `*`, the value that is stored in the caller's variable can be used or changed.

Call by value with a pointer argument. The first call on `odd2` is diagrammed in Figure 5.2. The arguments are the values of two pointers. When these arguments are stored in the parameter locations, the parameters point at the same locations as the pointers in `main()`. Writing `++aa` increments the local value of the pointer `aa`, which changes the array slot that the parameter points to (initially it was ar[1], after the increment it is ar[2]). Writing `++*bb` changes the integer in the caller's array slot.

Call by value with an `&` argument. The second call on `odd2` is shown in Figure 5.3. The effect is the same as the prior call; The arguments are the addresses of two slots in caller's array and are diagrammed as simple arrows. Writing `++aa` changes the array slot that the parameter points to (initially it was ar[1], after the increment it is ar[2]). Writing `++*bb` changes the integer in the caller's array slot.

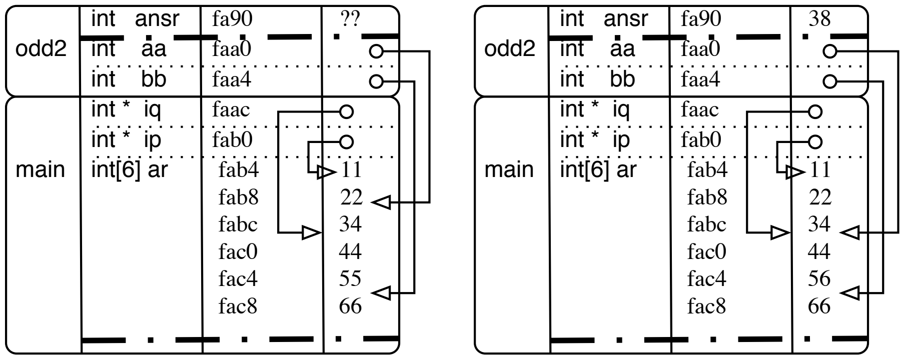


Figure 5.3: Execution of `odd2` after computing `ansr` (left) and after increments (right).

Call by reference (`&`). This form of parameter passing is normally used when the argument is or can be an object that is partly created by dynamic storage allocation.In call-by-reference (Figure 5.4), the parameter value is called a “binding” and is diagrammed as a shaded double arrow.

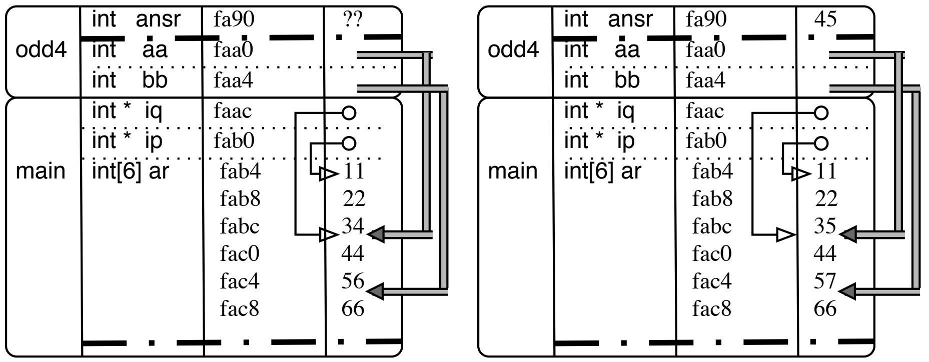


Figure 5.4: Execution of `odd4` after computing `ansr` (left) and after increments (right).

The value stored locally in the parameter cannot be changed because the parameter name becomes transparent: it is just an *alias*, or a synonym, for the caller’s variable. The program cannot read or change the address stored in the parameter. For example, line 29, using `++` changes the caller’s value, not the address stored locally in the parameter .

Call by reference vs. value. When call by value is used, the function cannot change values in the caller’s stack frame. Any reference to the parameter name is a reference to the local copy, and any assignment to the parameter changes only the local copy.

In contrast, when call by reference is used, the parameter name is an alias for the argument variable. Even though this is implemented by storing the address of the caller’s variable in the function’s stack frame, it does not function like a pointer. The dereference operator is not used because the dereference action is automatic. Any reference to or assignment to the parameter name IS a reference or assignment to the variable in the caller’s stack frame.

Call by reference vs. pointer. When call by pointer is used, the function can dereference the parameter and assign a new value to the argument variable in caller’s stack frame. Or, the function can change the address stored in the parameter and make it refer to a different variable.

In contrast, when call by reference is used, there is no choice; because the parameter name is an alias for the argument variable. It is not possible to change the address in the local stack frame. Also, syntactically, the parameter is not a pointer and cannot be dereferenced.

5.3 Function Returns

5.3.1 L-values and r-values.

An *l-value* is a value that can be used on the left side of an assignment operator; it is the location into which something will be stored. An *r-value* is a value that can only be used on the right side of an assignment operator; it is the value that will be stored. In the example below, the function `ret0()` returns an integer r-value, which we sometimes refer to as a “pure value”.

A reference (including one that was returned by a function) can be used directly as an l-value. When used as an r-value (on the right side of `=`) a reference is treated the same way that a variable name would be treated in that context: the value at that address is fetched from memory. In the example below, `ret3()` and `ret4()` return l-values.

A pointer is an r-value that can be dereferenced. If a function returns a pointer, the function’s result must be dereferenced before it can be used on the left side of `=`. In the example below, `ret1()` and `ret2()` return pointer r-values that can be dereferenced to get l-values. A dereferenced pointer can also be used as an r-value; when that is done, it will be automatically dereferenced again to get the underlying r-value.

5.3.2 Using L-values.

The following five functions all return a different flavor of int:

```
//----------------------------------------------------------------------------------
// Return by value, reference, and pointer, and how they differ.
// A. Fischer Apr 19, 1998                                         file: returns.cpp
//----------------------------------------------------------------------------------
int         ret0( int* ar ) { return  ar[0]; }    // Returns an integer R-value.
int*        ret1( int* ar ) { return &ar[1]; }    // A pointer R-value.
const int*  ret2( int* ar ) { return &ar[2]; }    // A read-only pointer R-value.
int&        ret3( int* ar ) { return  ar[3]; }    // An L-value (reference).
const int&  ret4( int* ar ) { return  ar[4]; }    // A read-only L-value reference.
```

Functions `ret1` through `ret4` all give the caller access to some variable selected by the function. Two of the return types provide read-only access; `ret2` returns a `const int*` and `ret4` returns a `const int&` . These return types are "safe", that is, they do not break encapsulation. They are often used in OO programming. In contrast, the other two return types, `int*` and `int&` can break encapsulation and give the caller a way to change a private variable belonging to the function's class. For this reason, returning non-const address and pointers is avoided whenever possible in C++.

In this example, functions `ret1()...ret4()` all return the address of one slot in the parameter array; the function simply selects `which` slot will be used. In such a situation, all these calls are "safe" and conform to OO principles, since the function is not giving the caller access to anything that is not already accessible to it. However, by using the same return mechanisms, a function could cause two serious kinds of trouble.

Violating privacy. Suppose the return value is the address of (or a pointer to) a private class variable. This breaks the protection of the class because it permits a caller to modify a class variable in an arbitrary way. Normally, non-class functions should not be permitted to change the value of class objects[1]. All changes to class members should be made by class functions that can ensure that the overall state of the class object is always valid and consistent. (The most important exception to this rule is extensions of the subscript operator.) For example, suppose we added these two functions to the stack class from Chapter 4:

```
int& how_full(){ return top; }
char* where_is_it(){ return s; }
```

A caller could destroy the integrity of a stack, St, by doing either of the following assignments:

```
St.how_full() = 0;
*St.where_is_it() = 'X';
```

Dangling pointers. A different problem would occur if a function returned the address of or a pointer to one of its local variables. At return time, the function's local variables will be deallocated, so the return-value will refer to an object that no longer exists. The resulting problems may occur immediately or may be delayed, but this is always a serious error.

5.3.3 Using Return Values in Assignment Statements

The following program calls each of the `ret` functions and assigns uses its return value to select one integer from `ar` array. This integer is saved in a variable and later printed. Four of the functions return the location of one slot in the `ar` array. Two of these slots are `const`, that is, read-only. The other two are non-const and we use them to modify the array.

```
//------------------------------------------------------------------------
// Calling functions with the five kinds of function-return values.
// A. Fischer Apr 19, 1998                                  file: retM.cpp
//------------------------------------------------------------------------
```

[1]There are a few exceptions, for example, when the subscript function is extended for new kinds of arrays, it returns a non-const reference because that is the entire purpose of subscript.

```
14  #include <iostream>
15  using namespace std;
16  #include "returns.cpp"
17
18  #define DUMPv(k) "\n" <<"    " #k " @ " <<&k <<"    value = " <<k
19  #define DUMPp(p) "\n" <<"    " #p " @ " <<&p <<"    value = " <<p \
20                        <<"    " #p " --> " << dec << *p
21  //----------------------------------------------------------------------
22  void print5( int* ap ) {            // Print the five values in an array.
23      for( int k=0; k<5; ++k )  cout << "    [" << k << "] = " << ap[k] ;
24  }
25
26  //----------------------------------------------------------------------
27  int main( void )
28  {
29      int ar[5] = {11, 22, 33, 44, 55};
30      int h, j, k, m, n;
31      int *p = ar;
32
33      cout << "Initially, variables are: ------------------------------\n";
34      print5( ar );
35      cout <<DUMPp(p);
36
37      h  =  ret0( ar );   // Answer should be 11
38      j  = *ret1( ar );   // Answer should be 22
39      k  = *ret2( ar );   // Answer should be 33
40      m  =  ret3( ar );   // Answer should be 44
41      n  =  ret4( ar );   // Answer should be 55
42      cout <<DUMPv(h) <<DUMPv(j) <<DUMPv(k) <<DUMPv(m) <<DUMPv(n) <<endl;
43
44      p = ret1( ar );     // Answer should be a pointer to ar[1].
45      *ret1( ar ) = 17;   // Storing through the pointer.
46      //*ret2( ar ) = 17; // Illegal: read-only location.
47      ret3( ar ) = -2;    // Assigning to the reference.
48      //ret4( ar ) = -2;  // Illegal: read-only location.
49
50      cout << "\nAfter return from functions variables are: -----------\n";
51      print5( ar );
52      cout <<DUMPp(p) <<endl;
53  }
```

The Dump macros. A macro lets us define a shorthand notation for any string of characters. We use macros when the same thing must be written several times, and that thing is long and error-prone to write. In this program, we want to be able to print out information that will be useful for understanding the program. We define one macro, `DUMPv` that will print the address and contents of its parameter, and a second, `DUMPp`, that prints the address of a pointer variable, the address stored in it, and the value of its referent.

A macro differs from an inline function in several ways. A very important difference is that macro parameters are not typed, and macro calls are not type checked. Thus, the macro defined here can be used to dump any object for which `<<` is defined.

The main function. We define an array, initialize it to easily-recognized values, and print it before and after calling the `ret` functions. By comparing the lines at the beginning and end of the output, you can verify that lines 45 and 46 changed the contents of the array. Calls on the `DUMP` macros in lines 35, 42, and 50 allow you to trace the effects of each function call.

The output is:

```
Initially, variables are: ------------------------------
    [0] = 11    [1] = 22    [2] = 33    [3] = 44    [4] = 55
    p @ 0xbffffd40    value = 0xbffffd18    p --> 11
```

```
        h @ 0xbffffd2c    value = 11
        j @ 0xbffffd30    value = 22
        k @ 0xbffffd34    value = 33
        m @ 0xbffffd38    value = 44
        n @ 0xbffffd3c    value = 55

    After return from functions variables are: -----------
        [0] = 11    [1] = 17    [2] = 33    [3] = -2    [4] = 55
        p @ 0xbffffd40    value = 0xbffffd1c    p --> 17
```

Return by value. This is the commonest and safest kind of function return; the returned value is an r-value. (An r-value can be used on the right side of an assignment but not on the left.) Among the functions above, `ret0()` returns an integer by value. The sample program calls `ret0()` (line 37) and assigns the result to an integer variable.

Return by pointer. The return value of `ret1()` is a non-const pointer r-value (the address of ar[1]) that can be stored in a pointer variable or dereferenced to get an l-value. It is called three times:

- On line 38, the result is explicitly dereferenced to get an integer l-value, then implicitly dereferenced again to get an integer r-value, which is stored in the variable `j`.

  ```
   j = *ret1( ar );    // Copy the referent of the return value.
  ```

- On line 44, the result is implicitly dereferenced to get a pointer r-value, which is stored in the pointer variable `p`.

  ```
  p = ret1( ar );     // Store the return value in a pointer variable.
  ```

- On line 45, the result is on the left side of an assignment. It is explicitly dereferenced to get an integer l-value, which is used to store an integer constant.

  ```
  *ret1( ar ) = 17;   // Store through the pointer into its referent.
  ```

Return by const*. The return value is an address that provides read-only access to the underlying variable. The function result cannot be used on the left side of an assignment to change the underlying variable. In the sample program, `ret2()` is called once (line 39). Its result is dereferenced to get an integer l-value, which is automatically dereferenced again because it is on the right side of an assignment statement. The result is an `int` r-value, which is stored in `k`:

```
k = *ret2( ar );    // Copy referent of return value into k.
```

Return by reference. A reference can be used in the same ways that a variable name is used. The function `ret3()` returns an integer reference: an l-value that can be used (without *) on either side of an assignment operator. This function is called twice and used both as an r-value (line 40) and as an l-value (line 46):

```
m = ret3( ar );     // Coerce function result to r-value and store.
ret3( ar ) = -2;    // Store -2 in address returned by function.
```

Return by const reference. The function `ret4()` returns a const int`&` (a reference to a const int). This result can be only be used for read-access, because the underlying variable is a constant. `ret4()` is called once (line 41) and used with implicit dereference on the right side of an assignment statement:

```
n = ret4( ar );    // Copy referent of return value into n.
```

Illegal calls. The following lines won't compile. For both, my compiler gives the error comment *assignment of read-only location.*

```
*ret2( ar ) = -5;    // return by const *
ret4( ar ) = 25;     // return by const &
```

Chapter 6: Objects, Allocation, and Disasters

By analyzing errors, we can learn how to make programs function well:

> **Knowledge rests not upon truth alone, but upon error also.**
> —Carl Jung, one of the fathers of modern psychotherapy.

6.1 Objects: Static, Automatic, Dynamic, and Mixed

6.1.1 Storage Class

- Auto objects are created by variable and parameter declarations. An object is allocated in the stack frame for the current block when a declaration is executed and deallocated when control leaves that block.
- Static objects are created by variable declarations with the "static" qualifier. These are allocated and initialized at load time and exist until the program terminates. They are only visible within the function block that declares them. The word "static" comes from "stay". Static variables stay allocated and stay in the same memory address, even while control leaves and reenters their defining block many times.
- Dynamic objects are created by explicit calls on `new`. They continue to exist until they are deleted or until the program ends. All objects that contain dynamic memory also have an auto or static portion (the core of the object) which provides access to the dynamic portion or portions (the extensions of the object).

It is common for a single object to have members of two or three storage classes. In Figure 6.1.1, the class declaration creates a simple class that is partly auto and partly dynamic, with an appropriate constructor and destructor. The diagram to the right of the declaration shows the object that would be created by the declaration `Mixed A( "Joe" )`. The core portion is shown against a shaded background, the extensions against a white background.

```
class Mixed {
    int len;
    char* name;
public:
    Mixed( char* nm ) {
        len = strlen( nm );
        name = new char[ len+1 ];
        strcpy( name, nm );
    }
    ~Mixed() { delete[] name; }
}
```

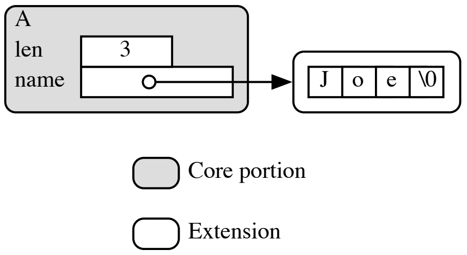


Figure 6.1: An object built partly of auto storage and partly of dynamic storage.

6.1.2 Assignment and Copying

The `=` operator is implicitly defined for all basic C++ types and for all types that the programmer might define. If `A` and `B` are of the same type, then `B = A;` is always defined and (by default) does a shallow copy.

- Shallow copy (left side of Figure 6.1.2): a component-wise copy of the non-dynamic portion of the object. This kind of copy rule is used to implement call-by-value, unless a different copying rule is explicitly defined.

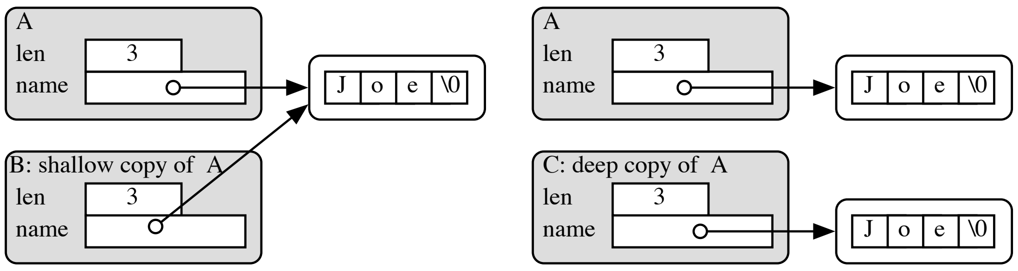


Figure 6.2: Shallow and deep copies. In the shallow copy, two objects share the same extension.

- Deep copy (right side of Figure 6.1.2): a component-wise copy of the `auto` portion of the object, except for fields that are pointers to dynamically allocated parts. These members are set to point at newly allocated and initialized replicas of the object's extensions. This is never predefined; a function must be written for the class that allocates, initializes, and copies parts appropriately, one at a time.

Copying problems. When an object, `A`, is shallow-copied into another object, `B`, the two objects share memory and a variety of nasty interactions can happen:

- If, before the copy, a member of `B` is a pointer to an extension, there will be a memory leak because that pointer will be overwritten by the corresponding member of `A`
- Changes to the data in the extensions of either object affect both.
- Deleting an extension of either object will damage the other. Ultimately, when destructors are run on the two objects, the first will delete properly, the second will cause a double-deletion error.

Because of these potential problems, shallow copy must be used carefully in C++. On the other hand, making a deep copy of a large data structure consumes both time and space; so making many deep copies can kill a program's performance.

6.2 Static Class Members

Static function members. The purpose of a static class function is to allow a client to use the expertise of a class before the first object of that class exists. This is often useful for giving instructions and for validation of the data necessary to create an object.

A static function is called using the name of its class and `::`. For example, suppose a class named Graph has two static functions, `instructions()` and `bool valid( int n )`. Calls on these static functions might look like this:

```
Graph::instructions();
if (Graph::valid( 15 )) ...
```

Since a static class function must be usable before the first class object is created, the code within the function cannot use any of the data members of that class. It can use parameters and global constants, however.

Static data members. The purpose of a static class variable is perfectly simple: it is used to communicate among or help define all class instances. A static data member is a shared variable that is logically part of all instances of the class (and all instances of all derived classes).

For example, consider a dynamically allocated matrix whose size is not known until run time. It is important to create all matrix rows the same length, but that length cannot be set using `#define`. A solution is to declare the array length as a private static class member. Then it can be set at run time, but it is visible only within the class. It can be used to construct all class instances, but only occupies space once, not space in every instance.

Sometimes a set of constants is associated with a class, for example, a Month class might have three associated constant arrays, an array of valid month abbreviations or names, and the number of days in each month. Since these are constant, it is undesirable to keep a copy inside every Month object, and undesirable to keep creating and initializing local arrays. The ideal solution is to make a set of shared (static) arrays within the class.

Declaration. To declare a static class member, you simply write "static" before the member name in the class declaration. Since the variable must be created at load time, it cannot be initialized by the class constructor. Also, since a static data member is shared by many class instances, it cannot be created by any one of them. Therefore, it is allocated and initialized at load time. The C++ designers (in their great wisdom) decided that static class members must be declared outside the class as well as within and initialized outside the class (unless they are integer variables or enum variables and are also const).

Here are some declarations that could be in a Month class. Since these are all arrays, we are not permitted to initialize them within the class declaration. The int constant can be initialized here.

```
// Subscript 1 corresponds to January.  Subscript 0 is unused.
static const int miy = 12;       // Number of months in a year.
static const int dim[13] ;       // Number of days in each month.
static char* mAbbrev[13];        // 3-letter abbreviations of the names.
```

Creation and initialization. Static data members are created and initialized at program-load time. Because of this, a separate statement must be written, outside the class definition, to declare and (optionally) initialize each static class member. The initializer, if present, may depend only on data that is known at compile time. Note that the defining declaration *does* contain the class name and *does not* contain the word "static":

```
int Matrix::columns;      // Will be initialized after execution begins.
int Customer::count = 0;  // Run time counter for instances of Customer.

const int Month::miy;     // Initialized within the class; not needed here.
const int Month::dim[13]  = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
char* Month::mAbbrev[13]  = {"---","Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
```

The defining declaration for a static data member must be compiled exactly once in all the modules that comprise the program. Since header files are often included by more than one module, they are not an appropriate place for the defining declaration. modules that comprise the program. If a class has a .cpp le, the static initialization can be written at the head of that file. This is good because it keeps the information about class members in the class own le. However, some classes are entirely contained within the .hpp le and have no corresponding .cpp le. For such a class, we could make a le just for one line, or we could put the static initialization just before the beginning of the main program. This placement guarantees that the dening declaration will be included exactly once in the completed, linked program but it separates an important fact (the initial value) from the denition of the class.

If a class has a `.cpp` file, the static initialization can be written at the head of that file. This is good because it keeps the information about class members in the class' own file. However, some classes are entirely contained within the `.hpp` file and have no corresponding `.cpp` file. For such a class, we could make a file just for one line, or we could put the static initialization just before the beginning of the main program. This placement guarantees that the defining declaration will be included exactly once in the completed, linked program but it separates an important fact (the initial value) from the definition of the class.

Usage. Static does not mean the same thing as const; a static data member is used in the same way that a non-static class variable would be used. Above, we showed declarations for two static variables. In the Matrix class, the variable named `columns` will be initialized by the Matrix constructor and used to create all the rows of the Matrix. In the Customer class, the counter will be incremented every time a Customer is created and decremented every time one is deallocated. There is nothing in the code that uses a static member to indicate that it is static or special in any way.

6.3 Common Kinds of Failure

1. **Memory leak.** C++ does not have an automatic garbage collector; it is the programmer's responsibility to deallocate dynamic storage when it is no longer useful. If the last pointer to a dynamic object is lost before it is deleted, we have a "memory leak". The memory locations occupied by that object are not recycled and are therefore no longer available to the program. Some programs run continuously for days or weeks. In such a program, a gradual memory leak can cause system performance to slowly degrade. When the swap space or paging space is full, the program will hang.

   In a properly working system, memory leaks are recovered when the program ends. However, some common current systems do not free memory properly, and the memory is not recovered until someone reboots the system.

2. **Amnesia.** You allocated and initialized dynamic memory correctly. Immediately after doing so, the information stored there was correct and complete. However, later, part or all of that information is gone, even though you have not stored anything in the object since it was created.
3. **Bus error.** Your program crashes because you have tried to use a memory address that does not exist on your computer.
4. **Segmentation Fault.** This happens on multiprogrammed systems with memory protection. Your program crashes because you have tried to use a memory address that is not allocated to your process. It may be part of the system's memory space or it may belong to another process that is in memory.
5. **Throwing an exception.** The program is in a permanent wait state or an infinite loop.
6. **Waiting for Eternity.** The program is in a permanent wait state or an infinite loop.

6.4 Causes and Cures

6.4.1 A shallow copy disaster.

The following short program creates two objects of class `Mixed`, then assigns one to the other. By default, a shallow copy operation is performed. An unusual function named `change()` is included in this class to illustrate the shared nature of the extension. No normal class would have a public function, like this, that permits private data to be corrupted. The output illustrates all three properties of a shallow copy operation. First, the allocation area that was originally attached to `B` is no longer attached to anything and has become a leak. Second, because `B` and `A` now share storage, assigning `'T'` to the first letter of `B` changes `A` also, as shown in the output, below. Finally, my run-time system detects a double-deletion error during the program termination process, and displays an error comment:

```
#include "tools.hpp"
class Mixed {
    int len;
    char* name;
    public:
    Mixed( char* nm ){  //------------------- Constructor
        len = strlen(nm);
        name = new char [len+1];
        strcpy( name, nm );
    }
    ~Mixed() { delete[] name; }
    void print(){ cout << name << " " << len <<'\t'; }
    void change( char x ) { name[0] = x; }
};

int main( void )
{
    Mixed A( "Joe" );
    Mixed B( "Mary" );
    cout << "Initial values A, B:\t";  A.print();  B.print();
    B = A;
    cout << "\nAfter copying B = A:\t";  A.print();  B.print();
    B.change( 'T' );
    cout << "\nAfter setting B.name[0] = 'T':\t";  A.print();  B.print();
    cout <<endl;
}
```

```
Initial values A, B:    Joe 3   Mary 4
After copying B = A:    Joe 3   Joe 3
After setting B.name[0] = 'T':  Toe 3   Toe 3
*** malloc[2029]: Deallocation of a pointer not malloced: 0x61850; This could be a double free(),
or free() called with the middle of an allocated block; Try setting environment variable
MallocHelp to see tools to help debug
```

Advice. Deep copy `can be` implemented for passing function arguments and assignment `can be` redefined to use deep copy. However, it is probably not desirable in most cases because of the great expense of allocating and initializing copies of large data structures. Object-oriented style involves constant function calls. If a deep copy is made of every dynamic object every time a function is called, performance will suffer badly. It is "safe", but the cost of that safety may be too great. A much more efficient way to handle arguments that is just as safe is to pass the argument as a constant reference.

6.4.2 Dangling pointers.

A dangling pointer is a pointer that refers to an object that has been deallocated. Using a dangling pointer is likely to cause amnesia (problem 2. In the following program, a dangling pointer is created when the function `badread()` returns a pointer to a local array. (My compiler did give a warning comment about it.) The string "hijk" was in that array when the function returned, but was freed and overwritten by some other part of the program before the output statement in the last line.

```
#include "tools.hpp"
char* badread( int n ){
    char buf[n];
    cin.get( buf, n );
    return buf;
}
//------------------------------------------------
char* goodread( int n ){
    char* buf = new char[n];
    cin.get( buf, n );
    return buf;
}
//------------------------------------------------
int main(void)
{
    cout << "Enter the alphabet: ";
    char* word1 = goodread( 8 );
    char* word2 = badread( 5 );
    char* word3 = goodread( 8 );
    cout << "Your letters: " <<word1 <<" " <<word2 <<" "<<word3 <<endl;
}
```

Output:

```
Enter the alphabet: abcdefghijklmnopqrstuvwxyz
Your letters: abcdefg  lmnopqr
```

6.4.3 Storing things in limbo.

Limbo is an unknown, random place. Storing things in limbo can cause problem 2, 3, or 4. A common error is to try to store character data in a char* variable that does not point at a character array, as shown in the program below. On my Mac OS-X system, the first input line, `cin >> fname >> lname`, causes a bus error. The compiler does not detect this error because, to it, char* and char[] are compatible types. Caution: there should be a call on `new` or `malloc()` for every char* variable that holds input.

```
#include <iostream>
using namespace std;
typedef char* cstring;

int main( void )
{
    char* fname;                  // Not initialized to anything.
    cstring lname;                // Points off into limbo.
    cout << "Enter two names: ";
    cin >> fname >> lname;        // Storing in Limbo..
    cout << fname << " " << lname << endl;
}
```

Output:

```
Enter two names: Andrew Bates

Limbo has exited due to signal 10 (SIGBUS).
```

6.4.4 The wrong ways to delete.

Shallow copy and deep deletion. Class objects with extensions should be passed by reference, not by value. If you pass a class object to a function as a call-by-value argument, a shallow copy of it will be used to initialize the parameter. When the function returns, the class constructor will be executed on the parameter. If the argument had an extension, and the destructor is written in the normal way, the extension will be deleted and the original copy of the object will be broken. Later, when the freed memory is reallocated, the program will experience amnesia (problem 2). This extension of the output operator, and the call on it in the third line, illustrate the most common deep-delete error:

```
inline ostream& operator<<( ostream& out, Dice d ) { return d.print(out); }
Dice mySet(5);
cout << mySet;
```

The second line declares a set of 5 dice, which are implemented by a dynamically allocated array of 5 integers. The Dice destructor (properly) deletes that array. The third line calls the faulty output operator, and, as part of the call, makes a copy of mySet (which shares extensions with the original). When printing is finished, this copy will be freed, its destructor will be run, and the dice array will be gone. This is Not the intended result!

Wrongful deletion. Trying to delete a static or auto object may cause problem 3 or 4 or 5 or may result in a delayed error. If a pointer stores any address other than one that was returned by `new`, it must never be the argument of `delete`. In the next example, we try to delete an auto array. The result is a segmentation fault.

```
#include <iostream>
using namespace std;

int main( void )
{
    char buf[80];
    char* input = buf;
    cout << "Enter some wisdom: ";
    cin.getline( input, 80 );
    cout << "Today’s words for the wise: " <<input <<endl;
    delete input;
    cout << "Normal termination";;
}
```

Output:

```
Enter some wisdom: To forgive is difficult.
Today’s words for the wise: To forgive is difficult.
wisdom(43266) malloc: *** error for object 0x7fff5fbff410: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort
```

Partial deletion. The same kinds of disasters will happen when you delete through a pointer that points at the middle of a dynamically allocated array, as in the next example. The newest Gnu C++ compiler was used here; it trapped the faulty delete command and produced an intelligent error comment instead of crashing. Older compilers simply crash with a segmentation fault.

```
#include "tools.hpp"
int main(void)
{
    char* scan, * buf = new char[80];
    cout << "Enter some words and some punctuation: \n\t";
    for(;;) {
        cin.getline( buf, 80 );
        scan = strchr( buf, ’.’ );
        if (scan != NULL) break;
        cout << "I don’t like " <<buf <<" Try again.\n\t";
    }
    cout << "I like the last one.\n" <<scan <<endl;
    delete scan;
}
```

Output:

```
Enter some words and some punctuation:
        It is sunny today!
I don't like It is sunny today! Try again.
        It is rainy today.
I like the last one.
.
delestack(43229) malloc: *** error for object 0x100100091: pointer being freed was not allocated
*** set a breakpoint in malloc_error_break to debug
Abort
```

Double deletion. Deleting the same thing twice is likely to cause problem 4, 3 or a delayed fatal error. The outcome would depend on the memory management algorithm used by your compiler. This commonly happens when the program correctly deletes through the pointer to the head of a dynamic area, but also deletes through a copy of the head pointer. This error is trapped by the new Gnu C++ compiler.

Failure to delete. If an area is not deleted when its useful life is over, problem 1 , a memory leak, will result. This is a serious problem for real-life programs that run in the background for long periods of time.

6.4.5 Walking on memory.

We say that a program "walks on memory" when it stores data beyond the end of the area (usually an array) that has been allocated for the target object. This can cause problem 2, 4, or 3. This happens when:

1. You store beyond the end of the allocated array because of careless array processing. (See example below.)
2. The input string is longer than the buffer during a call on an unlimited string input function.
3. The "from" string is longer than the "to" array during a call on strcpy().
4. The "to" array is too short to contain the combined strings during a call on strcat().

In the next program, the array `nums` is used to hold a series of numeric inputs. The program runs, reads input, and (because of careless use of subscripts) stores the last number past the end of the `nums` array, on top of the array allocated for the last name. The results are machine dependent, but on both of my systems, we get the output shown below the program. Note that the last name was correctly printed on the second line, but it is gone from the fifth line. Why? When the loop went past the end of the nums array, it stored the number 5 (0x05) on top of the last name. The 0 byte (`'\0'`) fell on top of the 'W', so nothing was printed.

```
#include "tools.hpp"
int main( void )
{
    char fname[10], lname[10];
    int nums[4], total = 0;
    cout <<"Enter two names: ";  cin  >>fname >>lname;
    cout <<"You typed: " <<fname << " " << lname << endl;
    cout <<"Enter four numbers: ";
    for( int k=1; k<=4; ++k)  {
        cin >> nums[k];
        total += nums[k];
    }
    cout <<"First: " <<fname << "\nLast: " <<lname <<"\nTotal = " <<total <<endl;
}
```

Output:

```
Enter two names: Margaret Wilkenson
You typed: Margaret Wilkenson
Enter four numbers: 2 3 4 5
First: Margaret
Last:
Total = 14
```

Parentheses are not square brackets. Square brackets [ ] are used to allocate an array. Parentheses () are used to enclose an argument for a constructor function or an initializer for a variable. Using () when you need square brackets can cause amnesia (problem 2) because the program will allocate one variable, not an array of variables, and eventually will overrun memory. In the next program, we want to allocate a new array of chars, but we use parentheses where the brackets belong. The line still looks "normal", and the mistake is easy to make and easy to miss:

```
#include "tools.hpp"
int main( void )
{
    char* buf1 = new char(100);
    char* buf2 = new char(100);
    cout <<"Enter two lines of text: \n";
    cin.getline( buf1, 100 );
    cin.getline( buf2, 100 );
    cout <<"Echo [(" <<buf1 <<endl <<buf2 <<")]\n" ;
}
```

Output:

```
Enter a line of text:
I once saw a doggie in the window,
Echo [(I once sEcho [(I once sEcho [(I on)]
```

The result is that you THINK there is enough space for a long string, but only 1 byte (initialized to the letter 'd', whose ASCII code is 100), plus padding was provided. In this case, it appears that eight bytes were allocated altogether, but some compilers might allocate less. Your program will probably not bomb, but when you (or the run-time system) allocates the next new object, all but the first N bytes of the data will be overlaid by new data. (Where N-1 is the number of bytes that were used for padding.) The output in this particular case is bizarre: I can't explain why there are three copies of some letters, and this result would probably not be repeatable on another system.

Changing a string literal. Many compilers store literals in a read-only storage area. An attempt to change the contents of a string literal can cause problem 4.

6.4.6 NULL pointers.

To avoid trouble, be careful not to walk off the end of a linked list, process an uninitialized portion of an array, or delete the referent of a `NULL` pointer. An attempt to dereference a NULL pointer may cause problem 4, or 3, depending on your system. The same is true of any attempt to apply a subscript or `++` to the NULL pointer. The nature of the error will be system dependent. I have three computers with different processors and different operating systems, all running the gnu C++ compiler. On one computer, I got a segmentation error, on the other two a bus error, as shown below.

```
#include "tools.hpp"

int main( void )
{
    char* name[5] = {"Jane", "Yonit", "Theo", "Xin"};
    char* selection;

    for(int k=0; k<5; k++){
        selection = name[k];
        cerr << "Name [" << k <<"] is " << selection << " \t";
        cerr << "[" << k <<"][" << k <<"] = "<<selection[k] << "\n";
    }
    bye();
}
```

Output:

```
Name [0] is Jane        [0][0] = J
Name [1] is Yonit       [1][1] = o
Name [2] is Theo        [2][2] = e
Name [3] is Xin         [3][3] =
Name [4] is (null)      Bus error
```

Printing. On many systems, garbage will be printed if you attempt to print a string but the string pointer is NULL On my system, however, it is always safe to print a string because the error is trapped by the print routines, and the output printed is `(null)`, as in the output above.

Illegal access. The program on line 119 is a simplified version of the Partial deletion program on line 69. The error that caused the earlier version to crash was eliminated. However, this still crashes with problem 3 or 4 or 5 if the input does not contain the "?" that we are searching for. The problem happens because `strchr()` returns a NULL pointer when it does not find the required character. That is not a problem. However, line 127 uses the result of the `strchr()` without testing it first and sends the memory address 1 to `<<`. This address is used by the system's BIOS (basic input-output system) and must not be used by an application program. On a system with memory protection, the result will always be a fatal error.

```
119  #include <iostream>
120  using namespace std;
121
122  int main(void)
123  {
124      char buf[80];
125      cout << "Enter some words and some punctuation: \n\t";
126      cin.getline( buf, 80 );
127      char* scan = strchr(buf, '?');
128      cout << "I like the last part: " <<scan+1 <<"\n\n";
129      cout << "buf: " <<buf <<endl;
130      cout << "Normal termination";
131  }
```

Output:

```
Enter some words and some punctuation:
        Hungary. Turkey, Chile!
Segmentation fault
```

Garbage in. Garbage in the input stream can cause problem 6. For example, each line in a DOS text file ends in two characters: carriage-return line feed. Unix lines have only one char at the end of a line. If you attempt to read a DOS data file in a program compiled for UNIX, the end-of-line characters will be garbage and will probably cause an infinite loop. The I/O demonstration program at the end of Chapter 3 illustrates this and other file-handling issues.

Chapter 7: C++ Bells and Whistles

A number of C++ features are introduced in this chapter: default parameters, const class members, and operator extensions.

7.1 Optional Parameters

Purpose and Rules. Default parameters are never necessary but they are a great convenience. They allow us to write a single function definition that defines two or more function variations with different numbers of parameters. A function with one default parameter is really two functions that share a name. (This is sometimes called overloading a name.) A function with several default parameters is a shorthand for several function definitions. The basic requirements are:

- A default value for a parameter is given in the class declaration, not the function definition. It is written (like an initializer) after the parameter name in the function prototype.

  The default values may either be given or omitted in the function definitions that are written in a separate .cpp file. However, if the function definition *does* gives a default parameter value, it must match the one given in the class declaration.
- Some or all of the parameters of any function may have default values.
- All parameters with default values must come after the parameters without defaults.
- When a function is called, the programmer may omit writing arguments if the default values are acceptable.
- Documentation should always explain clearly which parameters have default values and what the values are.

The most familiar function defined with optional arguments is istream::get(). It can be called to read a string with either three or two arguments:

```
cin.get( input, MAX, ',' );
cin.get( input, MAX );
```

The calls both read characters into an array named input. Both stop after MAX-1 characters have been read. The first call will also stop if a comma is read; the second call will stop when the default character (newline) is read. The corresponding prototype of istream::get() might look something like this:

```
void cin.get( char* buf, int max, char c='\n' );
```

7.1.1 A Short Example

Consider the short program that follows. It constructs and prints three instances of a Box class, each with a different number of parameters. The definition of the Box constructor makes this possible by using default parameters.

- By using default parameters, we can replace two or more function definitions by one (line 26). An earlier version of this program had two constructors:

  ```
  Box( int ln, int wd, int ht ){ length=ln; width=wd; high=ht; }
  Box(){ length = width = high = 1; }
  ```

- This single definition is equivalent to a family of four function methods, and allows us to omit the final zero, one, two, or three arguments when calling the constructor:

```
Box( int ln, int wd, int ht );
Box( int ln, int wd );
Box( int ln );
Box( );
```

```
 1  //--------------------------------------------------------------------------
 2  // Array construction example, October 8, 2000                  file: BoxM.cpp
 3  //--------------------------------------------------------------------------
 4  #include "tools.hpp"
 5  #include "box.hpp"
 6  //--------------------------------------------------------------------------
 7  int main( void )
 8  {   cout << "\nTesting the default parameters\n";
 9      Box B1;             // Make a default Box.
10      Box B2(2);          // Make Boxes with various parameters.
11      Box B3(2, 3);
12      cout <<"\nDumping the boxes:\n" << B1 << B2 << B3 <<endl;
13  }
```

```
14  //--------------------------------------------------------------------------
15  // Demonstrate syntax and usage of default parameters.
16  // A. Fischer February 10, 2009                                  file: box.hpp
17  //--------------------------------------------------------------------------
18  #pragma once;
19  #include "tools.hpp"
20  //--------------------------------------------------------------------------
21  class Box {
22    private:
23      int length, width, high;    // The three dimensions of a box.
24
25    public:
26      Box(int ln=1, int wd=1, int ht=1){
27          length=ln; width=wd; high=ht;
28      }
29
30      ~Box() {}
31
32      ostream& print( ostream& out = cout ){
33          return out <<length <<" by " <<width <<" by " <<high <<".  ";
34      }
35  };
36
37  //--------------------------------------------------------------------------
38  inline ostream& operator<< (ostream& out, Box& B) { return B.print( out ); }
39
```

Usage. Clearly, optional parameters are useful in constructors. They also save time and writing whenever a function must sometimes handle an unusual case. The parameter connected to the unusual situation is coded last and omitted from most calls. For example, suppose you wanted a print function to (normally) write output to `cout`, but occasionally you want to write to a file. You can accomplish this with an optional parameter. Suppose B is a Box object and file_out is an open output stream. Then you could call the Box::print function (line 32) three ways:

```
B.print( cout );     // Write to the screen, but we do not need the parameter.
B.print();           // Means the same thing as the line above.
B.print( file_out ); // Write to an output file.
```

7.2 Const Qualifiers

Purpose and usage. The proper use of const can be a powerful documentation technique. When you are using classes defined by someone else, you often see only the declarations and prototypes of the public members of the class. It is very important to know which of the members are constant and which functions do not modify their parameters. Here are a few rules and guidelines:

- You rarely need to use const. If a program will work with it, it will work without it.
- Use const to explicitly state your intentions: if your intention is that the variable will not change, then you should define it as a const.
- The most common use of const is to restrict the usage of reference and pointer parameters and return values, so that they give read-only access to another program unit.
- In class functions, const can also be used with the implied parameter; in this case it declares that no statement in the function changes the value of any member of `*this`.
- The other common use of const is at the beginning of a function, to declare a local variables whose value depends on one of the parameters, but whose value will be constant throughout the execution of one call on that function.
- If they are used at all, global variables should be const. It is not bad style to use a global const variable because you can do things with it that you cannot do with #define (initialize a constant object of a structured type).

Finally, be aware that compilers vary with respect to const. It is the job of the compiler (not the programmer) to make sure that the const rules are never broken. Writing the code that enforces the const declarations is difficult. Some compilers enforce the constant rules very strictly; others do not, so your program might compile without errors on a sloppy compiler but fail on a compiler that checks the const rules carefully.

Syntax for const. Everywhere a prototype contains the keyword const, the matching function definition must have a matching const. Here are the rules that apply to writing const in a function prototype and definition. There are four places the keyword const can appear:

1. The const can be written before the function's return type. This is used only when the value returned by the function is a * or an & type.

   ```
   const int* f( int x );  // A pointer to a constant integer.
   const int& g( int x );  // A reference to a constant integer
   ```

   The return value of f() points at a constant integer (possibly part of an array). It can be stored in a non-constant pointer variable. You could increment that pointer to point at the next element in an array of constant ints, but you could not change any of them.

   The return value from g() is a reference to a constant int. It gives read access but not write access to that integer.

2. The const can be written before the type of one of the parameters in the parameter list. This is used only when the parameter is a * or a &. It means that the function cannot modify the caller's variable that corresponds to that parameter.

   ```
   void List::Search( const Item& s );
   void List::Search( const char* name );
   ```

   Here, we declare that the Search function will not modify the Item (in the first prototype) or the string (in the second prototype) that is its argument. Class objects are usually passed by reference to avoid triggering the class destructor when the function exits. In this case, the const plays an important role in isolating the caller from the actions of the function.

3. The const can be written between the end of the parameter list and the ; that ends the prototype or the { that opens the function body. In this position, const means that the function does not modify the implied parameter.

   ```
   void Stack::Print() const;
   int Stack::Space() const { return Max-N; };
   ```

   In this location, the const is used as a form of documentation. It declares that the Print and Space functions use the members of the Stack class but do not modify them. This is obvious if you look at the function's code, but can be helpful documentation when only the prototype is available (not code itself).

4. Within the body of a function, const can be applied to any local variable. This is often done when the variable depends on one of the parameters, but does not change within the function.

7.3 Operator Extensions

Purpose and usage. The built-in arithmetic operators in C and C++ are generic; they are defined for all built-in numeric types. One of the purposes of an object-oriented language is to enable the programmer to define new classes and use them in the same ways that the built-in classes are used. Combining these two facts, we see a need to *extend* the built-in generic operators to work on new classes. An operator extension implements a built-in function for a new type in such a way that the intention and character of the original operator are preserved.

The assignment operator and a copy constructor are automatically defined for all types, as member-by-member shallow copies. Both can be redefined, if needed, to do something else; the new definition *overrides* the default definition. This is appropriate when the programmer wants to eliminate the possibility of an accidental use of the built-in definition, and even more appropriate if something useful can be implemented instead.

A third kind of operator definition *overloads* the operator by giving it a new meaning that is only distantly related to its original meaning. For example, the `+=` operator is extended, below, to add a new item to a list of items. This is not at all like numeric addition, so it is an overload of `+=`, not an extension of the original meaning of the operator (add and store back).

C has both binary and unary operators. When a unary operator (such as increment or type cast) is extended, the definition is always given as part of the relevant class. When a binary operator is extended, the definition is normally given as part of the class of the left operand. If that is not possible, it is given globally (outside all classes). A general rule for an operator definition is that the the number of operands and precedence must match the built-in operator. Default arguments are illegal in these definitions.

Below, we consider some contexts in which an operator definition might make sense and give one or more examples of each.

7.3.1 Global Binary Operator Extensions

These are appropriate for extensions of an operator to a non-class type, or for a pre-defined class that cannot be modified. The syntax for this kind of extension prototype is:

```
return-type operator op-symbol (left-op-type, right-op-type );
```

Three extensions of the `<<` operator were defined in the Bargraph program. These are defined globally because we cannot add them to the predefined ostream class. The right operand (second parameter) belongs to the new class in each case.

```
ostream& operator << ( ostream& out, Item& T);
ostream& operator << ( ostream& out, Graph& G);
ostream& operator << ( ostream& out, Row& T);
```

The preferred way to write this kind of extension is as an `inline` function just below the bottom of the class declaration. If the function is too long for inline expansion, then the prototype should be given at the bottom of the .hpp file for the class, and the function definition should be in the main module, after the #include command for the class header and before the beginning of main().

7.3.2 Binary Operator Extensions in a Class.

Binary functions within a class are defined using this prototype syntax:

```
return-type operator op-symbol ( right-op-type );
```

Extending arithmetic and comparison. Suppose we defined a new numeric class such as `Complex`. It would be appropriate to extend all the basic arithmetic and comparison operators for this class, so that complex numbers could be used, like reals, in formulas. Assume that the class Complex has two data members, rp (the real part) and ip (the imaginary part). Several definitions are required to define all of the relevant operators; here are two samples:

```
Complex operator + ( Complex plex ){   // Add two complex numbers.
    return Complex( rp + plex.rp, ip + plex.ip );
}
bool operator == ( Complex plex ){  // Compare two complex numbers.
    return rp == plex.rp && ip == plex.ip;
}
```

In such definitions, the implied argument is the left operand and the explicit argument is the right operand. Any reference to the left operand is made simply by using the part name. References to the right operand are made using the parameter name and the part name.

Adding to a collection: an overload. Another appropriate extension is a += operator in a container class. The Stack::Push() function could be replaced by the following extension. (Only the function name would change, not the function's code.)

```
void Stack::operator += ( char c );    // Push an Item onto the Stack.
```

The Stack::push function is called from three loops in the main program. The first loop could be replaced by this intuitively appealing code:

```
do { cin >> car;
     west += car;
} while (car != QUIT);            // Engine has been entered.
```

Assignment: an override. Sometimes the default definition of assignment is changed. For example, here is a definition of = for Stacks that moves the data from one stack to another, rather than doing a shallow copy. This definition illustrates the use of the keyword `this` to call the += function (defined above) to push a character onto the implied Stack parameter:

```
void operator= ( Stack& z ) {    // Pop one stack, push result onto other.
    char c;
    while (!z.empty()) {
        c = z.pop();
        *this += c;
    }
}
```

Extending subscript. The remaining important use of operator extensions is to define subscript for data structures that function as arrays but have a more complex representation. A good example is the resizeable array data structure that was used to implement the unlimited storage capacity for the stack in the Stack program. A flexible array class needs an extension of the subscript operator so that the client program can refer to the object as if it were a normal array.

The subscript operator is a little different in two ways. First, it must return a reference to the selected array slot so that the client program can either read or write the selected element. Second, the syntax is a little different because the left operand is enclosed between the square brackets, not written after them. An added advantage of extending subscript is that a bounds check can be incorporated into the definition, as illustrated below. Here is the part of the class declaration and the extension of the subscript operator that could be part of a FlexString class (a resizeable array of characters).

```
class FlexString {
  private:
    int Max;                 // Current allocation size.
    int N;                   // Number of data chars stored in array.
    char* Buf;               // Pointer to dynamic array of char.
  private:
    char& operator[]( int k );
    ...
};

// ----------------------------------- access the kth char in the string.
char& FlexString ::  operator []( int k ) {
    if (k >= N) fatal(" FlexString bounds error.");
    return Buf[k];
}
```

7.3.3 Unary operators.

Extending type cast. The syntax for extending the type cast operator is:

operator `target-type() {` **return** `member-name; }`

An extension of the type cast operator is used to convert a value from a class type to another type. This is also called "member conversion", because it converts a class member to some other type. It is particularly useful in linked-list classes such as Cell (below) for converting from a Cell to a pointer type such as Item*, by extracting the pointer from the Cell.

```
class Cell {
  private:
    Item*  Data;
    Cell* Next;
  public:
    operator Item() { return *Data; } // Type cast from Cell to Item
    ...
};
```

Type coercion. Extensions of a type cast operator can be used explicitly, but they are also used by the system to coerce operands and parameters. For example, in a program that contained the above extension of type cast, if you used a Cell object in a context in which a Cell* was needed, the compiler would use the Next field of the object to carry out the operation.

Increment and decrement: extensions or overloads. The Prefix forms of increment and decrement are normal unary operators; The postfix forms are different from everything else in the language. The prototypes for these operators have the form:

Preincrement:	`return-type` **operator ++()**;
Postincrement:	`return-type` **operator ++(int)()**;

The "int" after the ++ in the postfix form serves only one purpose, and that is to distinguish the prefix and postfix forms. The "(int)" looks like a parameter list, but the parameter does not need a name because it is not used; it is just a dummy. Definitions of these two operators inside the complex class might look like this:

```
Complex operator ++ (){ rp++; return *this; }
Complex operator ++ (int){ Complex temp = *this; ++rp; return temp; }
```

Another use of an increment extension is to make the syntax for a linked list imitate the syntax for an array, so that the exact nature of a class implementation can be fully hidden. This technique is used extensively in implementations of container classes in the Standard Template Library, and can be applied in any list class that has a "current element" member. Here is an appropriate extension for a List class that has members named Head, Curr and Prior:

```
Cell* operator ++ (){ Curr = Curr->Next; return Curr; }
Cell* operator ++ (int){ Cell* tmp = Curr; Curr = Curr->Next; return tmp; }
```

Given this definition and an appropriate extension for <= in the Item class, we could write a List::Search function as follows:

```
bool List::Search( Item T ){   // Search for T in list with ascending order.
    for (Curr= Head; Curr!=NULL; )
        if (T <= Curr->Data) break;
        Prior = Curr++;
    }
    return T == Curr->Data     // true for exact match, false for >.
}
```

Chapter 8: Interacting Classes

8.1 The Roles of a Class

Three more basic OO design principles:

> **A class is the expert on its own members.**
> **Oil and water don't mix.**
> **Delegate! Simplify!**

In this chapter we begin to build complexity by introducing classes that interact, or `collaborate` with each other to get the job done. Classes serve a variety of purposes in OO programming, and we try to define our classes so that they serve these purposes well. A class is...

- A type from which objects (instances) can be formed. We say the instances *belong to* the class.
- A collection of things that belong together; a struct with its associated functions.
- A way to encapsulate behavior: a public interface with a private implementation.
- A way to protect the integrity of data, providing the rest of the world with functions that provide a view of the data but no way to modify it.
- A way to organize and automate allocation, initialization, and deallocation
- A reusable module.
- A way to break a complex problem down into manageable, semi-independent pieces, each with a defined interface.
- An entity that can collaborate with other classes to perform a task.

8.1.1 Finding the Classes

In designing an OO application, we must first identify the classes that we need, based on the problem specification. This is more of an art than a science and experience plays a large role in making good choices. For a beginner, it helps to have some idea of the kind of classes that others have defined in many kinds of OO applications. These include:

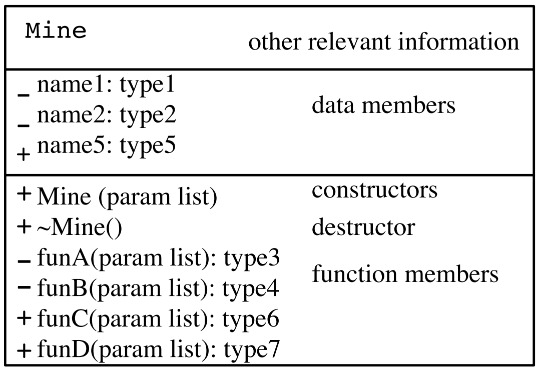
- Physical or tangible objects or places: airplane, airport, document, graph
- Non-tangible objects or systems: creditAuthorization, authorizationSystem
- Specifications for, or records of, objects: floorPlan, receipt
- Events or transactions: sale, charge, reservation, crash
- Organizations: csDepartment
- Roles of people: administrator, clerk
- Containers of things: jobList, facultySet, partsInventory, cardCatalog
- Things that go into the container: job, faculty, part, card

8.1.2 Diagramming One Class

A class is represented by a rectangle with three (or, optionally four) layers, as shown on the right:

- class name
- data members (type and name)
- function members (prototypes)
- other relevant information: friends, etc

Members are marked with + for public, # for protected, or – for private. In simplified diagrams, some of the layers may be omitted.


Mine
other relevant information
_ name1: type1
_ name2: type2
+ name5: type5
data members
+ Mine (param list)
+ ~Mine()
– funA(param list): type3
– funB(param list): type4
+ funC(param list): type6
+ funD(param list): type7
constructors
destructor
function members

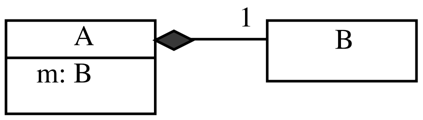
8.2 Class Relationships

Several relationships may exist between two classes: composition, aggregation, association, derivation, and friendship. These are explained in the paragraphs below and illustrated by class relationship diagrams. The four relationships at the top of this list provide the most protection for members of class B. As we move down the list, protection decreases, that is, class A functions get more privileges with respect to the private parts of B.

8.2.1 Composition.

Class A *composes* class B if a B object *is part of* every A object. The B member must be born when A is born, die when A dies, and be part of exactly one A object. Class A could compose any definite number of B objects; the number is indicated at the B end of the A-B link . The black diamond on the A end of the link indicates composition.

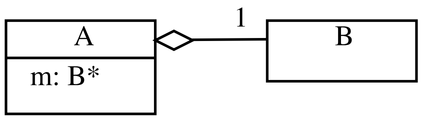
In this diagram, class A has one member named m of class B, which is constructed when an A object is constructed. When the A object is deallocated, it will cause the deallocation of the corresponding instance of B. A can use the public functions of B to operate on m.



8.2.2 Aggregation.

Class A *aggregates* class B if a B object *is part of* every A object, but the lifetime of the B object does not match the lifetime of the A object. The B object may be allocated later and attached to A with a pointer. Also, one B object might be aggregated by more than one A object, and A could aggregate more than one B object.

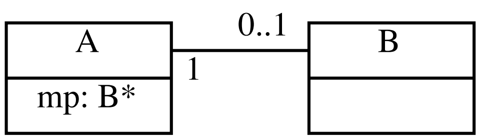
In the diagram, the white diamond on the A end of the A-B link indicates aggregation. Class A has one member named mp of class B*. When an instance of A is deallocated, it might or might not cause the deallocation of the corresponding instance(s) of B. A can use the public functions of B by using mp->.



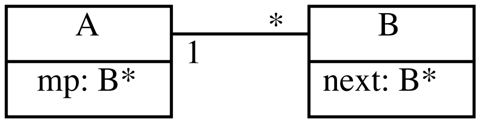
8.2.3 Association.

Class A is *associated* with class B if a member of A points at an instance of class B (or a set of instances), but both kinds of objects have an independent existence. Neither one is "part of " the other. The B object is generally not created by A's constructor; it is created independently and attached to A.

Simple Association. In a simple association, class A is associated with a definite number of B objects, often one. This is usually implemented in A by one pointer member that may be NULL or may be connected to an object or array of objects of type B. A can use the public functions of B by using mp->.

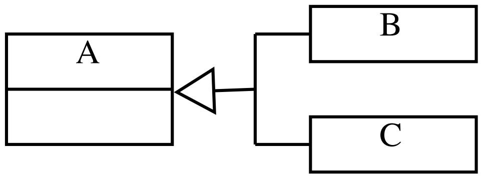


One-many Association. Class A is associated with a set of objects of class B. This set may be ordered or not. The relationship can be implemented by a member of A that points at a data structure containing objects of type B. In this case, A can use the public functions of B by using `mp->`. Alternatively, using STL classes, a member of A is an iterator for the set of B objects.



8.2.4 Derivation.

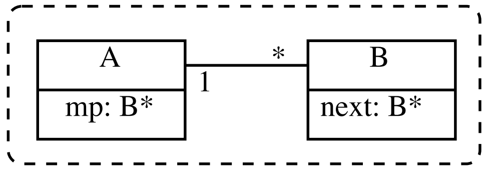
Class B can be *derived from* class A, which means that a B object is one possible variety of type A. Using derivation, a single object may have more than one type at the same time. Every instance of B or C is also an instance of A; it has all the properties of A, and inherits all of A's functions. The derived classes will extend either the set of A's properties, or functions, or both, and B and C will extend A differently.



In the diagram, the triangle with its point toward A indicates that classes B and C are derived from class A. We will study derivation after the midterm.

8.2.5 Friendship.

Class B can give *friendship* to A. This creates a tightly-coupled pair of classes in which the functions of A can use the private parts of B objects. This relationship is used primarily for linked data structures, where A is a container for items of type B and implements the interface for both classes. All members of class B should be private so that outside classes must use the interface provided by A to access them. (An alternative is to declare the class B inside class A, with all members of B being public.)



In the diagram, we show a dashed "rubber band" around the two class boxes, indicating that each one relies on the other.

8.2.6 An example.

Suppose you wish to implement a family datebook:

- The Datebook will have an array of linked lists, one for each family member.
- Each List points at a Node.
- Each Node contains one appointment and a pointer to the next Node.
- One appointment can be shared by two or more lists.

The resulting data structure is illustrated in Figure 7.1. In this structure, the Datebook *composes* four Lists, each list has a one-many *association* with a set of Nodes, and each Node *aggregates* one Appointment. Finally, *friendship* is used to couple the List and Node classes because the List class will provide the only public interface for the Node class. The UML diagram in Figure 7.2 shows these relationships.

8.2.7 Elementary Design Principles

Several design issues arise frequently. We list some here, with guidelines for addressing them and the reasons behind the guidelines.

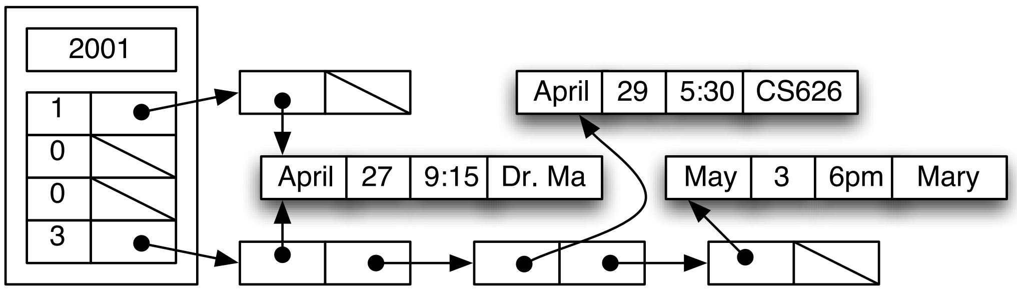


Figure 8.1: The data structure: an array of linked lists.

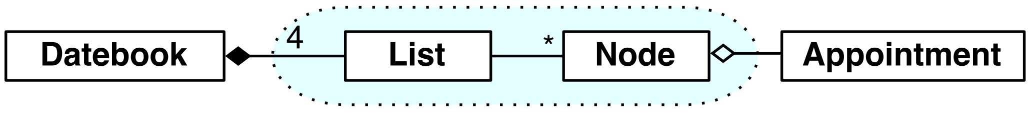


Figure 8.2: UML diagram for the Datebook data structure.

Privacy. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another. In the Bargraph program, later in this chapter, all data members of all classes are private.

Expert. Each class is its own expert and knows best what should be done with its members and how to do it. All functions that use data members should be class functions. If a class seems to need access to a member of another class in order to carry out an operation, it should delegate the operation to the class that is the expert on that data member.

There is one exception to this rule: To implement a linked structure, such as a List in the previous example, an auxiliary class is needed (Node). These two classes form a tightly coupled pair with an interface class (List) and a helper class (Node). A "friend class" declaration is used to create this relationship. All access to Nodes is through functions of the List class.

In the Bargraph program (later in this chapter) the two members of class Item are private. All access to Items is through the constuctor, destructor, and Print functions that belong to the class. However, in the tightly coupled pair of classes, Row and Cell, Row provides the interface for Cell, so Cell gives friendship to Row, allowing Row to set, change, and print the members of Cell.

Creation. The class that composes or aggregates an object should create it. [Why] This minimizes the ways in which one class depends on another (coupling).

In the Stack class, stack allocates storage for the objects (characters) stored in it. When that storage becomes full, the class allocates more space. In the Bargraph program, the Row class allocates space for Cells because the Row class contains the list of cells. It allocates the Items also, because Cell is a subsidiary class for which Row supplies the interface.

Deletion. The class that aggregates an object should generally delete it. [Why] To minimize confusion, and because nothing else is usually possible.

The array that was allocated by the Stack constructor and aggregated by the Stack class is freed by the Stack destructor. Also, in the push() function, each time the stack "grows", the old storage array is freed as soon as the data is copied into the new allocation area. In the Bargraph program, allocation happens at two levels. The Graph constructor allocates and aggregates 11 Rows, and the Graph destructor deletes them. The function Row::Insert allocates an Item and a Cell, which are then associated with the Row. At termination, the Row destructor deletes all the Items and all the Cells associated with it.

Consistency. Only class functions should modify class data members. There should be no "set" functions that allow outsiders to freely change the values of individual data members. [Why] To ensure that the object always remains in a consistent state.

Consider a class such as Row, that implements a linked list. All actions that modify the list structure should be done by Row class functions. The only class that should even know about tis structure is Row, itself. Under no circumstances should the Row class provide a function that lets another class change the pointer in a Cell. If another class is allowed to change list pointers, the list may be damaged beyond recovery.

Delegation. If a class A contains or aggregates a data member that belongs to class B, actions involving the parts of that member should be delegated to functions in class B. [Why] To minimize dependency between classes.

In the Bargraph program, each public class has its own print function. The Graph class delegates to the Row class the job of printing each Bar. That class, in turn, lets the Item class print the data fields of each Item.

Don't talk to strangers. Avoid calling functions indirectly through a chain of references or pointers. Call functions only using your own class members and parameters and let the member or parameter perform the task by delegating it in whatever way makes local sense. [Why] To avoid and minimize the number of indirect effects that happen when a class definition changes.

In the Bargraph program, the Graph class aggregates the Row class, the Row class associates with the Cell class and aggregates the Item class. So the Graph class is free to call Row functions, but it should not be calling Cell functions or Item functions. To do a Cell or Item operation, Graph should delegate the task by calling a Row function (and Row should provide functions for all such legitimate purposes.)

Responsibility. Every class should "take care of" itself, validate its own data, and handle its own emergencies. [Why] To minimize coupling between classes.

In the Stack program, the Stack checks whether it is full and if it is, it allocates more storage. There is no reason that a client program should ever ask about or be aware of a "full" condition.

8.3 The Program: Making a Bar Graph

8.3.1 Specification

Scope: Make a bar graph showing the distribution of student exam scores.

Input: A file, with one data set per line. Each data set consists of three initials and one exam score, in that order. (You may assume that all lines have valid data.) A sample input file is shown below, with its corresponding output.

Requirements: Create and initialize an array of eleven rows, where each row is initially empty but will eventually contain a linked list of scores in the specified range. Then read the file and process it one line at a time. For each line of data, dynamically allocate storage for the data and insert it into the appropriate row of scores.

Do this efficiently. When adding an item to a row, compute the subscript of the correct row. Do not identify it using a switch or a sequential search. Also, do not traverse that row's list of entries to find the end of the list; insert the new data at the head of the list. This is appropriate because the items in a list may be in any order.

Formulas: Scores 0. . . 9 should be attached to the list in array slot 0; scores 10..19 should go on the list in slot 1, etc. Scores below 0 or above 99 should go on the last list. The order of the items in each row does not matter.

Output: A bar graph with 11 horizontal bars, as shown below.

Input :

```
AWF  00
MJF  98
FDR  75
RBW  69
GBS  92
PLK  37
ABA  56
PDB  71
JBK  -1
GLD  89
PRD  68
HST  79
ABC  82
AEF  89
ALA  105
```

Output :

```
Put input files in same directory as the executable code.
Name of data file: bar.in
File is open and ready to read.

00..09:  AWF 0
10..19:
20..29:
30..39:  PLK 37
40..49:
50..59:  ABA 56
60..69:  PRD 68 RBW 69
70..79:  HST 79 PDB 71 FDR 75
80..89:  AEF 89 ABC 82 GLD 89
90..99:  GBS 92 MJF 98
Errors:  ALA 105 JBK -1
```

8.3.2 The Main Program and Output

```
1   //===========================================================================
2   // Bargraph of Exam Scores: An introduction to class interactions.
3   // Read scores from a user-specified file; display them as a bar graph.
4   // A. Fischer, September 30, 2000                           file: graphM.cpp
5   // Modified M. & A. Fischer, September 17, 2009
6   //===========================================================================
7   #include "tools.hpp"
8   #include "graph.hpp"
9
10  int main ( void )
11  {
12      banner();
13
14      Graph::instructions();          // Call of static class method
15      char fname [80];
16      cout << "Name of data file: ";
17      cin >> fname;                   // *** IRRESPONSIBLE CODE ***
18      // cin.getline(fname, 80);      // Prevents buffer overrun
19      ifstream infile (fname);        // Declare and open input stream
20      if (!infile)  fatal( "Cannot open %s for input - aborting!!\n", fname );
21      else say ( "File is open and ready to read.\n");
22
23      Graph curve( infile );          // Declare and construct a Graph object.
24                                      // Realizes data structure from a file
25      cout <<curve;                   // Print the graph.
26      // OR: curve.print( cout );     // Print the graph.
27
28      bye();
29      // Storage belonging to curve will be freed by destructors at this time.
30  }
```

Notes on the main program. The main program forms the interface between the user and the top-level class, Graph. It gives instructions (line 14) sets up an input stream for Graph's use and gives appropriate feedback (lines 15. . . 21), then calls the Graph constructor (line 23) and print function (line 25) to construct and print a bargraph. Although four classes are used to implement this program, three of them are subsidiary to Graph. No other classes are known to or used by `main()`. Thus, the only class header file included by this module is `"graph.hpp"`.

The output. The output consists of three sections. First are the banner, instructions and input prompt printed by main. Following this is the normal program output (a bar graph, exactly like the one shown in the specifications). Last is the trace that is printed by the class destructors. By including trace output in your destructors, you can check that all parts of your data structures are freed properly and prove that you have no memory leaks.

Below, we give the first and last parts of the trace to illustrate the process of memory deallocation. (The dotted line marks the part of the trace that was omitted.) The lines printed by the Graph and destructors are not indented, those printed by the Item destructor are indented, and the Cell destructor prints one word on the same line as the Item destructor. Rows and Items tell us which instance is being deleted. Note that the parts of a data structure are freed before the whole, that is, the Item before the Cell, the Cells before the Row, and the Rows before the Graph.

Here is the beginning and end of the program termination trace:

```
Normal termination.
   Deleting Item AWF ...  Cell
 Deleting row 00..09:   >>>
 Deleting row 10..19:   >>>
 Deleting row 20..29:   >>>
    Deleting Item PLK ...  Cell
 Deleting row 30..39:   >>>
 Deleting row 40..49:   >>>
    Deleting Item ABA ...  Cell
```

```
   .............................
Deleting row 80..89:   >>>
   Deleting Item GBS ...  Cell
   Deleting Item MJF ...  Cell
Deleting row 90..99:   >>>
   Deleting Item ALA ...  Cell
   Deleting Item JBK ...  Cell
Deleting row Errors:   >>>
Deleting Graph
```

8.3.3 The Data Structure and UML Diagrams

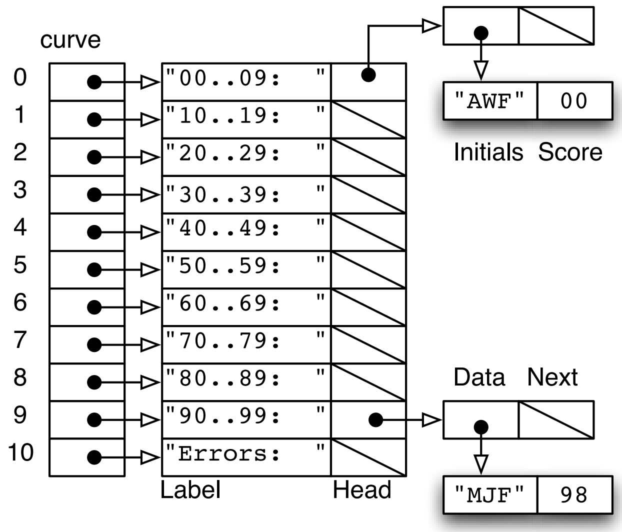


Figure 8.3: The graph data structure, after inserting two scores.

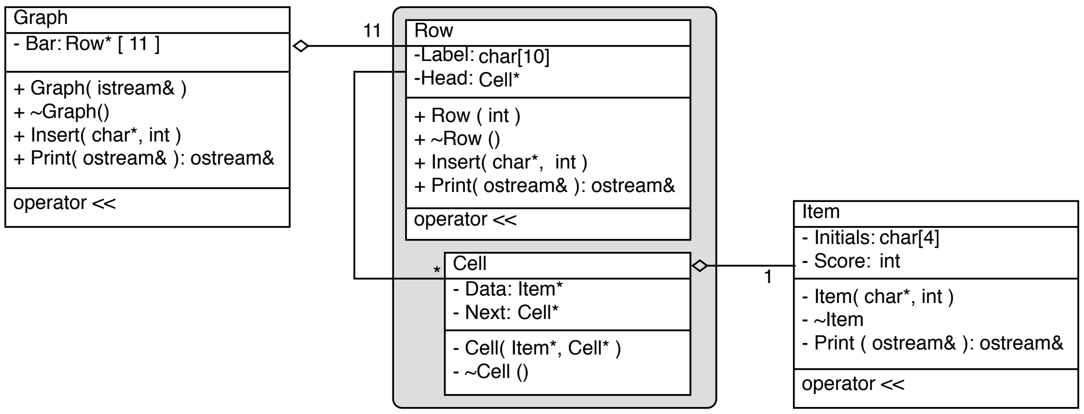


Figure 8.4: UML class diagram for Bargraph program.

Class relationships.

1. Part of initializing the Graph class is to allocate eleven new Rows and attach them to the backbone array (Bar). Since these are dynamically allocated, the relationship is not composition. But because there are exactly eleven Rows and their lifetime will end when the Graph dies, this is aggregation (not association).
2. Classes Row and Cell are friends, with Row providing the interface for both classes and Cell serving as a helper class for Row.
3. One Row contains an indefinite number of cells (zero or more), all allocated and deallocated at different times. This is association.
4. Each Cell contains a pointer to an Item. The Item is created by the Cell constructor and dies with the cell. This is aggregation. It is not composition because the Item is not physically part of the Cell (it is connected to the Cell by a pointer).

8.3.4 Class Item

```
//===========================================================================
// Item: A student's initials and one exam score.
// A. Fischer, October 1, 2000                                file: item.hpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#pragma once
#include <iostream>
#include <string.h>
using namespace std;
//---------------------------------------------------------------------------
// Data class to model a student exam score
// Alice's design pattern:
//     oil and water don't mix -- keep data and data structure separate
// Constructor and destructor contain tracing printouts for debugging
class Item                      // One name-score pair
{
  private:                      // Variable names are private
    char initials [4];          // Array of char for student name
    int  score;                 // Integer to hold score

  public:
    inline Item (char* inits, int sc);
    ~Item (){ cerr <<"    Deleting Item " <<initials <<" ...\n"; }
    ostream& print ( ostream& os );
};

//---------------------------------------------------------------------------
// Inline constructor, defined outside of the class but in header file
// Precondition: strlen(inits) <= 3
Item::Item (char* inits, int sc){
    strcpy( initials, inits );
    score = sc;
}

//---------------------------------------------------------------------------
inline ostream&    // inline can be declared here or within class (or both)
Item::print ( ostream& os ){
    return os <<initials <<" " <<score <<" ";
}

// Extend global output operator << -----------------------------------------
// Item::print() is expert at printing Item data
inline ostream&
operator << (ostream& out, Item& x){ return x.print( out ); }
```

We consider this class first because it is the simplest and relates to only one other class.

- The four functions defined here should be provided for every class: a constructor, a destructor, a print function, and an operator extension for `<<`.
- Every class function has a class object as its implied parameter, in addition to the explicit parameters listed in the parentheses. Thus, the Item constructor has three parameters (one implied and two explicit). We rarely refer to the implied parameter as a whole, but when we use the names of class members, we mean the members of the implied parameter.
- This class is defined entirely within the .hpp file because there are only three class functions and they are all short. The destructor fits entirely on one line, so it is given on line 49. The other two functions are declared immediately after the right-brace that closes the class declaration (lines 53. . . 65). Both are "inline". Item is declared "inline" on line 48; print is declared "inline" on line 62. Better style would have the word "inline" on both the declaration and the later definition.
- Lines 62–65 extend the `<<` operator to work on Items by defining it in terms of Item::Print(). Some books do this by declaring a friend function. However, friend declarations should be avoided wherever possible because they give free access to all class members. The technique shown here lets us extend the global operator without making data members public and without using a "friend function".
- The Print function returns the stream handle it received as an argument. This makes it easier to extend the `<<` operator for the new class. The operator `<<` *must* return an `ostream&` result to conform to the built-in definitions of `<<` and to make it possible to chain several `<<` operators in one expression. The definition given for `Print` returns an `ostream&` result to make it work smoothly with the operator extension given for `<<`. Both local definitions could be changed, as shown below, but they need to be paired correctly: the second definition of `Print` won't work with the first definition of `<<`.

```
void Print ( ostream& os ){ os <<Initials <<" " <<Score <<" "; }
inline ostream& operator << (ostream& os, Item& T){ T.Print( os );  return os; }
```

- The C++ language designers defined `<<` as a global (non-class) function so that it could be easily extended for every new class. Even though it works closely with the Item class, the operator extension does not belong to that class because `<<` is a global function. Therefore, the extension must be written outside the class.
- The extension of `<<` is written within the .hpp file for a class so that it will be included, with no extra effort, wherever the Item class is used. It must be declared "inline" to avoid linker errors due to multiple inclusion. This is a confusing issue that we will defer to later. For now, copy this pattern.
- The class definition file has a minimal header; it gives the author's name, last modification date, and the name of the file. In a commercial setting, a more extensive header would be required.
- The preprocessor command, `#pragma once` is used routinely in header files to prevent the same declarations from being included twice in the same compile module and to prevent circular `#includes`.

8.3.5 Class Graph

Notes on the Graph declaration.

- The class declaration starts and ends with the usual preprocessor commands. We include the header file for the Row class because a graph is built of Rows. We do not need the header for the Item class because the Graph functions do not deal directly with Items; all actions are mediated by the Row class.
- A Graph has only one data member, an array of pointers to Rows. Since the number of bars is known at compile time, it would be possible to declare a Graph as an array of Rows, not an array of Row*. Composing Rows is more efficient, but aggregating Row*s is more flexible. The Row* implementation was chosen so that the Row constructor could be called eleven times, to construct eleven rows, with the row number as an argument each time. This lets each Row be initialized with a different label. If an array of Row were used, all rows would need to be initialized identically.
- The constructor and destructor for this class are public because the main program will declare (create) an instance of Graph named "curve". The Graph destructor will be called automatically, from main, after the call on bye(), when curve goes out-of-scope.

- The argument to the constructor is the stream that contains the data, which should already be open. The argument is passed by reference for two reasons. First, call by value will not work here because reading from a stream changes it. Second, the stream is an object with dynamic extensions. All such objects are passed by reference or pointer to avoid the problem of shallow-copy and deep-deletion.
- Since all calls on Insert are within the Graph class, it is defined to be a private function. However, it is quite reasonable that someone might want to reuse this class in a program that does interactive input as well as file input. In that case, we would need to make the Insert function public.
- The extension of `operator<<` is included in the class header file because it is used as if it were a class function. However, it cannot be an ordinary class function because its first parameter is an ostream&, not a Graph. Being outside the class, the definition of `operator<<` cannot access the private parts of the class. It does its job by calling Graph::Print, which is a public function. Print, in turn, accesses the private class members. This is a typical illustration of how the privacy of class members can be maintained.

```
//===========================================================================
// Declarations for a bar graph.
// A. Fischer, April 22, 2000                                 file: graph.hpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#pragma once
#include "tools.hpp"
#include "row.hpp"
//#include "rowNest.hpp"  // alternative way to define a recursive type
#define BARS 11

//---------------------------------------------------------------------------
class Graph {
  private:
    Row* bar[BARS]; // Each list is one bar of the graph.  (Aggregation)
    void insert( char* name, int score );

  public:
    Graph ( istream& infile );
    ~Graph();
    ostream& print ( ostream& out );
    // Static functions are called without a class instance
    static void instructions() {
        cout <<"Put input files in same directory as the executable code.\n";
    }
};
inline ostream& operator<<( ostream& out, Graph& G){ return G.print( out ); }
```

```
//==========================================================================
// Implementation of the Graph class.
// A. Fischer, April 22, 2000                                file: graph.cpp
// Modified M. & A. Fischer, September 17, 2009
//==========================================================================
#include "graph.hpp"
//-------------------------------------- Use an input file to build a graph.
Graph::Graph( istream& infile ){
    char initials [4];
    int score;
    for (int k=0; k<BARS; ++k) bar[k] = new Row(k);
    for (;;){
        infile >> ws;              // Skip leading whitespace before get.
        infile.get(initials, 4, ' ');  // Read three initials ... safely.
        if (infile.eof()) break;
        infile >> score;           // No need to skip ws before using >>.
        insert (initials, score);   // *** POTENTIAL INFINITE LOOP ***
    }
}

```

```
// ---------------------------------------------------------- Delete all Rows.
Graph::~Graph() {
    for (int k=0; k<BARS; ++k)  delete bar[k];
    cerr <<" Deleting Graph\n";
}

// ------------------------------------------------ Insert a node into a Row.
void
Graph::insert( char *initials, int score ){ // Function is private within class.
    const int intervals = BARS-1;
    int index;
    // Calculate insertion index for score
    if (score >= 0 && score < 100)      // If score is between 0-99, it
        index = (score / intervals);    // belongs in one of first BARS-1 rows.
    else
        index = BARS-1;                 // Errors are displayed on last row
    bar[index]->insert( initials, score );  // delegation
}

// ---------------------------------------------- Print the entire bar graph.
ostream&
Graph::print( ostream& out ){
    out << "\n";
    for (int k=0; k<BARS; ++k)
        out << *bar[k] <<"\n";          // Delegate to Row::print()
    return out;
}
```

Notes on the Graph implementation. We put the definitions of all four class functions in a separate .cpp file because all are longer than a line or two.

- After allocating eleven empty Rows, the Graph constructor reads lines from the data file and calls the Graph::Insert function to store the data. Three lines of code are used to read each line of data because one of the data fields is a string. When `>>` is used to read a string, there is no protection against long inputs that overflow the bounds of the data array and start destroying other values in memory. For this reason, we use `get` rather than `>>` to read all strings. It is necessary to skip whitespace before the call on `get`, and calls on `get` and `>>` cannot be mixed in the same statement. Thus, three lines of code are required. If we do not care whether the input is done "safely" or not, these three lines could be condensed into one:

  ```
  infile >> Inits >> Score;
  ```

- Eleven Rows are created by the Graph constructor, so the Graph destructor contains a loop that deletes eleven Rows.
- The Insert function decides which row the new data belongs in, then delegates the insertion task to the appropriate Row. This is a typical control pattern; as smaller and smaller portions of the data structure are used, control passes from each class to the class it contains.
- Expertise: The Graph class is an expert on the meaning of its rows, so it selects the row to be used. The Row class is the expert on how to store data in a Row, so it accepts the data, creates a storage cell, and does the insertion.

8.3.6 Classes Row and Cell

Notes on the Cell class declaration.

- Together, the Cell and Row classes implement a linked list in which any number of data items can be stored. A Cell consists of a pointer to an Item and a pointer to the next Cell.
- The Row and Cell class declarations have been placed in a single .hpp file because they form one conceptual unit. Each class depends on the other and neither makes sense alone.

- Friendship. The Cell class is a private class (all of its members are private). It is used only to implement the Row class. It gives friendship to Row, which implements the interface for both classes. The private class (Cell) comes first, because the interface class (Row) refers to it. No way of any kind is provided for other parts of the program to access Cells.
- Each Cell aggregates one Item, which holds the data for one student. On the UML diagram, Cell and Item are connected by a line with a white aggregation diagram at the Cell end and an optional "1" at the Item end. As with Graph and Row, composition could be used, rather than aggregation, to put the Item in the Cell. In that case, the type of Data would be Item, not Item*. Aggregation is less efficient because it involves an extra layer of pointers, but it permits us to construct Items that are independent of the Cells. These items can be constructed and used in other parts of the program that have no relation to Cells or Rows.
- The Cell class constructor and destructor are short and routine and are defined inline. They, also, are private because they will be used only by Row functions. No other part of the program has any reason to create or destroy a Cell.

```
//===========================================================================
// Class for a linked-list row and its cells
// A. Fischer, October 1, 2000                                   file: row.hpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#pragma once
#include <iostream>
#include "item.hpp"
using namespace std;
//---------------------------------------------------------------------------
// Dependent class.  Holds an Item and a link to another Cell
class Cell
{
  friend class Row;
  private:
    Item* data;           // Pointer to one data Item    (Aggregation)
    Cell* next;           // Pointer to next cell in row (Association)

    Cell (char* d, int s, Cell* nx){ data = new Item(d, s); next = nx; }
    ~Cell (){ delete data; cerr <<"  Deleting Cell " <<"\n"; }
};

//---------------------------------------------------------------------------
// Data structure class
class Row   {   // Interface class for one bar of the bargraph.
  private:
    char label[10];             // Row header label
    Cell* head;                 // Pointer to head of row

  public:
    Row ( int n );
    ~Row ();
    void insert ( char* name, int score );  // delegation
    ostream& print ( ostream& os );
};

//---------------------------------------------------------------------------
inline ostream& operator << ( ostream& out, Row& T){ return T.print( out ); }
```

The Row class declaration.

- Row has a One-to-Many association with Cell, denoted in the UML by a "1" at the Row end and a "*" at the Cell end. Together, these classes form a classical container class for Items.
- Unlike a typical C or Pascal implementation of a linked list, the type Row is not the same as the type Cell*. The head of the linked list of Cells is one (but only one) of the members of Row. Other data

members include scanners and functions for processing lists and, in this example, an output label for the Row.

- The constructor and destructor are not defined inline because they are long and contain loops. The destructor for a linked-list class needs to free all the storage allocated by the constructor and by other class functions that add items to the linked list.
- As always, a Print function (and a corresponding extension for <<) are defined for Rows. No such functions are defined for Cell because Row provides the interface for both classes.
- Many times, a container class is used to implement an active database, so it has functions to Insert, Delete, Modify, and Lookup items. In this program, the container is used as a way to sort exam scores, and needs only one database function: insert an Item.

Notes on the Row class implementation. All of the functions in the `row.hpp` file belong to the Row class because the Cell functions were all defined inline. This code file is the heart of the program.

- The Row constructor creates an output label for the row and initializes the pointer, Head, that will point at the list of Item Cells. Head is set to NULL because, at this time, the list is empty. A scanning pointer, Curr, is also set to NULL but could be left uninitialized.
- Following the expert pattern, the Graph::Insert function calls the Row::Insert function to do the actual insertion. In general, this is the way operations should be implemented: the task is handed down the line, from the user interface to the primary class, and from class to class below that, until it reaches a low-level class that knows how to carry out the task.

```
//===========================================================================
// Implementation of class Row.
// A. Fischer, April 22, 2000                                  file: row.cpp
// Modified M. & A. Fischer, September 17, 2009
//===========================================================================
#include "row.hpp"
//---------------------------------------------------------------------------
// Row number is used to construct a label for the row
Row::Row( int rowNum ){
    if (rowNum == 10) strcpy( label, "Errors:  " );
    else {
        strcpy( label, " 0.. 9:  " );
        label[0] = label[4] = '0'+ rowNum;  // example: label=="70..79"
    }
    head = NULL;  // represents empty list of data
}
//---------------------------------------------------------------------------
// Row is responsible for deleting everything created by this class
Row::~Row(){
    Cell* curr;
    while (head != NULL){
        curr=head;
        head=head->next;
        delete curr;
    }
    cerr << " Deleting row " << label <<" >>> \n";
}
//---------------------------------------------------------------------------
// Create and insert Cell into linked list at head
// Design pattern: creator.  Item is created by Cell constructor.
void
Row::insert( char* name, int score ){
    head = new Cell( name, score, head );  // put new cell at head of list
}
//---------------------------------------------------------------------------
// Design decision: print Cell data directly; no delegation of print
ostream&
```

```
Row::print( ostream& os ){
    Cell* curr;
    os << label;
    for (curr=head; curr!=NULL; curr=curr->next)
        curr->data->print( os );    //  OR: os << *(curr->data);
    return os;
}
```

- Row::Insert does not pass the insertion command on to Cell::Insert because Row is the lowest level class that is "expert" on multiple Cells. The Row class allocates a new Cell and passes the data to it because the Cell is the "expert" on Items. The Cell constructor finally creates an item and stores its pointer. In C, eight routine and tedious lines of code are needed to to the allocation and initialization:

```
Item* tempI = (Item*) malloc( sizeof(Item) );
strncpy( tempI->Initials, 4, name );
tempI->Score = score;
Cell* tempC = (Cell*) malloc( sizeof(Cell) );
tempC->Data = tempI;
tempC->Next = Head;
Head = tempC;
```

  A good demonstration of the power and convenience of C++ constructors is that they permit us to do the same job in one line: `Head = new Cell( new Item(name, score), Head );`

  In this program we do part of the construction in the Row class and part in the Cell class, following the principles of Creator and Expert:

```
In Row:    Head = new Cell( name,  score, Head );
In Cell:   Data = new Item( name, score );
```

- The Print function loops through the Cells on the linked list and sends a Print command to the Item in each cell. Arrows are used in this command (rather than dots) because each Item is attached to the Cell, and each Cell is attached to the Row through a pointer.
- The Row destructor uses a loop to delete, or free, all of the memory blocks allocated by Row::Insert. When it deletes each Cell, the Cell destructor will delete the Item it contains.

8.4 An Event Trace

In one common programming pattern, a program has five phases of operation:

- During the setup phase, the main program acquires resources and creates its primary data structure(s).
- During the input phase, files and interactive entry are used to store data in that structure.
- Processing accesses and modifies the information.
- The output phase prints an analysis of it.
- Finally, during the cleanup phase, the data is written back onto some permanent storage medium, streams are closed, and storage is freed.

In C, and even more in C++, some phases are often combined. For example, setup can be partially merged with input, and processing can be merged with either input or output. In the Bargraph program, there is no processing step. This leaves four major phases: setup, input, output, and cleanup. The event traces in the next four diagrams illustrate these four phases of the program's operation.

Interpreting a trace. An event trace provides a dynamic 2-dimensional view of the phases of program execution. It shows the order in which things are created and freed and how control passes back and forth between the classes. Figures 7.5. . . 7.8 show the event trace for the bargraph program.

- **The columns.** An event trace has dashed vertical lines. Each represents the "territory" of one program module: either main() or one of the classes used by the program.

- **Passing control.** Arrows represent the flow of control. When an arrow stops at a vertical line, control enters that class. The label on an arrow indicate the function call that caused control to leave one class and enter another.
- **Loops.** Loops are indicated by large ovals and are documented informally.

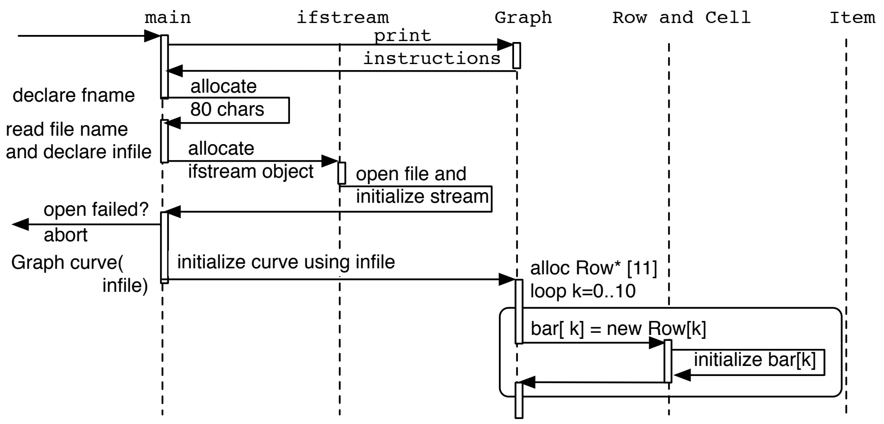


Figure 8.5: Event trace phase 1–Constructing the Graph.

- **The passage of time.** When the program begins, the system sends control into main() at the upper left corner of the graph. As long as control remains within the same class, time flows downward along its dotted line. A broad white time line indicates that control is within a class. Solid arrows represent function calls. As time passes, they take control from one class to another and also lead down the page.

Phase 1: Setup. During this phase (Figure 7.5), the main program interacts with the user to set up the input stream. Then two major data structures are declared and initialized: an istream named `input` and a Graph named `curve`. When curve is declared, the system creates storage for an array of eleven pointers, then calls the Graph constructor to initialize that array and construct its extensions.

During construction, space is allocated for the eleven rows of the graph. As each Row is created, the Row constructor is called to initialize it. (It creates a label for the row and sets the list pointers to NULL.)

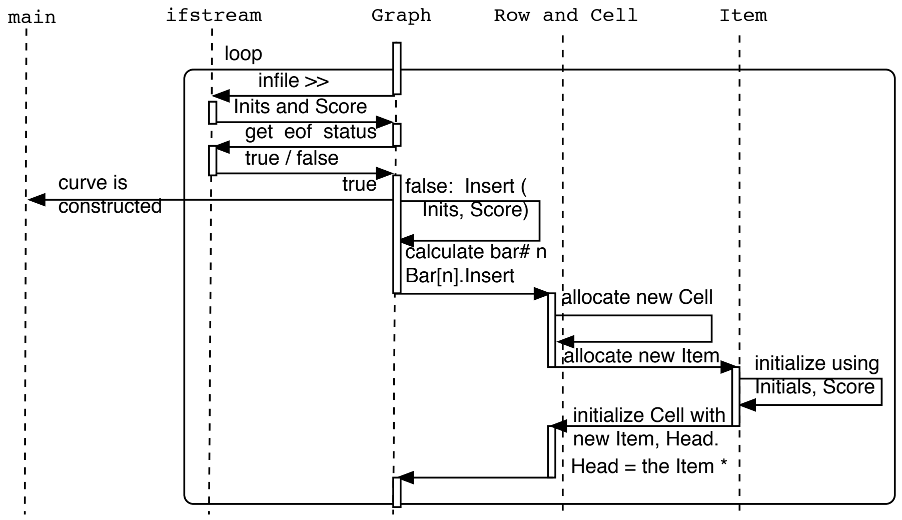


Figure 8.6: Phase 2, Filling the Graph.

Phase 2: Input. A loop is used to read data lines from the input stream. For each, storage is allocated, and the resulting structures are linked into the appropriate Row, at the head of its list of Cells. The new storage is

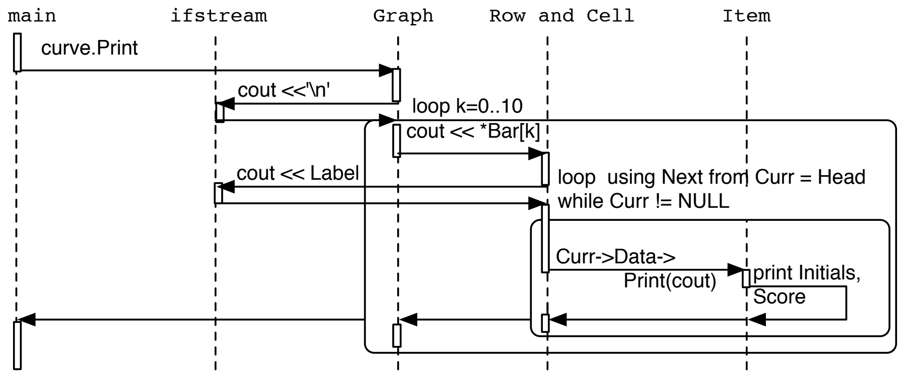


Figure 8.7: Phase 3, Output.

initialized by the constructors in the Row, Cell, and Item classes. (Figure 7.6) Nothing is deleted during this process because no extra or temporary locations are created. We are building a database; every object allocated gets connected to it and used later in the program.

Phase 3: Output. The purpose of this program is to organize the data into a meaningful format, a bar graph. The organization is done by the constructors during the input phase. No other processing is required, so printing the graph is the next step (Figure 7.7). Most programs print header and footer titles. Here, none are required by the specifications, so we print the graph by simply printing each of its 11 rows. To do so, we use the extended << operator, which calls on Row::print() (the expert on displaying rows) to do the job.

After Row::print() prints the row labels, it must print the data. Since Item::print() is the expert on how an Item should look, Row::print() calls it to print the Item in each Cell, starting with the Head cell and ending with NULL.

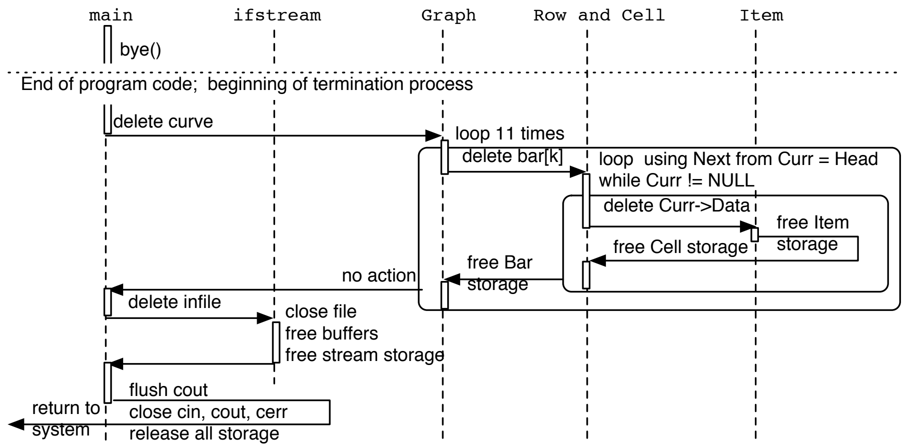


Figure 8.8: Phase 4, Deallocation.

Phase 4: Deallocation. When the data in the primary data structure has been created or modified during execution, it must usually be written to a file at the end. It is appropriate to have the destructors write the output file during the process of freeing the memory. This has the nice property that the data resides in exactly one place at any time; there is always one copy of it, not two, not zero.

In this program, no data was created during the run, so no output file is needed. The destructors are therefore simple: they call delete for each pointer that was created by new, and finish by printing trace comments.

Declared objects are deleted automatically in the order opposite their creation order (Figure 7.8). Thus, the Graph is deleted, then the istream, then the character array. The trace shows calls on destructors for the class objects.

Chapter 9: Array Data Structures

A fundamental principle of good construction and good programming:

Function dictates form. First, get the right blueprints.

This chapter covers some major array-based data structures that have been or will be used in the sample programs and/or assignments in this course. These are:

1. Arrays of objects and arrays of pointers.
2. The resizeable array.
3. The stringstore, an efficient way to store dynamically allocated strings.
4. The hashtable.

At the end of the chapter, these data structures are combined in a hashing program that uses an array of pointers to resizeable arrays of string pointers.

General Guidelines. Common to all data structures in C++ are some fundamental rules and guidelines about allocation and deallocation. Here are some guidelines, briefly stated: **The method of deallocation should imitate the method of creation.**

- If you allocate memory with new (and no []), you should free memory with delete.
- If you allocate memory with new in a loop, you should free memory with delete in a loop.
- If you allocate memory with new and [], you should free memory with delete [].

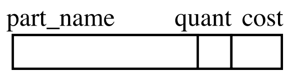
Basic Rules. The guidelines are derived from more fundamental rules about how C++ memory management works:

- If you allocate memory with a declaration, is is allocated on the run-time stack and is freed automatically when control leaves the block that contains the declaration.
- Dynamic memory is allocated using `new` and freed using `delete`. It is allocated in a memory segment called "the heap". If it is not freed, it continues to exist in the heap until program termination.
- If the base type of an array is a class type, you must use delete[] to free it. This automatically calls the base-type destructor function for each element of the array, to free the dynamic extensions attached to it.
- As a general habit, use delete [] to free arrays, even when the [] are not needed. Some programming tools and IDE's become confused when you omit the []. Note, however, that if the base type of an array is a pointer type or a non-class type, it does not matter whether you use delete or delete[] because there are no destructors to run. Dynamic memory attached to the elements of an array of pointers must be freed by explicit calls on delete.
- In one common data structure, the flexible array, an array "grows" by reallocation, as needed. After growth, the array contents are copied from the old array to the new, longer array. In this case, you must free the old array but you must not free anything more than that, so delete must be used without the brackets.

Which should I use? A fundamental dat modeling decision is whether to create the program's objects using declarations (stack allocation) or using pointers and dynamic (heap) allocation. Even in a simple data structure, such as an array of structures, four basic combinations of dynamic and stack allocation are possible, and all are commonly used. We look at these next.

9.1 Allocation and Deallocation of Arrays.

In this section, we will be using arrays of a representative class type named Item, which has three data members, as shown below.



```
class Item {
    char part_name[20];
    int quant;
    float cost;
  public:
    Item();
    ~Item();
};
```

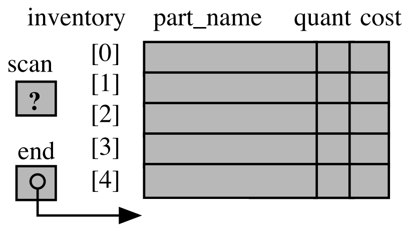
Four array data structures are presented, all of which implement some form of two-dimensional structure. However, they have different storage requirements and dynamic possibilities. It is important for you to understand how they differ theoretically and practically, how to create and use each one, and when each one might be preferred.

A diagram is given of the storage allocated for each data structure. Core portions of these objects are allocated on the run-time stack and are colored gray. Extensions are allocated in the heap and are colored white. To the right of each diagram is a code fragment that shows how to use new and delete to create and deallocate the structure.

The first data structure given is a simple array of Items whose size is fixed at compile time. Each succeeding example adds one level of dynamic allocation to the data structure, until the last is fully dynamic. For the sake of comparison, these four data structures are implemented as uniformly as possible. All are five slots long and have offboard end pointers to assist in sequential array processing. (Note: inventory+5 is the same as &inventory[5].)

1. **A declared array.** This data structure is used if the length of the array is known at the time control enters the block containing the declarations. All storage is allocated by the array declaration, so no "new" is necessary. For an array of class objects, a default constructor must be provided and will be used to initialize the array.

   No "delete" is necessary and using "delete" or "delete[]" is a fatal error because this storage will be automatically deallocated at the end of the function that contains the declarations and storage must not be deallocated twice.

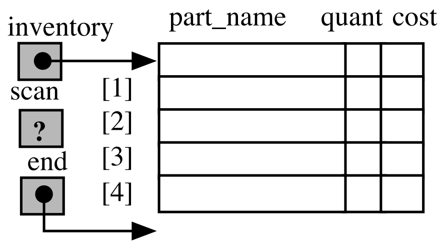


```
Item  inventory[5];          // Allocate array.
Item* end = &inventory[5];   // Connect end pointer.
Item* scan;                  // Processing pointer.
```

2. **A single dynamic array of class objects.** This data structure is used if the length of the array is not known at compile time but is known at run time, before any data is stored in the array. It could be combined with automatic reallocation to store data sets that might grow larger than the initially allocated size. However, this requires that all data in the array be copied into the reallocated memory area, a potentially large cost.

   The core portion of this object is created by declaring two Item pointers. The extension is created by a call on "new", often in the constructor for some class that contains the array. A default Item constructor must be present and will be used to initialize the array. Dynamic arrays of non-class base types cannot be initialized.

   Since "new" was used with [] to create this array, "delete[]" must be used to deallocate it.

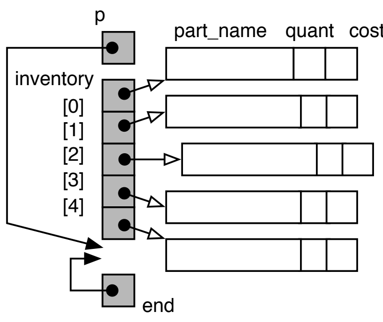


```
Item* inventory;          // Head pointer (core).
Item* end;                // Sentinel pointer (core).
Item* scan;               // Processing pointer (core).

inventory = new Item[5]; // Allocate array (extension).
end = &inventory[5];     // Connect end pointer.

delete[] inventory;      // Delete the array.
```

3. **A declared array of pointers to individual dynamic objects.** This data structure can be used if the Item constructor needs parameters or if the maximum amount of storage needed is predictable, but you wish to avoid allocating the storage until you know it is needed. An array of Item pointers is pre-allocated and initialized to NULL. A new Item is allocated and attached to the data structure each time new storage is needed. Any constructor (not just a default constructor) can be used to initialize these items because they are individually created. Since "new" was used without [] to create the Items, "delete" must be used without [] to deallocate them. Since a loop was used for allocation, a loop must be used for deallocation.

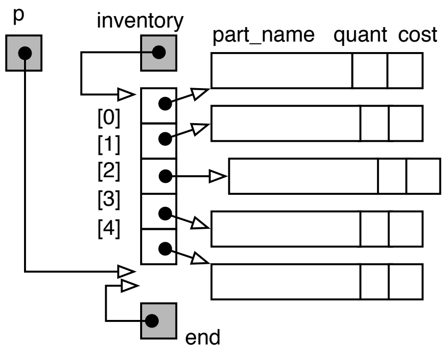


```
Item* inventory[5];                    // Create core.
Item** end = &inventory[5];            // Tail sentinel.
Item** p;                              // Scanner.

for (p=inventory; p<end; ++p) {        // Allocation.
    // Read and validate data.
    *p = new Item(field_1...field_n); // Install data.
}
for (p=inventory; p<end; ++p) {
// Deletion loop.
    delete *p;
}
```

4. **A dynamic array of pointers to individual dynamic objects.** This data structure is used when the amount of storage needed is unpredictable. An Item** is declared and initialized at run time to a dynamic array of Item pointers, each of which should be initialized to NULL. By using a flex-array data structure, the array of pointers can be easily lengthened, if needed. The cost of the reallocation and copy operations is minimal because only the pointers (not the Items) need to be copied.

   Another Item is allocated and attached to the data structure each time new storage is needed. Any constructor (not just a default constructor) can be used to initialize these items because they are individually created. Since "new" was used without [] to create the Items, "delete" must be used without [] to deallocate them. Since a loop was used for allocation, a loop must be used for deallocation. After the deallocation loop, the main array (the backbone) must also be freed by using delete with [].



```
Item** inventory = new Item*[5];       // Allocate *s.
Item** end = &inventory[5];            // Tail pointer.
Item** p;                              // Scanner.
for (p=inventory; p<end; ++p) {        // Allocation.
    // Read and validate data.
    *p = new Item(field_1...field_n);  // Install data.
}

for (p=inventory; p<end; ++p) delete *p;
delete[] inventory;
```

9.2 The Flexible Array Data Structure

A flexable array is a container class, that is, a class whose purpose is to contain a set of objects of some other type. The C++ standard template library (formerly called STL) contains a template class, named `vector`, from which a flexible array of any type may be created. Java supports a type named `vector` and another named `arraylist`, that are basically similar. All of these classes are based on a dynamically allocated array that is automatically reallocated, when necessary to contain more data. Class functions are used to store data in the container and retrieve it.

A flexible array is a good alternative to a linked list for applications in which an array would be appropriate and desirable but the amount of storage needed is unpredictable. It works especially well when sequential (not random) access is used to add data to the array, as in the simple program below. In this section, we present a simple version named FlexArray, which does not rely on the use of templates. We then use a FlexArray to create an array of characters that can hold an input string of any length.

9.2.1 Implementation in C++

The flexible array data structure consists of a pointer to a long dynamically-allocated array for data, an integer (the current allocation length), and the subscript of the first unused array slot. When there is data to be stored in the container, but it is full, the flexible array automatically doubles in length. The cost is a modest amount of time spent reallocating and copying storage. This can easily be less than the time required to manipulate links and traverse lists. Wasted space is minimal for a flexible array, often less than for a linked structure, but the major advantage of the flexible array is that a programmer can use array-based algorithms (quicksort) and random access to process the data in the array.

The class FlexChars is an implementation of a flexible array with base type char. It is a good example of how a class can (and should) take care of itself. It has one private function, `grow()` (lines 14 and 68–76), which is called whenever there is a object to store in the FlexChars, but the array storage space is full. The existence and operation of this function is completely hidden from the client program; the data structure simply "grows" whenever necessary.

Before enlarging the array, we test the value of Max. If it is a legal array length, it is doubled; otherwise it is set to the default length. The array is then reallocated at the new, longer, length, the data is copied from the old array into the new one, and the old one is freed.

The FlexChars class declaration.

- Lines 17–21 define the constructor. Note that the parameter `ss` has a default value of `FLEX_START`. This permits a client program to call the FlexChars constructor with or without an argument.
- Line 22 is the destructor. Note that it tests for a NULL pointer before attempting to free the array, and it uses `delete[]` because the corresponding constructor used `new` with `[]`.
- Two "get" functions, `length()` and `message()` (lines 28 and 29) provide read-only access to the number of chars actually stored in the flexible array and to the character string itself..
- A print function is (lines 31. . . 34) allows the FlexChars client program to print the contents of the array without knowing how it is implemented.
- The `put` function (lines 25 and 61–66) allows us to store information in the next free slot of the FlexChars. It checks for available space, calls `grow()` if necessary, stores the data, and increments `N`, the address of the first available slot. This function provides the only way to store new data in the FlexChars.
- The `grow` function (lines 14 and 68–76) is the heart of this class. It is called from `put` when the array is full and more input must be stored. The diagrams on the next page illustrate its operation, step-by-step.
- The extension of the `subscript` operator (lines 26 and 55–59) is not used in this application but will be used later in the chapter. It will be explained in Chapter 10.

```
 1  //-----------------------------------------------------------------------
 2  // Class declaration for a flexible array of base type char.
 3  // A. Fischer, February 7, 2003                          file: flexChars.hpp
```

```
#ifndef FLEXC
#define FLEXC
#include "tools.hpp"
#define CHARS_START 4       // Default length for initial array.

class FlexChars {
  private: // -----------------------------------------------------------
    int Max;        // Current allocation size.
    int N;          // Number of array slots that contain data.
    char* Data;     // Pointer to dynamic array of char.
    void grow();    // Double the allocation length.

  public: // ------------------------------------------------------------
    FlexChars( int ss = CHARS_START ) {
        Max = ss;                // Current allocation length
        N = 0;                   // Number of data items stored in array.
        Data = new char[Max];    // An array to contain objects of type char.
    }
    ~FlexChars() { if (Data != NULL) delete[] Data; }

    void inputLine( istream& ins );
    int  put( char data );       // Store data and return the subscript.
    char& operator[]( int k );

    int  length() { return N; }  // Provide read-only access to array length.
    const char* message() { return Data; }  // Read-only access to the chars.

    ostream& print(ostream& out) { // Print the filled part, ignore the rest.
        Data[N] = '\0';
        return out <<Data;
    }
};
inline ostream& operator<< (ostream& out, FlexChars& F){ return F.print( out ); }
#endif
```

The FlexChars class implementation.

```
//-------------------------------------------------------------------------------
// Implementation of the FlexChars class.
// A. Fischer, February 7, 2003                              file: flexChars.cpp

#include "flexChars.hpp"
void  FlexChars ::   // ------------------- Read and return one line from input stream.
inputLine( istream& ins ){                 // Discard prior contents of the buffer.
        char ch;
        for (N=0; ; ){                     // Start with slot 0 of the buffer.
            ins.get( ch );
            if (!ins.good() || ch == '\n') break;  // Go until eof or newline.
            put( ch );                     // The put function takes care if itself.
        }
        put( '\0' );                       // Terminate the string in the buffer.
        //cerr <<"Input length = " << N <<endl;
}

char& FlexChars :: //------------------------------- Access the kth char in the array.
operator[]( int k ) {
    if ( k >= N ) fatal("Flex_array bounds error.");
    return Data[k];                         // Return reference to desired array slot.
}

int FlexChars :: // --------------------------------------- Copy a char into the array.
put( char ch ) {
    if ( N == Max ) grow();     // Create more space if necessary.
    Data[N] = ch;
    return N++;                 // Return subscript at which item was stored.
}
```

```
67
68  void FlexChars :: // ----------------------------------- Double the allocation length.
69  grow() {
70      char* temp = Data;                      // hang onto old data array.
71      Max>0 ? Max*=2 : Max = CHARS_START;
72      Data = new char[Max];                   // allocate a bigger one.
73      memcpy(Data, temp, N*sizeof(char));     // copy info into new array.
74      delete temp;                            // recycle (free) old array.
75                  // but do not free the things that were contained in it.
76  }
```

The main program. The main program for this example shows a simple application for a flexible array: reading a line of input whose length is not known ahead of time. The main program creates a FlexChars named `sentence` (line 87) and uses it to read one line of input. The input is then echoed, with its length, as shown below:

```
77  //==========================================================================
78  // Using a FlexChars of characters.
79  // A. Fischer, May 4, 2002                                 file: flex_test.cpp
80  //
81  #include "tools.hpp"
82  #include "flexChars.hpp"
83  //--------------------------------------------------------------------------
84  int main( void )
85  {
86      banner();
87      FlexChars sentence; // A FlexChars of 20 chars.
88
89      cout <<"\nFlexible Array Demo Program \nEnter a long sentence:\n";
90      // Read an entire line of text; the buffer will grow when necessary.
91      sentence.inputLine( cin );
92
93      // End of input phase -- ready for output.
94      cout <<"\nSentence length is " <<sentence.length() <<endl;
95      cout << sentence.message() <<endl;  // Print an ordinary string.
96  }
```

The output:

```
Flexible Array Demo Program
Enter a long sentence:
Today is May 4.  It has been a lovely day outside -- I planted some flowers and enjoyed the soft,
warm, spring air.   Now evening is coming and the sun shines only on the treetops.

Sentence length is 181
Today is May 4.  It has been a lovely day outside -- I planted some flowers and enjoyed the soft,
warm, spring air.   Now evening is coming and the sun shines only on the treetops.
```

Tracing the operation of the FlexChars. The FlexChars data structure created by `main()` is illustrated in five stages of growth during the input operation. The object named sentence is shown on the left side of Figure 8.1, just after the constructor is done initializing it (line 21). The initial length of the array is `STARTSIZE`, or 4 characters.

Inside the `inputLine()` function, a loop is used to read characters one at a time and put them into the flexible char array. (A newline or eof condition will end the loop.) The right side of Figure 8.1, shows `sentence` again after four characters have been put into it, making it full.
When the command to put a fifth character into `sentence` is given, the FlexChars must grow. (Figure 8.2, left). This is done in the `grow()` function. In the C++ implementation, the Max length is doubled and a new data array of the new, larger length is allocated. then the data is copied from the old array to the new and the old array is deleted. Finally, the fifth character is stored in the newly-lengthened array.

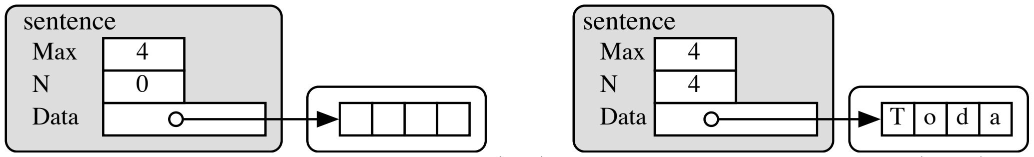


Figure 9.1: An empty FlexChars object (left) and one that is ready to grow (right).

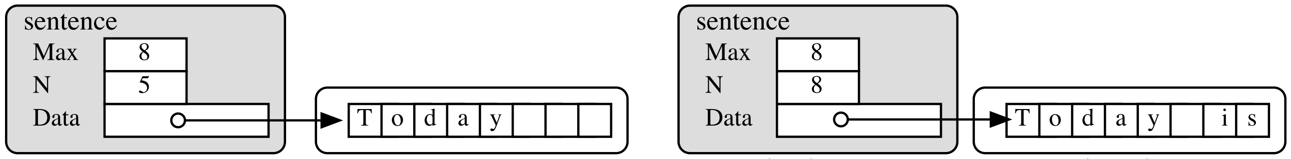


Figure 9.2: A FlexChars object after first growth spurt (left) and ready for second (right).

Input then continues normally, filling the rest of the slots (Figure 8.2, right). When the command is given to store the ninth character, the FlexChars must grow again. The doubling process is repeated, resulting in an array of 16 characters. The ninth character is then stored in the new space and input continues. Figure 8.3 shows the array after the tenth character is input, with ten slots in use and space for six more letters.

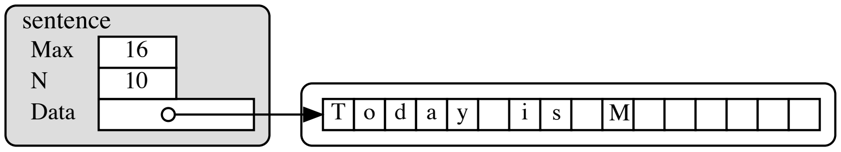


Figure 9.3: The same FlexChars object after doubling the second time.

Costs. This is an acceptably efficient algorithm. After the first doubling, the number of array slots allocated is always less than twice the number in use, and the total number of objects that have been copied, including all copy operations, is slightly less than the current length of the array.

Caution. Because the entire array is moved during the growth process, any pointers that point *into* the array must be re-attached after the reallocation. Because of this, a flexible array is not an appropriate data structure when many pointers point into an array. Pointers pointing *out of* the reallocated array are not a problem.

9.2.2 Implementation in C

In C, the doubling is accomplished using the `realloc()` function:

```
data = (char*)realloc( data, new_size );
```

This function takes any memory area that had been created by `malloc`, `calloc` or `realloc` and changes its size to the new size requested. If the new size is shorter, the excess memory is returned to the system for future re-use. If the new size is longer and the adjacent memory area is free, the existing area is simply lengthened. Otherwise, a new, larger area is allocated somewhere else in memory and the contents of the array are copied into the new area. This is easier and more efficient than the C++ implementation, in which the reallocation and copying must be done step-by-step.

9.3 Ragged Arrays

A ragged array is a convenient data structure for representing any list of phrases such as a menu, labels for a graph, or error comments. It has two advantages over the traditional 2-dimensional array of characters.

- Since the length of the strings in the ragged array can vary, the space they take is kept to a minimum. (Each phrase requires one byte per letter, one for the '\0', and four bytes of overhead.)
- Dynamic modification of the data structure is possible. The strings can be allocated and connected as needed and can be repositioned by swapping pointers.

Figure 8.4 illustrates the two kinds of ragged array: a constant version built from literals, and a dynamic version that can contain user input.

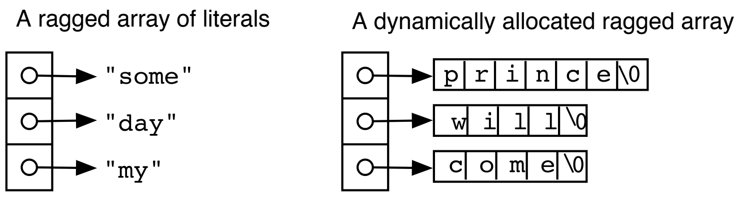

A ragged array of literals
"some"
"day"
"my"
A dynamically allocated ragged array
p r i n c e \0
w i l l \0
c o m e \0

Figure 9.4: Two kinds of ragged arrays.

A literal ragged array is most easily constructed by a declaration with a list of literals as the initializer:

```
char* vocabulary[] = { "Some", "day", "my" };
```

9.3.1 Dynamic Ragged Arrays

A dynamic ragged array is straightforward to build. Each "rag" is constructed separately and attached to a slot in a "backbone" array, which could be either declared or allocated dynamically. Several steps are required to construct a rag, as shown by the `getRag()` function:

```
 97  // ---------------------------------------------------------------------------
 98  // Read one string and store in dynamic memory.             file: getRag.cpp
 99  //
100  #include "tools.hpp"
101
102  char* getRag( istream& fin )
103  {
104      static char buf[80];        // Input buffer limited to 80 chars.
105      int len;                    // Length of input phrase.
106      char* item;
107
108      fin.getline( buf, 80 );     // Don't read past end of buffer.
109      len = fin.gcount();         // Measure the input string.
110      item = new char[len];       // With getline, len is always >= 1.
111      strcpy( item, buf );        // Copy input into the new space.
112      return item;                // Return dynamic storage containing input.
113  }
```

Attaching the rags. The following main program uses `getRag()` in a common way: to create a list of options from a file. This approach is better than building the menu into a program because it allows the menu to be changed easily.

For the backbone of the data structure, this main program uses an array with a fixed maximum size (line 124). However, a flexible array could be used and would often be a better choice. The main loop calls `getRag()` repeatedly (line 132) and attaches the resulting rags to the backbone. Filling all the array slots (line 131) or an end-of-file condition (line 133) ends the input loop. Finally `main()` prints the complete list (line 135).

Note that `main()` is the function that tests for eof, not `getRag()`. This is a common pattern in C++ because many input functions read only a single data set and deliver the resulting single object to a client program. The input function takes a stream as a parameter and does not know what kind of stream that might be: an istream or an ifstream or a istringstream.

In contrast, the calling function must get and store many inputs. It knows where the inputs come from and how many can be handled. Therefore, the calling function has the responsibility for ending an input loop.

```
114  #include "tools.hpp"
115  #define INPUT "menu.in"
116
117  char* getRag( istream& );
118  void print_menu( char*[], int );
```

```

// -----------------------------------------------------------------
int main (void)
{
    int k;                          // Line counter.
    char* menu[20];                 // Backbone of ragged array.

    cout <<"Ragged Array Demo Program\n";
    ifstream menufile( INPUT );     // Open file named by define.
    if (!menufile) fatal( "Cannot open file %s." INPUT );

    cout <<"Reading menu items from " <<INPUT <<"\n";
    for(k=0; k<20; ++k)  {
        menu[k] = getRag( menufile );
        if (menufile.eof()) break;  // User decided to quit.
    }
    print_menu( menu, k );
    return 0;
}

// -----------------------------------------------------------------
void print_menu( char* menu[], int n )
{
    int k;                          /* subscript variable for loop */
    cout <<"\n" <<n <<" skating sessions are available:\n";
    for (k = 0; k < n; ++k)  cout << '\t' <<menu[k] <<endl;
}
```

9.4 The StringStore Data Structure

Each time we allocate space dynamically, the C and C++ systems allocate extra bytes to store the length of the allocation. This can use a significant amount of storage if many separate small allocations are done. A StringStore is a more efficient way to store an unlimited number of variable-length strings, such as might be needed for a dictionary or a long menu. This data structure consists of one or more long dynamically allocated arrays called "pools", each of which will hold many strings. When the first pool is full, another is allocated. These pools are linked together to facilitate deallocation at the end of their useful lifetime. Figure 8.5 shows the first pool allocated for a stringstore, pointing at NULL. The oldest pool will always be at the end of the list of pools; the newest pool will be at the head of the list.

A StringStore has three fields: a pointer to the beginning of the most recently allocated pool, the amount of space remaining in that pool, and the subscript of its first unused slot. Figure 8.6 shows a StringStore with one Pool.

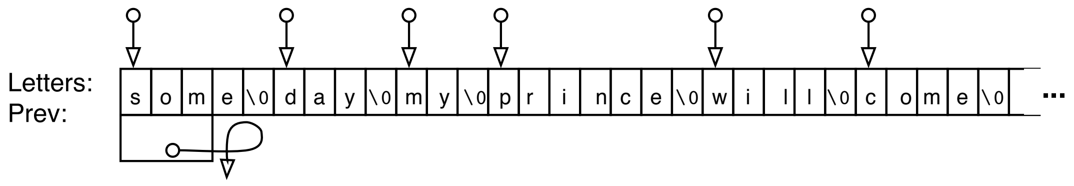

Letters:
Prev:
s o m e \0 d a y \0 m y \0 p r i n c e \0 w i l l \0 c o m e \0 ...

Figure 9.5: A Pool that contains 6 words.

9.4.1 The StringStore and Pool Classes.

```
//===========================================================================
// Class declarations for StringStore and Pool.
// A. Fischer, June 4, 2000                                  file: sstore.hpp
#ifndef SSTORE
#define SSTORE

#include "tools.hpp"
```

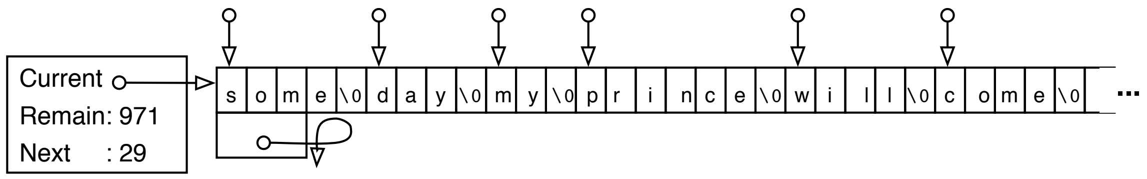

Current
Remain: 971
Next : 29
s o m e \0 d a y \0 m y \0 p r i n c e \0 w i l l \0 c o m e \0 ...

Figure 9.6: A StringStore with one Pool.

```
 8  #define MAXLETS  1000   // Pool size.
 9  //=====================================================================
10  class Pool {
11    friend class StringStore;
12    private:
13      char  Letters[MAXLETS]; // Memory behind stringarray.
14      Pool* Prev;             // Previous StringStore, for deallocation.
15
16      Pool( Pool* pv = NULL ) { Prev = pv; }
17      ~Pool(){}
18  };
19
20  //=====================================================================
21  class StringStore {
22    private:
23      Pool* Current;  // Pool that is currently being filled.
24      int Remain;     // Space remaining in current Pool.
25      int Next;       // Subscript of next available slot.
26
27    public:           //-----------------------------------------------
28      StringStore( int sz = MAXLETS ): Current(new Pool), Remain(sz), Next(0) {}
29      ~StringStore();
30      char* put( const char* s, int len );
31  };
32  #endif
```

The advantage of this structure over individually-allocated strings is that both space and time are used efficiently. There are fewer calls for dynamically allocation, so there are fewer areas to delete later and less space wasted in overhead. The disadvantage is that, if strings are deleted out of the middle of the array, the space cannot be easily reclaimed or reused. Thus, the data structure is appropriate for relatively stable lists of alphabetic information.

The StringStore interface. A StringStore serves only one purpose: to store a large number of strings efficiently, on demand. The strings are not accessed through functions of the StringStore class. Instead, the class that calls StringStore::put() receives a pointer to the stored string and provides access to the string through that pointer. Thus, the only functions that are needed in this class are a constructor, a destructor, and the put function. During debugging, it might help to have a print function, but that would never be called by a client class.

Each StringStore class has one or more Pools associated with it. A one-many relationship of this sort is usually implemented by having the primary class point at a list of objects of the secondary class. Neither class makes sense without the other; they form a tightly coupled pair. There are two ways a tightly-coupled pair of classes can be implemented: the secondary class can be defined inside the primary class, or the secondary class can be declared to be a friend of the primary class. I dislike placing the definition of the helper class inside the interface class because it is wordy and syntactically awkward.

I prefer using friendship. It is easier to write and clearer to read. Friendship gives the primary (or interface) class unrestricted access to all of the members of the secondary (or helper) class. The helper class only needs constructor and destructor functions; all other functionality can be handled by the primary class. In my StringStore definition, the Pool class gives friendship to the StringStore (line 11).

```
33  //=====================================================================
// StringStore data structure -- implementation.
// A. Fischer, May 29, 2000                                  file: sstore.cpp

#include "sstore.hpp"
//-----------------------------------------------------------------------------
StringStore::~StringStore() {
    Pool* p;
    while (Current != NULL) {
      p = Current;
      Current = p->Prev;
      delete p;
    }
}

//-----------------------------------------------------------------------------
char*
StringStore::put( const char* s, int len ) {
    char* where;
    len++;                             // number of characters including '\0'.
    if ( len > Remain ) {              // no room to store s in current Pool.
        Current = new Pool( Current );  // Attach new pool at head of chain.
        Next = 0;                      // New pool is empty.
        Remain = MAXLETS;              // All slots are unused.
    }
    where = &Current->Letters[Next];   // Number of characters including '\0'.
    Next += len;                       // Move pointer to next empty slot.
    Remain -= len;                     // Decrease number of remaining bytes.
    strcpy( where, s );                // Copy the new string into the array.
    //cerr << where <<" ";
    return where;                      // First letter of newly stored string.
}
```

The StringStore implementation. Internally, the StringStore and pool classes follow the usual form for a linked list. The interface class, StringStore, points at a linked list of pools, and the pool class gives friendship to StringStore. Each pool contains a long dynamically allocated char array and points at the previously allocated pool (or NULL). Each char array is long enough for 1000 characters, and we expect to be storing words or phrases that vary from a few to about 30 characters. Thus, we expect to waste less than 15 chars at the end of each pool array, a modest amount.

As strings are read in by a client program, they are stored by calling StringStore::put(). This function copies the string into the next available slot of the current pool, updates `next` and `remain`, and returns a pointer to the beginning of stored string. The client must store this pointer in some other data structure such as a linked list, array, flex_array, or hash_table. If the space remaining in the current pool is not large enough to hold the current input string, another pool is allocated and attached to the head of the list of pools. The length of the pools should be great enough so that the space wasted at the end of each pool is only a small portion of the space allocated. One of the pointers at the top of this StringStore diagram will be returned each time StringStore::put() is called. The caller must store the pointer in some data structure such as an array or a linked list.

9.5 The StringArray

A StringStore can be used with any data structure that stores string pointers. Of these, the simplest is the StringArray, shown below, which is simply an array of string pointers. This data structures has two strong advantages over a linked list of words: the elements of an array can be searched and sorted much more efficiently than the same data stored in a linked list. In the diagram below, we show a StringArray named `SA`, which has a StringStore named `Letters` as its first member. The second member is an array of string pointers named `Words`. Each of the six items stored in `Words` is a pointer to one of the strings in `Letters`.
The program below illustrates the use of a StringArray to organize a set of pointers to words in a StringStore. It also demonstrates how to use the standard quicksort function (from stdlib.h) to sort that array.

Notes on the StringArray class declaration.

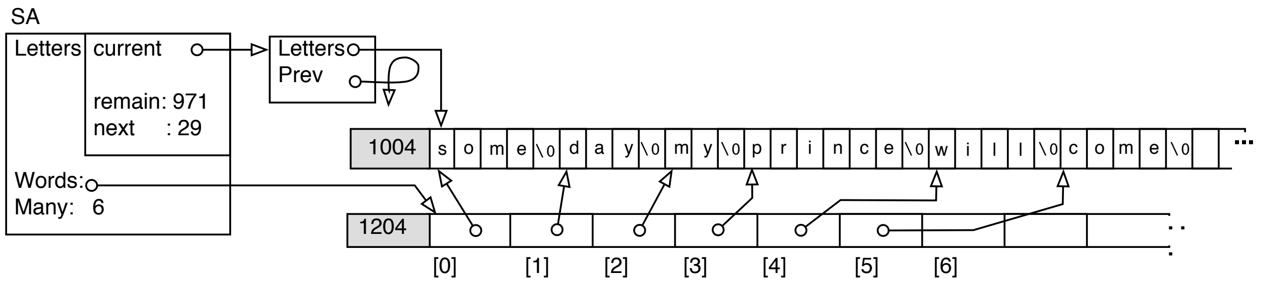


Figure 9.7: A String Array named SA.

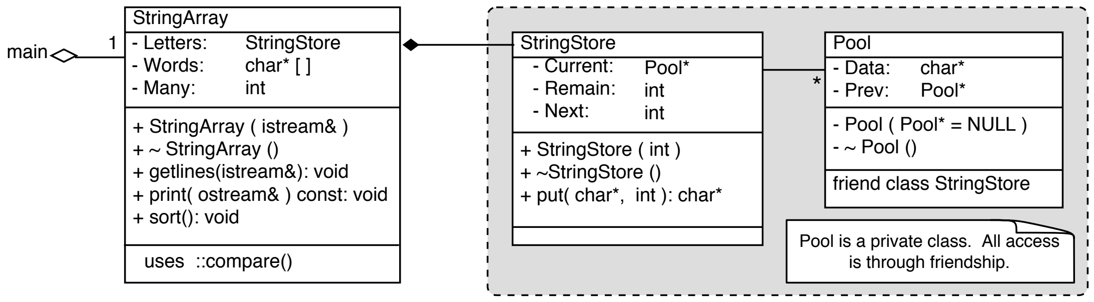


Figure 9.8: UML diagram for the String Array program.

- Three header files are included: `sstore.hpp` is needed because one member of this class is a StringStore. `FlexChars.hpp` is needed because we will use a flexible array as an input buffer. `compare.hpp` is needed because one of the StringArray class functions is `sort()` that cannot be part of the class because it must interact with which `qsort()`, a standard C library function.
- This implementation restricts the word list to 300 words. This is an arbitrary maximum, large enough to test the code but not large enough to be useful for most word lists. The `Vector` class or a flexible array of strings could be used to remove this limitation.
- The sort function (Line 87) is defined as a class member because it must refer to the data members of the class. It sorts the strings by calling `qsort()`. The arguments are (1) the array to be sorted, (2) the number of data items stored in that array, (3) the size of the base type of the array, and (4) a function that `qsort()` will call many times to compare pairs of array elements. The `compare()` function is a global function in the file `compare.hpp`.
- Line 85 is a do-nothing destructor. It is good style to define a destructor for every class, even if it does nothing.

```
65   //===========================================================================
66   // Class declaration for StringArray: an array of words using a StringStore.
67   // A. Fischer, October 3, 2000                                file: sarray.hpp
68   //
69   #ifndef SARRAY
70   #define SARRAY
71
72   #include "flexChars.hpp"
73   #include "sstore.hpp"
74   #include "compare.hpp"
75   #define MAXWORDS 300            // Limit on number of words in vocabulary.
76
77   //---------------------------------------------------------------------------
78   class StringArray {
79     private:                      //------------------------------------------
80       StringStore Letters;        // Storage behind string array.
81       char* Words[MAXWORDS];      // String array.
```

```
 82      int Many;                   // Number of words and first available slot.
 83    public:                       //-----------------------------------------
 84      StringArray( istream& in );
 85      ~StringArray(){}
 86      void print( ostream& out ) const;
 87      void sort(){ qsort( Words, Many, sizeof(char*), compare ); }
 88  };
 89  #endif

 91  //=====================================================================
 92  // Implementation of StringArray: an array of words using a StringStore.
 93  // A. Fischer, February 8, 2003                              file: sarray.cpp
 94
 95  #include "sarray.hpp"
 96  // ---------------------------------------------------------------------
 97  StringArray::StringArray( istream& sin )
 98  {
 99      Many = 0;                   // Array is empty; the first available slot is [0].
100      const char* word;
101      for (;;) {
102          FlexChars buf;          // Allocate a new flexible char array for the input.
103          buf.inputLine( sin );   // A flexible buffer has no fixed maximum length.
104          if ( sin.eof() ) break;
105          word = buf.message();
106          if ( word[0] != '\0') {
107              Words[Many++] = Letters.put( word, buf.length() );
108              //cout << "\n> " << word <<"\n";
109          }
110      }
111  }
112
113  // ---------------------------------------------------------------------
114  void StringArray::print( ostream& outs ) const
115  {
116      for (int k=0; k<Many; k++) cout << Words[k] << endl;
117  }
```

Notes on the StringArray class implementation.

- Line 103 reads the input into a temporary buffer; it is later put into a more efficient permanent data structure. Use of a flexible array for an input buffer enables us to safely see, validate, and measure any input string before we need to allocate permanent storage for it.
- Line 104 tests for the end of the file and breaks out of the loop if it is found. This is both correct and commonly done, even though some other function did the actual input.
- Line 107 copies the input word from the temporary buffer into the StringStore and saves the resulting pointer in the StringArray.
- The use of postincrement on the left side of line 107 is the normal and usual way to store a value in an array and update the array index to be ready for the next input.
- Line 108 is commented out; it was useful when I debugged this class.
- The `const` on lines 86 and 114 indicates that the `print()` function does not change the values of any class members. This is useful as documentation and as insurance that you have not accidentally assigned a value to a class member.

The compare() function.

```
#ifndef COMPARE
#define COMPARE
// ------------------------------------------------------------------------
// Called by qsort() to compare two strings (non-class objects).
// To compare two class objects, the last line would call a class function.
//
inline int
compare( const void* s1, const void* s2 )
{
    char* ss1 = *(char**)s1;
    char* ss2 = *(char**)s2;
    return strcmp( ss1, ss2 );      // This compares two non-class objects
}
#endif
```

- This function compares two strings in a way that is compatible with the standard qsort() function. The complexity of the call and the compare function are due to the general nature of `qsort()`: it can sort an array of any base type containing any number of data elements. Several casts and the use of pointer-pointers are necessary to achieve this level of generality.
- The parameters are type void*, which is compatible with any pointer type. The arguments will be pointers to strings, that is, char**. First, we need to cast the void* to type char**, to match the argument. Then, to use strcmp, we need to extract the char* from the char**.
- You can use `qsort()` and write `compare()` functions by following this example, even if you don't fully understand it. To sort a different data type, just change the char* to float or int or whatever you need, and change the strcmp to the appropriate comparison operator or function.

The main program.

```
//========================================================================
// StringArray, StringStore, and qsort Demo Program
// A. Fischer, February 8, 2003                    file: ArrayQsort/main.cpp
//========================================================================
#include "tools.hpp"
#include "sarray.hpp"

int main ( int argc, char* argv[] )
{
    banner();
    if (argc < 2) fatal( "Usage: ArrayQsort input_file");
    ifstream vocabIn( argv[1] );
    if (!vocabIn) fatal( "\nCould not open file vocab.in" );

    StringArray vocab( vocabIn );
    cout << "\nFile successfully read; ready for sorting.\n\n";
    vocab.sort();
    vocab.print( cout );
    return 0;
}
```

Notes on the main program. This program reads a list of words, prints it, sorts it, and prints it again. The relationships among the main program and all the functions called are illustrated by the call chart below. The main program follows the typical form. It includes only one application-specific header file (for the main data structure). Two objects (a stream and a string array) are declared. All work is delegated to class functions.

The output. The input file (left) and the output (right) have been arragned in two columns below, to fit the space. The banner and bye messages have been omitted from the output.

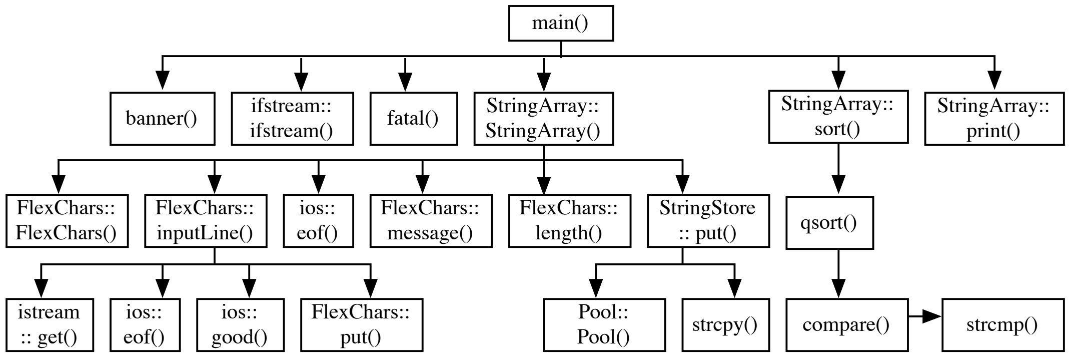


Figure 9.9: A call chart for the sorting example.

```
This is a file of strings.
It should end up sorted.
To demonstrate this, though,
I need several words.
This
is
the
end.
```

```
File successfully read; ready for sorting.

I need several words.
It should end up sorted.
This
This is a file of strings.
To demonstrate this, though,
end.
is
the
```

At the end of this chapter, we implement a Typing Tutor application using a hash table and a FlexString class, which is a built by combining the techniques of the flexible array and the String Array.

9.6 Hashing

A hashtable is a container class that can store and retrieve data items quickly and efficiently but does not keep them in a predictable or sorted order. The position of a data item in the hash table is derived from a key data field, or a key constructed from parts of several data fields. When implementing a hash table, three issues must be decided:

1. The backbone of a hash table is an array. How many slots will it have?
2. Each slot of the hash array is called a bucket. What data structure will you use to implement a bucket?
3. What algorithm should be used for your hash function? To assign an item to a bucket, a table subscript is calculated from the key field. The algorithm that does the calculation is called the *hash function.* It must be chosen to work well the expected data set. *Hashing* is the act of applying the algorithm to a data item's key field to get a bucket number.

In most hashing applications, an Item is a structure with several fields and the key field is just one of many. In such applications, distinct Items may have the same key. In this discussion, we imagine that we are hashing words for a dictionary. Since the Item will be a simple null-terminated string, the entire item will be the key field.

9.6.1 The Hash Table Array

Table structure. There are two basic ways to implement a hash table:

- Each bucket in the main hash array can store exactly one Item. Using this plan, the array must be longer than the total number of Items to be stored in it, and will not function well unless it is nearly twice that long. When a key hashes to a bucket that is already occupied, it is called a *collision* and some secondary strategy must be used to find an available bucket. Often, a second (and possibly a third) function is used; this is called *rehashing.*
- Each bucket in the main hash array can store an indefinite number of Items. Rehashing is not necessary; dynamic allocation takes its place. Ideally, the list of items in a bucket should be short so that it can be

searched quickly. This fact is used in calculating NBUK: if the goal is an average of B items per bucket, and we expect to have N items total, then NBUK should be approcimately N/B.

This kind of hash table is much easier to implement and will be used here. The bucket can be implemented by any dynamic data structure; most commonly used are linked lists, hash tables, and flex-arrays. In this discussion, we use a flex array to implement a bucket, as in Figure 8.6.

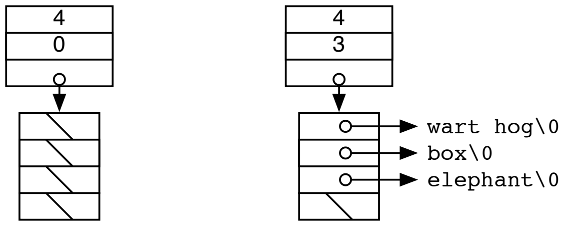


Figure 9.10: An empty Bucket (left) and one containing three Items (right).

Our chosen hash table implementaion is an array of NBUK buckets where each bucket is a flexarray, as shown in Figure 8.7. The buckets will be initialized in the Bucket constructor to be empty flex-arrays with 4 slots of type Item* where items can be attached later.

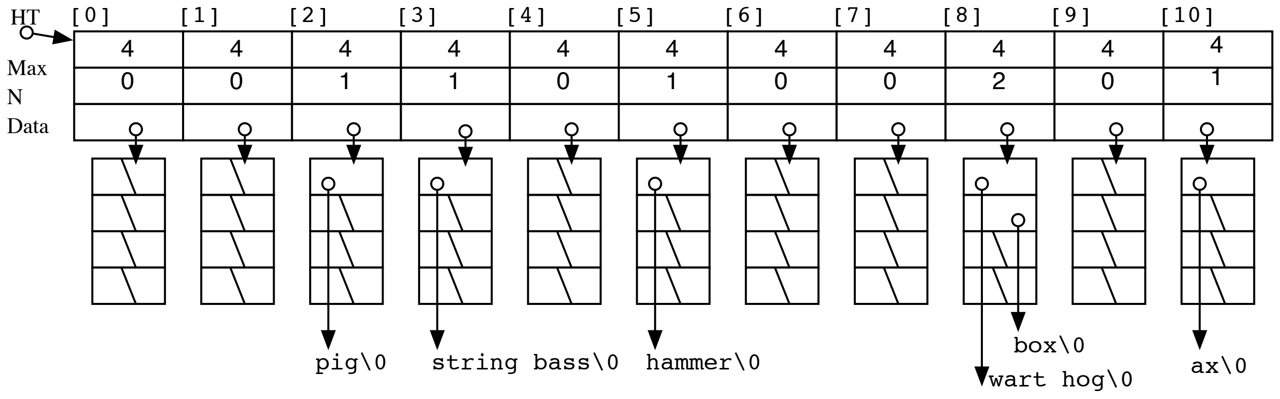


Figure 9.11: A HashTable with 11 Buckets and 6 Items.

Table length. For reasons related to number theory, and beyond the scope of this discussion, hashing works best when the number of the buckets is not a multiple of 2. Normally, the programmer would choose a large prime number for the table length. Here, we choose a small one so that the diagrams will fit on a page. When you implement your own hash table, start with a small number of buckets and do the initial debugging with a modest number of data items. When the program is ready for a full-scale trial, change the NBUK to a large prime.

Inserting a new Item. Suppose that D is a data item and that the program wants to insert it into a hash table. Before insertion, the program should search the table to determine whether it is duplicate data. If not, space whould be allocated for for the new data dynamically. The key field is then hashed to find the correct bucket number, and the FlexArray::put() function is used to insert the resulting pointer at the end of the appropriate list. Because a flex-array is being used, each list will automatically extend itself if it becomes overfull.

Retrieving an Item. To find an item in the table, we must hash its key field and search the correct bucket and search it for the desired key. Since every hash bucket should be fairly short, a simple sequential search is appropriate. If more than one item may have the same key field, more comparisons are needed to confirm whether or not the desired item (or another with the same key) has been located.

9.6.2 Hash Functions.

When choosing a hash function, several things are important.

1. It should use all of the information in the hash key, not just the beginning or end.
2. Its result must be a subscript between 0 and NBUK-1. This is easily accomplished by using the key to generate a larger number and using the *mod* operator (`%`) to reduce it to the desired range.
3. It should be efficient. Hash functions that make a lot of unnecessary calculations should be avoided. Bit operations (bitwise exor and shifts) are fast; addition and subtraction are slower, multiplication is slow, and division and mod are very slow. The last operation in the hash function must be mod, but, with that exception, the slower operators should be avoided. Remember—the object is to calculate a meaningless random number based on the key.
4. It should spread the data as evenly as possible throughout the table, that is, it should choose each bucket number equally often, or as near to that as possible.

Thus, the nature of the hash function depends on the data type of the key and the number of bytes in the key. Two sample hash functions are given below for strings; they are not perfect, but they give some ideas about what might be done. There is no limit on the number of possibilities.

Hash algorithm 1. This algorithm is appropriate for strings of 1 or 2 characters.

1. Initialize `hash`, an unsigned long, to the first char in the string.
2. Shift hash 7 bits to the left.
3. Add the second character.
4. Return `hash %` table length.

Hash algorithm 2. This algorithm can be used for strings of four or more characters. It uses the first four and last three letters of the string.

1. Set `len` = number of characters in the string, which must be four or greater. (Use `strlen()`.)
2. Set a pointer `p` to the first letter in the string.
3. Set a pointer `q` to the last letter in the string, = `&p[len-1]`.
4. Initialize an unsigned long, `hash` = `*p`;
5. Loop three times:
   (a) left shift `hash` 6 bits.
   (b) exor it with `*q` and decrement `q`.
   (c) left shift `hash` 3 bits.
   (d) exor it with `*(++p)`.
6. Return `hash %` table length.

9.7 Example: Combining Several Data Structures

This sample program is a typing tutor: it provides a list of target words to practice and uses a simple spelling checker to check the accuracy of the user's typing. Its real purpose, however, is to demonstrate the use of several of the data structures presented in this chapter:

- The spelling checker relies on a dictionary of words, which is implemented using a hash table.
- The hash table's buckets are an array of FlexArrays.
- The FlexArrays store pointers to a set of vocabulary words.
- The characters of these words are stored in a StringStore.

9.7.1 The Main Program

```
 1  //===========================================================================
 2  // HashTable, FlexArray, and StringStore Demo Program
 3  // A. Fischer, February 7, 2003                          file: Tutor/main.cpp
 4  //===========================================================================
 5  #include "tools.hpp"
 6  #include "dictionary.hpp"
 7
 8  int main ( int argc, char* argv[] )
 9  {
10      cout <<"Typing Tutor\n------------\n";
11      if (argc < 2) fatal( "Usage: tutor input_file");
12      Dictionary Dic( argv[1] );  // Create dictionary from words in the input file.
13      cout << "\n\nReady for typing practice? "
14           << "At each prompt, type one of the words shown above.";
15
16      int k, right = 0;           // Prompt for a word 10 times and count correct entries.
17      for (k=0; k<10; ++k) {
18          FlexChars buf;          // Allocate a new flexible char array for the input.
19          cout << "\n> ";
20          buf.inputLine( cin );   // A flexible buffer has no fixed maximum length.
21          if ( Dic.find(buf.message()) ) {    // Is input in dictionary?
22              cout << "\t Correct!";
23              ++right;                        // Keep score.
24          }
25          else {
26              cout <<"\t " << buf <<" is wrong.";
27          }
28      }
29      cout << "\nYou typed "<<right <<" words correctly, out of 10";
30      return 0;
31  }
```

Here is some sample output (several lines have been omitted):

```
Typing Tutor
------------
paper piper pauper pupa puppet pepper

Ready for typing practice? At each prompt, type one of the words shown above.
> paper
    Correct!
...
> puper
    puper is wrong.
> pepper
    Correct!
> payper
    payper is wrong.
You typed 8 words correctly, out of 10
Tutor has exited with status 0.
```

Notes on the main program. This main program is typical for an OO-application: it creates one object (a Dictionary, line 12) and uses that object in a simple way (line 21). Almost all of the work is done within the classes.

The name of the input file is supplied as a command-line argument so that it will be very easy to change the vocabulary. The main program picks up that argument and sends it to the Dictionary constructor (lines 11–12). When dictionary construction is complete, the user is prompted to enter a series of words from the given list, and the tutor checks each one for correctness and counts the correct entries. After ten trials, the typist's score is displayed.

The input buffer used here (line 18) is a flexible array of characters; this means that any input word, no matter how long, can be safely read. To test that property, I ran the program again and entered a string of 142 random characters. The program dutifully read and echoed them all, stated that the spelling was wrong, and went on with the game. Having an input buffer of unlimited length is overkill in this demo program. However, in a real application, being free from the need to limit the length of an input string can make it much easier for a programmer to achieve bug-free program operation. An added simplification is that a new FlexChars object is created for each input line, making it unnecessary to worry about reinitializing the data structure.

9.7.2 The Dictionary Class

```
32  //===========================================================================
33  // Class declarations for Dictionary.
34  // A. Fischer, April 25, 2001                              file: dictionary.hpp
35  #ifndef DICT
36  #define DICT
37
38  #include "tools.hpp"
39  #include "sstore.hpp"
40  #include "flexChars.hpp"
41  #include "flexString.hpp"
42
43  //===========================================================================
44  class Dictionary {
45    private:
46      StringStore SS;               // Vocabulary is stored in the StringStore.
47      FlexString* HT;               // Vocabulary is accessed using a HashTable.
48      ifstream fin;                 // Stream for reading the vocabulary file.
49      int NumBuk;                   // Number of buckets in the HashTable.
50      int CurrentBuk;               // Bucket currently in use; set by hash.
51      void hash( const char* s );   // Sets CurrentBuk to correct bucket for s.
52      void put( char* s );          // Put new words in the dictionary.
53
54    public:                         //-----------------------------------------
55      Dictionary( char* infile, int buckets=101 );
56      ~Dictionary(){ delete[] HT; }
57      bool find( const char* s );
58  };
59  #endif
```

This class implements the dictionary to be used by the typing tutor. Dictionary words are stored in a StringStore (SS) and organized by a hash table (HT) which is implemented as an array of FlexArrays. These powerful data structures are not needed for the little list of words used by the typing tutor demo. However, the hash table and StringStore would be essential tools in a real spelling checker that must handle thousands of words.

Line 104 is commented out, but was very helpful during debugging. You will see similar debugging lines, commented out, in several classes in this program. They have been left in the code as examples of the kind of debugging information that you are likely to need.

The data members of Dictionary All data members are private because all classes should protect data members. They should not be accessed directly by a client program.

Two purposes of the first two data members are obvious: `StringStore SS` and `FlexString* HT` implement the primary data structure of the dictionary. Note that the hash-table array is dynamically allocated (line 101). The third data member is an `ifstream`. It is part of this class because the vocabulary list that is in the input file is only relevant to the dictionary, not to the main application. Declaring an input or output stream within a class is common when the class ”owns” the stream. It must be a class member (rather than a local variable in one function) if more than one class function needs to use the stream.

```
60  //===========================================================================
61  // Class implementation for the Dictionary.
62  // A. Fischer, April 25, 2001                              file: dictionary.cpp
63
64  #include "dictionary.hpp"
65  //---------------------------------------------------------------------------
66  void
67  Dictionary::hash( const char* s ){
68      int len = strlen(s);
```

```
 69         const char* p = s;
 70         const char* q = &p[len-1];
 71         unsigned long hashval = *p;
 72         for(int k=0; q>=s && k<3; ++k){
 73             hashval = hashval << 6 ^ *q--;
 74             hashval = hashval << 3 ^ *(++p);
 75         }
 76         CurrentBuk = hashval % NumBuk;
 77     }
 78
 79     //-----------------------------------------------------------------------
 80     void
 81     Dictionary::put( char* word ){
 82         char* saved;            // Pointer to beginning of word in StringStore.
 83         if (!find( word )){
 84             saved = SS.put( word, fin.gcount() -1 );
 85             HT[CurrentBuk].put( saved );
 86         }
 87     }
 88
 89     //-----------------------------------------------------------------------
 90     bool
 91     Dictionary::find( const char* s ){
 92         hash( s );
 93         int result = HT[CurrentBuk].find( s );
 94         return result >= 0;         // -1 indicates not found.
 95     }
 96
 97     //-----------------------------------------------------------------------
 98     Dictionary::Dictionary( char* infile, int buckets ){
 99         fin.open( infile );
100         if (!fin)  fatal( "Cannot open %s for input - aborting!!", infile );
101         HT = new FlexString[ NumBuk = buckets ];
102
103         char buf[80];
104         //cerr <<"Reading words from file " <<infile <<".\n";
105         for (;;) {
106             fin >> ws;
107             fin.getline( buf, 80 );
108             if (!fin.good()) break;
109             put( buf );
110             cout << buf <<" ";
111         }
112         if ( !fin.eof() ) fatal( "Read error on tutor file" );
113         fin.close();
114     }
```

Using the state of the object. The last two data members, `Numbuk` and `CurrentBuk` are *state variables.* A state variable is a private data member that describe some essential property of the class or stores the current setting of some internal pointer or index. `Numbuk` is set by the constructor and used to compute every hash index.

In an OO-program, two functions in the same class often use state variables to communicate with each other. In this program, `CurrentBuk` stores the number of the bucket in which the current data item belongs. It is set by hash() and usedlater by find() and put(). This is more efficient and more secure than recalculating the bucket number every time it is needed.

Private function members of Dictionary.

- NumBuk (line 49) is the number of buckets in the hash-table array. The hash() function (lines 51 and 66. . . 77) selects a bucket number (0. . . NumBuk-1) by making a meaningless computation on the bits of the data to be stored in the table. These two class members are both private because they relate to the inner workings of the Dictionary class.

- A hashing function is supposed to compute a legal array subscript that has no apparent relationship to its argument. It must also be efficient and repeatable, that is, the same argument must always hash to the same subscript. However, if the hash function is called many times to hash different arguments, it is supposed to distrubute the data evenly among the legal bucket numbers. Bit operations (such as xor and shift) are often used to combine, shift around, and recombine the bits of the argument so that they end up in a seemingly-random (but repeatable) pattern (lines 73 and 74). The mod operator `%` is always used as the last step to scale the final number to the size of the array (line 76).

- The put() function (lines 52 and 80. . . 87) is used by the constructor to build the vocabulary list. It searches the table for a word, then puts the word into the table if it is not there already. This function is private because it is used only by the class constructor; it is not designed to be used by the client program. A more general implementation of a hash table would also provide a public function (with a reasonable interactive interface) that would allow the user to enter new words and store them in the Dictionary by calling this private put() function.

Public function members of Dictionary.

- The constructor. Given the name of a file containing vocabulary words, the Dictionary constructor builds and initializes a dictionary. There are two parameters: the input file name is required, and the length of the hash array is optional. If the optional integer is not supplied by main, in the Dictionary declaration, a default value of 101 will be used. (See Chapter 9, Bells and Whistles, for an explanation of default parameters.)

  This is a long function because it opens (lines 99–100), reads (lines 106–07), processes (lines 108–11), and closes (line 113) the input file. To process one word or phrase from the file, we first test for valid input, then delegate the rest of the job to the `put` function.

- The find() function (lines 57 and 90. . . 95) searches the dictionary for a given word and returns a boolean result. This is a public function becausethe purpose of the class is to allow a client program to find out whether a word is in the dictionary.

  To find out whether or not a word is in the dictionary, we must first select the right bucket then search that bucket. The `hash` function computes the number of the bucket in which the argument string, `s`, should be stored. We then *delegate* the remainder of the task, searching one bucket, to the `Bucket::find` function, and return the result of that function.

- The put() function. It is quite common to name a function `put` if its purpose is to enter a data item into a data structure. Following this naming convention, we would define put() instead of push() for a stack or enqueue() for a queue.

  This function takes care of its own potential problems. Because it is undesirable to have the same item entered twice into a hash table, the first action of `put()` (line 83) is to call `find` to check whether the new word is *already* there. If so, there is no further work to do. If not, two actions are required: to enter the word into the stringstore (line 84) using StringStore::put() , and to enter the resulting pointer into the appropriate bucket of the hash table (line 85) using FlexString:put(). Both actions are delegated to the `put()` functions of the servant classes.

- The destructor. The task of a destructor is to free any and all storage that has been dynamically allocated by class functions. In this class, only the constructor calls new, so the destructor has only one thing to delete. The StringStore was created by declaration as part of the Dictionary, and will be freed automatically when the Dictionary is freed, at the end of `main()`.

  The form of the delete command follows the form of the call on new: Line 101 allocates an array of class objects, so line 56 uses delete[] to deallocate that array; Line 101 stores the allocation pointer in HT, so line 56 deletes HT. Attempting to delete anything else, such as SS or CurrentBuk would be a fatal error.

9.7.3 The FlexArray and StringStore Classes

The code for StringStore was given earlier in this chapter. The code for a flexible array of characters (FlexChars) was also given. We use both classes here without modification.

For this application, we add another class, `FlexString`, a flexible array of C strings. It is like the `FlexChars` except:

- The base type of the flexible array is `char*`, not `char`.
- We added a function to search the array for a given string.
- We added an extension of the subscript operator to permit random access to the array.

The revised and extended class is given below. Later in the term we will see how two powerful techniques, templates and derivation, let us create variations of important data structures without rewriting all fo the code.

The FlexString class declaration. The basic operation of a flexible array was described in Section 8.2 and will not be repeated. However, some new techniques were used here and they deserve some mention. Note the extensive debugging print statements that were needed in the course of writing this code.

- The typedef (line 122). This is a flexible array of strings, or `char*`. The pointer to the head of the array is, therefore, a `char**`. We have defined a typedef name, `handle`, for this type because It is syntactically awkward and confusing to use types that are pointers to pointers. Traditionally, a *handle* is a pointer to a pointer to an object; this term is used frequently in the context of windowing systems.
- The constructor, Line 132, was rewritten to use ctor initializers instead of assignment to initialize the members of a new FlexString. Ctors will be described fully in a future chapter; at this time, you should just be aware that an initializer list (starting with a colon) can follow the parameter list of a constructor function. In the initializer list, member names are given, followed by initial values in parenthess. Following the list is the opening bracket of the body of the constructor function. More code can be given in the body, if needed, or it can be empty, as it is here.
- The find() function. When searching for one data item in an array of data items, you must use a comparison function. The function strcmp() is appropriate for comparing cstrings, so it is used here. The `find` function must compare its argument to elements in the array. Since the kind of comparison to be used depends on the base type of the array, we must either write a new `find` for every array base type or define a `find` that takes a comparison function as a parameter. We chose the first option here.
- The subscript function. Chapter 10 will describe operator extensions. Here, it is enough to note that we can extend the meaning of subscript to access array-like structures that are not just simple arrays. Like the built-in subscript function, the new function extension returns a reference, allowing us to either store data in the array or retrieve it.

```
115  //--------------------------------------------------------------------------
116  // Class declaration for a flexible array of base type T.
117  // A. Fischer, A. Fischer, February 7, 2003            file: flexString.hpp
118  #ifndef FLEXS
119  #define FLEXS
120  #include "tools.hpp"
121  #define STRINGS_START 20    // Default length for initial array.
122  typedef char** handle ;
123
124  class FlexString {
125    private: // --------------------------------------------------------------
126      int Max;                // Current allocation size.
127      int N;                  // Number of array slots that contain data.
128      handle Data;            // Pointer to dynamic array of char*.
129      void grow();            // Double the allocation length.
130
131    public: // ---------------------------------------------------------------
```

```
    FlexString( int ss = STRINGS_START ): Max(ss), N(0), Data(new char* [Max]){}
    ~FlexString() { if (Data != NULL) delete[] Data; }

    int put( char* data );          // Store data and return the subscript.
    int length() { return N; }      // Provide read-only access to array length.

    char*& operator[]( int k );     // Extend subscript operator for new class.
    int find( const char* );        // Return subscript of argument, if found.
    handle extract();               // Convert a flex-array to a normal array.

    ostream& print(ostream& out) {      // Print the filled part, ignore the rest.
        for (int k=0; k<N; ++k) out <<Data[k] <<endl;
        return out;
    }
};
inline ostream& operator <<( ostream& out, FlexString F ){ return F.print(out); }
#endif
```

```
//--------------------------------------------------------------------------
// Implementation of the FlexString class.
// A. Fischer, A. Fischer, February 7, 2003             file: flexString.cpp

#include "flexString.hpp"
int //------------------------ search for the word in the dictionary array.
FlexString::find( const char* word) {
    int k;
    //cerr <<"    Looking for " <<word <<"  " <<"N is " <<N <<"  ";
    for (k=0; k<N; ++k) {
        //cerr <<Data[k] <<"  ";
        if (strcmp( word, Data[k] ) == 0) break;
    }
    return (k < N) ? k : -1;
}

int // --------------------------------- Copy a char* into the FlexString.
FlexString::put( char* data ) {
    if ( N == Max ) grow();    // Create more space if necessary.
    //cerr <<"Putting " <<data <<" into bucket " <<N <<endl;
    Data[N] = data;
    return N++;                // Return subscript at which item was stored.
}

char*& //------------------------------- Access the kth string in the array.
FlexString::operator[]( int k ) {
    if ( k >= N ) fatal("Flex_array bounds error.");
    return Data[k];               // Return reference to desired array slot.
}

void  // ------------------------------------- Double the allocation length.
FlexString::grow() {
    handle temp = Data;                     // hang onto old data array.
    Max>0 ? Max*=2 : Max = STRINGS_START;
    Data = new char*[Max];                  // allocate a bigger one.
    memcpy(Data, temp, N*sizeof(char*));    // copy info into new array.
    delete temp;                            // recycle (free) old array.
                // but do not free the things that were contained in it.
}
```

9.7.4 A Better Way

The FlexChars class and FlexString class are the same except for the type of data item being stored in the array and minor differences in the functions provided by the class. Yet, because of these small differences, we have written and debugged the code twice and used both versions in the same program. There's got to be a better way to accomplish this goal. In fact, there are several C++ tools that, together, provide a better way.

1. If we want to use a standard data structure, and if we want only one version of that data structure, we can define the class with an abstract base type name, like T, and use a `typedef` at the beginning of the program file to map T onto a real type. This method was illustrated by the generic insertion sort in Chapter 2.

2. The C++ standard library supports several basic data structures in the form of *templates*. These templates are abstract code with type parameters, written using techniques very much like those used in the generic insertion sort. These templatea are often referred to as the "Standard Template Library", or STL. `Vector` is the template type corresponding to our flexible arrays. To use a template from the library, you must supply a real type in any the declaration that creates an object of the template class; technically, we say that you `instantiate` the template. The complier will combine your type with the template's abstract code at compilation time, to produce normal compilable code.

   STL is an amazing piece of work. It covers all of the most important data structures, and brings them into a unified interface, so that a program written using one data structure could be changed to use another simply by changing one line of code: the line that instantiates the template. The implementations it provides are efficient and correct, and their performance characteristics are thoroughly documented.

3. You could write your own template type. Templates will be covered in Chapter 13, where we will revisit the flexible array class.

4. You could start with a standard template type or with your own template, and use derivation to add more functions to the class. Such functions might be needed for some base types but not for others. Derivation will be covered in Chapters 12, 14, 15, and 16.

Chapter 10: Construction and Destruction

The way of existence in C++:

> **Because of my title I was the first to enter here. I shall be the last to go out.**—The Duchess d'Alencon, refusing help during a fire at a charity bazaar in 1897.

10.1 New C++ Concepts

Dumps. Some of the most troublesome program bugs are caused by mishandling storage allocation, or failure to understand what happens, when, and why. During the long process of mastering C++, it is a good idea to track the constant allocation and deallocation actions:

- By printing comments in constructors and destructors.
- By using dump functions, as in the Van and Box classes.
- By printing an object (or part of it) when it is created.

10.1.1 Talking About Yourself

The keyword `this` is used to refer to the implied paramter within a class function. In class T, the type of `this` is `T*` because "`this`" is a self-pointer; it points at the implied argument. Thus, "`cout << this`" prints the address of the current object (in hexadecimal) and "`cout << *this`" prints the value of that object. If a class had a member named `data`, you could use `this->data` to refer to it. However, this is bad style: anywhere it is legal to say `this->data` it means the same thing to say, simply, `data`. There is never a need to write the `this->` in front of a member name.

In this chapter, we use "`this`" three times in the Van program (lines 25 and 86), once in the Layers program (line 129) and once in the Copy program (line 225). These lines show correct usage of `this` and are repeated below:

```
(line 25)  out <<"    Box @ " <<this <<" : " <<*this;
(line 86)  out <<"  Van @ " <<this <<"  Load @ " <<&Load <<".\n";
(line 129) out << "Instance is: " <<this <<" name: " <<name <<endl
(line 225) return *this;
```

In lines 25 and 86, we wish to print the memory address of the implied parameter, so we write `<< this`. At the end of line 25, we wish to print the object itself, not its address, so we write `*this`. Line 225 returns a `Copy&`, which is a reference to the implied parameter.

10.1.2 Constructor Initializers

In C and C++, initialization follows rules that are more permissive than the rules for assignment. Anything written in the body of a constructor function follows the rules for assignment. If an action can only be done with initialization, it must be done using a ctor (constructor-initializer).

The term "constructor initializer" is shortened to "ctor". Ctors exist to provide a way to initialize an object during construction, rather than by using assignment later, in the body of the constructor function. The initial value for any non-static data member may be supplied this way and ctors are often used if the initial value is a parameter to the constructor. To illustrate ctor syntax, the constructor for the Stack class (Chapter 4) is repeated, below, then rewritten using three ctors. In this case, the choice is a matter of style.

```
Stack::Stack( cstring label, int sz )  {
    s = new char[max=sz];
    top = 0;
    name = label;
}

Stack::Stack(char* label, int sz): max(sz), top(0), s(new char[sz]), name(label){}
```

Why use ctors? Ctors provide a compact and easy way to initialize the data members of any object, but they are actually essential in three situations:

- A ctor is required if a class member is declared as const but optional for member variables.
- An object can compose other objects. Sometimes the constructor for the composed object has parameters; these must be supplied by a ctor.
- When class derivation is used, an object (the derived object) may be an extension of another object (the base object). If the constructor for the base class needs arguments, they must be supplied by a ctor.

What is the syntax? How you write a ctor depends on the type of the member it will initialize.

- If a class member is a non-class type, the name of the member is written followed by its initial value in parentheses. The initial value follows the same syntactic rules as an initializer in a declaration.
- If a data member is a class type, you must supply arguments for the constructor of the component class. To do so, write the member name followed by parentheses that enclose the name of its class and arguments for its constructor.
- To initialize the base portion of an object in a derived class, the name of the base class is written followed by the arguments for its constructor, in parentheses.

Where do you write them? The ctors are written before the beginning of the body of a constructor, between the closing parenthesis and the opening curly bracket. The list of ctors is preceded by a colon and separated by commas. Each ctor is used to initialize one data member of the class. If there is more than one ctor, they should be listed in the same order as the members are listed in the class declaration.

10.2 Dynamic Allocation Incurs Overhead.

The allocation functions "malloc" and "new" must calculate and store bookkeeping information for later use by "free" or "delete". In this discussion, we will use the declarations below (an Item* initialized to a dynamically allocated Item) to illustrate the way things work. Figure 10.1 shows how the allocation area is configured on many systems.

The memory management system needs to know the block length. In both C and C++, the system must know how many bytes are in each dynamically allocated storage block. This could be stored as part of the block, computed from the sizeof the base type, or kept in a hidden table, indexed by the first address of the block.

```
struct Item { char name[20]; int count; float price; };
Item* purchase = new Item;
Item* inventory = new Item[5];
```

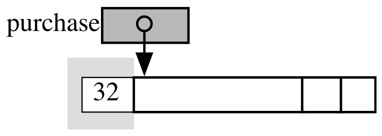


Figure 10.1: Allocation overhead.

There is no rule in the language standard that the system must allocate a particular amount of overhead space, or where it must be. However, every compiler must do something of this sort. Here we illustrate an old and common strategy, which was used for both C and C++ for many years. Extra space was allocated at the beginning of every dynamically allocated area and initialized to the actual number of bytes in the allocation block, including the overhead. In Figure 10.1, this length field is shown against a pale gray background. Note that this field precedes the user's storage area, and the address returned by `malloc()` was the address of the beginning of the user's area.

A C++ system needs to know the number of slots in an array. When "new" is used to allocate an array of class objects, it stores the number of allocated array slots immediately before slot 0. This number is used by "delete[]" to loop through the array elements and call the appropriate destructor to recycle the extensions of each element before recycling the array itself. Figure 10.2 is a diagram of the actual storage allocated for the dynamic array on the third line of code, above. The array length is shown against a pale gray background.

If, in addition, the allocation length is stored with the array, it would be stored before the number of array elements. The remaining diagrams will show the array length but not the overall block length.

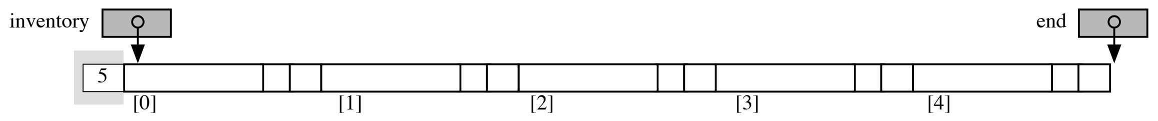


Figure 10.2: Overhead for an array of class objects.

10.2.1 Allocation Example: a Van Containing Boxes

The following short demonstration program will be used throughout this chapter to illustrate the allocation and deallocation semantics of C++. The first portion consists of a main program and two classes, Box and Van (which contains an array of Boxes). A second example builds on these two classes, introducing a third class named Layers that composes Van and Box. The memory diagrams shown are taken from actual runs of the program compiled by Gnu C++ for OS-X.

The Box class.

```
 1  //-----------------------------------------------------------------------------
 2  // A. Fischer October 9, 2000                                      file: box.hpp
 3  //-----------------------------------------------------------------------------
 4  #pragma once;
 5  #include "tools.hpp"
 6  //-----------------------------------------------------------------------------
 7  class Box {
 8    private:
 9      int length, width, high;   // The 3 dimensions of a box; occupies 12 bytes.
10
11    public:
12      Box(int ln, int wd, int ht){
13          length=ln; width=wd; high=ht; cout<<"\n  Real Box ";
14      }
15      Box(){ length = width = high = 1; cout <<"Default Box ";}
16
17      ~Box() {  cerr<< "  Deleting: ";  dump( cerr ); }
18
19      ostream& print( ostream& out ){
20          return out <<length <<" by " <<width <<" by " <<high <<".  ";
21      }
22      inline void dump( ostream& out );
23  };
24  //----------------------------------------------------------------------------
25  inline ostream& operator<< (ostream& out, Box& B) { return B.print( out ); }
26  inline void Box::dump(ostream& out){ out <<"  Box @ " <<this <<" : " <<*this <<endl;}
```

1. In order to declare an array of class objects, the class must have a default constructor, that is, one that can be called without parameters. This class has two constructors: a default constructor (line 15) for

initializing the array and a real constructor (lines 12–14) for building data objects. By using default parameters, these two definitions could be combined into one:

```
Box (int ln=1, int wd=1, int ht=1) { length=ln; width=wd; high=ht; }
```

2. A *dump* is a kind of debugging output that shows the machine addresses of your objects as well as their values. Dumps are helpful in C++ to verify what is happening with allocation, copying, and deallocation. This class contains a Dump function (line 22 and line 26).

3. Box::Dump() uses the keyword "`this`" to access its own address and print out its own contents.

4. `operator<<` must be defined before Box::Dump() because Dump() uses `<<` to print a Box.

The Van class.

```
27  //-----------------------------------------------------------------------------
28  // Van is a container class that can hold a load of Boxes.
29  // A. Fischer October 9, 2000                                      file: van.hpp
30  // Hex addresses from a run on OS-X, March 1, 2009
31  //-----------------------------------------------------------------------------
32  #pragma once;
33  #include "tools.hpp"
34  #include "box.hpp"
35  //-----------------------------------------------------------------------------
36  class Van {
37    private:
38      int Count;      // @ f718, 4 bytes
39      int Max;        // @ f71c, 4 bytes
40      Box* Load1;     // @ f720, 4 bytes -> 1001b4, 0x40 = 48 bytes: 12*3 + 4 + ?
41      Box* Load2;     // @ f724, 4 bytes -> 1001f4
42
43    public:
44      Van (int n=6);
45      ~Van();
46      ostream& print( ostream& out );
47      void dump( ostream& out );
48      int load_dim(){ return *( (int*)(&Load1) - 1 ); }
49  };
50  //--------------------------------------------------------- Global declaration
51  inline ostream& operator<< (ostream& out, Van& V) { return V.print( out ); }

52  //------------------------------------------------------------------------------
53  // Array construction example: Class implementation for Van.
54  // A. Fischer October 4, 2000                                       file: van.cpp
55  //------------------------------------------------------------------------------
56  #include "van.hpp"
57  Van::Van (int n){   //----------------------------------------------------
58      int a, b, c;
59      Count = 0;
60      Load1 = new Box[Max=n];
61      Load2 = new Box[2];
62
63      cout <<"\nEnter the boxes; use a 0 dimension to quit\n";
64      for (Count=0; Count<Max; ++Count) {
65          cout <<"  Dimensions: ";
66          cin >> a >> b >> c;
67          if (a*b*c == 0) break;
68          Load1[Count] = Box(a, b, c); // Make a local box, copy it using assignment.
69      }                                // Delete the local box at end of loop body.
70  }
71
72  Van::~Van(){    //------------------------------------------------------------
73      delete[] Load2;                     // Delete the boxes in the Load array.
74      cerr <<"  Deleted Load2.\n";
75      delete[] Load1;                     // Delete the boxes in the Load array.
76      cerr <<"  Deleted Load1.\n";
77  }
78
79  ostream& Van::print( ostream& out ){    //--------------------------------------
80      out <<"Load 1 has " <<Count <<" Boxes.\n";
81      for (int k=0; k<Count; ++k)  out <<"        " <<Load1[k];
82      return out << endl;
83  }
84
85  void Van::dump( ostream& out ) {        //--------------------------------------
86      out <<"  Van @ " <<this
87          <<"\n  Count @ " <<&Count <<" = " << Count
88          <<"\n  Load2 @ " << &Load2 <<" = " << Load2<<".\n"
89          <<"\n  Load1 @ " << &Load1 <<" = " << Load1
90          <<" slots " << load_dim() <<endl;
91      for (int k=0; k<Count; ++k)  Load1[k].dump( out );
92      out << endl;
93  }
```

Only two comments are given here. Most of the commentary on this class is in the next section, where allocation, initialization, and deallotion actions are examined in detail.

1. The Van constructor (lines 44 and 57–70) has a default parameter. This lets us declare a Van with or without parentheses and a dimension following the name, as in "`Van V1`" or "`Van V2(10)`".

2. Van::Dump() calls the function defined at the bottom of van.hpp (line 48). This functions accesses the hidden dimension field that precedes the beginning of the array named `Load1`. After printing this information, Van::Dump() calls Box::Dump() in a loop (line 91), to dump the Boxes in the van.

A test run on OS-X.

```
A dynamic array of class objects.
Default Box Default Box Default Box Default Box Default Box Default Box Default Box
Enter the boxes; use a 0 dimension to quit
  Dimensions: 2 2 2

  Deleting:   Box @ 0xbffff328 : 2 by 2 by 2.
  Real Box   Dimensions: 3 3 3

  Deleting:   Box @ 0xbffff328 : 3 by 3 by 3.
  Real Box   Dimensions: 4 3 2

  Deleting:   Box @ 0xbffff328 : 4 by 3 by 2.
  Real Box   Dimensions: 0 0 0

About to dump the van:
  Van @ 0xbffff3d8
  Count @ 0xbffff3d8 = 3
  Load1 @ 0xbffff3e0 = 0x500184 slots 5 length 3
  Load2 @ 0xbffff3e4 = 0x5001c4.
  Box @ 0x500184 : 2 by 2 by 2.
  Box @ 0x500190 : 3 by 3 by 3.
  Box @ 0x50019c : 4 by 3 by 2.


Normal termination.
  Deleting:   Box @ 0x5001d0 : 1 by 1 by 1.
  Deleting:   Box @ 0x5001c4 : 1 by 1 by 1.
  Deleted Load2.
  Deleting:   Box @ 0x5001b4 : 1 by 1 by 1.
  Deleting:   Box @ 0x5001a8 : 1 by 1 by 1.
  Deleting:   Box @ 0x50019c : 4 by 3 by 2.
  Deleting:   Box @ 0x500190 : 3 by 3 by 3.
  Deleting:   Box @ 0x500184 : 2 by 2 by 2.
  Deleted Load1.
```

The main program.

```
 94  //-----------------------------------------------------------------------------
 95  // Array construction example, October 8, 2000                     file: vanM.cpp
 96  //-----------------------------------------------------------------------------
 97  #include "tools.hpp"
 98  #include "van.hpp"
 99  //-----------------------------------------------------------------------------
100  int main( void )
101  {   cout << "\nA dynamic array of class objects.\n";
102      Van V;                              // Make a Van with space for 5 Boxes.
103      // Van V(2);                        // Alternate version has 2 Boxes.
104      cout <<"\nAbout to dump the van:\n";
105      V.dump( cout );
106      bye();
107  }
```

10.2.2 How Allocation and Deallocation Work

Consider the Van program above, which declares a Van named V (line 102). In response, the system allocates space for the four core data members of a Van (dark gray in Figure 10 .3) and calls the Van constructor to initialize that space.

Construction of the Van. The Van constructor allocates space for an array of five Boxes and stores the pointer in V.Load. Automatically, the default constructor for Boxes is run five times, once for each array slot. The five Box objects are initialized by the default Box constructor, so the dimensions of all five are initialized to 1 1 1. Figure 10 .3 illustrates the actual storage that was allocated for this object on my machine, with the initial values from the default constructor. The core portion of the object "V" is colored dark gray. The extensions consist of overhead (light gray) and usable storage (white).

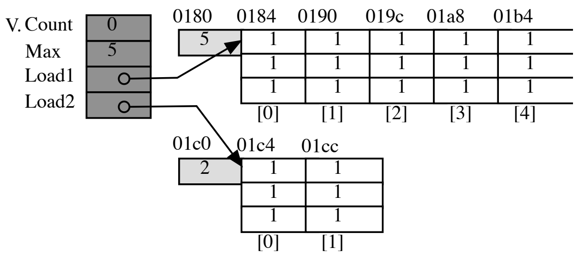


Figure 10.3: Creating a Van full of Default Boxes

Storing Boxes in the Van. Then control passes to the code part of the Van constructor. Line 63 prints instructions and the following loop prompts the user for the dimensions of a series of boxes. For each set of dimensions entered, line 68 constructs a new temporary Box and copies it into one slot of the array of Boxes. This replaces one set of dummy data in the Van by real information. The default assignment operator is used for this purpose.

```
    Load1[Count] = Box(a, b, c);
```

When control reaches the end of the loop body (line 69), the temporary object is automatically deleted and we see a deletion comment. In the run shown above, data for three boxes were entered, followed by a dummy box with a volume of 0, which ended the input phase. Figure 10.4 illustrates the contents of the Van at the end of the input phase.

By the end of the input loop, ten Boxes have been allocated and seven still exist:

- Five are part of Load1.
- Two are part of Load2.
- Three were allocated, initialized, copied into Load1, and freed.

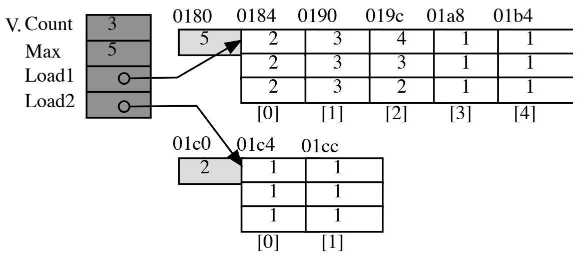


Figure 10.4: After storing real Box data in the Van.

As proof of this deletion, we see three trace comments:

```
Deleting:   Box @ 0xbffff328 : 2 by 2 by 2.
Deleting:   Box @ 0xbffff328 : 3 by 3 by 3.
Deleting:   Box @ 0xbffff328 : 4 by 3 by 2.
```

Note that the addresses of these three boxes are the same! that happens because the first one is created, used, and deleted, then the second is created in the same memory location. Also note that these boxes were allocated at an address that is very distant from the Van boxes (shown at the end of the output). This difference happens because the temporary Boxes are allocated on the run-time stack while the Van Boxes are allocated in the dynamic storage area.

Dumping the Van. Line 105 of main() dumps the Van, producing the second block of output. This block shows the machine addresses of the Van and its core parts, as well as the addresses of its two dynamically allocated extensions. From the dump, we see that the data from the temporary boxes has, indeed, been copied into the permanent boxes that are attached to the van.

Deallocation of the Van with delete[ ]. When we wish to delete dynamic storage, we use the name of the variable that points at the dynamically allocated block. Assume this pointer is named X. There are two forms of the deallocation command: `delete X` and `delete[] X`. The conservative rule for usage is:

> If you allocate with [], use delete[] with [].

Different treatment is necessary for arrays and non-arrays. For a non-array, the only memory space that needs deletion is the single variable that X points at. But for an array, we need to deal with the array itself *and* with the objects stored in it. Allocating the array caused the constructors to be run for the objects in the array, so deleting the array must cause the destructor to be run for each individual array object.

In the Van program, the statements that create the dynamic storage are on lines 60 and 61: `Load1 = new Box[Max = n];` and `Load2 = new Box[2];` Since we allocate this array in the Van constructor, and since this statement stores the allocation pointers in Load1 and Load2, we write `delete[] Load1` and `delete[] Load2` in the Van destructor. The result is:

- The Box destructor will be run for each Box in the array, in the opposite order from their creation. The number that is stored just before the `&Load1[0]` will control the deletion loop.
- Last of all, the array, itself, will be deallocated.

The last line of `main()` is the call on `bye()`, which prints the "Normal termination" message. The program's work is done and it is time for it to return to its caller (the system). We say that the variables in main() *go out-of-scope*, which causes the destructors for main's variables to be run. (`Van::~Van` is run to delete V.) You do not need to write anything to cause this to happen, and there is no way to prevent it. In our test run, the third block of output shown is the result of the automatic destruction. First, the Van destructor was called by the system. On line 73, it called the Box destructor: `delete[] Load2`. When the 2 Boxes have been deleted, control returns to the Van desctuctor. In the output, we see the trace comments from deleting two boxes, followed by the trace for deleting the array. On line 75, the Box destructor is called again: `delete[] Load1`. When all the Boxes have been deleted, control again returns to the Van desctuctor. In the output, we see the trace comments from deleting five boxes, followed by the trace for deleting the array.

10.3 Construction of C++ Objects

10.3.1 A Class Object is Constructed in Layers

- Space is allocated for the core portion of the whole object.
- Starting with the first component member, the constructor functions for the members are called. These initialize the components and may cause extensions to be allocated and initialized. If any of these constructors have parameters, they must be passed as ctor initializers (see below).
- Members of the class are initialized from ctor initializers, if any.
- The constructor for the class, itself, is called. It may store values in components and may also create extensions.

10.3.2 Layering Demo Program

The goal of the Layers program is to illustrate the order in which aggregate objects are constructed and deleted. The trace comments in the Layers functions, together with those in Van and Box provide step-by-step evidence of how C++ storage management works. The Layers class is given next, followed by comments that lead you through the step-by-step construction process and, finally, the output.

The Layers class.

```
110  //-----------------------------------------------------------------------------
111  // The Layers class aggregates Van and Box.
112  // A. Fischer October 10, 2000                                  file: layers.hpp
113  //-----------------------------------------------------------------------------
114  #pragma once;
115  #include "van.hpp"
116
117  class Layers {      //--------------------------------------------------------
118      char* name; // Memory is allocated by body of Layers constructor.
119      Box B;
120      Van V;
121
122   public:            //--------------------------------------------------------
123      Layers( char* nm ) : B(8,7,6), V(4) {               // constructor with ctors
124          cout <<"  Constructing " <<nm <<"\n";
125          name = new char[strlen(nm) + 1];
126          strcpy(name, nm);
127          cout <<"  " <<nm <<" is complete.\n\n";
128      }
129
130      ~Layers() {                                        // destructor
131          cout << "  Deleting name @ " <<(unsigned)name <<": " <<name << endl;
132          delete name;
133      }
134
135      void print( ostream& out ){
136          out << "Instance is: " <<this <<" name: " <<name <<endl
137              <<"   Box: " <<B <<"\n   Van: " <<V;
138      }
139  };
```

- An object of class Layers has three members: a char*, a Box, and a Van.
- The Layers constructor has a char* parameter; this name will be used in the output to help clarify which object is being constructed or deleted. Most of the output, below, came from the class constructors and destructors.
- Line 123 uses two ctors to initialize the Box and Van portions of the object. This is necessary because we want a real box, not a dummy, and we want a Van with 3 slots, not 5 (the default).

- A cast is used in the destructor to print the address of the name string (line 131). Without the cast, we would print the word itself. The contents of `name` is the address of the dynamic allocation area (a pointer). We are allowed to cast pointers to type unsigned.
- The last part of the trace, after the "normal termination" comment, shows that six Boxes were created as part of the Layers object created when line 149 of main was executed. The first box created was the loose box declared on line 119 and initialized by a ctor in the Layers constructor on line 123. The ctor supplies initialization paramters, so this box was initialized by the "Real" Box constructor, the one with parameters.
- The other five Boxes are inside the Van that is declared on line 120 and initialized by the second ctor on line 147. The Van constructor is on lines 57. . . 70. On lines 60 and 61, the constructor creates two new arrays of Boxes. Since these boxes are in an array, they are initialized by the default constructor.
- The first half of the output and shows the user-dialog that occurred during the process of constructing three real boxes and storing them in the Van, lines 63. . . 69 in the Van constructor. The Box objects created there are local temporary variables that are deallocated when control reaches line 70. These variables exist just long enough to copy their contents into the blank box that is part of the Van array (using the default definition of the assignment operator, which is the same as the default copy constructor).
- The parts of an object are initialized before calling the constructor for the whole. Note that control enters the Van constructor (output line 170) after initializing all the Boxes to default values. It enters the Layers constructor and prints the object's name, "PackRat", after initializing the Van.
- The main program creates an instance of class Layers (line 149) and prints it. From the output, you can see that the Layers object contains a Box and a Van, and the Van contains two more real Boxes (the default boxes are not printed here).
- The post-termination output verifies the order in which objects are deleted: the destructor for the outside class triggers the destructors for the inside classes before it executes itself. Within an array, as in Van.Load1, the array elements with the largest subscripts are deleted first and the object in slot 0 is deleted last. Then the array itself is deleted.

The main Layers program.

```
140  //-----------------------------------------------------------------------------
141  // Layered construction and destruction example.
142  // A. Fischer October 10, 2000                                  file: layersM.cpp
143  //-----------------------------------------------------------------------------
144  #include "tools.hpp"
145  #include "layers.hpp"
146  //-----------------------------------------------------------------------------
147  int main( void ) {
148      cout << "Creating an object of class Layers. ";
149      Layers f1( "PackRat" );
150      f1.print(cout);
151      // Layers f2(f1);        // A call on the copy constructor.
152      bye();
153  }
```

The trace.

```
Creating an object of class Layers.
Real Box Default Box Default Box Default Box Default Box Default Box
Enter the boxes; use a 0 dimension to quit
Dimensions: 3 2 3

Deleting:   Box @ 0xbffff624 : 3 by 2 by 3.
Real Box   Dimensions: 2 2 2

Deleting:   Box @ 0xbffff624 : 2 by 2 by 2.
Real Box   Dimensions: 4 2 1

Deleting:   Box @ 0xbffff624 : 4 by 2 by 1.
```

```
Real Box   Constructing PackRat
PackRat is complete.

Instance is: 0xbffff708 name: PackRat
Box: 8 by 7 by 6.
Van: Load 1 has 3 Boxes.
3 by 2 by 3.         2 by 2 by 2.         4 by 2 by 1.

Normal termination.
Deleting name @ 1049120: PackRat
Deleting:   Box @ 0x1001c0 : 1 by 1 by 1.
Deleting:   Box @ 0x1001b4 : 1 by 1 by 1.
Deleted Load2.
Deleting:   Box @ 0x10019c : 4 by 2 by 1.
Deleting:   Box @ 0x100190 : 2 by 2 by 2.
Deleting:   Box @ 0x100184 : 3 by 2 by 3.
Deleted Load1.
Deleting:   Box @ 0xbffff70c : 8 by 7 by 6.
```

10.4 Constructors and Construction

C++ objects are created and initialized in three situations:

1. A call on `new`
2. Declaration of a class-type variable.
3. A call on a constructor function embedded in an expression.

In Java, the the first way listed is the only way to create non-array objects. n both languages, an object created with `new` is allocated in the dynamic memory area called the "heap". These objects are used through references (in Java) or pointers (in C++). Because they are in the heap, they can outlast the execution of the function that created them and the lifetime of the variables that store them. In C++, these objects must be specifically deallocated by calling `delete` when they are no longer needed.

Unlike Java, C++ also permits local objects to be created by a declaration or by calling any constructor. These objects are allocated in the current stack frame and deallocated when they go "out of scope". (That is, when control leaves the block of code that contains the declaration or constructor call.)

Kinds of Constructors The terminology regarding constructors is confusing. However, the types of constructors and the various terms for them must be mastered before you will understand C++ reference books. The types of constructors are:

1. **Constructors with parameters.** This is the usual case and is the kind we have been using. Examples are found in the Van program on lines 14, 44 / 60–73, and are in the Layers program on lines 146–151. These constructors can be used in declarations (lines 106 and 129), ctors (line 146), and calls on new that allocate a single object, not an array. Line 71 calls a constructor in an assignment statement. This builds a temporary object that will be discarded as soon as its value is copied into the array.
2. **Null constructor.** A null constructor is one that initializes nothing. A null constructor for a class named named `Zilch` looks like this:

   ```
   Zilch(){}
   ```

   Sometimes we add trace output to constructors; these are still null constructors, even though the function body is non-blank. A null constructor with a trace might look like this:

   ```
   Zilch(){ cerr << "Constructing a Zilch object."; }
   ```

   Every class has a constructor. If you don't provide any constructor at all, C++ generates a null constructor. If you provide some sort of constructor, a null constructor will not be supplied automatically, although you can define one yourself.

3. **Default constructor.** A default constructor is a one with no parameters (line 15) or one that has a default value for every parameter (line 44 / 60–72). A null constructor is a default constructor, but non-null constructors constructors can also be default constructors. A declaration that uses the default Van constructor is on line 105.

   If you declare an array of class objects, the class *must* have a default constructor which will be used to initialize the elements. The calls on new on lines 63–64 use the default Box constructor to initialize all the boxes.

4. **Default copy constructor.** A copy constructor is used whenever there is a need to initialize a newly constructed object from another object of the same class. The most common and important application is call-by-value parameter passing, but uses of the copy constructor are also shown on lines 71 and 130. Each class has exactly one copy constructor (either by default or by explicit definition) with one of these prototypes:

   ```
       Classname( const Classname& );
       Classname( Classname& );
   ```

   A default copy constructor is supplied automatically if no explicit copy constructor is present. It performs a shallow copy from the initializing object into the newly-constructed object. If a user-defined copy constructor is present, it is used instead of the default copy constructor for call-by-value. If `operator=` is not redefined for a particular class C, then the default copy constructor is used to implement `c1 = c2`, where c1 and c2 are objects of class C.

5. **A deep-copy constructor.** A deep copy is a complete copy of a data structure, including its extensions. A programmer-defined copy constructor that performs a deep copy is shown on Lines 220. . . 224 of the Ball program (next). Calls on this copy constructor are made on lines 200. . . 201.

   If a data structure structure is small, the time and space used to create this copy are often not important. However, if the data structure is large, making a deep copy for every call on every function would waste an excessive amount of time and space and would cause an unacceptable performance slowdown. For this reason, deep-copy constructors are rarely used.

10.4.1 A Variety of Constructors

This section introduces a class named Ball with four constructors and a redefinition of the assignment operator. The main program calls each one to illustrate the differences in syntax.

Calling the constructors:

1. A static class member is shared by all instances of the class. In this program, we use a static member to count the total number of instances that have been created. This static member is created and initialized on line 306.
2. The ID field of each instance is the value of the counter at the time the instance was initialized. This counter is incremented by every constructor except the null constructor which has been commented out.
3. The declaration on line 311 has just the class name and an object name; this calls the constructor on lines 345–49 and uses the default value for the parameter. Line 378 of the output shows that the Name field has been initialized to the default value, 'B', the ASCII version of the argument 66.
4. Line 312 calls the same constructor as line 311 but supplies the optional parameter. The result is shown on line 379: the Name field has been initialized to 'K', rather than to the default, 'B'.
5. The declaration on line 313 provides a char* argument for the constructor; this calls the normal constructor (lines 350–54). Line 380 of the output shows that the Name field has been initialized to the argument provided in the declaration.

```
300   // ------------------------------------------------------------------------
301   //  Syntax and semantics for copy constructors, initialization, assignment.
302   //  Alice E. Fischer May 6, 2001                            file: ballM.hpp
303   // ------------------------------------------------------------------------
304   #include "tools.hpp"
305   #include "ball.hpp"
```

```
306  int Ball::Counter = 0;     // Create & initialize the static class member.
307
308  // ---------------------------------------------------------------------
309  int main( void ) {
310      cout <<"                                 ID  Name\n";
311      Ball One;           cerr <<"a. " << One <<endl;     // Default constructor
312      Ball Two( 75 );     cerr <<"b. " << Two <<endl;     // Normal constructor
313      Ball Three("Ali");  cerr <<"c. " << Three <<endl;   // Normal constructor
314      Ball Four( One );   cerr <<"d. " << Four <<endl;    // Copy constructor
315      Ball Five = Three;  cerr <<"e. " << Five <<endl;    // Copy Constructor
316      cout << "\nReady to construct an array:" <<"\n";
317      Ball Six[2];        cerr <<"\t" <<"f. "<< Six[0] <<"\t   " <<Six[1] <<"\n";
318      // Ball Bad = Ball(77);                      // Bad: Ball(77) not a Ball&
319      cerr <<"\nNow " << One.getcount() <<" Ball objects have been created.\n";
320      Four = Two;         cerr << "g...........................\n" // Assignment
321                              <<"\t" << Four <<"\n\t"  << Two <<endl;
322      bye();
323      return 0;
324  }
```

6 The argument in the declaration on line 314 is an object of class Ball; this calls the copy constructor (lines 355-59). This copy constructor gives the new object a unique ID#, then makes a deep copy of the rest of the argument. It does not change the argument. Within this function, the parameter name is used to refer to the object being copied and the corresponding fields of the new object are called by the member name alone. Line 381 of the output shows the result: the Name field has been initialized to "B", a copy of the Name in One.

7 Line 315 uses another syntax to call the copy constructor, initializing the new object, Five, as a copy of Three. The result is on line 382.

8 At this time, five objects have been created and intialized. We now create an array of two more Balls, on line 317, giving us a total of seven Ball objects. The output (lines 384–87) shows that the constructor on lines 345–49 was used, with the default value of the parameter, to construct both.

9 We now use the redefined assignment operator (line 320 / 360–64). This is a very strange definition for `operator=` because it only copies one character from the right operand into the left operand, but it does illustrate how redefinition works. Initially, Four.Name was "`B`". After executing line 362, Four.Name has been changed to "`K`", as shown on line 390.

10 We have reached the end of the program; the call on `bye()` prints a termination comment (lines 322 / 393). After that, we see the trace comments printed by the destructor on line 345. As each object is deleted, we decrement the object counter. The last line of the trace tells us that the Ball class does not have a memory leak.

11 A class cannot have two functions with the same calling sequence. The null constructor on line 339 is commented out because it conflicts with the default constructor on lines 345–49. When both are in the code, the compiler produces comments like this:

```
ballM.cpp: In function ‘int main ()’:
ballM.cpp:12: call of overloaded ‘Ball()’ is ambiguous
ball.hpp:17: candidates are: Ball::Ball ()
ball.hpp:27:                 Ball::Ball (int = 66)
```

In this program, we fix the problem by commenting out the null constructor. If, instead, we comment out the default constructor on line 345…349 (and the call on line 312), we see output from the null constructor:

```
Null constructor........a. 782336
```

The ID and Name fields contain garbage because they have not been initialized.

12 The system's final status comment (line 402) assures us that the Ball program' s memory has been freed without crashing.

The Ball class.

```
// ---------------------------------------------------------------------
//  Syntax and semantics for copy constructors, initialization, assignment.
//  Alice E. Fischer May 6, 2001                               file: ball.hpp
// ---------------------------------------------------------------------
#pragma once;
class Ball {    // ----------------------------------------------------
  private:
    static int Counter;
    const int ID;
    char* Name;

  public:       // ----------------------------------------------------
    //Ball(){ cerr<< "Null constructor........"; }  // Null constructor
    //~Ball(){}                                     // Null destructor.
    ~Ball(){
        --Counter;  delete Name;
        cerr <<"\tDestructor..." << Counter <<" Balls remain.\n";
    }
    Ball( int alpha = 66 ): ID( ++Counter ) {  // --- Default constructor.
        cerr <<"Default or normal constructor. ";
        Name = new char[ 2 ];
        Name[0] = alpha; Name[1] = '\0';
    }
    Ball( char* arg ): ID( ++Counter ) {    // -------- Normal constructor.
        cerr <<"Normal constructor............";
        Name = new char[ 1+strlen(arg) ];
        strcpy( Name, arg );
    }
    Ball( Ball& a ): ID( ++Counter ) { // ---------- Deep copy constructor.
        cerr <<"Copy constructor..............";
        Name = new char[ 1+ strlen(a.Name) ];
        strcpy( Name, a.Name );
    }
    Ball& operator = ( Ball& source ){ // -- Changes data but not identity.
        cerr<<"Assignment function..........\n ";
        Name[0] = source.Name[0];
        return *this;
    }

    void print() { cerr <<ID <<"  " <<Name ; }
    int getcount() { return Counter; }
};
ostream& operator <<( ostream& out, Ball& C ){ C.print(); return out; }
```

The output:

```
                                 ID  Name
Default or normal constructor. a. 1  B
Default or normal constructor. b. 2  K
Normal constructor............c. 3  Ali
Copy constructor..............d. 4  B
Copy constructor..............e. 5  Ali

Ready to construct an array:
Default or normal constructor. Default or normal constructor.   f. 6  B    7  B

Now 7 Ball objects have been created.
Assignment function..........
 g..........................
   4  K
   2  K
```

```

Normal termination.
    Destructor...6 Balls remain.
    Destructor...5 Balls remain.
    Destructor...4 Balls remain.
    Destructor...3 Balls remain.
    Destructor...2 Balls remain.
    Destructor...1 Balls remain.
    Destructor...0 Balls remain.

The Debugger has exited with status 0.
```

10.5 Destruction Problems

Deep deletion after a shallow copy When call-by-value is used to pass an argument to a function, the class copy constructor (default or user-defined) is used to initialize the parameter variable with a copy of the argument value. The default copy constructor does a shallow copy.

At function-return time, the class's destructor will be run on the parameter. A correctly-written destructor deallocates object extensions and frees dynamic memory (deep destruction). However, when shallow copy is used to pass an argument, the parameter variable and the underlying argument variable both point to the same object extensions. Using deep destruction will cause a serious problem because the object extensions will be deleted. However, the calling program still needs them.

Therefore, call-by-value generally cannot be used with parameters of class types. Use call by reference (more efficient, less modular) or define a copy constructor that does a deep copy (miserable efficiency, great modularity). The next sample program shows a class with a copy constructor that makes a deep copy.

Important reminders.

- Deletion of stack objects happens when control leaves the block in which the object was declared.
- Deletion of dynamically-allocated (new) objects happens when delete or delete[] is called explicitly. An attempt to use delete on a stack-allocated object is a fatal error.
- In both cases, the destructor from the appropriate class is called to deallocate memory.
- Memory leaks happen when you fail to delete a dynamic object at the end of its useful life, before the last pointer to that object is reassigned to point at something else.
- Fatal (but subtle and delayed) errors happen when you attempt to use an object after it has been deleted, when you delete the same thing twice, or when you delete something that was not created by new.

Chapter 11: Modules and Makefiles

11.1 Modular Organization and makefiles.

From the Fanny Farmer cookbook:

Before beginning to mix, be sure that all ingredients and utensils are on hand.

You can't bake a good cake if you don't have enough eggs or if your butter is spoiled. You can't compile or run a program successfully if you need files that are missing or out of date.

11.1.1 History

The C++ standard does not address the issues of code files and header files. These and makefiles developed as part of the Unix system environment because of a clear and evident need to mechanize and automate the production of large-scale applications.

Projects. Both Unix-based and Windows-based systems have supported *projects* as part of an integrated development environment (IDE). The project facilities vary greatly in both power and convenience of use, and can be more difficult for a newcomer to use than the old-fashioned makefile. At this time, the Microsoft project facility is both confusing and intolerant of error. Mac's XCode is flexible and less intolerant of error, but highly complex. Both organize the project using a special project file. The cross-platform Eclipse IDE gives the programmer a choice of producing a traditional makefile or a non-portable project file.

A modern IDE offers much more than just a project facility. Most useful are syntax coloring, global renaming and refactoring, and global search-and-replace. It is clear that, as IDEs improve and our experience with them increases, they will gradually replace the old ways of doing the job: compiling from the command line and hand-built makefiles.

Whether you are using a project or a makefile, one kind of information must be supplied: the full pathname of every source code file and every header file that is needed by the project. Given this information, a project facility and some makefile facilities will generate a list of which modules depend on which other modules. This dependency list is used to determine which modules are unaffected by a change to part of the program. When you ask to build the application, the only modules that will be recompiled are those that have been affected by some editing change since the last build. Thus, if one module of a massive application is changed, we avoid recompiling almost all of the program. Re-linking will happen if anything was recompiled.

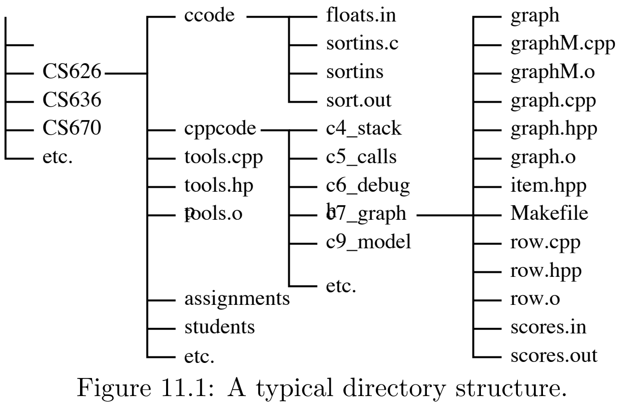


Figure 11.1: A typical directory structure.

Namespaces. Very recently, the C++ standard adopted the concepts of namespaces, and Visual C++ implemented them in conjunction with workspaces. These facilities may or may not replace the traditional makefiles and separate compilation. At the moment the added facility leads more to confusion than to ease of use. The purpose of namespaces is the same as one of the purposees of separate compilation: to allow many modules to define objects and functions freely without concern about conflict with other objects of the same name in other modules.

File and Directory Organization. A multi-module program in C++ consists of pairs of .cpp and .hpp files, one pair for each class (or closely related set of classes), and one .cpp file that contains the main program. (Exceptions: he main program may or may not have a matching .hpp file, and some simple classes are written entirely within the class .hpp file.) Each .cpp file includes just its own .hpp file. Each .hpp file in the project must include the other .hpp files that contain the prototypes it uses.

Experienced programmers define a separate directory for each project. (This saves much pain in the long run.) This directory contains the .cpp, .hpp, and .o files for the project as well as the makefile, the executable file, the input, and the output. Since many projects may use the tools files, they are placed a level or two above the project directories, as indicated by Figure 11.1.

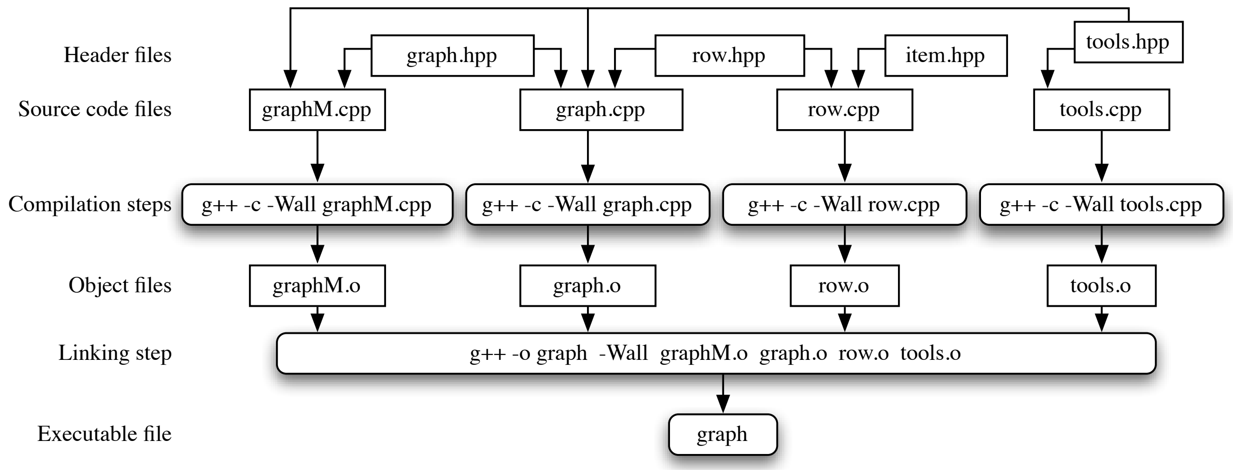


Figure 11.2: The BarGraph application with four code modules.

11.1.2 General Concepts

- A program that is physically located in many files will be compiled as a single module if the main program includes (directly or indirectly) the .cpp files for the other modules. In a multi-module compilation, main() includes the header files of the other modules, *not* their source code.
- When doing a multi-module compile, each module is compiled separately to make a .o file, as shown in Figure 11.2. Then the linker integrates these .o files with the .o files for the libraries to make an executable file.
- A makefile (in Unix) automates the process of separate compilation and linking. The project files in Windows systems accomplish the same goals as a makefile, but a makefile has somewhat more power and flexibility because it can be hand-coded and/or hand-modified. Project files are automatically generated and, therefore, rigid. Makefiles are universally portable in the Unix world.
- One .hpp file may be included by several others, so we must do something to prevent definitions from being included twice in the same module. (This would cause the compiler to fail.) Many systems support this directive at the top of a header file:

  ```
  #pragma once;
  ```

  This is advice to the compiler to include the header only once per module. If your system does not support this, you must surround the entire .hpp file with conditional compilation commands.

```
#ifndef MYCLASS_H
#define MYCLASS_H
... put the class declaration here ...
#endif
```

If an attempt is made to include the same header twice in one module, this "wrapper" will cause the inclusion to be skipped the second time, avoiding conflicts due to duplicate definitions. The "wrapper" always follows the pattern shown above, where MYCLASS is traditionally the same as the class name, but written in upper-case (instead of mixed case) letters. Anything that might be the same as a standard library header file name should be avoided.

- Naming. Three names are associated with each module. These could be completely unrelated, but it is better to make them alike. One suggestion is to use the name of the major class within the module, but write that name in three ways:
  - The name of the pair of the .hpp and .cpp files. *Use all lower case* here because it is the most portable from file system to file system.
  - The name of the class within the file. Use *Mixed Case* here because that seems to be becoming customary for class names: List, Cell, BarGraph
  - If your system does not recognize `#pragma once;`, you need to put conditional compilation directives at the top and bottom of each header file. Use *ALL UPPER CASE* letters for the `#define` symbol at the top of the .hpp file, since that is customary for defined symbols. It is also customary to end each one with `_H`. Example: `LIST_H`.
- The linker can fail for three reasons:
  - Some required symbol (usually a function name) is not in any of the modules that are being linked, usually caused by a missing module.
  - Some symbol is defined in two of the modules (double inclusion).
  - Lack of disk space to store the executable file.
- The linker will not succeed if same object is defined *outside the class declarations* in two .hpp files, or in one .hpp file that is included by two others. This causes double inclusion. The `#pragma once;` and `#ifndef` directives cannot help with this problem because they operate at the level of a single module. The linking problem occurs when one method or object is part of the .o file for two modules.

  However, if a function is defined entirely *inside* a class declaration (in an .hpp file), or defined as an inline function *after* the class (in an .hpp file), the .hpp file may be included by several other modules without causing a conflict.

  If a data object (such as a constant array of strings) must be directly visible to two modules, you must define it in one module and use `extern` declarations to link it to the other modules.

11.1.3 Enumerations and External Objects

The problem. An enum declaration creates a type, and types are usually defined globally. Several parts of a program typically need to use the enumerated type, so it cannot be conveniently put inside a single class. The best place is a header file, enums.hpp, that contains the enum declarations for the whole application.

Further, to use enum types effectively, you need a civilized way to input an enum value and to output it. Numeric codes are not civilized. Input can be easily handled by reading a menu selection and processing it with a switch. Output is a harder problem because we must go from the enum constant to an English word that the user will understand. We can do this by creating an array of output strings, parallel to your enum list, and using the enum value to subscript the array when we want output.

However, this array is an object, so it cannot go into the .hpp file with the corresponding enum declaration. If that .hpp file were included in more than one compile module, the array object would be compiled more than once and be part of two .o files. The linker would then be unable to link the application because it would see two objects with the same name.

The Solution The solution to this linking problem is to use the `extern` storage class. Identical extern declarations are included by several files, but this does not cause a linking problem because an extern declaration *with no initializer* does not create an object. (It instructs the linker to look for and link to an object that was created elsewhere.) An extern declaration *with an initializer* creates the object that all the other modules will link to. Therefore, we write one copy of an extern declaration *with an initializer* in some .cpp file. For this purpose, you might use the file that contains main(), a .cpp file that contains the application's primary class, or a separate file called enums.cpp.

Details of the Solution

1. Create a file named enums.hpp.
   (a) Into it, put all the enum declarations needed by your application. Suppose you need an enumeration of colors: red, white, blue. Sometimes you need an additional code for errors:
   ```
   enum Color { ERROR, RED, WHITE, BLUE };
   ```
   (b) For each enum declaration, you need an array of strings for making civilized output. Below each enum declaration, put a declaration similar to this just :
   ```
   extern const char* colorNames[4];
   ```
   (c) `#include enums.hpp` in the .hpp file for every class that needs to use the colors or the column status. For output, simply use an enum constant to subscript this array of words.
   ```
   Color itemColor = RED;
   cout << color ordered << " \t" << colorNames[itemColor];
   ```
2. In enums.cpp or some other .cpp file, put a matching extern declaration for
   ```
   extern const char* colorNames[4] = {"Error","Red","White","Blue" };
   ```

11.1.4 Rules

It should be clear by now that the physical division of source code into .hpp files and .cpp files is not easily compatible with the logical division of code into classes. Thus, the traditional compilation and linking methods that worked well for C are not a good fit for C++. These rules need to be understood when trying to debug the global structure of a project file or makefile:

1. Every function, variable, or class must be declared before it can be referenced.
2. If two definitions need to reference each other, the cycle must be broken by declaring one, declaring and defining the second in terms of it, then giving the full definition of the first one.
3. When given in different places, a definition must exactly match its declaration.
4. Every function (and external variable) that is used must be defined exactly once in the completed, linked, project.

11.2 A Makefile defines the project.

In a Unix environment, a makefile automates and speeds up the process of compiling and linking the application and makes it much easier to keep everything consistent.

- It allows you to list the object modules and libraries that comprise the application and say how to produce and use each one.
- For each module, it allows you to specify the code and header files on which it depends.
- When you give the command to make or build the project, the facility generates the executable program. Initially, this means it must compile every module and link them together.
- When you say make, control goes immediately back to you if nothing has changed since the last linking.
- When you change a source-code module, the corresponding object module becomes obsolete. The next time you make the application, it will be recompiled.

- When you change a header file, the object files for all modules that depend on it become obsolete. The next time you make the application, they will be recompiled.
- When you recompile a module, the corresponding executable program becomes obsolete, so the application will be relinked.
- It allows you to easily specify a set of flags that control how the compiler will work. Some useful flags are described later in this handout.
- It lets you automate the process of deleting files that are no longer needed.

Makefile syntax.

- Any line that starts with a `#` is a comment. (Lines 1, 6, 9, 18, 22.)
- To define a macro, write a name followed by an = sign and the definition. We define symbols this way on lines 7 and 10. To use a macro, enclose its name in `$(...)`. We use the defined symbols in lines 12, 13. 16, and 20.
- Object and executable files are built from source and header files and, therefore, depend on them. To define a dependency, list the name of a target file followed by a colon and the list of source files that are used to create it. Lines 12, 19, 23, 24, and 25 define dependencies. If any file on the right changes, the file on the left will be regenerated the next time "make" is executed.
- Any line that is indented MUST start with a single tab character; it won't work if the tab is missing or if it is replaced by spaces.

11.2.1 Making the Bargraph Application

The makefile.

```
 1  # Rule for building a .o file from a .cpp source file
 2  .SUFFIXES: .cpp
 3  .cpp.o:
 4      g++ -c $(CXXFLAGS) $<
 5
 6  # Compile with debug option and all warnings on.
 7  CXXFLAGS = -g -Wall -I../..
 8
 9  # Object modules comprising this application
10  OBJ =   graphM.o graph.o row.o ../../tools.o
11
12  graph: $(OBJ)
13      g++  $(CXXFLAGS) -o graph $(OBJ)
14
15  clean:
16      rm -f $(OBJ) graph
17
18  # Use tools source file from grandparent directory --------
19  ../../tools.o:   ../../tools.cpp ../../tools.hpp
20      g++ -c $(CXXFLAGS) ../../tools.cpp -o ../../tools.o
21
22  # Dependencies
23  graph:      graphM.cpp graph.hpp row.hpp item.hpp ../../tools.hpp
24  graph.o:    graph.cpp graph.hpp row.hpp
25  row.o:      row.cpp row.hpp item.hpp
```

- Please note that the line numbers are part of this explanation, they do *not* belong in the makefile.
- Lines 3 and 4 are a template, or general rule, for creating a compilation command. Line 3 says that this rule tells how to convert a .cpp file to a .o file. The command is on line 4. It refers to two macro symbols, `CXXFLAGS` (defined on line 7) and `$<` which stands for the name of a .cpp file that will be supplied later. This template will be used automatically generate a compilation command every time a .o file is needed and there is no specific rule for creating it.

- Line 7 defines the symbol CXXFLAGS, a listof the options, or flags, to be used by the C++ compiler and linker. I sometimes use these flags:
  - `-Wall` to turn on all warnings and `-Wno-char-subscripts` to turn off the single warning about using characters as subscripts.
  - Since I keep my tools one or two levels higher in my directory tree, I use the `-I..` or `-I../..` flag to define the search path for files included from this other directory.
  - If you want to use the on-line debugger, you must also give a `-g` flag.
- Line 10 defines the symbol OBJ to be the list of object files that compose this application. We define the list once, and use it three times, in lines 12, 13, and 16. The defined symbol makes it easy to write three commands that work with the same list of object files and makes it easy to modify that list when a module is added or removed.
- This makefile defines two commands (lines 12. . . 16) that can be used from the command line:

`make graph`	Lines 12 and 13	Compile and link the graph application.
`make`		This is the same as make graph.
`make clean`	Lines 15 and 16	Clean out any existing .o files and, finally, delete the executable file.

- Line 13 is the linking command that will be used.
- Line 16 is a command that will delete all the generated files. We use it to clean up a directory after finishing a project or to force a fresh compilation of all modules.
- Lines 19 and 20 define the dependency and the compile command for the tools file. This rule is given separately because it is different from the general rule on line 4.
- Line 20 is the command that will be used to compile the tools file in the grandparent directory. If the .o file already exists, this step will be skipped. After the first project is compiled once, tools.o will be in the grandparent directory where the linker will find it for other projects.
- If the tools were only one directory level up, instead of two, the following lines would be changed:

```
7.  CXXFLAGS = -g -Wall -I..
10. OBJ =   graphM.o graph.o row.o ../tools.o
18. # Use tools source file from parent directory ----------
19. ../tools.o:   ../tools.cpp ../tools.hpp
20.     g++ -c $(CXXFLAGS) ../tools.cpp -o ../tools.o
23. graph:      graphM.cpp graph.hpp row.hpp item.hpp ../tools.hpp
```

- This file should be named `Makefile` or `makefile` and should be in the same directory as the majority of the source code files.

Chapter 12: Derived Classes

A consequence of inheritance:

> **The sins of the father are to be laid upon the children.**
> . . . Euripides, Exodus, and Shakespeare.
> . . . And so are the strengths and skills of the father, in C++.

12.1 How Derivation is Used

Purposes. A class may be derived from another class for several possible reasons:

1. To add functions to those defined by an existing class.
2. To extend and specialize the actions of a function defined in an existing class.
3. To mask a function in an existing class and prevent further access to it.
4. To create a different interface for an existing class.
5. To add data members to those included by an existing class.
6. To further restrict the protection level of data members that belong to an existing class.

Declaration syntax.

1. The first line of the class declaration declares the derivation relationship. The following two lines are taken from the demo program below.

   ```
   class Printed : private Pubs { ... };
   class Book : public Printed { ... };
   ```

2. The style of derivation (public or private) will be clarified later. It affects future derivation steps but not the current derived class.
3. The base class must be defined first; in this case, it is Pubs. The second class (Printed) was derived from Pubs using private derivation. The third class, Book, was derived from Printed by public derivation.

Usage patterns and rules.

1. Typically, more than one class is derived from a base class, but the derived classes form a bush, not a chain.
2. Inheritance chains such as `Pubs -◁-- Printed -◁-- Book` do occur in real programs but are not the most important use of derivation.
3. Two or more derived classes at the same level (brothers) are used to implement polymorphic types and/or multiple interfaces for the same type.
4. It is also possible to derive one class from two or more printed classes. When that is done, an object of each base class is a part of the derived class.

UML for derived classes. To diagram a derived class, we use a line and a triangle with its point toward the printed class and its flat side toward the derived class. The sign written on the lines (`+`, `#`, or `-`) denotes public, protected, or private derivation, respectively. Figure 12.1 gives a UML diagram for the Publications demo program, below. (Note: the Online and Periodical classes are not implemented.)

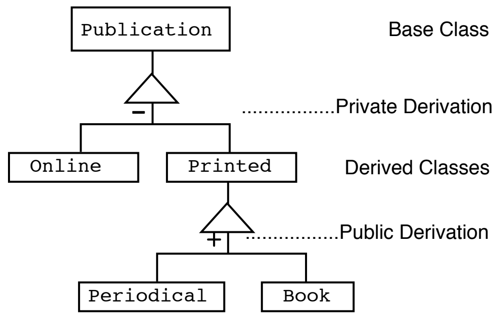


Figure 12.1: UML diagram for private and public inheritance.

12.1.1 Resolving Ambiguous Names

It is normal for a base class and a derived class to have methods with the same name (purposes 2 and 3, above). In this case, we say that the method in the derived class *overrides* the base method. This relationship is the basis of polymorphism (covered in a later chapter). Most of the time, it causes no confusion to have two methods (or data members) in the same class hierarchy that share the same name. When a function method in class A refers to a class member named `mem`, the instance of `mem` in class A will be used if there is one. Otherwise, the compiler will look for an inherited instance of `mem`. (It will look in A's base class and continue searching up the hierarchy until a definition of `mem` is found.)

Sometimes, however, a class may contain a definition for `mem` but one of its methods must refer to a different instance of `mem` defined in the base class. For example, suppose you wish to print the data for a Book, and class Book is derived from class Printed. To print all the Book data, we first tell the base class to print all its data, then we output the local data. To do this, we must be able to call the print method in the base class. This is the purpose of the *scope-resolution operator*, which is written with two colons (`::`).

In the demo program that follows, all three classes have members named `serial`, `name`, and print(), so these names are ambiguous. Line 69 shows the `::` operator used to call the base-class `Pubs::print()` from within the print method of the derived class, `Printed::print()`. On line 93 of Book::print, we write `<<name` to print Book::name. On the next line, we want to print the name from the base class, so we write `<<Printed::name`.

```
93     out <<name <<" #" <<serial
94         <<"\n\tis " <<Printed::name <<endl
95         <<"\tmy Pubs name is hidden from me." <<endl;
```

On line 100 we want to print Pubs::name, but this member is not visible to the Book functions because of the private derivation step even though this member is public for the world at large (a strange combination). In earlier versions of standard C++, the Pubs data could be accessed by using a relabeling cast to convert `this` to a Pubs* and use the result to access the name. In the current ISO standard C++, even this cast is prohibited, and there seems to be *no way* that a function in the Book class can access data in the Pubs class, even when that data is public.

12.1.2 Ctor Initializers

The difference between initialization and assignment is important in both C and C++ because the rules for initialization are more liberal than the rules for assignment. For example, you can and must initialize a const variable but you cannot assign a new value to it.

In C++, the differences are even more important because operator= and the copy constructor may have different definitions. Thus, ctor initializers are a necessary part of C++, and we must be able to use ctors to initial the base-class portion of an object in a derived class. Construction and initialization happen in this order:

- Space on the run-time stack is allocated for the core portion of the new object. This includes space for the core portion of all inherited and new data members of the class. Base class members come first in this object, followed by members declared in the derived classes, in order.

Once the core space exists, the extensions must be created and both core portions and extensions must be initialized. These tasks are always done in the same order, starting at the first data member and progressing downward (in the source code) or to higher addresses (in memory).

- If the object is of a derived class, the members of the base class must be initialized first, since other parts of the object might refer to them. To do this, one of the constructors for the base-class is selected and run. It *initializes* the core portion of the base-class object (which has already been allocated). In the process, it constructs the extensions for this part of the new object.
- If the constructor for the base class requires parameters, they must be supplied by a ctor of this form:

  ```
  BaseClassName( argument-list )
  ```

  If the base class has more than one constructor, the compiler will select the one whose parameter list matches the list of agruments given by the ctor. If no ctor is given for the base class, the compiler will use the default constructor, if it exists.

  If an object (like a Book object) has a base class (such as Printed) that is also a derived class, the constructor for the middle-level class must pass on parameters to the constructor of the original base class.
- Next, the remaining ctor initializers are used to initialize the data members of the object. These ctors have the form:

  ```
  member_name( initial_value )
  ```
- Finally, the code portion of the object constructor is run. This code can do anything. It is usually used to allocate the extension portion of the object, set various fields to 0, and connect pointers into a legal and meaningful data structure.

12.2 Visibility and Protection Level

Protected members. Previously, we have used just two protection levels: *private* and *public.* A third level, *protected*, is intermediate between these two. A public member can be read or written by any part of the program in which its name is known. A private member can be used only by the functions in the same class. A protected class member can also be used by functions of any derived class. In "family" terminology, a Printed keeps his bedroom private, shares protected resources (the home) with his Bookren and his grandBookren but not with strangers, and may provide some resources (such as a sidewalk) for public use.

Public and private derivation. Members of a base class can be declared as private, protected, or public. During a derivation step, the protection level can be kept the same (by using public derivation) or tightened up (by using private or protected derivation). If public derivation is used, inherited members have the same protection in the derived class as in the base class. With protected derivation, public members become protected in the derived class. With private derivation, all inherited members become private.

The chart in Figure 12.3 summarizes the effects of each of the possible protection combinations. In it, the first column lists members of the Pubs class with protection levels, the second lists the three ways the Printed class can be derived from the Pubs class. In all cases, the Book class is derived publicly from the Printed. The fourth and sixth columns show which members of the Pubs class are visible in the derived class after the first and second derivation steps. From the chart you can see:

- The *protection level* of a member in a derived class (Printed or Book) is the maximum of the protection level in the base class (Pubs) and the styles of all following derivation steps.
- *Accessibility* in the second (Printed) is determined by the *protection level* of a member in the first class (Pubs). The functions of the derived Printed class can access the public and protected members that have been inherited, but not the private members. The inherited private members are there and take up space in a Printed object, but they are "invisible" to the functions of the Printed class. To use such members, a Printed class function would call a public or protected Pubs function.

- *Accessibility* in the third class, (Book), is determined by the *style* of the first derivation (Printed:Pubs). For example, suppose we tried to write the following code in Book::print():

  ```
  cout <<"\n\tMy Pubs is " << Pubs::name<<'\n';
  ```

  If Printed is derived privatly from Pubs, this code generates a compile-time protection error: "member 'Pubs::name' is private in this context". However, the same code in Printed::print() is legal because in the context of the Printed class, name is public.

  In earlier versions of standard C++, the Pubs data could be accessed by using a relabeling cast to convert `this` to a Pubs* and use the result to access the name:

  ```
  out   <<" my Pubs is " <<((Pubs*)this)->name <<endl;
  ```

  In the current ISO standard C++, even this cast is prohibited, and there seems to be *no way* that a function in the Book class can access data in the Pubs class, even when that data is public.

  In contrast, if protected derivation is used to derive Printed from Pubs, the same code is legal in the Book class. It is OK because Pubs::name is protected in the Printed class and, therefore, visible within Book.

Original protection level in Pubs	Style of 1st Derivation	After 1st derivation Accessible in Printed	Protection in Printed	After 2nd derivation Accessible in Book	Protection in Book
Grands (private)	public	No	private	No	private
serial (protected)		Yes	protected	Yes	protected
name (public)		Yes	public	Yes	public
Grands (private)	protected	No	private	No	private
serial (protected)		Yes	protected	Yes	protected
name (public)		Yes	protected	Yes	protected
Grands (private)	private	No	private	No	private
serial (protected)		Yes	private	No	private
name (public)		Yes	private	No	private

Figure 12.3: How derivation affects protection level.

12.2.1 Inherited Functions

Functions (as well as data members) are inherited, and the protection rules for inherited functions are the same as for inherited data members. Inherited functions play an important role in maintaining privacy: they enable an object to use its inherited private parts that otherwise would be "invisible" to objects in the derived class. The derived class can use the inherited function without modification or redefine it. A redefinition can take the form of an extension or an override.

Function redefinition. The functions of a class are closely tied to the representation of the class. Also, certain functions should be defined for every class (`print, operator<<`) using the same name and with the same general meaning (output the values of all data members in an appropriate format). Taking these two facts together, we see that function-naming conflicts will almost always occur between a derived class and its base class.

An `extension` is a redefinition of the function that calls the inherited version and also does additional work. Almost every class needs to have a redefinition of the print() function. Normally, this redefinition will call the inherited function using the scope-resolution operator, like this: `return Pubs::print( out );`

An `override` is a redefinition that fundamentally changes or adds to the meaning of the inherited function or blocks access to it completely, preventing any further derived classes from using it. In each of the two derived classes above, a new method for `print` is defined that overrides the inherited method. As is typical, the function in the derived class prints some data itself, then calls the inherited function to print the rest of the data.

In a derivation hierarchy, a middle-level class sometimes overrides an inherited function in this way. The effect is to remove that function from the set of functions available to classes further down the inheritance hierarchy. This is a powerful tool, but rarely used.

12.3 Class Derivation Demo

The main program. We instantiate some Pubs, Printeds and Books and use them to illustrate the syntax and semantics of derivation and inheritance.

```
//------------------------------------------------------------------------------
// Class Derivation and Static Class Members                      file: main.cpp
// A. Fischer March 2, 2009
//------------------------------------------------------------------------------
#include "Pubs.h"
#include "Printed.h"
#include "Book.h"

int Pubs::pubCount = 0; // Number of Pubs objects that now exist.
int Printed::prinCount = 0;     // Number of Printed objects that now exist.
int Book::bookCount = 0;        // Number of Book objects that now exist.

//------------------------------------------------------------------------
int main( void ) {
    Pubs A("A-Wesley");
    Printed B("Trade Books", "Pearson");
    Book D( "Anatomy", "Out-of-print", "P-Hall" );
    Book G( "Applied C", "Textbook", "McGraw-Hill" );
    cout <<"\nThe population is:\n"<< A << B <<  D << G;
    bye();
    return 0;
}
/*  ---------------------------------------------------------------------------
 *  Pubs: the base class.                                           file: Pubs.h
 *  Created by Alice Fischer on 3/2/09.
 */
#pragma once;
#include "tools.hpp"

class Pubs {
    private:    static int pubCount;    // Number of Pubs that exist.
    protected:  const int serial;       // Serial number of this instance.
    public:     const char* name;       // Name given in the declaration.

    //--------------------------------------------------------------------------
    Pubs( char* g ): serial(++pubCount), name(g) {
        cerr <<"Creating " <<name <<endl;
    }
    ~Pubs(){
        cerr <<"deleting "<<name
        <<", leaving " <<--pubCount <<" Pubs\n";
    }
    ostream& print( ostream& out) const {   // Name, rank, serial number.
        out <<name <<" #" <<serial <<" out of " <<pubCount <<'\n';
        return out;
    }
};
inline ostream& operator<< (ostream& out, const Pubs& x){ return x.print(out); }
```

12.3.1 Inherited Data Members

Each derivation step adds members to the ones present in the base class. We say that the base-class members are *inherited* by the derived class. An object of the derived class starts with the data members of the base class, followed by the new members declared within the derived class. You could say that the derived object is an extension of the base object. (To do the same derivation in Java, you would write `class Printed extends Pubs`.) Some classes have static data members that are allocated in a non-contiguous part of memory. These members are also inherited.

For example, the first part of a Printed object is a Pubs object. In addition, a Printed has three more data members (name, serial, and prinCount) making a total of six data members in a printed object. Similarly, as shown in the diagram below, the first part of a Book object is a Printed object, to which Book adds another static member (bookCount) and two more ordinary members (name, serial), for a total of nine data members. A book *has* all these members even though some of them are allocated remotely and others are private and cannot be accessed from methods in the Book class.

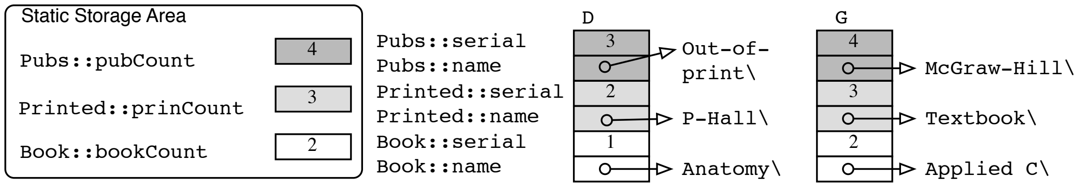


Figure 12.2: A derived object contains an object of its base class.

The data diagram in Figure 12.2 shows the last two objects allocated by `main()`, two Books named `D` and `G`. Each has nine data members: three shared members in the static storage area and six instance members. Medium gray denotes the members that were inherited from the Pubs class; those that came from the Printed class are light gray. Null characters are denoted by a backslash. This diagram is complicated by that fact that all three classes have static members that are stored in another area of storage. The three static members are shared by all `Book` objects in the program.

```
/*  ---------------------------------------------------------------------------
 *  Illustrates private derivation and static class members      file: Printed.h
 *  Created by Alice Fischer on 3/2/09.
 */
#pragma once;
#include "Pubs.h"
//-------------------------------------------------------------------------------
class Printed : Pubs {
    private:    static int prinCount;   // Number of Printed objects that exist.
    protected:  const int serial;       // Serial number of this instance.
    public:     const char* name;       // Name given in the declaration.

    Printed( char* np, char* pub ): Pubs(pub), serial(++prinCount), name(np) {
        cerr <<"Creating " <<name <<" based on #" <<Pubs::serial
        <<" Pubs named " <<Pubs::name <<endl;
    }
    ~Printed(){ cerr <<"deleting "<<name <<", leaving " <<--prinCount <<" Printeds\n";}

    ostream& print( ostream& out) const {
        out <<name <<": " <<" #" <<serial  <<"\n\twhose base object is ";
        return Pubs::print( out );
    }
};
inline ostream& operator<< (ostream& out, const Printed& x){ return x.print(out);}
```

```
/*  ---------------------------------------------------------------------------
 *  Public Derivation and static const class member.               file: Book.h
 *  Created by Alice Fischer on 3/2/09.
 */
#pragma once;
#include "Printed.h"
//-------------------------------------------------------------------------------
class Book : public Printed {
    private:    static int bookCount;   // Number of Book objects that exist.
    protected:  const int serial;       // Serial number of this instance.
    public:     const char* name;       // Name given in the declaration.

    Book(char* b,char* np,char* pub): Printed(np,pub), serial(++bookCount), name(b){
        cerr <<"Creating " <<name
        <<" based on #" <<Printed::serial <<" Printed named " <<Printed::name
 88             <<endl;
 89         }
 90         ~Book(){cerr <<"deleting "<<name <<", leaving " <<--bookCount <<" Book\n";}
 91
 92         ostream& print( ostream& out) const {
 93             out <<name <<" #" <<serial
 94                 <<"\n\tis " <<Printed::name <<endl
 95                 <<"\tmy Pubs' name is hidden from me." <<endl;
 96
 97             // out << Pubs::name;       // error: Pubs::name is inaccessible.
 98             // Pubs pp = (Pubs)(*this); // error: Pubs is an inaccessible base of Book.
 99             // Printed p = (Printed)(*this);   // This line compiles, triggers copying.
100             // out << p.Pubs::name;     // error: Pubs is an inaccessible base of Printed.
101             return out;
102         }
103     };
104     inline ostream& operator<< (ostream& out, const Book& x){ return x.print(out); }
```

Ctors Required. Every C++ object is built by constructing the base portion, then constructing the composed parts, then calling the class constructor. This means that various parts must be initialized by ctors – doing it in the constructor is too late. In this demo, the constructors for Printed and Book illustrate this principle.

- The constructor for Printed starts on line 61 and has three ctors, all required. The base class, Pubs, does not have a default constructor, so we must pass parameters to it. This is done by a ctor giving the name of the base class, with parentheses enclosing the appropriate arguments: `Pubs(pub)`.
- The serial number is a const int, so it must be initialized in a ctor, not assigned later. This ctor must follow the ctor for the base class, since the base class members form the first part of the object and the ctors must be in order. We see code that increments the shared counter and initializes the serial number: `serial(++prinCount)`.
- The third ctor, `name(np)` initializes the name field to the string literal written in the program. This does not work for strings read as input, since it does not allocate any new space to store the characters of the string.
- The constructor for Printed starts on line 85, It also has three ctors for its three parts. Note that the ctor for the base class (Printed) has two arguments in parentheses because the Printed constructor needs two parameters.

Access to inherited members. Various methods in the Printed and Book classes access inherited members.

- The Printed constructor prints trace comments that include two data members from the base class: `Pubs::serial` and `Pubs::name`. The scope-resolution operator must be used to print them because the Printed class has members with the same names.
- These two inherited members are visible in the Printed class because `Pubs::serial` is protected (not private) and `Pubs::name` is public. The derivation method was private derivation, but that does not restrict visibility `at this level`.
- The Printed::print() method also uses the scope-resolution operator to call the inherited print() method. Without the `Pubs::`, this would be a recursive call (a bug).
- The constructor and print() method in Book do similar things using `Printed::`.
- Since private derivation was used to create Printed, all the members of the Pubs class become private in Printed. Therefore, Book cannot see any of them, even those that were originally public. The evidence for this is on lines 97..100. The compiler will not compile anything in the Book class that tries to breach the privacy of the Pubs class.

The output. The middle block of output is the “normal” program output. The first block of lines was printed by the constructors, the last block by the destructors. This makes clear the order in which the pieces of these objects are created and deleted. Note that as objects are created, the serial numbers increase in each class that is part of the created object.

```
Creating A-Wesley
Creating Pearson
Creating Trade Books based on #2 Pubs named Pearson
Creating P-Hall
Creating Out-of-print based on #3 Pubs named P-Hall
Creating Anatomy based on #2 Printed named Out-of-print
Creating McGraw-Hill
Creating Textbook based on #4 Pubs named McGraw-Hill
Creating Applied C based on #3 Printed named Textbook

The population is:
A-Wesley #1 out of 4
Trade Books:  #1
    whose base object is Pearson #2 out of 4
Anatomy #1
    is Out-of-print
    my Pubs' name is hidden from me.
Applied C #2
    is Textbook
    my Pubs' name is hidden from me.


Normal termination.
deleting Applied C, leaving 1 Book
deleting Textbook, leaving 2 Printeds
deleting McGraw-Hill, leaving 3 Pubs
deleting Anatomy, leaving 0 Book
deleting Out-of-print, leaving 1 Printeds
deleting P-Hall, leaving 2 Pubs
deleting Trade Books, leaving 0 Printeds
deleting Pearson, leaving 1 Pubs
deleting A-Wesley, leaving 0 Pubs
```

Chapter 13: Templates

Templates are patterns. According to Merriam-Websters Unabridged dictionary:

> **template:** Something that establishes a pattern.
>
> **archetype:** The original model, form, or pattern from which something is made or from which something develops.

From a review of *Effective STL : 50 Specific Ways to Improve Your Use of the Standard Template Library* by ScottMeyers:

> It's hard to overestimate the importance of the Standard Template Library: the STL can simplify life for any C++ programmer who knows how to use it well. Ay, there's the rub.

Templates are the archetypes behind the "container" data structures whose processing methods depend only on the structuring method and not on the type of the data contained in the structure. The C++ standard library supports a set of pre-defined templates that implement stacks, queues, trees, and hash tables. A programmer might use the STL templates, extend them, or define entirely new template classes.

A template consists of declarations and functions with one or more type parameters or non-type parameters. Type parameters are used to represent the base type of a data structure. Non-type parameters are relatively new in C++; they are generally integers and are used to define such things as array lengths. Both kinds of parameters can be given default values.

By itself, a template is not compilable code. Before it can be compiled, actual types must be supplied to replace the type parameters. The replacement process is called *instantiation* and happens at compile time. The result of instantiation is a class declaration and definitions of the class functions.

13.1 Basic Template Syntax

A complete class template consists of a parameterized class declaration together with all its class functions and all related functions such as an extension of operator¡¡. The entire template is written within a .hpp file because it is a declaration, not compilable code and all these parts must be available at compile time for instantiation by a client class.

- A template declaration starts with a "template" line: the keyword `template` followed by angle brackets enclosing another keyword, `class`, and a name for the type parameter (or parameters). It is customary to name template parameters with single, upper-case letters. Most templates have only one parameter. If there are two or more, the parameter names are separated by commas. Examples:

  ```
  template <class X>
  template <class K, class T, int n>
  ```

- If a class template has remote functions or an extension for the output operator, each function (lines 32, 36, 44, 51) must start with a "template" declaration like the one that begins the class declaration (line 9). C++ supplies no reasonable syntax that lets us declare `template <class T>` once and include all the parts that are needed within its scope.
- Following each template declaration is a class or function declaration with normal syntax. Within this declaration, the name(s) of the type parameter(s) are used to refer to the future base type of the class.
- Being part of a template declaration does not require any extra words or punctuation within the class. Specifically, we refer to the class name within this part of the code without using angle brackets.

- A prototype or the definition of an inline function in a template is the same as for a non-template. (Note the definition of `FlexArray::flexlen` and the prototype for `FlexArray::operator[]`.)
- The definition of each remotely defined function starts with the name of the template WITH angle brackets and the name(s) of the parameters. For example, the name of the put function is: `FlexArray<T>::put`
- Even though operator `<<` is not part of the class, its definition must be given as part of the template (lines 32–33) because it depends on the type parameter, T.

A template for a recursively-defined type. Some data structures require definition of a pair of classes such that each class definition refers to the other class. An example is a linked list defined as a friendly pair of classes: List and Cell. The List class must have a data member of type Cell*. The Cell class must have a friendship declaration for List. This circular dependency poses no problem for a non-template declaration: both classes can be declared in one .hpp file, with the dependent class first, followed by the interface class.

However, when we transform this pair of classes into a template pair, a difficulty arises when the compiler tries to process the friend declaration. Suppose we were defining the `Cell<T>` template. In that definition, we must say: `friend class List<T>`. However, because the `Cell<T>` declaration comes before `List<T>`, the compiler gives a fatal error comment for the friend declaration: *Stack is not a template.* This can be cured by adding a *forward declaration*, like this, at the top of the file, above the `Cell<T>` declaration:

```
template <class T> class Stack;        // A forward declaration for Stack<T>.
```

Note: every reference to Cell in the List class must say `Cell<T>`, and every reference to List in the Cell class must say `List<T>`.

13.2 A Template for FlexArrays

```
 1  #ifndef FLEX
 2  #define FLEX
 3  // ========================================================================
 4  // Template declaration for a flexible array of base type T.
 5  // A. Fischer, May 14, 2001                                 file: flexT.hpp
 6  #include "tools.hpp"
 7  #define FLEX_START 4    // Default length for initial array.
 8
 9  template <class T>
10  class FlexArray {
11    protected:// -------------------------------------------------------------
12      int Max;          // Current allocation size.
13      int N;            // Number of array slots that contain data.
14      T* Data;          // Pointer to dynamic array of T.
15
16    private: // --------------------------------------------------------------
17      void grow();      // Double the allocation length.
18
19    public: // ---------------------------------------------------------------
20      FlexArray( int ss = FLEX_START ) : Max(ss), N(0), Data( new T[Max] ) {}
21      ~FlexArray() { if (Data != NULL) delete[] Data; }
22
23      int put( T data );
24      T&  operator[]( int k );
25      int flexlen() const { return N; }
26      T*  extract() { T* tmp=Data; Data=NULL; Max = N = 0; return tmp; }
27      ostream& print( ostream& out ) const {
28          for (int k=0; k<N; ++k) out << Data[k] <<" ";
29          return out;
30      }
31  };
32  template <class T> inline ostream&
33  operator<< ( ostream& out, FlexArray<T>& F){ return F.print(out); }
```

```

// ------------------------------------------- copy a T into the FlexArray.
template <class T> int
FlexArray<T>::put( T data ) {
    if ( N == Max ) grow();    // Create more space if necessary.
    Data[N] = data;
    return N++;            // Return subscript at which item was stored.
}

//------------------------------------------- access the kth T in the array.
template <class T> T&
FlexArray<T>::operator[]( int k ) {
    if ( k >= N ) fatal( "Flex_array bounds error." );
    return Data[k];               // Return reference to desired array slot.
}

// ------------------------------------ double the allocation length.
template <class T> void
FlexArray<T>::grow() {
    T* temp = Data;                    // hang onto old data array.
    Max>0 ? Max*=2 : Max = FLEX_START;
    Data = new T[Max];                 // allocate a bigger one.
    memcpy(Data, temp, N*sizeof(T));    // copy info into new array.
    delete temp;                       // recycle (free) old array.
            // but do not free the things that were contained in it.
}
#endif
```

Turning a working class into a template. If you have a class that implements a general data structure, changing it into a template is easy.

1. Move all the code from the .cpp file to the end of the .hpp file.
2. Insert a template declaration before the class declaration and before every function that is outside of the class.
3. Using the search and replace feature of your editor, replace the name of the class's base type by a parameter name like T.
4. Insert a pair of angle brackets enclosing the parameter name before every occurrence of :: .

The FlexArray class used in prior chapters is similar to, but much simpler than, the STL Vector class. We use it to demonstrate how to define, extend, and use a template class.

13.3 Adapting a Template

All classes, including template classes, should encapsulate their members and provide public functions for all essential tasks. This leads to a design conflict: should a template definition include every possible function that someone might someday need? Or should the class be clean and focus all functions on its primary purpose?

Derivation solves this problem: the base template should be clean and derived templatess should be used to add functionality for special purposes. Sometimes it is useful to have two or more templates available for variations on one basic data structure. For example, the FlexArray template does not supply any functions for searching the array, yet some potential applications of a FlexArray may require searching. In such situations, the solution is to derive one or more variations from a basic template class.

Deriving from a template. Both template classes and non-template classes can be derived from a template, and both derivation patterns are common. Template derivation is used when the original template provides a useful data structure, but a different or more extensive interface is needed. Two FlexArray variations are presented here: the FlexFind template adds a single capability (a find() function) to the class, and the Stack template provides a wholly different interface. A few details should be noted:

- The data parts of the FlexArrray class are protected, not private, because we intend to use derivation with this class.
- Public derivation was used to build the derived template so that all the functionality of FlexArray will remain available to a client program that derives from FlexFind.
- When a class is derived from a template, its constructor **must** have a ctor that supplies a type parameter for the template class, plus any other parameters the template constructor needs.

Changing a template's interface. The stack is a familiar data structure with restrictions on access. The rule for pushing an item onto a stack fits well with the append-on-the-end rule of the FlexArray. However, a Stack does not support random access or sequential search. Because of the basic similarity, we can use a FlexArray to implement a stack. In this example, new functions provide the familiar stack interface, and private derivation is used to hide the inherited FlexArray class members.

```
61  #ifndef STACKT
62  #define STACKT
63  // ==========================================================================
64  // Template class definition for a stack of T objects
65  // Alice E. Fischer  June 10, 2000                          file: stackT.hpp
66  //
67  #include "flexT.hpp"
68
69  template <class T>
70  class Stack : FlexArray<T> {
71      char* Name;
72    public:          //-------------------------------------------------------
73      Stack(char* nm, int sz=4): FlexArray<T>(sz), Name(nm) {
74          cerr << " Create " <<Name <<"  ";
75      }
76      Stack(const Stack<T>& s){ fatal( " Can't shallow-copy stack %s", s.Name ); }
77      ~Stack() { cerr << " Delete " <<Name <<"  ";}
78
79      void  push(T& c)    { put( c ); };
80      T     pop()         { return empty() ? (T)0 : Data[--N]; }
81      T     top()   const { return empty() ? (T)0 : Data[N-1]; }
82      bool  empty() const { return N == 0; }
83      int   depth() const { return N; }
84      ostream& print(ostream& out) const;
85  };
86
87  template <class T> //-------------------------------------------------------
88  ostream&
89  Stack<T>::print(ostream& out) const {
90      out <<"\nStack has " <<depth() <<" items: bottom<[";
91      FlexArray<T>::print( out );
92      out << " ]>top\n";
93      return out;
94  }
95
96  template <class T> inline //------------------------------------------------
97  ostream& operator<< ( ostream& out, Stack<T>& s ){ return s.print( out ); }
98  #endif
```

Adding new functionality. The FlexArray template implements a general strategy for storage management that can be useful in many kinds of situations. The FlexFind template extends the FlexArray template by adding a function, `find`, that performs a sequential search. A `find` function is not part of the basic class because searching is irrelevant in many array applications. The entire definition of the derived class is very brief, having only one constructor and one added function.

The comparison on line 111 illustrates a typical template problem. A template is supposed to work with many or all template parameter types, and the template creator cannot know what types will be used in the future as template paramters. However, he must write code that works in a broad range of situations, and this must be done without breaking class encapsulation. The solution is to require a potential class parameter to provide one or more public functions. The template code is then written to use those functions, In this case, any type `F` that is used with this template must provide a definition for `operator ==`.

```
 99   // ===========================================================================
100   // Template variant of the FlexArray that supports sequential search.
101   // This class relies on an etension of == in class F to compare two F's.
102   // Alice E. Fischer  June 10, 2000                            file: stackT.hpp
103   //---------------------------------------------------------------------------
104   template <class F>
105   class FlexFind : public FlexArray<F> {
106     public:
107       FlexFind(int ss = FLEX_START): FlexArray<F>(ss){}
108       bool find( const F key ) const{
109           int k;
110           for (k=0; k<N; ++k) {
111               if (Data[k] == key) return true;
112           }
113           return false;
114       }
115   };
```

13.4 A Precedence Parser: Instantiation of a Template

Instantiation is the process of using a template, with actual arguments, to create a class declaration and implementation. This is done at compile time, when the compiler reaches an instantiation command. In this example, we instantiate the derived template class, Stack$< T >$ twice to make two different kinds of stacks that are used to implement an infix expression evaluator. This simplified evaluator could easily be extended to handle parentheses and non-binary operators, but our purpose here is to explore the use of C++, not to write a general and powerful evaluator. The Eval class is presented first, followed by a small Operator class, the main program, and some output.

Using the Stack template.

- To instantiate a template, the programmer writes a declaration or a call on new using the class name with an argument list in angle brackets. Two examples are given here, lines 134 and 135:
  ```
  Stack<double> Ands;
  Stack<Operator> Ators;
  ```
- Even though the keyword `class` is used in the template declaration, a non-class type such as `double` may be used as an argument. A `struct` type may also be used.
- The name of the resulting class includes the `< >` and the type argument, and each instantiation with a different parameter creates a new class with a unique name. It also creates a new module of compilable code. Here, we create two complete and separate Stack classes, `Stack<double>` and `Stack<Operator>`. Each Stack function is compiled twice, once for `Stack<double>`, then again for `Stack<Operator>`.
- If a class template is instantiated twice with the same arguments, a compilable module is produced the first time and reused the second time. Suppose we added a third instantiation: `Stack<double> Results`. This third call would refer to the class created by the first instantiation, `Stack<double> Ands`, and both Results and Ands would be objects of the same class.

```
116   #ifndef EVAL_H
117   #define EVAL_H
118   // =======================================================================
119   // A. Fischer, June 9, 2002                                 file: eval.hpp
120   //
121   // Parse and evaluate a prefix expression.  All operators are binary;
122   // they are: +, -, *, /, % (a mod b) and ^ (a to the power b)
123   // Whitespace must be used to delimit both operators and operands.
124   // Operands must start with a digit and may or may not have a decimal point.
```

```
125   // Operands must not exceed 30 keystrokes.
126
127   #include "tools.hpp"
128   #include "stackT.hpp"
129   #include "operator.hpp"
130
131   class Eval {
132     private: // -------------------------------------------------------------
133       enum Intype { bad, number, op, end };
134       Stack<double> Ands;     // Stack of operands and intermediate results.
135       Stack<Operator> Ators;  // Stack of operators.
136       Intype classify( char ch );
137       void   dispatch();
138       void   force( int rprec );
139       double expError();
140
141     public: // --------------------------------------------------------------
142       Eval(): Ands("Ands"), Ators("Ators") {};
143       ~Eval(){}
144       static void instructions( void );
145       double evaluate( istream& in );
146       ostream& print( ostream& out );
147   };
148   #endif
```

A private type.

- A private enumerated type is declared (line 133) and used to simplify the input and parsing operations. One enumeration symbol is listed for each legal kind of keystroke and one for bad data. The enum symbols are returned by the function classify() and used in evaluate().
- Note that the type of the value returned by `classify()` is declared within the class (line 136) as simply `Intype`, and is used the same way to declare a variable in the definition of the `evaluate()` function (line 169). However, the return type of the remote function (line 212) must be written as `Eval::Intype` because:
  - The definition of the enumeration is inside the `Eval` class.
  - A remote function is defined outside its class and translated in the global context, not in the context of the class.
  - The encapsulated enum type is not visible to the compiler unless you qualify it with the class name.

A static class function. Static class functions can be called even when no object of the class type exists.

- Line 144 declares a static class function named `instructions`. It is static because we want `main` to be able to call this function before creating the first Eval object. It is a class function (not global) because it gives expert instructions about the usage and requirements of the `Eval` class.
- Note that the word `static` is used in the declaration on line 144 but not in the remote definition on line 156. This function is called from line 300, in main.

```
149   // ============================================================================
150   // A. Fischer, June 9, 2002                                     file: eval.cpp
151   //
152   #include "eval.hpp"
153   #include "operator.hpp"
154
155   // -------------------------------------------------- Instructions for the operator.
156   void Eval::instructions( void ){
157       cout << "This is an infix expression evaluator.\n"
158            << "* Operands start with a digit and may or may not have a decimal point.\n"
159            << "* Operands must not exceed 31 keystrokes.\n"
160            << "* All operators are binary operators.  Parentheses are not supported.\n"
161            << "* Operators are: +, -, *, /, % (a mod b) and ^ (a to the power b).\n"
162            << "* Whitespace must be used to delimit both operators and operands.\n"
```

```
         << "* End each expression with a semicolon.\n\n"
         << "To quit, type semicolon instead of an expression.\n";
}
//------------------------------------------------ Read input and evaluate expression.
double
Eval::evaluate( istream& in ) {
    Intype next;    // Classification of next input character.
    Operator inOp;  // Operator object constructed from inSymbol.
    double inNum;   // Read input operands into this.
    char ch;

    for(;;) {
        in >> ws >>ch;
        if (in.eof()) next = end;
        else next = classify( ch );
        switch( next ){
          case number:
                in.putback(ch);
                in >> inNum;
                if ( Ands.depth() != Ators.depth() ) return expError();
                Ands.push( inNum );
                break;

          case op:
                inOp = Operator(ch);
                if ( Ands.depth() != Ators.depth()+1 ) return expError();
                force( inOp.precedence() );
                Ators.push( inOp );
                break;

          case end:
                if ( Ands.depth() != Ators.depth()+1 ) return expError();
                force( 0 );
                return  Ands.pop();
                break;

          case bad:
          default: return expError();
        }
    }
}

// ------------------------------- Evaluate all higher precedence operators on stack.
void
Eval::force( int rprec ) {
    while( Ators.depth()>0 && Ators.top().precedence() >= rprec ) dispatch();
}

//--------- Decide whether next input char is an operator, a semicolon, the beginning
Eval::Intype                                           // of an operand, or garbage.
Eval::classify( char ch ){
    if (isdigit( ch )) return number;
    switch(ch){
      case '+':
      case '-':
      case '*':
      case '/':
      case '%':
      case '^': return op;
      case ';': return end;
      default : return bad;
    }
}
```

```
226
227  // ------------------------------------------------------------ Evaluate one operator.
228  void
229  Eval::dispatch() {
230      double result;
231      double right = Ands.pop();
232      double left = Ands.pop();
233      Operator op = Ators.pop();
234      switch (op.symbol()) {
235          case '+': result = left + right;        break;
236          case '-': result = left - right;        break;
237          case '*': result = left * right;        break;
238          case '/': result = left / right;        break;
239          case '%': result = fmod(left, right);   break;
240          case '^': result = pow (left, right);   break;
241      }
242      Ands.push( result );
243  }
244
245  // ------------------------------------------------------------------- Error comments.
246  double
247  Eval::expError(){
248      cerr << "\tIllegal expression.\n";
249      print(cerr);
250      return HUGE_VAL;
251  }
252
253  // ----------------------------------------------------------------- Print the stacks.
254  ostream&
255  Eval::print( ostream& out ){
256      out << "\tRemaining contents of operator stack: ";
257      out << "\tRemaining contents of operand stack: " <<Ands;
258      return out;
259  }
```

Evaluation using precedence. Precedence and associativity are used to define the meaning of operators in most modern languages. The evaluate function implements both. Basically, operators are kept on one stack and operands on another.

- When each operator is read, a decision must be made whether to evaluate the *previous* operator or stack the new one.
- If the precedence of the incoming operator is greater than the precedence of the stack-top operator, it is not yet time to evaluate either one, and the incoming operator is added to the stack.
- If the precedence of the two operators is equal, the rule for associativity comes into play. Arithmetic operators need left-to-right associativity, so we evaluate the leftmost, which is the one on the top of the stack. So we pop the stacked operator and evaluate it. Then we compare the incoming operator to the next one on the stack.
- Lower incoming precedence also means that the stacked operator should be popped and evaluated immediately (its operands are at the top of the Ands stack). Once that is done, the incoming operator must be compared to the new top-of-stack, and so on.
- Lines 193–197 handle the end of the expression. In this situation, all operands have been read and stacked, and all operators that are still on the stack must be evaluated, so we force everything with precedence greater than 0 to be dispatched. If there are no errors in the expression, the value that remains on the operand stack is the answer.
- The `force` function is the heart of a precedence parser. It is responsible for comparing the precedence of the incoming operator with the precedence of the operator on the top of the stack and evaluating as many operators as are appropriate, according to precedence. To evaluate an operator, it calls the `dispatch` function.

- The `dispatch` function (lines 228–243) pops two operands from the `Ands` stack and one operator from the `Ators` stack, interprets the operator, calls the appropriate C operator or library function, and puts the result back on the `Ands` stack.

Other notable things.

- `HUGE_VAL` is defined in `math.h`. It is the largest representable floating-point value, and is used here to signify an error.
- Error checking is done in the `classify` function (line 223) and throughout the `evaluate` function (lines 182, 188, 194, and 200) so that unsupported operators and ill-formed expressions are caught as soon as possible.

13.4.1 The Operator Class

```
260  #ifndef OPERATOR_H
261  #define OPERATOR_H
262  // ========================================================================
263  // A. Fischer, June 9, 2002                                file: operator.hpp
264  //
265  #include "tools.hpp"
266  class Operator {
267    private: // ---------------------------------------------------------------
268      char symb;
269      int  prec;
270
271    public: // ----------------------------------------------------------------
272      Operator( char op = ’!’) : symb(op) {
273          switch (op){
274            case ’+’: case ’-’:           prec = 1; break;
275            case ’*’: case ’/’: case ’%’: prec = 2; break;
276            case ’ˆ’:                     prec = 3; break;
277            default:                      prec = -1; break;
278          }
279      }
280      ˜Operator() {}
281      int precedence() const { return prec; }
282      char symbol()    const { return symb; }
283      ostream& print(ostream& out)
284          { return out <<"Symbol: " <<symb <<"  Precedence: " <<prec <<endl; }
285  };
286  inline ostream& operator<<( ostream & out, Operator& op) {return op.print(out); }
287  #endif
```

- We represent an operator as a two-member object: the symbol used to denote the operator, and the precedence of the operator with respect to other operators that are supported. The switch statement in the Operator constructor defines the precedence.
- The precedence is used on line 189, where we move down the stack, popping and evaluating all operators that are higher precedence than the new input.
- The symbol is used on line 234 to select the C++ operator to use to evaluate the current subexpression.
- We use simple “get” functions (lines 281–282) to make these private members of Operator available to the application in a read-only form.

13.4.2 The Main Program

I have written this algorithm in C and Pascal, also. The C++ version is considerably cleaner, simpler, easier to write, and easier to understand. The improvement is amazing to me, and is made possible by having classes with their own constructors, print functions, error handling, etc.

```
288  // ==========================================================================
289  // Template example -- Using a stack template.
290  // Alice E. Fischer June 9, 200file: Evaluate/main.cpp
291  //
292  #include "eval.hpp"
293  // ==========================================================================
294  int main( void )
295  {
296      char buf[256] = "Hello";
297      double answer = 0;
298
299      banner();
300      Eval::instructions();
301      for(;;){
302          cout <<"\n\nEnter an expression: ";
303          cin >> ws;
304          cin.get( buf, 256 );
305          if ( buf[0] == ;) break;
306          istringstream inst( buf);
307          Eval E;
308          answer = E.evaluate( inst );
309          cout <<\n << answer <<" = " <<buf <<endl;
310      }
311      bye();
312      return 0;
313  }
```

Overall operation.

- We start by calling the instructions function. The instructions could be written as part of main, but since they are very closely tied to the capabilities of the Eval class, it is better to define an instructions function there and call it from main.
- Main contains the loop that interacts directly with the user. It reads a line of input, quits if it finds the end-of-job sentinel. Otherwise, it creates and uses a new expression evaluator and prints the result.
- We declare the evaluator as a local object (line 307) for each new expression. Using local declarations for the stringstream and the evaluator makes initialization very simple and avoids any possibility of having the remains of one expression affect the evaluation of the next one.

String-streams.

- Line 306 declares an input string-stream; this class uses a string as its source of data instead of an input file. Here, we initialize the string-stream to an array of characters that was read on line 304. The buf array is passed into the stringstream constructor and its contents become the contents of the stream. This stream is used as the argument to `evaulate` (line 308), which expects the data in the stream to be an infix expression.
- When a string-stream is used, all the standard input functions are available. Here, we use ws (line 175) to skip leading whitespace, eof() (line 176) to test for the end of the input, putback (line 180) to put the most recent character *back* into the stream, and `operator >>` (lines 175 and 181) to read the data into the appropriate type of variable and convert ASCII strings to floating-point numbers. We could do all this without a string-stream, but it would be a lot more work and much less clear.
- The `putback` function may be unfamiliar to some. Basically, it "backs up" the stream pointer by one keystroke (line 180) so that the input character may be read again in a different format. Here, we first read the input into a char variable, then, when we discover that it is numeric, put it back and reread it (line 181) using numeric input conversion. This is much simpler and nicer than using `strtol` or `atoi`.

The output.

```
This is an infix expression evaluator.
* Operands start with a digit and may or may not have a decimal point.
* Operands must not exceed 31 keystrokes.
* All operators are binary operators.  Parentheses are not supported.
* Operators are: +, -, *, /, % (a mod b) and ^ (a to the power b).
* Whitespace must be used to delimit both operators and operands.
* End each expression with a semicolon.

To quit, type semicolon instead of an expression.


Enter an expression: 2 + 3
 Create Ands   Create Ators
5 = 2 + 3
 Delete Ators   Delete Ands

Enter an expression: 3 ^ 2 * 94 % 7 + 2
 Create Ands   Create Ators
8 = 3 ^ 2 * 94 % 7 + 2
 Delete Ators   Delete Ands

Enter an expression: 26 ^ 0.5
 Create Ands   Create Ators
5.09902 = 26 ^ 0.5
 Delete Ators   Delete Ands

Enter an expression: ;

Normal termination.
```

13.4.3 UML for Templates

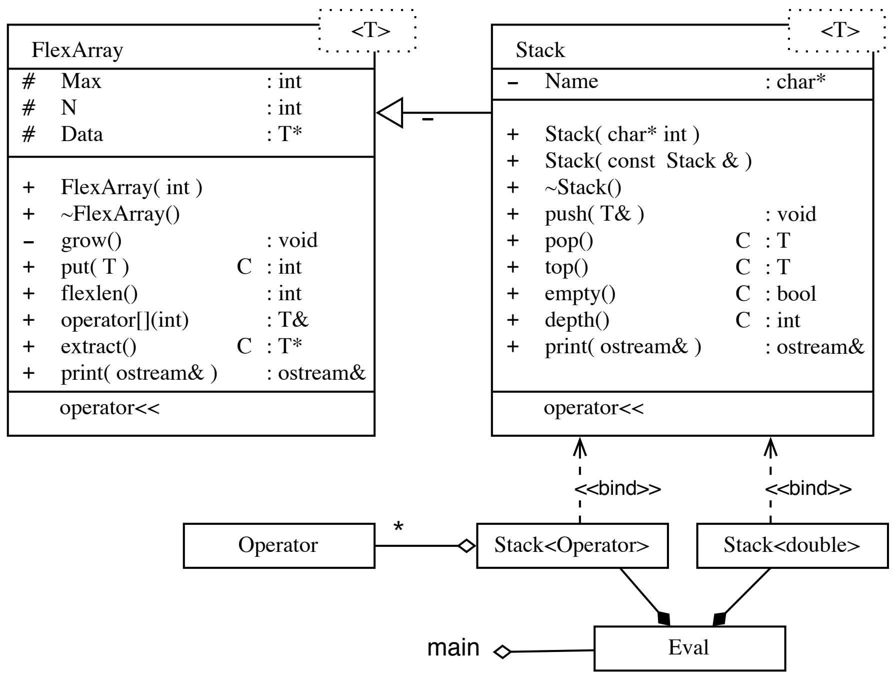


Figure 13.1: UML notation for a template, a derived template, and two instantiations.

We represent a template class in UML by drawing an ordinary class rectangle with a small dotted rectangle covering its upper-right corner. The names of the template parameters are written in the dotted rectangle. Figure 13.4.3 shows two template classes, FlexArray and Stack.

A derived template class is diagrammed like any other derived class. In Figure 13.4.3, the Stack class is joined to the FlexArray class with an ordinary derivation symbol.

In UML, an *instantiation* of a parameterized class is called a *bound element*. A bound element is diagrammed as a class box with a name like `Stack<char>`. with a dotted arrow joining it to the box for the template class. For example, our program that evaluates infix expressions uses two stacks: one for operators, the other for values, so we instantiate the Stack template twice, once as `Stack<Operator>` and once as `Stack<double>`. The instantiations create two new classes that are fully bound and can be compiled. These classes are composed by the Eval class.
It is quite common to derive and instantiate in one step. Suppose class GoodStuff were derived from the instantiation of the stack template with type char:

```
        class GoodStuff :  Stack<char> { ... }
```

The diagram would look like this:

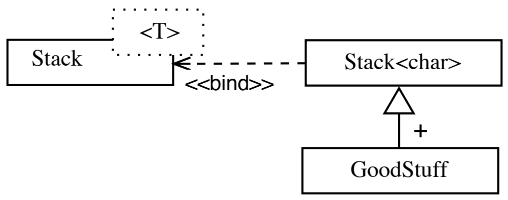


Figure 13.2: UML notation for simultaneous derivation and instantiation.

13.5 The Standard Template Library

STL was designed with extreme care so that it is complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

- To provide standardized and efficient implementations of common data structures as templates, and of algorithms that operate on these structures.
- To produce efficient code. Instantiation of the generic container class is done at compile-time, producing code that is both correct and efficient at run time. This contrasts with derivation and polymorphism which can be used to achieve the same ends but are much less efficient at run time.
- To unify array and linked list concepts, terminology, and interface syntax. Code can be written and partially debugged before making a commitment to one kind of implementation or to another. This permits code to be designed and built in a truly top-down manner,

There are three major kinds of components in the STL:

- Containers manage a set of storage objects (list, tree, hashtable, etc). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.
- Iterators are pointer-like objects that provide a way to traverse through a container.
- Algorithms are computational procedures (sort, set_union, make_heap, etc.) that use iterators to act on containers and are needed in a broad range of applications.

In addition to these components, STL has several kinds of objects that support containers. These include pairs (key–value pairs, for the associative containers), allocators (to support dynamic allocation and deallocation) and function-objects (to "wrap" a function in an object). Function objects can be used in algorithms instead of function pointers.

13.6 Containers

The definition of each container class consists of template code, of course, but that code is not part of the standard. Instead, the standard gives a complete definition of the functional properties and time/space requirements that characterize the container. Two groups of containers are supported: sequence containers (lists, vectors, queues, etc.) and sorted associative containers (maps, sets, etc). The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: any semantics that is operationally equivalent to the model code is permitted. Big-O notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time O(n), is never worse than O(n) but may often be better.

The basic building blocks are precisely organized and, within a group, interchangeable.

Member operations. Some member functions are defined for all containers. These include:

- Constructors: A null constructor, a constructor with one parameter of the container type, and a copy constructor. The latter two constructors operate in linear time.
- Destructor: It will be applied to every element of the container and all memory will be returned. Takes linear time.
- Traversal initialization: begin(), end(), rbegin(), rend(). These mark beginning and ending points for a traversal or reverse-traversal.
- The object's state: size() – current fill level, max_size() – allocation size, empty() – true or false.
- Assignment: `=` Assign one container to another. Linear time.
- Equality: `a == b, a != b` – Returns true or false. Two containers are equal when the sequenes of elements in both are elementwise equal (using the definition of operator== on the element type. Otherwise they are not equal Both take linear time.
- Order: `<, <=, >, >=` Lexicographic comparisons; linear time.
- Misc: a.swap(b) – swaps two conainers of the same type. Constant time.

Sequence Containers

These classes all have the following requirements:

- Constructor with two parameters, int n and base-type element t; Construct a sequence with n copies of t.
- Constructor with two forward-iterator parameters, j and k. Construct a sequence equal to the contents of the range [j, k).
- Traversal initialization: begin(), end(), rbegin(), rend(). These mark beginning and ending points for a traversal or reverse-traversal.
- The object's state: size() – current fill level, max_size() – allocation size, empty() – true or false.
- Assignment: `=` Assign one container to another. Linear time.
- Equality: `a == b, a != b` – Returns true or false. Two containers are equal when the sequenes of elements in a and b are elementwise equal (using the definition of operator== on the element type. Otherwise they are not equal Both take linear time.
- Order: `<, <=, >, >=` Lexicographic comparisons; linear time.
- Misc: a.swap(b) – swaps two conainers of the same type. Constant time.

Sorted Associative Containers

All associatiative containers have two parameters: a type `Key` and an ordering function `comp`, called the "comparison object" of the container, that implements the idea of `<=`. Two keys, `k1, k2` are *equivalent* if `comp(k1, k2` and `comp(k2, k2` both return false. These classes all define the following functions:

- Constructors that include a comparison-object as a parameter.
- Selectors that return the key_type, comparison object-type, and comparison objects.
- Insertion and deletion: four insertion functions with different parameters named `insert()` and `uniq_insert()`. Three `erase()` functions.

- Three iterator functions: `lower_bound(), upper_bound(), and equal_range()`
- Searching: `find(k)`, which returns a pointer to the element whose key is k, (or `end()` if such an element does not exist), and `count(k)` which returns the number of elements whose keys equal `k`.

13.7 Iterators

An iterator provides a general way to access the objects in a container, unifying and replacing both subscripts and pointers. It allows the traversal of all elements in an container in a uniform syntax that does not depend on the implementation of the container or on its content-type.

The underlying value (the value stored in the container) can be either mutable or constant. Exceptional values are also defined:

- Past-the-end values (not dereferenceable)
- Singular values (garbage).

There are five types of iterators, related as shown in figure 13.7. The different types of iterators allow a programmer to select the kind of traversal that is needed and the restrictions (read-only or write-only) to be placed on access. The last two types are called "reverse iterators": they are able to traverse a container backwards, from end to beginning. The examples in Section 13.8 will make all this clearer.

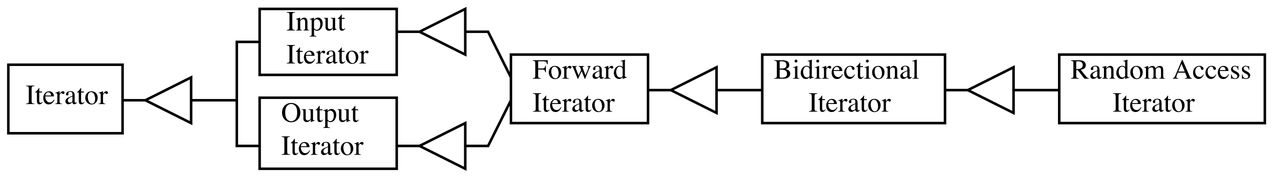


Figure 13.3: Iterator classes form a type hierarchy:

- Input iterators A input iterator class must have a constructor and support the operators `->, ++, *, ==, and !=`, and the `*` operator cannot be used as an l-value in an assignment.
- Output iterators support writing values to a container (using `=` but not reading from it. Restriction: assignment through the same iterator happens only once.
- Forward iterators can traverse the container from beginning to end (using ++)
- Bidirectional iterators can go back and forth in the container; they support the operator `--` in addition to the basic list.
- Random-access iterators must support these operators in addition to the basic list: `[], --, <.  >, <=, >=, -=, and +=`. Further, the * operator must support assignment..

13.8 Using Simple STL Classes

The three STL classes introduced here are string, vector, and map. Vector is a sequence container and Map is a sorted associative container. String is derived from Vector, but is presented separately here because of its great usefulness. Please thank Joe Parker for producing the examples in this section.

13.8.1 String

```
// STL string example. Joseph Parker, modified by A. Fischer, March 26, 2003
#include <string>                                // Header file for STL strings
#include <iostream>
using namespace std;

```

```
 6  int main( void )
 7  {
 8      string str1 = "This is string number one.";   // Allocate and initialize.
 9      // string size
10      cout << "String str1 is: \"" << str1.c_str() <<"\". "
11          <<" Its length is: " << str1.size() << "\n\n";
12
13      cout << "Get a substring of six letters starting at subscript 8: ";
14      string str2 = str1.substr(8,6);
15      cout << str2.c_str() << endl;
16
17      // search first string for last instance of the letter 'e'
18      unsigned idx = str1.find_last_of("e");
19      if (idx != std::string::npos)
20          cout << "The last instance of the char 'e' in string str1 is at pos "
21              << idx <<endl;
22      else cout << "No char 'e' found in string str1" <<endl;
23
24      // search second string for first instance of the letter 'x'.
25      idx = str1.find_first_of("x");
26      if (idx != std::string::npos)
27          cout << "The first instance of the char 's' in string str2 is at pos "
28               << idx <<"\n\n";
29      else cout << "No char 'x' found in string str2\n\n";
30
31      cout << "Now replace \"string\" with \"xxxyyyxxx\".\n";
32      idx = str1.find("string");
33      if (idx != std::string::npos)
34          str1.replace(idx, string("string").length(), "xxxyyyxxx");
35      cout << "str1 with replacement is \"> " << str1.c_str() << "\"\n";
36      return 0;
37  }
```

The output:

```
String str1 is: "This is string number one.".  Its length is: 26

Get a substring of six letters starting at subscript 8: string
The last instance of the char 'e' in string str1 is at pos 24
No char 'x' found in string str2

Now replace "string" with "xxxyyyxxx".
str1 with replacement is "> This is xxxyyyxxx number one."

StringDemo has exited with status 0.
```

Please note these things:

- Line 2: the header function needed for this class.
- Line 10: how to get a C-style string out of a C++ string, and print it.
- Line 11: how to get the length of the string.
- Line 14: creating a new string from a substring of another.
- Lines 18 and 25: searching a string for a letter (first and last occurrenctss).
- Line 34: replace a substring with another string. Note that the old and new substrings do not need to be the same length.

13.8.2 Vector

```
40  // STL_Vector_Example.cpp by Joseph Parker, modified by A. Fischer, March 2003
41  #include <iostream>
42  #include <vector>
```

```
#include <algorithm>

using namespace std;

//-------------------------------------------------------------------
void print(vector<int>& v)      // print out the elements of the vector
{
    int idx = 0;
    int count = v.size();
    for ( ; idx < count; idx++)
        cout << "Element " << idx << " = " << v.at(idx) << endl;
}

//-------------------------------------------------------------------
int main( void )
{
    vector<int> int_vector;         // create a vector of int's

    // insert some numbers in random order
    int_vector.push_back(11);
    int_vector.push_back(82);
    int_vector.push_back(24);
    int_vector.push_back(56);
    int_vector.push_back(6);

    cout << "Before sorting: " <<endl;
    print(int_vector);                          // print vector elements
    sort(int_vector.begin(), int_vector.end()); // sort vector elements
    cout << "\nAfter sorting: " <<endl;
    print(int_vector);                          // print elements again

    // search the vector for the number 3
    int val = 3;
    vector<int>::iterator pos;
    pos = find(int_vector.begin(), int_vector.end(), val);
    if (pos == int_vector.end())
        cout << "\nThe value " << val << " was not found" << endl;

    // print the first element
    cout << "First element in vector is " << int_vector.front() << endl;

    // remove last element
    cout << "\nNow remove last element and element=24 "<< endl;
    int_vector.pop_back();

    // remove an element from the middle
    val = 24;
    pos = find(int_vector.begin(), int_vector.end(), val);
    if (pos != int_vector.end())
        int_vector.erase(pos);

    // print vector elements
    print(int_vector);
    return 0;
}
```

The output:

```
Element 0 = 11
Element 1 = 82
Element 2 = 24
Element 3 = 56
Element 4 = 6

Element 0 = 6
Element 1 = 11
Element 2 = 24
Element 3 = 56
Element 4 = 82

The value 3 was not found
First element in vector is 6
Now remove last element and element=24
Element 0 = 6
Element 1 = 11
Element 2 = 56
```

Please note: A STL vector is a generalization of a FlexArray. You might want to use it because It is not as restricted and presents the same interface as the other STL sequence container classes. The FlexArray, however, is simpler and easier to use for those things it does implement.

- Line 42: the header function needed for this class.
- Line 59: we construct a vector, given the type of element to store within it. Initially this vector is empty.
- Lines 62–66: se put five elements into the vector (at the end).
- Line 69 and 72: we print the vector before and after sorting.
- Line 70: `begin()` and `end()` are functions that are defined on all containers. They return iterators associated with the first and last elements in the container. (It appears that `end()` is actually a pointer to the first array slot past the end of the vector.)
- Line 70: sort is one of the algorithms supported by vector. The arguments to sort are two iterators: one for the beginning and the other for the end of the portion of the vector to be sorted.
- Line 76: we declare an iterator variable of the right kind for vector and use it on the next line to store the position at which a specific element is found.
- Lines 77 and 90: the `find()` function searches part of the vector (specified by two iterators) for a key value (the third argument).
- Line 78 tests for the value `end()`, which is a unique value returned by the find function to signal failure to find the key value.
- Line 82: get the first element in the vector but do not erase it from the vector.
- Line 86: remove the last element from the vector: very efficient.
- Line 92: remove an element from the middle of a vector, using an iterator: not as efficient as `pop()`.

13.8.3 Map

A map is a collection of key/value pairs. When a value is stored into a map, it is stored using one of its fields, called the key field. To retrieve that value, you use the key field to locate it. As STL map can hold only unique keys; if more than one item with the same key can exist, you must use a multimap instead.

This class could be implemented as a hash table or a balanced search tree. Either one would make sense and serve the purpose. We do not actually know which is used, since that is not dictated by the standard. The standard guarantees performance properties, but not implementation. (One might be able to deduce the implementation from the performance properties, though.)

```
100   // STL_Map_Example.cpp by Joseph Parker, modified by A. Fischer, March 2003
101   #include <map>
102   #include <iostream>
103   #include <string>
104   using namespace std;
105
106   int main( void )
107   {
108       // create a map
109       map<int, string> myMap;
110       map<int, string>::iterator it;
111
112     // insert several elements into the map
113     myMap[1] = "Andrea";
114     myMap.insert(pair<int, string>(2, "Barbara"));
115
116     // print all elements
117     for (it = myMap.begin(); it != myMap.end(); ++it)
118         cout << "Key = " << it->first
119             << ", Value = " << it->second
120             << endl;
121
122     // try some operations
123     it = myMap.find(2);
124     if (it == myMap.end()) cout << "\nKey value 2 not found" << endl;
125     else cout <<"\nValue for key 2 = " << it->second << endl;
126
127     it = myMap.find(3);
128     if (it == myMap.end()) cout << "Value for key 3 not found\n";
129
130     // get # of elements in map
131     cout << "\nThe number of elements in myMap is " << myMap.size() << endl;
132     cout << "Now erase one element from map.\n";
133     myMap.erase(2);
134     cout << "The number of elements in myMap is " << myMap.size() << endl;
135     return 0;
136 }
```

The output:

```
Key = 1, Value = Andrea
Key = 2, Value = Barbara

Value for key 2 = Barbara
Value for key 3 not found

The number of elements in myMap is 2
Now erase one element from map.
The number of elements in myMap is 1

MapDemo has exited with status 0.
```

Please note:

- Line 101: the header function needed for this class.
- Lines 109 and 110 create a map object and an appropriate iterator. Note that both require two parameters: the type of the key field and the type items stored in the container.
- Lines 113 and 114 insert two pairs into the map. Note that the second pair is constructed within the argument list of the insert function.
- Lines 116–120 iterate through the container and print each element. Note that each pair has two members, named first and second, and that these members are used to access both the key value and the data.
- Lines123 and 127 call `map::find()`, supplying the key. Compare this to to the call on `vector::find()` in line 77: the required parameters are different but both return an iterator that points to the found item.
- Lines 124 and 128 test for success of the find operation by comparing the result to `myMap.end()`. This is exactly like the test on line 78 in the vector example.
- Lines 131 and 134 get the number of pairs stored in the map.
- Line 133 removes a specific pair from the map, given its key value.

Conclusion The three STL classes introduced here are among the simplest and most useful. But these examples are "only the tip of the iceberg"; the capabilities of these and other STL classes and algorithms go far beyond what is shown here. The three examples are only a starting point from which the student can continue to learn and master this important aspect of modern programming practice.[1] [2]

[1]The Schildt textbook contains detailed and up-to-date material covering all of the standard templates.

[2]I am also using the first STL book – Musser and Saini, *STL Tutorial and Reference Guide*, and I find it quite readable.

Chapter 14: Derivation and Inheritance

In life and in understanding a complex program...

A picture is worth a thousand words.

This chapter consists of one interactive game program that uses templates, derivation, a makefile, a stringstore, a flexarray, and some C coding "tricks" that are worth knowing. The game output is presented first to familiarize you with how the game works. Following that are the makefile, main program, and pairs of .hpp and .cpp files, with notes on each. These are documented by several kinds of "pictures", each of which illustrates a different aspect of the application: a module dependency chart, a data structure diagram, UML class diagrams, and a function call chart.

14.1 Playing

This program plays an interactive word-guessing game called hangman. In this game, the leader (the computer) selects a secret word and displays a line of dashes with one dash for each letter. The player must guess letters, one at a time, and try to figure out what the hidden word is. A sample game is included here, for those who are unfamiliar with it. Suppose the computer chose "hippopotamus" as the secret word. The player would see:

```
---------Constructing Hangman ----------
Please enter name of vocabulary file (or ENTER to quit): vocab2.in

---------- Welcome to Hangman ----------
You win if you can guess the hidden word.
You lose if you guess 7 wrong letters.


Puzzle is: <[ _ _ _ _ _ _ _ _ _ _ _ _ ]>

    Letters left--> a b c d e f g h i j k l m n o p q r s t u v w x y z
    Bad guesses---> _ _ _ _ _ _ _
Guess a letter:
```

After one wrong guess (e) and one correct guess (a), the board would look like this:

```
 You guessed 'a'.  You scored!


Puzzle is: <[ _ _ _ _ _ _ _ _ a _ _ _ ]>

    Letters left--> b c d f g h i j k l m n o p q r s t u v w x y z
    Bad guesses---> e _ _ _ _ _ _
```

After several more guesses (i,o,y,u,t,p,m, and finally s), the game is won and you see:

```
 Puzzle is: <[ h i p p o p o t a m u s ]>

    Letters left--> b c d f g j k l n q r v w x z
    Bad guesses---> e y _ _ _ _ _
Congratulations -- you win!

You won 1 time out of 1 try.

Type p to play another round, q to quit: q

----------- Have a good day! -----------
```

14.1.1 The Hangman Application

A makefile defines its application: it lists the required parts and describes the relationships among them. The information in the makefile is presented graphically in Figure 14.1.

The makefile.

```
# Rule for building a .o file from a .cpp source file -------
.SUFFIXES: .cpp
.cpp.o:
    c++ -c $(CXXFLAGS) $<

# Compile with debug option and all warnings on. ------------
CXXFLAGS = -g -Wall

# Object modules comprising this application ----------------
OBJ = main.o game.o board.o sstore.o rstrings.o words_d.o tools.o

game: $(OBJ)
    c++  -o game $(CXXFLAGS) $(OBJ)

# Delete .o and exe files and force recompilation. ----------
clean:
    rm -f $(OBJ) game

# Use tools source file from grandparent directory ----------
tools.o:   tools.cpp tools.hpp
    c++ -c $(CXXFLAGS) tools.cpp -o tools.o

# Dependencies ----------------------------------------------
main.o:     main.cpp game.hpp  board.hpp flexT.hpp
game.o:     game.cpp game.hpp board.hpp rstrings.hpp sstore.hpp flexT.hpp
board.o:    board.cpp board.hpp words_d.hpp
words_d.o:  words_d.cpp words_d.hpp words.hpp tools.hpp
rstrings.o: rstrings.cpp rstrings.hpp sstore.hpp flexT.hpp
sstore.o:   sstore.cpp sstore.hpp tools.hpp
```

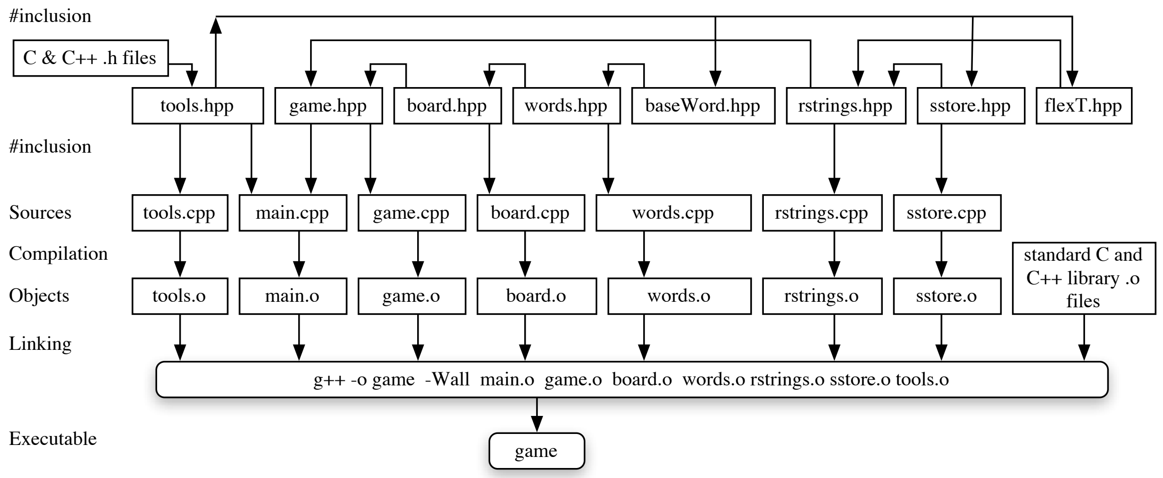


Figure 14.1: Makefile graph for Hangman: The files and their dependencies.

14.1.2 Hangman: The Main Program

The vocabulary file. This program is designed to construct a hangman game then play one or more rounds of it. Line 46 asks whether the user has given the name of a vocabulary file as a command-line argument. If so, that file name is sent to the Game constructor. If not, the NULL pointer signals the Game constructor to use a default file.

A conditional operator is used appropriately in the argument list for the Game constructor. It tests a condition and returns something (a pointer) in either case. C syntax requires that the same type of object must be returned by both clauses of a conditional operator. In this program, the ? asks whether the user typed more than one thing (the program name) on the command line. If so, the additional command field is returned (it should be a file name). If not, a NULL is returned. The overall code is simplified considerably by using the conditional operator instead of an if-else statement.

```
30  // ====================================================================
31  // Hangman program:    Let the user guess words from the vocabulary file.
32  // A. Fischer, May 13, 2001                                file: main.cpp
33  #include "tools.hpp"
34  #include "game.hpp"
35  //---------------------------------------------------------------------
36  int main( int argc, char* argv[] )
37  {
38      char response;              // For query, "Play again?"
39      int wins = 0, rounds = 0;   // For keeping score.
40      const char* timeword;       // For grammatical output: time, times.
41      const char* tryword;        // For grammatical output: try, tries.
42
43      cout << "\n--------- Constructing Hangman ----------\n";
44      Game g( argc>1 ? argv[1] : NULL );      // Get optional file name.
45
46      cout << "\n---------- Welcome to Hangman ----------\n"
47              "You win if you can guess the hidden word.\n"
48              "You lose if you guess " << HANG_MAX << " wrong letters.\n";
49      do {
50          wins += g.play();                   // Play one round of game.
51          rounds++;
52          timeword = (wins == 1) ? "time" : "times";
53          tryword = (rounds == 1) ? "try" : "tries";
54          cout << "\nYou won " << wins << " " << timeword
55               << " out of " << rounds << " " << tryword
56               << ".\n\nType p to play another round, q to quit: ";
57          cin >> response;
58      } while (tolower(response) == 'p');
59      cout << "\n----------- Have a good day! -----------\n\n";
60  }
```

Instructions. The instructions given on lines 46 through 48 will not be repeated before each round. The loop on lines 49 through 58 plays one round and queries the user about whether to continue.

One round of the game. Line 50 plays one round of the game and returns the result: 1 for a win, 0 for a loss. The wins and losses are tallied and displayed when user asks to quit. Lines 52 and 53 use string variables and conditional operators to select singular or plural wording so that the final score message will be displayed in correct English. This kind of care is not necessary in student projects but makes a difference in the perceived quality of a commercial job.

14.1.3 Call Graphs

Different kinds of documentation lead to different insights into the structure of an application. An object diagram shows us the way our actual storage is organized and used at run time. A class diagram tells us what properties each kind of object has, and which ones can be used in other classes. A flow chart shows us possible execution paths. An event trace (also called a *sequence diagram* is like a flow chart but also shows how control passes back and forth between classes during execution. A fifth kind of graphical documentation is the call graph—a static chart showing which functions can call which other functions at some point in the program. Call graphs can be useful during debugging for tracking down all possible ways for control to get to any particular function.

Figure 14.1.3 is a call graph for Hangman. To minimize the complexity of the diagram and focus on the class functions, calls on fatal, new, delete, and iostream functions have been omitted from the chart. From this chart, you can see that execution is divided into two major phases. First, a game is constructed, then played. When one round is played, a random string is selected, a board is constructed, and finally, the board is played.

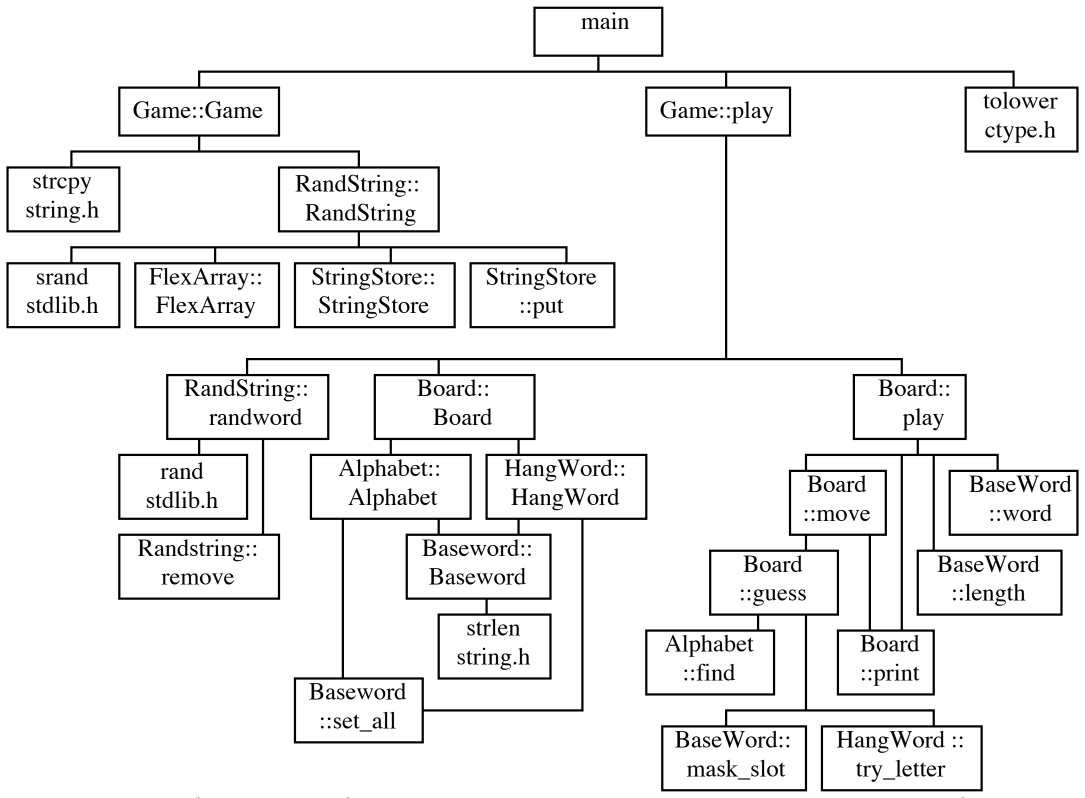


Figure 14.2: A call graph for the Hangman game: How control can reach each function.

14.1.4 UML Diagrams: A View of the Class Relationships

A UML diagram gives another static view of the application; it illustrates the data types being used, the protection level of each part of each class, and the possible ways that one class can access or use another.

The classes used in this application fall into two nearly separate subsystems: Game and Board, together with the three Word classes implement the form and function of the game itself, while RandString, StringStore, and FlexArray implement the database from which the game selects puzzle words. These two subsystems are diagrammed separately – the association between Game and RandString is the only connection.

In Figure 14.1.4 we see that two classes are derived from BaseWord. BaseWord defines a data structure and a set of functions that implement one basic behavior. From it, we derive two sub-classes that define variations on the basic theme and have different initialization, search, and display rules.

From the UML, you can see that the BaseWord class is not associated with or aggregated by any other classes and, in fact, is not instantiated by the program. It is used only as a basis for deriving Alphabet and Hangword, which form the basis for the gameboard display. This follows a basic OO-design guideline:

Don't instantiate a class that you also derive from.

The goal of this rule is to minimize the conflict between the requirements of a base class, which must be clean and general, and a class that must serve the specific needs of an an application.

In Figure 14.1.4 we see an application-specific class, RandString, that is built out of two utility data-structure classes: it is derived from FlexArray and it aggregates StringStore. The RandString class is a container for words that randomly selects and returns one word at a time. The word is then removed from the container so it cannot be reused. In one sense, the new class, RandString "wraps" the two familiar classes in a new behavior, with only a fraction of the work that would be required to program the new class from scratch. It is a typical demonstration of the potential power of class libraries.

The FlexArray class has been rewritten, finally, in its proper form: as a class template. In going from FlexArray to RandString, we simultaneously bind the template parameter to `char*` and derive from the resulting class. This combination of template instantiation and derivation is very common because you normally want to write application-specific functions to handle the general data structures for which templates are used.

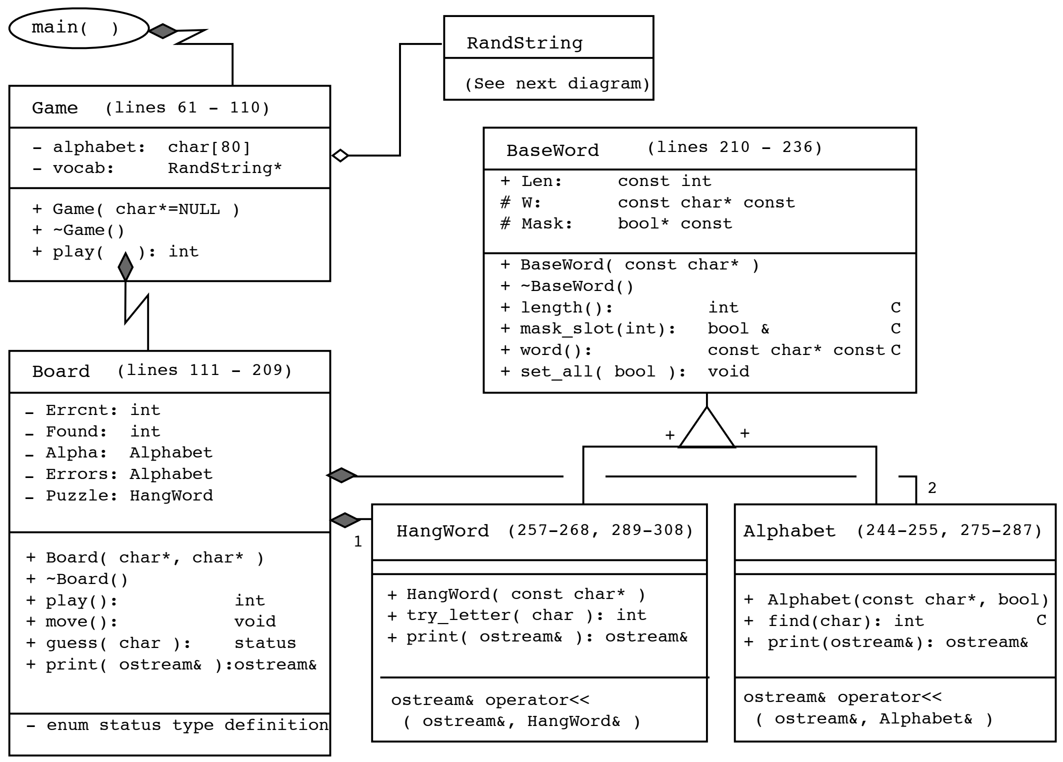


Figure 14.3: The game classes in Hangman: UML describes class relationships.

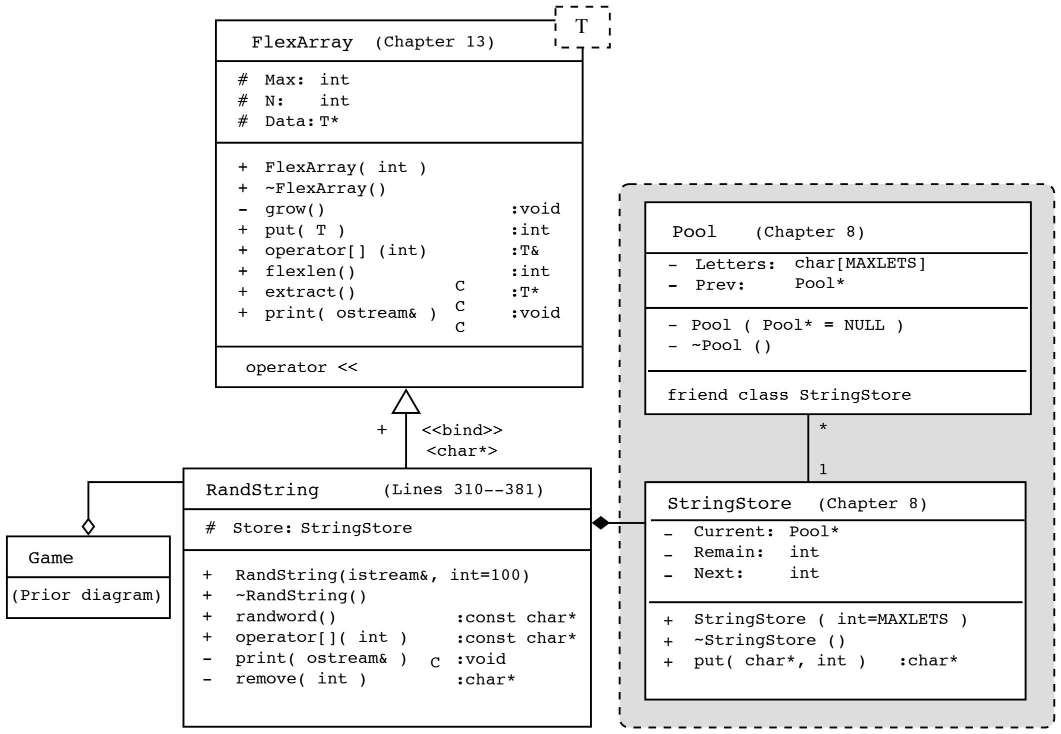


Figure 14.4: The vocabulary classes in Hangman.

14.1.5 The Hangman Data Structures

A data diagram illustrates the allocations, connections and contents that have been created by the program at one specific moment at run time, given a specified sequence of inputs. This kind of picture is particularly helpful when pointers are used to build a complex structure. It does not help us know how the program reached the particular state that is illustrated, but it gives us an appreciation for the overall complexity of the application. A data diagram can be a great help to a reader who is trying to understand how and why a program works (or fails to work).

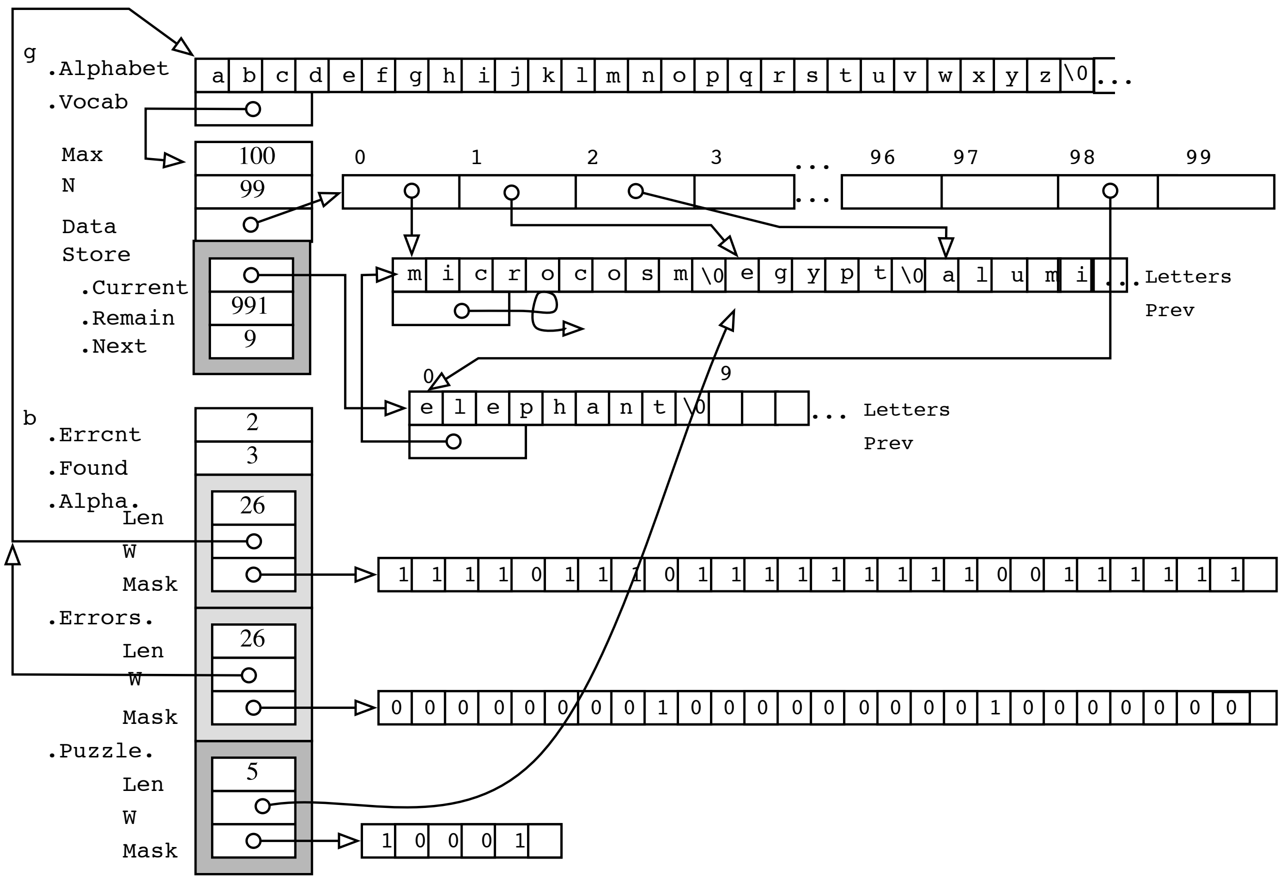


Figure 14.5: Data structures for the Hangman program: A snapshot during execution.

Program notes throughout this section will refer to the data diagram in Figure 14.1.5. It illustrates the data structures constructed for a representative game using a vocabulary file with 99 words, when the puzzle word is “egypt”. The data structure, above, are shown after the fourth guess. (Two guesses were correct, two were wrong). Here is the current gameboard:

```
Puzzle is: <[ e _ _ _ t   ]>

    Letters left--> a b c d f g h j k l m n o p q r u v w x y z
    Bad guesses---> i s _ _ _ _ _
Guess a letter:
```

14.2 Hangman: The Game and Board Classes

The main program creates a Game (s 44) which, in turn, creates a Board. Game::play() calls Rand-String::randword() to select a random string from the vocabulary, builds a gameboard around it and plays one round of hangman. The Board class implements the gameboard with the help of Alphabet and HangWord, both derived from BaseWord.

A new gameboard is created by Game::play() (lines 50, 108) for each round. Since the array members of Board are built around the hidden word, each new word requires a new construction job. We could have done the same thing by declaring a Board* as a member of Game, but there would be no advantage in doing it that way because there is no need to use a board after returning from Game::play().

14.2.1 The Game Class

```
// ==========================================================================
// Build a board, play it, and return the score to main.
// A. Fischer, June 4, 2000                                    file: game.hpp
#ifndef GAME
#define GAME

#include "rstrings.hpp"
#include "board.hpp"
// ==========================================================================
class Game {
  private:
    char Alphabet[80];      // Normally the English Alphabet.
    RandString* Vocab;      // The randomized vocabulary list.

  public:
    Game( cstring wordfile = NULL );
    ~Game(){ delete Vocab; }
    int play();             // Play hangman.
};
#endif
```

```
// ==========================================================================
// Implementation of the Hangman game.
// A. Fischer, June 4, 2000                                   file: game.cpp

#include "game.hpp"
// ==========================================================================
Game::Game( char* wordfile ){
    char file_name[80];
    //cerr << "Constructing Game.  ";       // For debugging.
    if (wordfile) strcpy( file_name, wordfile );
    else strcpy( file_name, "vocab.in" );

    ifstream source( file_name );
    if (!source) fatal( "Could not open %s.\nEnding Hangman.\n", file_name );
    //cerr << "File " <<file_name <<" is open for reading." <<endl;

    source.getline( Alphabet, 80 );
    Vocab = new RandString( source );       // Create vocabulary database.
    source.close();
    //Vocab->print( cerr );                 // For debugging.
}

//--------------------------------------------------------------------------
int Game::play() {
    const char* puzzle_word = Vocab->randword();    // pick puzzle word.
    //cerr << puzzle_word;

    Board b( Alphabet, puzzle_word );               // construct game board
    return b.play();                                // play the round
}
```

Notes on the Game code. The Game constructor's first task is to find and open the vocabulary file. If no name was typed on the command line, wordfile will be NULL, otherwise it will point to the selected filename. A filename must be supplied, one way or another. If it was not given on the command line, the program uses a default file, `vocab.in`. In either case, a copy of the file name is copied into the local array and used to open an input stream.

Once the stream is open, the first line of the file is read (line 97) and used to initialize the class member named `Alphabet`. This permits the program to be used with vocabulary from any language. The remainder of the file is used (line 98) to build the vocabulary array (a new RandString structure). the result is shown at the top of Figure 14.1.5: g.Vocab points to a RandString object consisting of a FlexArray and a StringStore with two Pools.

The Game::play() function is the heart of the program: it selects a mystery word (line 105), uses it to construct a board (line 108), plays the board (line 109), and returns the score (1 win or 0 wins) to main(). Details of how to play the board and how to calculate the score are delegated to the Board class.

14.2.2 The Board Class

```
111  // ===========================================================================
112  // Choose a word, use it to create a playing board.
113  // A. Fischer, June 4, 2000                                   file: board.hpp
114  #ifndef BOARD
115  #define BOARD
116
117  #include "words_d.hpp"
118  #define HANG_MAX 7
119  // ===========================================================================
120  class Board {
121      enum status {GOOD_GUESS, BAD_GUESS, NOT_IN_ALPHA, USED_ALREADY};
122      int Errcnt;         // Wrong guesses so far,
123      int Found;          // Number of Letters correctly filled in.
124      Alphabet Alpha;     // Masked alphabet.
125      Alphabet Errors;    // Masked alphabet for error list.
126      HangWord Puzzle;    // Masked mystery word.
127
128    public:
129      Board(const char* a, const char* puz);  // Alphabet and puzzle word.
130      ~Board(){}
131      int play();                             // play a board
132      void move();                            // user interaction for one move
133      status guess(char c);                   // process a guess
134      ostream& print(ostream&);               // print a board
135  };
136  #endif
```

```
137  // ===========================================================================
138  // board.cpp: Implementation for hangman board
139  // A. Fischer, June 4, 2000                                  file: board.cpp
140
141  #include "board.hpp"
142  // ===========================================================================
143  Board::Board(const char* a, const char* puz) :
144      Errcnt(0), Found(0), Alpha(a, true), Errors(a, false), Puzzle(puz){
145      //cerr << "\nConstructing Board.  ";
146  }
147
148  //---------------------------------------------------------- display the board
149  ostream&
150  Board::print( ostream& out ) {
151      out << "\n\nPuzzle is: " << Puzzle << "\n\n";
152      out << "    Letters left-->" << Alpha << "\n";
153      out << "    Bad guesses--->" << Errors;
154
155      for (int k = Errcnt; k < HANG_MAX; k++) out << " _";
156      return out << endl;
157  }
158
159  //------------------------------------------------------------ process a guess
```

```
Board::status
Board::guess(char c) {
    int where = Alpha.find(c);
    if (where == -1) return NOT_IN_ALPHA;
    if ( !Alpha.mask_slot(where) ) return USED_ALREADY;
    Alpha.mask_slot(where) = false;

    int matches = Puzzle.try_letter(c);
    if (matches <= 0) {
        Errors.mask_slot(where) = true;
        Errcnt++;
        return BAD_GUESS;
    }
    Found += matches;
    return GOOD_GUESS;
}

//-------------------------------------------- user interaction for one move
void
Board::move() {
    char ch;
    cout << "Guess a letter: ";
    cin >> ch;
    cout << "You guessed '" << ch << "'";
    switch (guess( ch )) {
       case NOT_IN_ALPHA:
            cout << " -- but it's not in the alphabet." << endl;    break;
       case USED_ALREADY:
            cout << " -- but you guessed it once before." << endl;  break;
       case BAD_GUESS:
            cout << " -- too bad." << endl;                         break;
       case GOOD_GUESS:
            cout << ".  You scored!" << endl;                       break;
    }
    print(cout);
}

//----------------------------------------------------------- play a board
int
Board::play() {
    print(cout);
    while (Errcnt < HANG_MAX && Found < Puzzle.length()) move();
    if (Found == Puzzle.length()) {
        cout << "Congratulations -- you win!" << endl;
        return 1;
    }
    cout << "Sorry, you lose!" << "\nThe answer is: "
         << Puzzle.word() << endl;
    return 0;
}
```

To play a round of hangman, we need a hidden word and an alphabet. To make a good interface, we also need a list of wrong guesses. These three arrays of characters (Puzzle, Alpha, and Errors) are updated and displayed after every guess to help the player make skillful guesses. Thus, our gameboard displays three arrays of letters:

- Puzzle, the mystery-word consisting of underscores and correctly guessed letters.
- Errors, a list of incorrect guesses.
- Alpha, the list of letters that have not yet been guessed.

In addition, the Board object must keep score and end the round when the number of correct guesses (Found) equals the length of the puzzle word, or when the number of bad guesses (Errcnt) reaches the limit, `HANG_MAX`. In Figure 14.1.5, you see the Board, b, and its five components in the lower part of the diagram.

It is much easier to believe in the correctness of a function when its return values have meaningful names than when integer codes are used. For this reason, we define a private enumerated type to describe the possible outcomes of a guess. A value of this type is returned by guess() and used by move() to select a response message for the player. We use this device to clarify, simplify, and modularize the code. The resulting two functions are much clearer that the alternatives: one very long function and/or cryptic integer codes for the outcomes.

Constructing a playing board. The parameters to the constructor are an alphabet and a word that was randomly selected from the vocabulary. This code uses ctors (line 144) to initialize all of the data members, although they are only necessary for the last three. The body of the constructor contains only a debugging message, currently commented out. To return to a debugging phase, one would remove the // marks.

The last three ctors convey arguments from the parameter list of the Board constructor to the constructors of the aggregated class objects. The mystery word is sent to the Puzzle constructor where it becomes the basis for a new playing board. Class members Alpha and Errors are both constructed from the alphabet. Initially, all letters of the Alpha alphabet are set to "true", meaning that they are available, and all letters in the Errors alphabet are set to false, meaning that no errors have yet been made. These initializations will be discussed more fully in the relevant classes.

One round of play. Board::play() is called by Game::play() to play one round of hangman on a newly constructed Board. This function calls Board::move() in a loop (line 201), until the round has been won (all letters have been guessed, lines 202–204) or lost (seven errors have been made, lines 206–207). In both cases, the function announces the result to the player and returns the score to Game::play(). The details of how a move is made are handled by the move() function.

One move. Board::move() prompts for and reads a guess from the user, (line 181–183) calls Board::guess() to analyze the guess (line 184), and displays a message about the result and the new board position. The switch statement illustrates how well-chosen enumeration constants can make code much clearer. A reasonable question is, "why don't we combine the code from Board::move() and Board::guess() into one function and get rid of both the switch and the enumeration. There are two answers. First, the combined function would be very long. Second, it is very helpful to keep high-level logic separate from low-level logic. The switch implements the high-level logic and provides a road map for the other function. In contrast, Board::guess() is filled with many detailed comparisons and counters. The enumeration constants guide the reader and help clarify the purpose of the operations.

Is the guess correct? Given a puzzle, an alphabet, and a guess, there are four possible outcomes. We do a case analysis and use four return statements to simplify the logic by keeping the cases maximally separate from each other.

If the guess is not in the game-alphabet or it has been guessed previously, the player will not be "charged" for the bad guess. To find out, we call Alphabet::find() (line 162) to search for the guess among the legal letters. The return value is the subscript of the guess in the alphabet, if it is a legal letter, otherwise -1. For example, when 'e' is guessed, the subscript 4 will be returned because 'e' is the fifth letter in the alphabet.

If the letter is valid, we then check (line 164) whether it was previously guessed. To do this, we use the subscript returned by Alphabet::find() to index the mask array that parallels the alphabet array. (A result of false means the letter has been used, true means it is still available.)

If the letter is legal and still possible, we set the appropriate position in the mask array (line 165) to "false" to indicate that the letter is now used. Since 'e' has been guessed, we see that Alpha.Mask[4] is false and 'e' has disappeared from the display.

Next, on line 167, we determine whether the guess is correct or wrong by calling Hangword::try_letter() . The result will be the number of letters in the puzzle that match the guess (zero or more). If the answer is zero, we set the corresponding position of the error array to "true", increment the error-counter, and return with an error code (lines 168–171). This will cause the letter to "appear" (in alphabetical order) in the error array the next time it is displayed. In Figure 14.1.5, the letter 's' has been guessed and it is wrong. In response, Alpha.Mask[18] was set to false and Errors.Mask[18] to true. If the guess is correct, we reach line 173, where we add the number of new matches to the match-counter (Found) and return a success code. Play for this board will end when the Found total equals the length of the mystery word.

The return type must be given here as “Board::status”, even though it is written simply as “status” on line 121. This is necessary because the status type is defined inside the Board class and this function definition is outside the class declaration.

14.3 The Word Classes

A mask marks a subset. A masked data structure is the simplest way to denote a subset of the data that is to be used (or not used) for a particular purpose. One or several masks might be used to mark one or several different subsets. The mask field or fields might be members of the structure or might exist in a parallel data structure. The mask fields might be type bool (to denote a simple yes/no choice) or any other enumerated type. Masks are most useful if the condition that they represent is not simple to test for, and if the status of an object changes over time or is checked frequently.

A masked structure lets us sort or process data efficiently and easily. For example, suppose a club membership database has one record for each member. At least two sets of masks might be helpful: one for age (juvenile, teenager, adult, senior) and another for dues status (guest, lifetime, paid-up, due, lapsed). Masks can be useful here because age is messy to categorize and dues status is based on the member’s history.

14.3.1 The BaseWord Declaration

The class BaseWord creates a basic masked string, that is, a string with a corresponding array of bools to indicate whether each letter in the string is currently valid or not. If a letter is valid, it is searched and displayed. If not, it is hidden both from sight and from the computations. You could say that the mask controls access to the individual letters in the array. This is an easy and fast way to select a subset of the data stored in an array. To add an element or remove it, simply change the bit from true to false, or vice versa.

```
// ===========================================================================
// Maskable word base class.
// A. Fischer, June 4, 2000                                    file: words.hpp
#ifndef WORDS
#define WORDS
#include "tools.hpp"
// ===========================================================================
class BaseWord {
  protected:
    const int Len;      // length of word, excluding terminator.
    const char* const W;    // partially concealed word
    bool* const Mask;       // concealment mask

  public:
    BaseWord(const char* st) : Len(strlen(st)), W(st), Mask(new bool[Len+1]){
        //cerr << "\nConstructing BaseWord.  ";
    }
    ~BaseWord() { delete [] Mask; }
    int length() const { return Len; }
    bool& mask_slot(int k) const { return Mask[k]; }
    const char* const word() const { return W; }

    void set_all(bool on_off){  // set false to hide letter, true to expose.
        for (int k=0; k<Len; k++) Mask[k] = on_off;
    }
};
#endif
```

The BaseWord class. In this class, everything is based on a particular string (the argument to the BaseWord constructor) and its length. The members named Len and Mask are constants, so they must be initialized using ctors, *after* the length of the argument string is known. The first ctor that must be executed is the one that measures the string and stores its length in Len; the mask cannot be constructed until that is done. For this reason, we declare Len first in the class.

The class member named W is a constant pointer to a constant string, either the alphabet or a word that was selected randomly from the vocabulary and is still stored there. No copy is made of this string because we do not need to modify it. The word, itself, is constant. The mask array is modified during the play, and those modifications determine which letters are displayed.

14.3.2 The Derived Word Classes

```
// ===========================================================================
// Derived Word classes.
// A. Fischer, June 4, 2000                                  file: words_d.hpp
#ifndef WORDSDERIVED
#define WORDSDERIVED

#include "words.hpp"
// ===========================================================================
class Alphabet : public BaseWord {
  public:
    Alphabet(const char* st, bool on_off) : BaseWord(st) {
        set_all(on_off);
        //cerr << "Constructing Alphabet.";
    }
    int find(char c) const;     // return index of first c in word
    ostream& print (ostream&);  // print an alphabet
};

inline ostream& operator<<(ostream& out, Alphabet& x){ return x.print(out); }

// ===========================================================================
class HangWord : public BaseWord {
  public:
    HangWord(const char* st) : BaseWord(st) {
        set_all(false);
        //cerr << "Constructing HangWord. " <<st;
    }
    int try_letter(char);
    ostream& print (ostream&);  // print a hang word
};

inline ostream& operator<<(ostream& out, HangWord& x){ return x.print(out); }
#endif
```

```
// ===========================================================================
// Implementation for maskable words.
// A. Fischer, June 4, 2000                                  file: words_d.cpp

#include "tools.hpp"
#include "words_d.hpp"
// ================================================= Alphabet class functions
int                             // Return index of first occurrence of c in W
Alphabet::find(char c) const {
    int k;
    for (k=0; k<Len && c != W[k]; k++);     // Loop body is empty.
    return (k==Len) ? -1 : k;
}
//---------------------------------------------------------------------------
ostream&
Alphabet::print (ostream& out) {
    for (int k=0; k < Len; k++)  if ( Mask[k] ) out << " " << W[k];
    return out;
}

// ================================================= Hangword class functions
int       // Count the number of times letter c occurs in puzzle; unmask each
```

```
292 HangWord::try_letter(char c) {
293     int count = 0;
294     for (int k=0; k<Len; k++) {
295         if (c == W[k]) {
296             count++;
297             Mask[k] = true;
298         }
299     }
300     return count;
301 }
302 //-----------------------------------------------------------------------
303 ostream&
304 HangWord::print (ostream& out) {
305     out << "<[" ;
306     for (int k=0; k < Len; k++) out  << ' ' << (Mask[k] ? W[k] : '_') ;
307     out << " ]>";
308     return out;
309 }
```

The Alphabet and HangWord classes. Baseword is a polymorphic class that defines a basic data structure and some of its functions. Alphabet and HangWord are variations of BaseWord. They are derived from BaseWord, so they inherit functions and data from BaseWord. In addition, each has its own set of functions for initialization, use, and display. The first line of a class declaration declares the derivation relationships, if any. In Hangman, Alphabet and HangWord are both derived from BaseWord by public derivation:

```
class Alphabet :  public BaseWord { ...  };
class HangWord :  public BaseWord { ...  };
```

From this relationship we know that:

1. The two new classes are derived from BaseWord so that we can have the same structure but different print functions.
2. The first part of an Alphabet or HangWord object is a BaseWord object.
3. The constructors for Alphabet and HangWord use ctor initializers to provide arguments for the BaseWord constructor.
4. The public/protected/private status of all the inherited members in Alphabet and HangWord is the same as in class BaseWord.
5. The functions of the two derived classes can freely use the protected members of the base class: Len, W, and Mask.

Initializing the masks. A gameboard contains three Words (Puzzle, Alpha, and Errors); each consisting of a char array with a parallel mask array. When a word is displayed, its mask array is checked; a letter in the word is displayed if its mask bit is on (true), and ignored if the bit is off.

During Board construction, the function BaseWord::set_all() is called to initialize the three Word members of the Board. It will set all of the bits of a mask to either true or false, depending on how the instance will be used: bits for the game alphabet are set to true, for the errors to false, and for the mystery word to false. The bool parameter for set_all() comes from a call in the HangWord constructor (line 261) or from the ctor for Alphabet in the Board constructor (line 144). As the game progresses, two mask bits are changed each time a legal guess (correct or incorrect) is made. The alphabet that is displayed grows shorter (as letters are used) and the error list grows longer (as letters are added to it). The mystery word stays the same length, but dashes in the display are replaced by letters each time a correct guess is made.

Printing through a mask. The polymorphic class lets us create different means of displaying the same data structure.When a word is displayed, its mask array is checked, and each letter in the word is displayed if its mask bit is on (true). For Alphabet objects, nothing is displayed if the bit is false, but for the puzzle (a HangWord), dashes are shown.

Searching and updating the masks. When the player guesses a letter, Board::guess() searches the alphabet (line 162) to find out whether the letter is legal and saves the index of the letter for later use. If the letter is not in the alphabet or if it has already been used, the function returns immediately with an error code (lines 163–164). Otherwise, it turns off the mask bit (line 165) corresponding to the letter to indicate that the letter has been used. (This removes the letter from the display.)

Then try_letter() is called (line 167) to compare the guessed letter to the letters in the puzzle word (lines 294–299). This function turns on the mask bit corresponding to each matching letter in Puzzle (line 297) and returns the number of matches (0 or more) that were found. If no matches were found, 0 is returned and the mask field in Error that corresponds to the bad guess is turned on (line 169). This causes the letter to appear on the error list. Finally, lines 173–174 add the number of matches to the score and return a success code. In Figure 14.1.5, Puzzle.Mask[0] and Puzzle.Mask[4] have been set to true in response to the two correct guesses, 'e' and 't'. The same two letters have been marked as false in Alpha.Mask[4] and Alpha.Mask[19].The board will display "`e _ _ _ t`" and show the alphabet with these two letters missing.

Coding techniques. Two code segments are worth mentioning here. First, note the Alphabet::find() function on lines 277–282. Line 280 is a complete sequential search written in one line, as a `for` loop with no body. It positions `k`, the index for both Alpha and Errors, on the letter that was guessed. The conditional operator in line 281 returns this position or -1, an error code. This is very compact code, but is well within the bounds of readability.

HangWord::print() also uses a one-line loop and a conditional operator to check the mask and print either a puzzle letter or a dash. In contrast, an if statement is used to test the mask in Alphabet::print() and HangWord::try_letter(). The difference is that the last two functions are using a one-sided conditional; they do something if the condition is true, nothing otherwise. The conditional operator can only be used in symmetric situations where some value is returned whether the condition is true or false.

14.4 RandString Adapts a Reusable Data Structure

The RandString class is an *adapter*. It is derived from a general-purpose container class template, FlexArray, and changes the interface provided by the reusable class to one that is appropriate for this application. One new public function, randword(), is added and one existing function is removed from the interface by an override. This is a typical pattern that is repeated over and over when you use class libraries. The library seldom provides exactly the classes you want, but if you can find something close to your needs, you can add to, modify, or restrict its interface to meet you needs. In this application, the interface must be changed to prevent ordinary sequential access and to provide random word selection.

The class RandString is a vocabulary list from which strings may be randomly selected and removed. The letters that form the words are stored in a StringStore. The StringStore class that was presented in Chapter 8 is reused here with a few corrections. StringStore will "take care of itself" and construct as many Pools as needed to hold the characters in the vocabulary words. To store the string pointers, we use a flexible array (vector) because we need both random access and flexibility. Random access is not available with linked lists, so we must use some sort of an array. We don't use a simple array because the number of words in the vocabulary file is not known at compile time.

14.4.1 The RandString Declaration

We create the RandString class by deriving it from an instantiation of the FlexArray template:

```
class RandString : public FlexArray<char*> {...}
```

We aggregate the StringStore in the resulting derived class, and complete the package by adding functions to do random selection and removal of the puzzle words. By this means, we achieve a flexible, sophisticated data structure with very little new code.

The RandString constructor. A few small things deserve notice here:

- Because this is a derived class, we need a ctor to construct the base class (line 338).

- By default, the RandString constructor creates an initial FlexArray that can hold 100 words. This constructor is called from the Game constructor without an integer argument, so the default size is actually used.
- The default FlexArray size is specified in the .hpp file (line 326) but not in the function definition (line 338). This is correct usage.
- There is one restriction on vocabulary words: they must be shorter than 80 characters, since that is the length of the input buffer in the RandStrings constructor (lines 340, 345).
- On line 347, StringStore::put() is called to store the letters; it returns a pointer to the first letter in the new word. This pointer is sent to FlexArray::put(), to be stored in the word array. Very concise, very efficient. Perhaps difficult to understand.
- This is a typical eof-controlled input loop. The break on line 346 will happen if either a read error or and end-of-file occurs. In a game program like this, it is good enough to end input if a read error occurs. They are rare and we can play the game even of some of the words in the file remain unread.

```
310  // ==========================================================================
311  // Declaration for a string array with random selection
312  // A. Fischer, May 13, 2001                                 file: rstrings.hpp
313
314  #include "flexT.hpp"
315  #include "sstore.hpp"
316  // ==========================================================================
317  class RandString : public FlexArray<char*> {
318    protected:
319      StringStore Store;                 // Storage behind string array.
320
321    private:
322      inline cstring remove( int r );
323      void print( ostream& outs ) const;  // For debugging, make this public.
324
325    public:
326      RandString( istream& vocin, int sz = 100 );
327      ~RandString(){}
328      const char* randword();
329      const char* operator[] ( int index );
330  };
```

The functions randword() and remove(). Random selection and removal of a string is implemented by the randword() function, as follows:

- The RandString constructor, primes the standard C random number generator by calling srand(time(NULL)). This uses the current time of day as an initial value for the randomizing computation, ensuring that different games will start with different puzzle words.
- Game::play() calls RandString::randword() before constructing a new playing board (line 105).
- To select a random word from the N unused words in the vocabulary, randword() generates a random number R in the range $0 \ldots N-1$, then calls remove() to remove the selected word from the vocabulary.
- Inside remove(), the string pointer in the *Rth* position is copied into a local temporary and later returned. Then it is replaced in the FlexArray by a copy of the last string pointer in the array. Finally, N, the number of words in the vocabulary is decreased by 1. No actual words change position; only pointers are copied. This is a standard algorithm for "shuffling" a deck of cards or randomizing the order of the objects in any array. The strategy is simple, fast, and theoretically sound. It does leave a meaningless copy of a string pointer at the end of the vocabulary array each time a word is used and the array is shortened. Thus, the number of items remaining in the vocabulary, not its original size, must be used to select the next random word.

```
331  // ==========================================================================
332  // Implementation for a string array with random selection
333  // A. Fischer, June 4, Nov 14, 2000                         file: rstrings.cpp
334
335  #include "tools.hpp"
```

```
#include "rstrings.hpp"
// ======================================================================
RandString::RandString( istream& vocin, int sz ) : FlexArray<char*>(sz) {
    //cerr << "\nConstructing RandString  ";
    char line[80];                      // input buffer
    srand( time( NULL ) );              // start up random number generator.

    for(;;) {
        vocin >> ws;
        vocin.getline( line, 80 );
        if (!vocin.good()) break;
        put( Store.put(line, vocin.gcount())); // Add to SStore & FlexArray.
    }
    //cerr << "\nRead " <<Many <<" Data from vocabulary file " << endl;
    if ( !vocin.eof() ) fatal( "Read error on vocabulary file" );
}
// ---------------------------------------------------------------------
void
RandString::print( ostream& outs ) const {
    outs << "The vocabulary: \n";
    for (int k=0; k<N; k++) outs << Data[k] << endl;
}

// ---------------------------------------------------------------------
const char*
RandString::randword() {
    if (N < 1) fatal( "Sorry, out of Data!");
    int r = rand();
    return remove( r % N );
}

//----------------------------------------------------------------------
inline cstring
RandString::remove( int r ) {
    cstring ret = Data[r];      // Grab the word that was selected.
    Data[r] = Data[--N];        // Replace by last word in array.
    return ret;                 // Return word and decrease word count.
}

// ---------------------------------------------------------------------
const char*             // Override; block access to function in base class.
RandString::operator[] ( int index )
{
    cerr <<"No random access to vocabulary list";
    return "";
}
```

Overriding FlexArray::operator[]. When using class derivation, all properties of the base class are inherited by the derived class. Sometimes this is undesirable or destructive in the new context. For example, the FlexArray class provides a general subscript operator. However, the RandString class has a very special access and removal rule, and random-access subscripting would defeat its purpose of restrincting access. This problem is handled by *overriding* the inherited definition by a new definition with exactly the same parameters as the inherited method. We cannot eliminate the subscript operator altogether, but we can write it as a trap. In this class, we use a nonfatal trap: it prints an error message but does not abort the run. Clearly, this will never happen in a fully debugged program. However, during construction and debugging, traps like this can be very useful.

Chapter 15: Polymorphism and Virtual Functions

From Lewis Carrol, *Through the Looking Glass*:

> "When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean – neither more nor less."
>
> "The question is," said Alice, "whether you can make words mean so many different things."
>
> "The question is," said Humpty Dumpty, "which is to be master– that's all."

15.1 Basic Concepts

15.1.1 Definitions

Simple derivation. The Hangman program uses the simple form of derivation: two application classes are derived from the same base class. This kind of derivation serves two important purposes:

- To factor out the common parts of related classes so that the common code does not need to be written twice. An example of this is the BaseWord class with its derivatives, Alphabet, and Hangword.
- To facilitate reuse of library classes and templates. The base class will contain a generally-useful data structure and its functions. The derived class will contain functions that are specific to the application. An example of this is the RandString class, which is derived from an instantiation of the FlexArray template.

Polymorphic derivation. In this chapter, we introduce virtual functions and two complex and powerful uses for derived classes that virtual functions support: abstraction and polymorphism.

- A *virtual function* is a function in a base class that forms part of the interface for a set of derived classes. It is declared *virtual* in the base class and may or may not have a definition in that class. It *will* have definitions in one or more of the derived classes. The purpose of a virtual function is to have one name, one prototype, and more than one definition so that the function's behavior can be appropriate for each of the derived classes.
- A *pure virtual function* is a function that has no definition in the base class.
- An *abstract class* is a class with one or more pure vitual functions.
- A *polymorphic* class is a base class that supports a declared set of public virtual functions, together with two or more derived classes that define methods for those functions. Data members and non-virtual functions may be defined both in the base class and the derived classes, as appropriate. We say that the derived classes *implement* the polymorphic *interface* class.

15.1.2 Virtual functions.

A virtual function is shared by the classes in a derivation hierarchy such as the Employee classes in Figure 15.1.2. (Note: In this sketch, three shapes have been drawn on top of the ordinary rectangular class boxes. This is not proper UML; the shapes will be used later to explain polymorphism.)

We create a virtual function when the same task must be done for all objects of all types in the hierarchy, but the method for doing this task depends on the particular representation of an object. For example, suppose the function calculatePay() must be defined for all employees, but the formula for the calculation is different for union members and professional staff. We want a function with one name that is implemented by three or more defining *methods.* For any given function call, we want the appropriate function to be called. For example, suppose we have declared four objects:

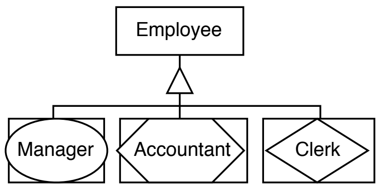


Figure 15.1: A polymorphic class hierarchy.

```
Manager M;
Accountant  A;
Clerk C, D;
```

Then if we call A.calculatePay() or M.calculatePay() we want the calculation to be made using the formula for professional staff. If we call C.calculatePay() or D.calculatePay(), we want the union formula used.

We implement this arrangement by declaring (and defining) calculatePay() as a virtual function in the Employee class, with or without a general method in that class. We also define calculatePay() in each of the derived classes, to make the appropriate calculations for the specific type of employee. The general method might do the parts of the calculation that are common to all classes. When calculatePay() is called, the system must select one of the methods to use. It does this by looking at the specific type of the object in the function call. This issue is explored more fully in the section on polymorphic classes.

Syntax. For examples, look at print() in Container, Linear and Queue, later in this chapter.

- The prototype for a virtual function starts with the keyword `virtual`. The rest of the prototype is just like any other function.
- The prototype for a pure virtual function ends in =0 instead of a semicolon. This means that it *has no definition* within the class.
- You must *not* write the keyword `virtual` before the definition of a virtual function that is outside the class.
- Any class with one or more virtual functions must have a virtual destructor.

How it works. Every object that could require dynamic dispatch must carry a type tag (one byte) at run time that identifies its relationship to the base class of the class hierarchy. (1st subclass, 2nd subclass, etc.) This is necessary if even one function is declared virtual, either in the class itself or in a parent class.

The run-time system will select the correct method to use for each call on a virtual function. To do so, it uses the type tag of the implied parameter to subscript the function's dispatch table. This table has one slot for each class derived from the class that contains the original virtual declaration. The value stored in slot k is the entry address for the method that should be used for the kth subclass derived from the base class. This is slightly slower than static binding because there is one extra memory reference for every call on every virtual function.

15.2 Polymorphic Classes

The purposes of polymorphism.

- To define an extensible set of representations for a class. One or two may be defined at first, more added later. It is a simple way to make an application extensible; you can build part of it now, part later, and be confident that the parts will work together smoothly.
- To allow a data structure to contain a mixture of items of different but related subtypes, such as the linked list of employees illustrated in Figure 15.2. The data objects in this list are the Employees defined in section 15.1.2; the shape of each object indicates which subtype of Employee it belongs to.
- To support run-time variability of types within a restricted set of related types, and the accompanying run-time dispatch (binding) of methods. The most specific appropriate method is dispatched. This lets us create different varieties of objects depending on input that is entered at run time.

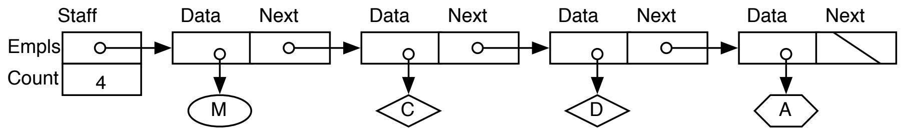


Figure 15.2: A list of polymorphic Employees.

Dispatching. Dispatching a function call is simple when the implied argument belongs to a derived class and a method for the virtual function is defined in that class. However, this is not always the case. For example, Figure 15.2 is a list of pointers to polymorphic Employees. Suppose a virtual print() function is defined in Employee and given a method there. Also, methods are defined in Accountant and Manager; these print the special data then call the Employee::print() to print the rest. However, no method is defined in Clerk because the general Employee method is appropriate. Now suppose we want to print the four Employees on the list and write this:

```
Employee* p;
for (p=Staff.Empls; p!=NULL; p=p->Next) p->Data->print(cout);
```

When we execute this loop, the Employee::print() function will be *called* four times. If it were an ordinary function, Employee::print() would be *executed* four times. However, that does not happen because Employee::print() is virtual, which means that the most appropriate function in the class hierarchy will be executed when Employee::print() is called:

1. The system will dispatch Manager::print() to print M because M is a Manager.
2. To print C and D, Employee::print() will be used because the Clerk class does not have a print function of its own.
3. To print the Accountant, A, the system will dispatch Accountant::print().

In all cases, the most specific applicable method is selected at run time and dispatched. It cannot be done at compile time because three different methods must be used for the four Employees.

Two limitations. Virtual functions cannot be expanded inline because the correct method to apply is not known until run time, when the actual type of the implied parameter can be checked. For this reason, using a large number of virtual functions can increase execution time.

If you have a function that you want a derived class to inherit from its parent, do not define another function with the same name in the derived class, even if its parameter list is different.

Two design guidelines. If you have a polymorphic base class:

1. . . . and you allocate memory in a derived class, you must make the base-class destructor virtual.
2. You can use the C++ dynamic_cast operator for safe run-time polymorphic casting.

15.3 Creating a Polymorphic Container

Overview of the demo program. The major example in this chapter is a polymorphic implementation of linear containers. The class Container is an abstract class from which linear containers (lists, queues) and non-linear containers (trees, hash tables) could be derived. Container supplies a minimal generic interface for container clases (those that can be used to store collections of data items).

In this chapter and the next, we focus on linear containers. The class Linear is the base class for a polymorphic family of list-based containers. From it we derive several data structures: Stack and Queue in this chapter, List and Priority Queue in the next. All of these conform to the Container interface and Linear implementation strategy but present different insertion and/or deletion rules. We use the containers in this chapter to store objects of class Exam, consisting of a 3-letter name and an integer.

The classes Linear and Cell should be defined as templates so that they do not depend on the type of the data in the list. Templates were not used here because they would complicate the structure of the program and the important issue here is the structure of a polymorphic class. In a real application, both would be used.

What to look for.

- A virtual function can be initially declared with or without a definition. For example, pop() is declared without a definition in Container, but insert() is declared with a definition in Linear.
- A virtual function can be redefined (directly or indirectly) in a derived class, as Queue::insert() redefines Linear::insert(). When redefinition is used, the derived-class method is often defined in terms of the inherited method, as Stack::print() is defined in terms of Linear::print().
- If a function is declared virtual, then it is virtual in all derived classes.
- When a virtual function is executed on a member of the base class, the method defined for that function in the appropriate derived class is dispatched, if it exists. Thus, when insert() is called from Linear::put() to insert a new Cell into a Queue, Queue::insert() is dispatched, not Linear::insert(). If Linear::insert() were not virtual, Linear::insert() would be executed.

15.3.1 Container: An Abstract Class

```
// ------------------------------------------------------------------------
// Abstract Containers
// A. Fischer   June 10, 2001                               file: contain.hpp
// ------------------------------------------------------------------------
#ifndef CONTAIN_H
#define CONTAIN_H
#include "exam.hpp"
#include "cell.hpp"

class Container {
  public:          // ----------------------------------------------------
    virtual void     put(Item*)      =0; // Put Item into the Container.
    virtual Item*    pop()           =0; // Remove next Item from Container.
    virtual Item*    peek()          =0; // Look but don't remove next Item.
    virtual ostream& print(ostream&) =0; // Print all Items in Container.
};
#endif
```

A container is a place to store and retrieve data items of any sort which have been attached to cells. Each container has its own discipline for organizing the data that is stored in it. In this section we develop a generic linear container class from which special containers such as stacks or queues can be derived. We use the program to illustrate the concepts, syntax, and interactions among a polymorphic base class and its implementation classes.

The Container class presented here supplies implementation-independent functions that are appropriate for any container and any contents type. Every container must allow a client program to put data items into the container and get them back out. A print() function is often useful and is necessary for debugging.

15.3.2 Linear: A Polymorphic Class

A linear container is one that has a beginning and an end, and the data inside it is arranged in a one-dimensional manner. It might be sorted or unsorted. The class Linear defines a simple, unsorted, linear container that is implemented by a linked list, using a helper class named Cell. It defines a set of list-handling function that can be written in a generic way so that they will apply to most or all linked-list linear containers. All of these functions are protected except the public interface functions that were inherited from Container. The other functions are not public because they allow a caller to decide how and where to put things into or take things out of the list. This kind of control is necessary for defining Stack and Queue but not safe for public use.

Since Linear is derived from Container, it inherits all of the function prototypes of Container. Since these functions are equally appropriate for use with an array or a linked list, Linear could be implemented either way. We commit to a linked list implementation when we define the Linear constructor in line 37. The code for reset() (line 39), end() (line 40), and all the functions in the .cpp file also rely on a linked list representation for the data.

```
// --------------------------------------------------------------------------
// Linear Containers
// A. Fischer June 12, 2001                                  file: linear.hpp
// --------------------------------------------------------------------------
#ifndef LINEAR_H
#define LINEAR_H
#include "contain.hpp"
#include "cell.hpp"
#include "tools.hpp"

class Linear: public Container {
  protected: // -------------------------------------------------------------
    Cell* head;      // This is a dummy header for the list.

  private:   // -------------------------------------------------------------
    Cell* here;      // Cursor for traversing the container.
    Cell* prior;     // Trailing pointer for traversing the container.

  protected: // -------------------------------------------------------------
            Linear(): head(new Cell), here( NULL ), prior( head ) {}
    virtual ~Linear ();
            void  reset()           { prior = head; here = head->next; }
            bool  end() const       { return here == NULL; }
            void  operator ++();

    virtual void  insert( Cell* cp );
    virtual void  focus() = 0;
            Cell* remove();
            void  setPrior(Cell* cp){ prior = cp; here = prior->next; }

  public:   // --------------------------------------------------------------
            void     put(Item * ep) { if (ep) insert( new Cell(ep) ); }
            Item*    pop();
            Item*    peek()         { focus(); return *here; }
    virtual ostream& print( ostream& out );
};
inline ostream& operator<<(ostream& out, Linear& s) {return s.print(out); }
#endif
```

The two private data members in this class, together with the functions reset(), end(), and `++` permit us to start at the beginning of the container and visit each member sequentially until we get to the end. The reset() function sets the pointer named `here` to the first item in the container and sets `prior` to NULL. The `++` operator sets `prior = here` and moves `here` to the next item. At all times, these two pointers will point to adjacent items in the container (or to NULL). The end() function returns true if `here` has passed the last item in the container.

Three of the functions inherited from Container are no longer virtual, so the run-time dispatcher will not look in the Stack or Queue class for an overriding definition when put(), pop() or peek() is called from within Linear. One would expect put() to be virtual, since stacks and queues need different methods for putting an item into the container. However, put() simply wraps the item in a Cell, then delegates the actual insertion operation to insert(), which `is` virtual so that the specific and appropriate version of insert() in the derived class (Stack, Queue, etc.) will always be dispatched. We say that the Linear class `collaborates` with Stack or with Queue to handle the insertion task.

In a similar way, remove(), which is not virtual collaborates with focus(), which *is* virtual to focus the deletion on the proper element of the list. Since this is a pure virtual function, Linear is an abstract class that cannot be instantiated. The print() function is virtual for the same reason: to pass the responsibility to a derived class that is the experts on how printing should be done. The destructor is virtual because C++ requires a virtual destructor in classes that have virtual functions.

The Container class refers to Cells but does not define Cell. From that class, all we know is that Cell is a helper class that must be used when information is placed into the Container. At that stage, a Cell could be the traditional linked-list cell or it could be just a typedef-synonym for Item*, appropriate for use with an array-based container. In the Linear class, we must commit to one representation or another, and we do commit to using a linked list. The definition given here for Cell is the usual two-part structure containing an Exam* and a Cell*. Nothing in Cell is virtual and everything is inline.

```
// -----------------------------------------------------------------------
// Linear Containers
// A. Fischer June 12, 2001                                file: linear.cpp
// -----------------------------------------------------------------------
#include "linear.hpp"
// -----------------------------------------------------------------------
Linear::~Linear () {
    for (reset(); !end(); ++*this) {
        delete prior->data;
        delete prior;
    }
    delete prior->data;
    delete prior;
}
// ----------------------------- Move index to next Item in the container.
void
Linear::operator ++() {
    if (!end() ){
        prior = here;
        here = here->next;
    }
}
// --------------- Put an Item into the container between prior and here.
// ------------ Assumes that here and prior have been positioned already.
void
Linear::insert(Cell* cp) {
    cp->next = here;
    here = prior->next = cp;
}
// --------------- Take an Item out of the container. Like pop or dequeue.
// -- Assumes that here and prior have been positioned to the desired Item.
Cell*
Linear::remove() {
    if (! here ) return NULL;
    Cell* temp = here;                  // Grab cell to remove.
    here = prior->next = temp->next;    // Link around it.
    return temp;
}
// ----------------------------------- Remove a Cell and return its Item.
Item*
Linear::pop(){
    focus();                 // Set here and prior for deletion point.
    Cell* temp = remove();   // Remove first real cell on list.
    if (!temp) return NULL;  // Check for empty condition.
    Item* answer = *temp;    // Using cast coercion from Cell to Item.
    delete temp;             // Take contents out of cell, delete Cell.
    return answer;           // Return data Item.
}
// --------------------------------------- Print the container's contents.
ostream&
Linear::print (ostream& out ) {
    out << "  <[\n";
    for (reset(); !end(); ++*this)  out << "\t" <<*here;
    return out << "  ]>\n";
};
```

As in previous linked list definitions, this Cell class gives friendship to Linear. One slight difference is the friendship declaration on line 122, which is needed for the operator extension on line 139. We need to choose one of three alternatives:

a. Using the friend function declaration,

b. Making the print function public, and

c. Not having an operator extension for class Cell.

In previous versions of this class, we chose strategy (c); this time we choose (a) because it makes the Linear::print function shorter.

15.3.3 Cell: The Helper Class

```
111  // -----------------------------------------------------------------------
112  // A cell contains an Item* and a link.  Cells are used to build lists.
113  // A. Fischer   June 13, 2000                               file: cell.hpp
114  // -----------------------------------------------------------------------
115  #ifndef CELL_H
116  #define CELL_H
117  #include "item.hpp"
118  #include <iostream.h>
119  // -----------------------------------------------------------------------
120  class Cell {
121    friend class Linear;
122    friend ostream& operator<<( ostream& out, Cell& c);
123
124    private:  // -----------------------------------------------------------
125      Item* data;
126      Cell* next;
127
128      Cell(Item* e = NULL, Cell* p = NULL ): data(e), next(p){ }
129      ~Cell(){ cerr <<"\n  Deleting Cell 0x" <<this << dec <<"..."; }
130      operator Item*() { return data; }   // Cast Cell to Item*. ------------
131
132      void print(ostream& out) const {    // -------------------------------
133          if (data) {
134              out << "Cell 0x" << this;
135              out << " [" << *data << ", " << next << "]\n";
136          }
137      }
138  };
139  inline ostream& operator<<(ostream& out, Cell& c){c.print(out); return out;}
140  #endif
```

One new technique is introduced in Cell: we define a cast (line 130) from type Cell to type Exam*. This cast can be used explicitly, just like a built-in cast function or it can be used by the compiler to *coerce* (automatically convert) the type of an argument. For example, Line 100, in Linear::pop(), could be written two ways, as shown below.

```
Item* answer = (Item*)(*temp);    // Explicit use of a cast from Cell to Item*.
Item* answer = *temp;              // Using cast coercion from Cell to Item*.
```

The first uses cast operation explicitly. The second supplies `*temp` in a context where an Item* is required. The compiler will find the cast operator on line 130 and use it to coerce the argument on the right to the type of the variable on the left.

15.3.4 Exam: The Actual Data Class

The linear containers defined above would be appropriate to store any kind of data. The data class used here is very simple but could be much more complex. It is used again in the next chapter where it is extended by adding comparison functions and a key() function so that the data can be sorted.

```
//==========================================================================
//  Exam: A student's initials and one exam score.
//  A. Fischer, October 1, 2000                                file: exam.hpp
//==========================================================================
#ifndef EXAM_H
#define EXAM_H
#include <string.h>
#include <iostream.h>
typedef int KeyType;
//--------------------------------------------------------------------------
class Exam                      // One name-score pair
{
  private:  //--------------------------------------------------------------
    char Initials [4];          // Array of char for student name
  protected://--------------------------------------------------------------
    int Score;                  // Integer to hold score
  public:  //---------------------------------------------------------------
    Exam (const char* init, int sc){
        strncpy( Initials, init, 3 );
        Initials[3] = '\0';
        Score = sc;
    }
    ~Exam (){ cerr << "    Deleting Score " <<Initials <<"..."; }
    ostream& Print ( ostream& os ){
        return os <<Initials <<": " <<Score <<"  ";
    }
};
//--------------------------------------------------------------------------
inline ostream& operator << (ostream& out, Exam& T){ return T.Print( out ); }
#endif
```

```
//==========================================================================
//  Bind abstract name ITEM to the name of a real class.
//  A. Fischer, June 15, 2000                                  file: item.hpp
//==========================================================================
#ifndef ITEM_H
#define ITEM_H
typedef Exam Item;
#endif
```

15.3.5 Class diagram.

Figure 15.3.5 is a UML diagram for this program, showing the polymorphic nature of Linear and its relation to Cell, Stack, and Queue.

15.4 Stack: Fully Specific

A Stack is a container that implements a LIFO discipline. In this program, we derive Stack from Linear and, in so doing, we represent a stack by a linear linked list of Cells. From Linear, Stack inherits a head pointer, two scanning pointers (here and prior) and a long list of functions.

Notes on the Stack code.

- The list head pointer is protected, not private in Linear, because some derived classes (for example Queue), need to refer to it. The scanning pointers are private because the derived classes are supposed to use Linear::setPrior() rather than setting the pointers directly. This guarantees that the two scanning pointers are always in the correct relationship to each other, so that insertions and deletions will work properly. It also saves lines of code in the long run, since they are written once in Linear instead of multiple times in the derived classes.

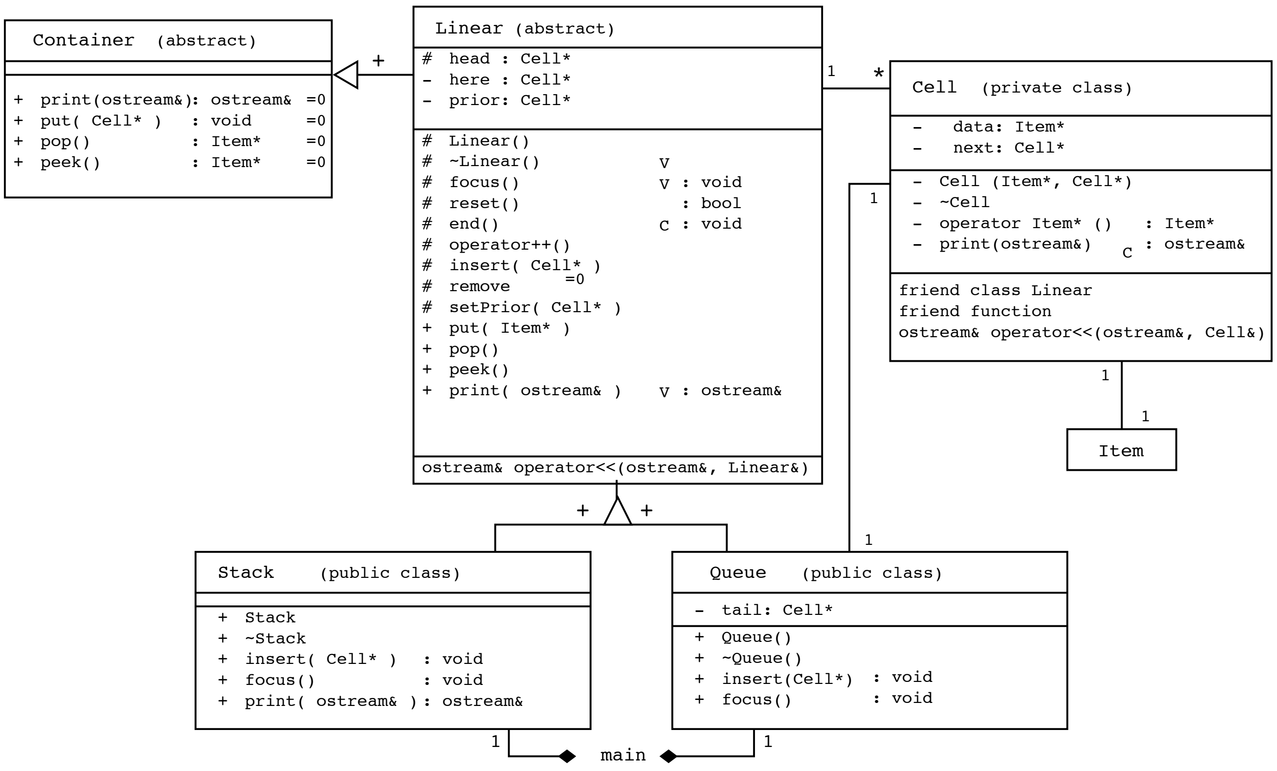


Figure 15.3: UML diagram for the Polymorphic Linear Container

- The Stack constructor and destructor are both null functions; they are supplied because it is bad style to rely on the defaults. The program does work perfectly well without either. Five functions are defined explicitly in Stack.

```
// ---------------------------------------------------------------------------
// Stacks, with an inheritance hierarchy
// A. Fischer  June 8, 2001                                      file: stack.hpp
// ---------------------------------------------------------------------------
#ifndef STACK_H
#define STACK_H
#include "linear.hpp"

// ---------------------------------------------------------------------------
class Stack : public Linear {
  public:
    Stack(){}
    ~Stack(){}
    void  insert( Cell* cp ) { reset(); Linear::insert(cp); }
    void  focus(){ reset(); }

    ostream& print( ostream& out ){
        out << "  The stack contains:\n";
        return Linear::print( out );
    }
};
#endif
```

- Stack defines focus(), which is abstract in Linear. This is called by Linear::remove() to set prior and here so that the first Cell in the container will be returned. (Last in, first out.)
- Since focus(), was the only remaining pure virtual function in Linear. Since we define it here, Stack becomes a fully specified class and can be used to create objects (that is, it can be instantiated.)

- Linear::put(Exam*) calls Insert, which is defined in both Linear and Stack. Because insert() is virtual, the version in Stack will be used to execute the call written in Linear::put(). We want to insert the new Cell at the head of the list because this is a stack. The actual insertion will be done by Linear::insert(), after Stack::insert() positions here and prior to the head of the list by calling reset(). The left side of figure 15.4 illustrates how control passes back and forth between the base class and the derived class during this process.

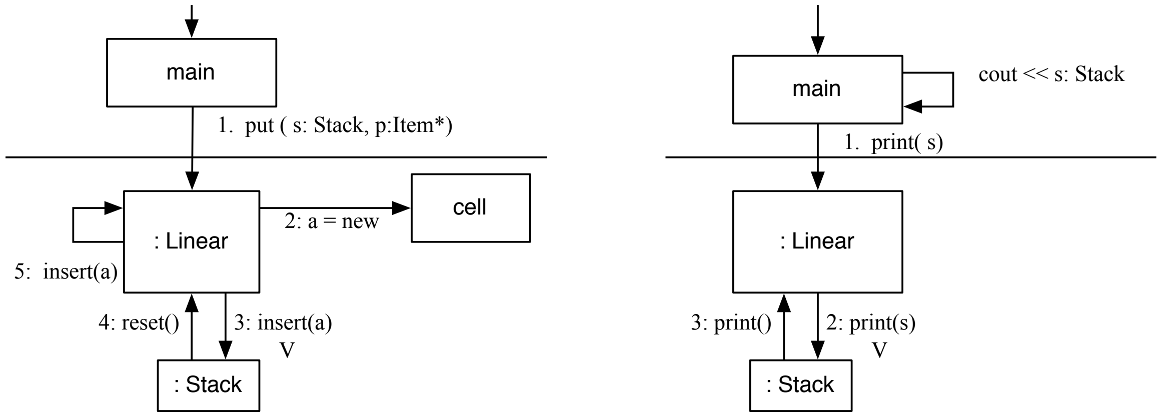


Figure 15.4: How the classes collaborate during Stack::put (left) and Stack::print (right).

- The redefinition of print() is not absolutely necessary here; all it does is to print a few words ("The stack contains:") and call Linear::print() to print the contents. For contrast, the Queue class does not define a print function. However, the collaboration between the two print() functions and the `<<` operator is instructive.

  In the main program, we never call the print functions directly. All printing is done by statements like `cerr << S;` But Stack and Queue do not supply definitions for `operator<<`, so the output task is given (by inheritance) to the definition in Linear (line 54). It, in turn, calls Linear::print(), which is virtual, and dispatches the job to Stack::print() because S is a stack. Stack::print() prints the output label and calls Linear::print() to finish the job. The class name is necessary only in the last call; all other shifts of responsibility are handled by inheritance or virtual dispatching. This activity is diagrammed on the right in Figure 15.4

15.5 Queue: Fully Specific

```
// ----------------------------------------------------------------------------
// Queues: derived from  Container-<--Linear-<--Queue
// A. Fischer   June 9, 2001                                     file: queue.hpp
// ----------------------------------------------------------------------------
#ifndef QUEUE_H
#define QUEUE_H
#include "linear.hpp"

// ----------------------------------------------------------------------------
class Queue : public Linear {
  private:
    Cell*   tail;

  public:           // ---------------------------------------------------
    Queue() { tail = head; }
    ~Queue(){}

    void  insert( Cell* cp ) { setPrior(tail); Linear::insert(cp); tail=cp;}
    void  focus(){ reset(); }
};
#endif
```

A queue is a container that implements a FIFO discipline. This Queue is a linear linked list of Cells with a dummy header. This declaration of Queue follows much the same pattern as Stack. However, Queue uses the inherited Linear::print(), instead of defining a specific version of its own. This is done for demonstration purposes; we recognize that better and more informative formatting could be achieved by defining a specific local version of print().

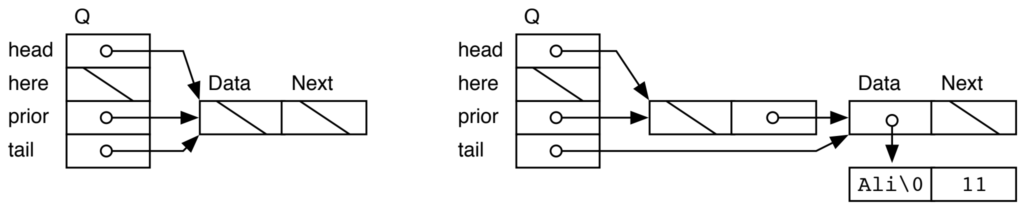


Figure 15.5: The Queue named Q, newly constructed and after one insertion.

Notes on the Queue code.

- Linear has three data members (head, prior and here), and Queue has one (tail). The constructor of each class must initialize the data members in its own class. When a Queue is constructed, all four are initialized to create an empty linked list with a dummy header, as in Figure 15.5. An important issue here is that the constructor of the base class, Linear, is executed first. It creates a dummy cell and attaches it to the head and prior pointers. When control gets to the Queue constructor, the dummy cell will exist and it is easy to attach the tail pointer to it.
- Queue::insert(Item*) set here and prior to the tail of the list so that insertions will be made after the last list cell. As before, Linear::insert() then finishes the job and does the actual insertion. Almost no code is duplicated because the classes collaborate.
- Queue defines focus(), which is abstract in Linear, to set the scanning pointers to the beginning of the Queue. The result is that the earliest insertion will be the next removal (FIFO).
- When an inherited virtual function (such as pop) is called, control passes back and forth between the base class and the derived class, as shown in Figure 15.4.

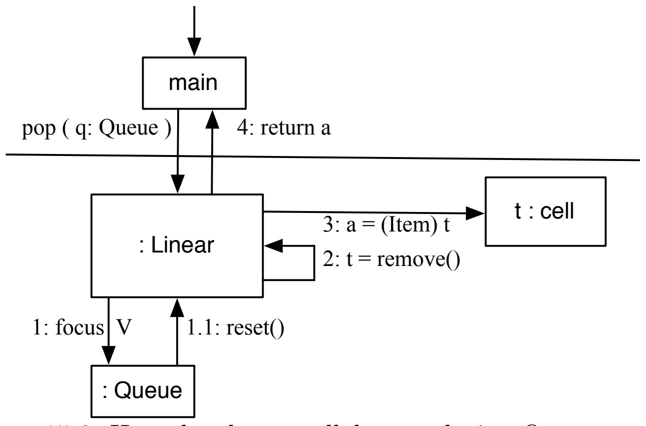


Figure 15.6: How the classes collaborate during Queue::pop().

- The output (following main) shows that the queue Q was printed by the inherited function, Linear::print().
- If Linear::insert() *is not* virtual, insertion is done incorrectly for queues. I removed the "virtual" property and got the following output :

```
Putting 3 items on the Queue Q: 11, 22, 33.
  ]>
  <[    Cell 0x0x804c498 [Cil: 33  , 0x804c478]
        Cell 0x0x804c478 [Bea: 22  , 0x804c458]
        Cell 0x0x804c458 [Ali: 11  , (nil)]
  ]>
```

Compare it to the correct output at the end of this chapter, where Ali comes first in the queue.

15.6 A Main Program and its Output

```
// -----------------------------------------------------------------------
// Demonstration of derived classes with virtual functions.
// A. Fischer  June 15, 1998                                 file: main.cpp
// -----------------------------------------------------------------------
#include "tools.hpp"
#include "exam.hpp"     // Must precede #include for item.hpp.
#include "item.hpp"     // Abstract base type for stacks and queues.
#include "stack.hpp"    // Base type is Item == Exam.
#include "queue.hpp"    // Base type is Item == Exam.
// -----------------------------------------------------------------------
int main( void ) {
    Stack S;
    Queue Q;

    cerr << "\nPutting 3 items on the Stack S: 99, 88, 77.\n" ;
    S.put( new Exam("Ned", 99) );       //cerr << S << endl;
    S.put( new Exam("Max", 88) );       //cerr << S << endl;
    cerr << "  Peeking after second insertion: " <<*S.peek() <<"\n";
    S.put( new Exam("Leo",77) );          cerr << S << endl;

    cerr << "Putting 3 items on the Queue Q: 11, 22, 33.\n";
    Q.put( new Exam("Ali",11) );        //cerr << Q << endl;
    Q.put( new Exam("Bea",22) );        //cerr << Q << endl;
    cerr << "  Peeking after second insertion: " <<*Q.peek() <<"\n";
    Q.put( new Exam("Cil",33) );          cerr << Q << endl;

    cerr << "Pop two Exams from Q, put on S. \n";
    S.put(Q.pop()); S.put(Q.pop());       cerr <<"\n" <<S << endl;

    cerr << "Put another Exam onto Q: 44.\n";
    Q.put( new Exam("Dan",44) );          cerr << Q << endl;

    cerr << "Pop two Exams from S and discard.\n";
    delete S.pop();
    delete S.pop();                       cerr <<"\n" << S << endl;
    bye();
}
```

To test the linear container classes, we wrote a meaningless main program that instantiates one Stack and one Queue and moves data onto both and from one to the other. The call graph in Figure 15.6 attempts to show the ways that functions are actually called, after dynamic dispatching. In this chart, grey circles that say "V" mark virtual dispatching and circles that say "I" show calls that were made through inheritance.

Enough function calls are made to demonstrate that the classes work properly; the contents of S and Q are printed just often enough to see the data move into and out of each container. Note that several diagnostic output commands were used during debugging and are now commented out.

- The Queue implements a first-in first-out order.
- The Stack implements a last-in first-out order.
- peek() returns the same thing that pop() would return, but does not remove it from the list.
- Stack::print() is used to print the stack but the inherited Linear::print() prints the queue.
- All dynamically allocated objects are properly deleted.

The output:

```
Putting 3 items on the Stack S: 99, 88, 77.
  Peeking after second insertion: Max: 88
  The stack contains:
  <[    Cell 0x0x804c988 [Leo: 77  , 0x804c968]
        Cell 0x0x804c968 [Max: 88  , 0x804c948]
        Cell 0x0x804c948 [Ned: 99  , (nil)]
  ]>
```

```
Putting 3 items on the Queue Q: 11, 22, 33.
  Peeking after second insertion: Ali: 11
  <[    Cell 0x0x804c9a8 [Ali: 11  , 0x804c9c8]
        Cell 0x0x804c9c8 [Bea: 22  , 0x804c9e8]
        Cell 0x0x804c9e8 [Cil: 33  , (nil)]
  ]>

Pop two Exams from Q, put on S.

  Deleting Cell 0x0x804c9a8...
  Deleting Cell 0x0x804c9c8...
  The stack contains:
  <[    Cell 0x0x804c9c8 [Bea: 22  , 0x804c9a8]
        Cell 0x0x804c9a8 [Ali: 11  , 0x804c988]
        Cell 0x0x804c988 [Leo: 77  , 0x804c968]
        Cell 0x0x804c968 [Max: 88  , 0x804c948]
        Cell 0x0x804c948 [Ned: 99  , (nil)]
  ]>

Put another Exam onto Q: 44.
  <[    Cell 0x0x804c9e8 [Cil: 33  , 0x804ca08]
        Cell 0x0x804ca08 [Dan: 44  , (nil)]
  ]>

Pop two Exams from S and discard.

  Deleting Cell 0x0x804c9c8...    Deleting Score Bea...
  Deleting Cell 0x0x804c9a8...    Deleting Score Ali...
  The stack contains:
  <[    Cell 0x0x804c988 [Leo: 77  , 0x804c968]
        Cell 0x0x804c968 [Max: 88  , 0x804c948]
        Cell 0x0x804c948 [Ned: 99  , (nil)]
  ]>
```

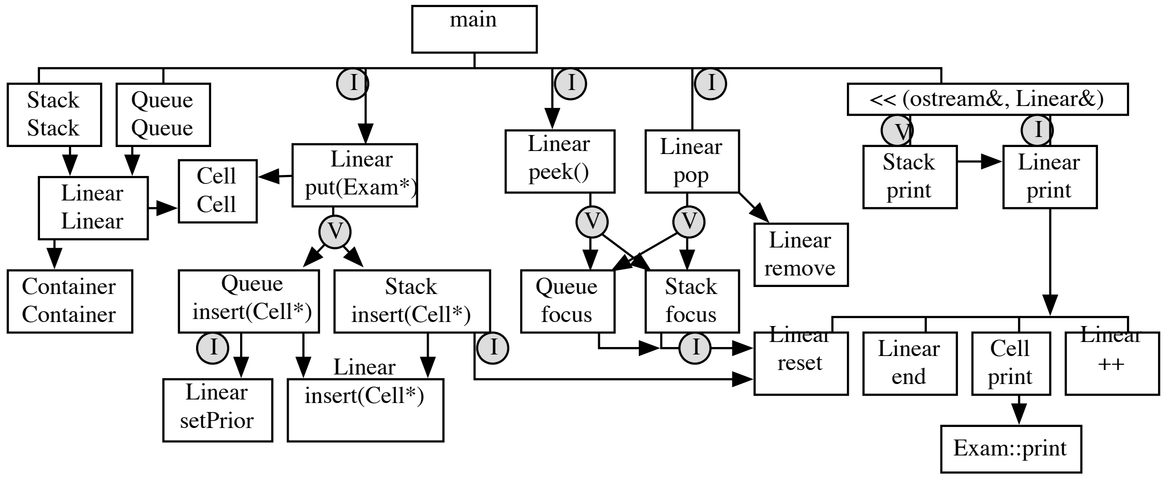


Figure 15.7: A call graph for the linear container program.

Termination. After printing the termination message, the objects Q and S will go out of scope and be deallocated. An output trace can serve as part of a proof that deallocation is done correctly and fully, without crashing. The output trace from main is given below, with a call graph (Figure 15.6) showing how control moves through the destructors of the various classes.

```
Normal termination.

  Deleting Cell 0x0x804c928...    Deleting Score Cil...
  Deleting Cell 0x0x804c9e8...    Deleting Score Dan...
  Deleting Cell 0x0x804ca08...
  Deleting Cell 0x0x804c918...    Deleting Score Leo...
  Deleting Cell 0x0x804c988...    Deleting Score Max...
  Deleting Cell 0x0x804c968...    Deleting Score Ned...
  Deleting Cell 0x0x804c948...
```

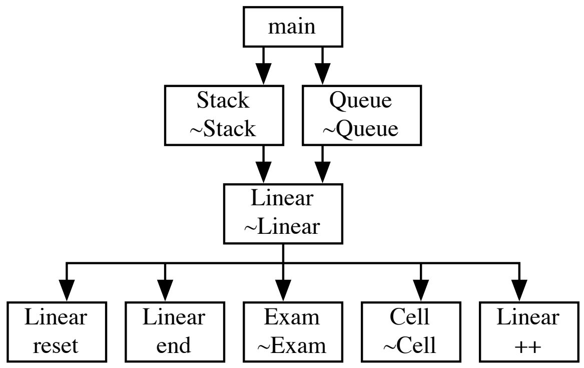


Figure 15.8: Terminating the linear container program.

Chapter 16: Abstract Classes and Multiple Inheritance

From the Holy Bible, New King James Version, Proverbs 22:6.

Train up a child in the way he should go, and when he is old he will not depart from it.

An abstract class declares a set of behaviors (function prototypes) that all of its descendents must follow (or implement). Documentation accompanying the abstract class must explain the purpose of each function and how each interacts with other parts of the class. Derived classes must implement all of the functions, and should obey the guidelines explained in the documentation.

16.1 An Abstract Class Defines Expectations

Abstract classes. When a base class includes even one prototype for a pure virtual function, it is an *abstract class* which cannot be used to create objects. However, such classes do have a purpose and they are important in the process of developing a large system. An abstract class lets us define and enforce a common interface, or behavior, for a set of related classes. Class derivation, combined with methods defined in the derived class(es) make the abstraction useful.

An abstract class specifies a set of virtual representation-dependent function prototypes for which definitions will be required in future derived classes. By doing so, it enables a large system to be developed in a top-down style.

The first development step is to define the major modules. The interface that each module provides is then written The compiler will ensure that no necessary part of A is forgotten, and that all functions conform to the prototypes that were promised.

Large systems developed by teams of programmers are built this way. First, each major system component is identified. As soon as its role in the system is clear, its public interface can be defined in the form of an abstract class declaration. The abstract class forms a *contract* for the programming team that will develop the component, and different people (or teams) will work on different subsystems simultaneously.

Suppose a system designer has specified modules named A, B, and C. Since all the prototypes provided by module A are defined, programmers working on modules B and C can begin to write code that calls the functions of A, even before A is fully implemented. Derivation is used to connect the abstract interfaces to their implementation modules. When modules are complete, the subsystems can be easily integrated because the established prototypes guarantee that functions in one module will be able to call functions in another module that were programmed by a different person.

Definitions and rules.

- The opposite of abstract is *concrete.* A concrete class can have virtual functions, but all of those functions must have methods defined within the class itself or its ancestor classes.
- An abstract class cannot be `instantiated`, that is, used to construct any objects.
- Any class that has one or more pure virtual functions is called an *abstract class.*
- Abstract classes is polymorphic if more than one concrete class is derived from it.

16.2 Abstraction Example: an Ordered Type

Two abstract classes are used in this chapter's program: `Container` (from the previous chapter) and `Ordered`.

- `Container` defines the interface that should be presented by any container class: a way to put data into the container (put()), and find it when needed (get()), take it out of the container (remove()), write the contents of the container to a stream (print()).

- Ordered defines prototypes for functions that are needed when you sort data items: comparison functions, sentinels, and a way to access the key field of the data.

```
// -----------------------------------------------------------------------
// Ordered base class -- An abstract class
// A. Fischer  June 8, 1998                                file: ordered.hpp
//
#ifndef ORDERED_H
#define ORDERED_H
#include <limits.h>
#include <iostream.h>
// -----------------------------------------------------------------------
class Ordered {
  public:
    virtual ~Ordered(){}
    virtual KeyType key() const                       =0;
    virtual bool    operator <  (const KeyType&) const  =0;
    virtual bool    operator == (const KeyType&) const  =0;
};
#endif
```

Specifying and enforcing requirements. The purpose of a container is to store a collection of items. The data stored in an item is not important; we use the class name Item as a representative of any kind of object that a containers might store. However, a few properties of an Item are essential for use with a sorted container:

- The Item must contains a key field and a key() function that returns the key.
- The operators `<` and `==` must be defined to compare two Items. Items will be compared using one or both of these operators. They will be stored in the container in ascending order, as defined by the operator `<`.
- A programmer who creates an Ordered class must supply the appropriate typedef for KeyType and appropriate definitions for the operators and sentinels that define the minimum and maximum possible values for a KeyType object.

The first two properties can be specified by defining an abstract class, Ordered, that gives prototypes (but no definitions) for the three required functions. We can enforce these requirements in a data class by deriving the data class from Ordered. When we do this, we instruct the compiler to guarantee that the derived data class does implement every function listed by Ordered. If one of the functions is missing, the compiler will give an error comment.

16.3 Multiple Inheritance

A class may be derived from more than one parent class. (We must `#include` the header files for each parent class.) The purpose of such *multiple inheritance* is:

- Simple form: to inherit properties from one parent and constraints from another.
- General form: inherit properties from two parent classes

The syntax is a simple extension of ordinary derivation. We use it here (line 30) to combine the properties of the Exam class from the previous chapter with the abstract Ordered class. The new class is a *wrapper* for Exam that provides more functions than the original class but does not duplicate the ones that Exam supplies (`Print()` and `operator<<`).

16.3.1 Item: The Data Class

In the previous chapter, we used a `typedef` to make `Item` a synonym for the `Exam` class. In this chapter, we do more with `Item`: we use multiple inheritance to add constraints and functionality to the original `Exam` class.

```
18  //------------------------------------------------------------------------
19  // Class declaration for data items.
20  // A. Fischer, May 29, 2001                                  file: item.hpp
21  //
22  #ifndef ITEM_H
23  #define ITEM_H
24  #include <limits.h>
25
26  typedef int KeyType;
27  #include "exam.hpp"
28  #include "ordered.hpp"
29  //------------------------------------------------------------------------
30  class Item : public Exam, public Ordered {
31    public:
32      static const KeyType max_sentinel = KeyType(INT_MAX);
33      static const KeyType min_sentinel = KeyType(INT_MIN);
34
35      Item(char* init, int sc): Exam(init, sc){}
36      ~Item() { cerr <<"Deleting Item " << key() <<"\n"; }
37
38      KeyType key() const                    { return Score; }
39      bool operator==(const KeyType& k) const { return key() == k; }
40      bool operator< (const KeyType& k) const { return key() < k; }
41      bool operator< (const Item& s)    const { return key() < s.key(); }
42  };
43  #endif
```

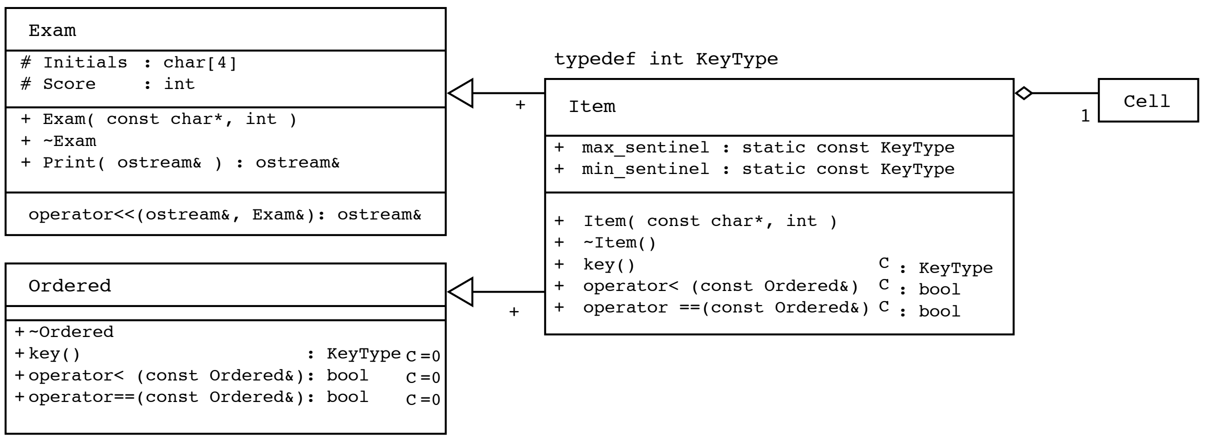

Exam
# Initials : char[4]
# Score : int
+ Exam(const char*, int)
+ ~Exam
+ Print(ostream&) : ostream&
operator<<(ostream&, Exam&): ostream&
typedef int KeyType
+
Item
+ max_sentinel : static const KeyType
+ min_sentinel : static const KeyType
+ Item(const char*, int)
+ ~Item()
+ key() C : KeyType
+ operator< (const Ordered&) C : bool
+ operator ==(const Ordered&) C : bool
1
Cell
Ordered
+
+ ~Ordered
+ key() : KeyType C =0
+ operator< (const Ordered&): bool C =0
+ operator==(const Ordered&): bool C =0

Figure 16.1: Inheriting both functionality and constraints.

Notes on the Item class. Two comparison operators and the key() function are required for Item because it was derived from Ordered. We define these three functions (lines 38–40) in such a way that Score (inherited from Exam) is the key field and the exams will be sorted in ascending order by Score. A third comparison function (line 41) is added for the convenience of client classes.

The Item class also defines two constants that are often needed for sorting algorithms: the maximum and minimum values of type KeyType. Line 24 tells us that KeyType is a synonym for int; the int values used here are supplied by the file `<limits.h>` in the standard library.

The Item constructor does nothing but pass its arguments through to the Exam constructor because it has no variables of its own that need initialization. The virtual destructor in the Ordered class is necessary to avoid warning comments in the Item class. For example, without that seemingly useless function, we get a warning comment about line 42:

```
item.hpp:42: warning:
    ‘class Item’ has virtual functions but non-virtual destructor
```

With these definitions, Item fulfills all the inherited obligations, the class is concrete and can be used normally. This class will be used with Linear and Cell from the prior chapter to build two new container classes: List and Priority Queue.

16.4 Linear Containers You Can Search

In this chapter we develop two new container classes from Linear. In a stack or a queue, all insertions and deletions are at one of the ends of the container; we never need to locate a spot in the middle. In contrast, a priority queue requires all insertions to be made in sorted order, and a simple list requires a search whenever an item is removed. To develop these classes in a general way, we assume that each Cell will contain an Item that is derived from Ordered, and we use the functions promised by Ordered to define three new functions in the Linear class:

```
bool Linear:: operator < ( Cell* cp ) { return (*cp->data < *here->data); }
bool Linear:: operator < ( KeyType k ){ return *here->data < k; }
bool Linear:: operator== ( KeyType k ){ return *here->data == k; }
```

These functions allow us to search or sort a linear container according to the key field of the Item.

16.4.1 PQueue: a Sorted Linear Container

Notes on the PQueue code Items are deleted from a priority queue at the head of the list, just like an ordinary queue. Preparation for a deletion is easy because Linear provides the reset() function to position its pointers at the head of the list.

In a priority queue, items are inserted in priority order in the list and removed from the head of the list. To do the insertion, we must scan the list to locate the correct insertion spot: the item at `prior` should have higher priority (a higher key number than the new item) and the item at `here` the same or lower priority. The loop on lines 60–62 performs such a scan using the `<` operator and list traversal functions (reset(), end(), and `++`) that are provided by Linear. Each reference to `*this` calls an operator defined by Linear and inherited by PQueue. When the right place is found, control is returned to Linear to do the actual insertion.

The only other functions here are a null constructor and a null destructor. Neither is necessary because the compiler will supply them by default. (Compare this class to List, below, in which the constructor and destructor have been omitted.) We write explicit functions because it is good style.

```
44  // ---------------------------------------------------------------------
45  // Priority queues: derived from  Container-<--Linear-<--PQueue
46  // A. Fischer   June 9, 2001                              file: pqueue.hpp
47  // ---------------------------------------------------------------------
48  #ifndef PQUEUE_H
49  #define PQUEUE_H
50  #include "linear.hpp"
51
52  class PQueue : public Linear {
53    public:         // -----------------------------------------------------
54      PQueue(){}
55      ~PQueue(){}
56      void focus(){ reset(); }    // Priority queue deletion is at the head.
57
58      // ----------------------- Insert new Cell in ascending sorted order.
59      void PQueue::insert( Cell* cp ) {
60          for (reset(); !end(); ++*this) {    // locate insertion spot.
61              if ( !(*this < cp) )break;
62          }
63          Linear::insert( cp );               // do the insertion.
64      }
65  };
66  #endif
```

16.4.2 List: An Unordered Container

Notes on the List code The List class provides a container with no special rules for insertion, deletion, or internal order. Since the order of items in the list does not matter, we use the easiest possible insertion method: insertion at the head, as in Stack. However, removing an item creates two new problems: how can we specify which item to remove, and how can we find it? The removal function required by Container does not have a parameter, but to remove an item from a List, we must know the key of the desired item. We solve the problem here by making the required focus() function interactive; it asks the operator to input a key. In a real program, the List class would probably have another public function that could be called with a parameter.

Once we know the key to remove, we search the list sequentially for a matching key. Since the list is unsorted, we must search the entire list before we know whether or not the key is in the list. If it is not, the pointer `here` will be NULL when we return from this function to Linear::remove(); the remove() will pass on the NULL it received to its caller, main().

```
// ------------------------------------------------------------------------
// Unsorted list: derived from  Container-<--Linear-<--List
// A. Fischer   June 9, 2001                                  file: list.hpp
// ------------------------------------------------------------------------
#ifndef LIST_H
#define LIST_H
#include "linear.hpp"
#include "item.hpp"

// ------------------------------------------------------------------------
class List : public Linear {
  public:
    void  insert( Cell* cp ) { reset(); Linear::insert(cp); }

    // --------------------------------------------------------------------
    void  focus(){
        KeyType k;
        cout <<"\n  What key would you like to remove? ";
        cin >> k;
        for (reset(); !end(); ++*this) if (*this == k) break;
    }
};
#endif
```

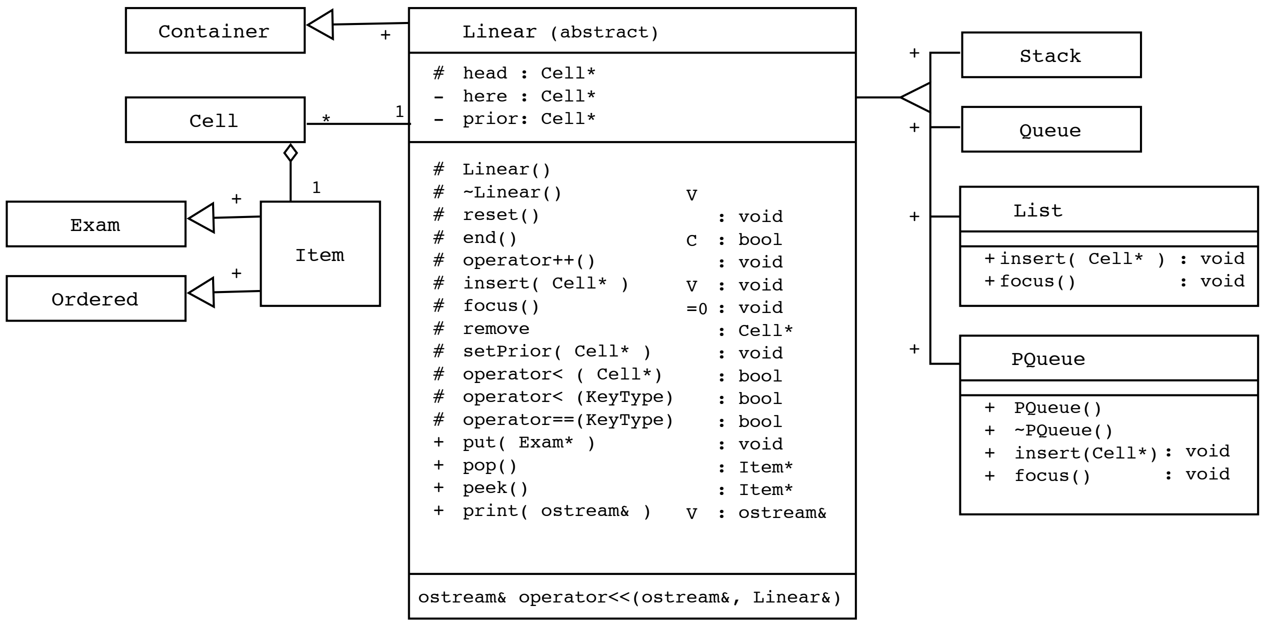


Figure 16.2: UML for all the Linear classes.

16.4.3 The Main Program

This main program has only one purpose: to test the two new classes. It puts meaningless data into two containers, takes some back out, and prints the results. The test results shown on the next page prove that insertions into the PQueue are implemented in the correct order, and that removal from the middle of the list works correctly. Other test runs verified that removal from head and tail of list, and attempted removal of a key that did not exist, also work properly. Trace comments after termination show that 9 cells (7 data cells and 2 dummy headers) and 7 Items were deleted by the destructors.

```
// -----------------------------------------------------------------------
// Demonstration of derived classes with virtual functionL.
// A. Fischer  June 15, 1998                                file: main.cpp
// -----------------------------------------------------------------------
#include "tools.hpp"
#include "pqueue.hpp"
#include "list.hpp"
#include "item.hpp"
// -----------------------------------------------------------------------
int main( void ) {
  PQueue P;
  List L;
  cerr << "  Print the empty List L.\n" <<L;
  cerr << "\n  Putting 3 items onto List L: 99, 77, 88.\n" ;
  L.put( new Item("Ned", 29) );     //cerr << L << endl;
  L.put( new Item("Leo", 37) );     //cerr << L << endl;
  L.put( new Item("Max", 18) );       cerr << L << endl;

  cerr << "  Putting 3 items onto PQueue P: 22, 11, 44.\n" ;
  P.put( new Item("Bea",22) );      //cerr << P << endl;
  P.put( new Item("Ali",11) );      //cerr << P << endl;
  P.put( new Item("Dan",44) );        cerr << P << endl;


  cerr << "  Remove one item from L and queue on P." ;
  P.put( L.pop()) ;

  cerr << "\n  Dequeue one item from P and push onto L." ;
  L.put( P.pop()) ;

  cerr << "\n  Pushing 33 onto P.\n" ;
  cerr << "\n  Peek at P: " << *P.peek() <<" \n";
  cerr <<"\n  The list contains: \n" << L;
  P.put( new Item("Cil",33) );
  cerr <<"\n  The priority queue contains: \n" << P << endl;
  bye();
}
```

The output.

```
  Print the empty List L.
  <[
  ]>

  Putting 3 items onto List L: 99, 77, 88.
  <[
Cell 0x0x33650 [Max: 18  , 0x33630]
Cell 0x0x33630 [Leo: 37  , 0x33610]
Cell 0x0x33610 [Ned: 29  , 0x0]
  ]>

  Putting 3 items onto PQueue P: 22, 11, 44.
  <[
Cell 0x0x336b0 [Dan: 44  , 0x33670]
Cell 0x0x33670 [Bea: 22  , 0x33690]
Cell 0x0x33690 [Ali: 11  , 0x0]
  ]>

  Remove one item from L and queue on P.
  What key would you like to remove? 37
  Deleting Cell 0x0x33630...
  Dequeue one item from P and push onto L.
  Deleting Cell 0x0x336b0...
  Pushing 33 onto P.

  Peek at P: Leo: 37

  The list contains:
  <[
Cell 0x0x336b0 [Dan: 44  , 0x33650]
Cell 0x0x33650 [Max: 18  , 0x33610]
Cell 0x0x33610 [Ned: 29  , 0x0]
  ]>

  The priority queue contains:
  <[
Cell 0x0x33630 [Leo: 37  , 0x376f0]
Cell 0x0x376f0 [Cil: 33  , 0x33670]
Cell 0x0x33670 [Bea: 22  , 0x33690]
Cell 0x0x33690 [Ali: 11  , 0x0]
  ]>


Normal termination.

  Deleting Cell 0x0x335f0...Deleting Item 44
    Deleting Score Dan...
  Deleting Cell 0x0x336b0...Deleting Item 18
    Deleting Score Max...
  Deleting Cell 0x0x33650...Deleting Item 29
    Deleting Score Ned...
  Deleting Cell 0x0x33610...
  Deleting Cell 0x0x335e0...Deleting Item 37
    Deleting Score Leo...
  Deleting Cell 0x0x33630...Deleting Item 33
    Deleting Score Cil...
  Deleting Cell 0x0x376f0...Deleting Item 22
    Deleting Score Bea...
  Deleting Cell 0x0x33670...Deleting Item 11
    Deleting Score Ali...
  Deleting Cell 0x0x33690...
multiple has exited with status 0.
```

16.5 C++ Has Four Kinds of Casts

16.5.1 Static Casts

A static cast is an ordinary type conversion. It converts a value of one type to a value with approximately the same meaning in another type. The conversion can be a lengthening (short to long), a shortening (int to char), or a change in representation (float to int). The C and C++ languages support the built in type conversions shown in Figure 16.5.1 These are called "static" casts because the compiler finds out that they are needed at compile time and generates unconditional conversion code at that time.

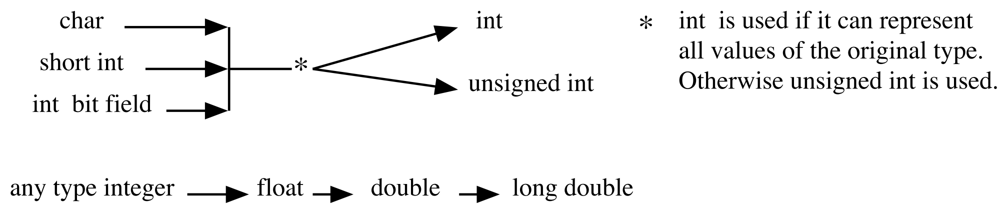

char
short int
int bit field
*
int
unsigned int
* int is used if it can represent all values of the original type. Otherwise unsigned int is used.
any type integer
float
double
long double

Figure 16.3: Built-in type conversions in C and C++.

Explicit casts. A static cast can be called explicitly using ordinary C syntax. In addition, C++ has two new ways to call a cast:

```
int k, m, *ip;
float f, *fp;
f = (float)k;                    // traditional C syntax.
```

```
f = float(k);                  // function-call syntax.
f = static_cast<float>(k);     // explicit C++ syntax.
```

Coercion. Coercion, or automatic type conversion, happens when a function call is encountered, and the type of an argument does not match the declared type of its parameter. In this case, the argument will be converted to the parameter type, if the compiler has a method for doing so. Coercion is also used to make operands match the type-requirements of operators. In C, coercion is limited to primitive types. However, in C++, it can also apply to a class type, say Cls:

- If an object of type T is used where a Cls object is needed, and the class Cls contains a constructor with one parameter of type T, the constructor will be used to coerce the T value to a value of type Cls.
- If an object of type Cls is used where a T object is needed, and the class Cls contains a cast operator whose result is type T, the cast operator will be used to coerce the Cls object so that the context makes sense.

16.5.2 Reinterpret Casts

A reinterpret cast is a type trick performed with pointers. It relabels the pointer's base type without changing any bits of either the pointer or its referent. This is like putting lamb's clothing on a wolf. Using a reinterpret cast, a program can access a value of one type using a pointer of a different type, without compiler warnings. (See Figure 16.5.2, line 147.) This lets us perform nonsensical operations such as adding incompatible values. For example, the reinterpret cast in the following program lets us relabel the integer 987654321 as a float, then add 1.0 to it to produce garbage:

```
140  #include <iostream.h>
141  using namespace std;
142  // ------------------------------------------------------------- File:  "assign.cpp"
143  int main( void )
144  {
145      int    my_int = 987654321;
146      int*   p_int = & my_int;
147      float* p_float  = (float*) p_int;
148      float  answer = *p_float + 1.0;
149      cerr <<answer <<"\n";
150  }
```

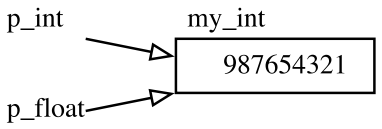


Figure 16.4: Relabeling the type of an object.

The answer is garbage: 1.0017, not 987654322 because The integer value and the floating point 1.0 were added to each other without any type conversion. The primary applications for reinterpret casts are hash functions and input conversion functions like strtod() and strtol().

Alternative syntax. Ordinary C syntax or C++ syntax with angle brackets can be used to invoke a reinterpret cast:

```
fp= (float*)p_int                    // ordinary C syntax.
fp= reinterpret_cast<float*>(ip);    // explicit C++ syntax.
```

16.5.3 Const Casts

A const cast provides a way to remove the const property from a pointer variable just long enough to change the value of its referent. This lets us use a constructor (or any class function) instead of a ctor to initialize a const class member.

```
#include <iostream.h>
int main( void )
{
    int w = 99;
    const int* cip = &w;

    cout <<"  &w is " <<&w <<" referent of cip is " <<cip <<endl;
    cout <<"  w= " <<w  <<"  *cip= " <<*cip <<endl;

    * const_cast<int*>(cip) = 33;
    //*cip = 33;
    cout <<"  w= " <<w  <<"  *cip= " <<*cip <<endl;
}
```

When we write `*cip = 33;` without a const cast, we get a const violation error:

```
const.cpp: In function ‘int main ()’:
const.cpp:165: assignment of read-only location
```

With a const cast, we are permitted to change the location and we get output:

```
  &w is 0xbffff714 referent of cip is 0xbffff714
  w= 99  *cip= 99
  w= 33  *cip= 33
```

16.5.4 Dynamic Casts

Dynamic casts are used with polymorphic classes, and can cast either pointers or references. Dynamic casts can move either upward or downward on the derivation tree. In Figure 16.6, a cast from B to D would be a downward cast, a cast from B to A would be an upward cast.

Suppose class D is derived from class B, as in the figure, and suppose p is a pointer (or a reference) to an object of class D. Then we can use a dynamic cast to relable p as a B pointer (or reference). An upward dynamic cast (from D to B) is always meaningful because any derived-class object includes a base-class object as part of itself.

According to my compiler, no downward casts are permitted at all. Accordiing to the Schildt text, some downward casts are permitted, but they are more complex and require a run-time check. Here is what Schildt says: If a pointer, p, has type A* it could be pointing at an object of any one of the four types A, B, C, or D. A down-cast of p from A* to B* would be meaningful if p's referent were actually type B or D, because these classes actually have all the members required by type B. However, such a down-cast would not be meaningful if p's referent were actually type A or C because some of B's members would be missing. It could cause a run-time crash if such a cast were permitted. To prevent such problems, the legality of every down-cast is checked at run time. A bad pointer down-cast will return a NULL result. A bad reference down-cast will throw an exception.

When do I use a dynamic-cast? You don't need to use an explicit dynamic cast to move up a derivation tree. The dynamic cast is done automatically whenever you use a derived-class object in a base-class context, or set a base-class pointer to point at a derived-class object. For this reason, it is hard to find a good use for an explicit dynamic cast in a simple program.

The rules for using dynamic casts are complicated by private parts, private derivation, and multiple inheritance. Some examples are given in the next section to illustrate these issues and show what kind of dynamic casts are and are not permitted in a multiple-inheritance situation.

16.6 Virtual Inheritance and Dynamic Casts

As long as derivation is used singly—so that a derived class has only one parent with data members—inheritance works smoothly, the storage model is easy to implement, and it is not hard to understand how it works. However, when a class inherits data members from two parents, two serious problems can occur:

- The `this` pointer points at the beginning of the entire object, consisting of the parts of the first parent, followed by the parts of the second, and finally, the parts of the derived class. To apply a function inherited from the second class, the compiler must compute where `this` should point for that part of the object. Dynamic casts are related to this problem.
- A class could inherit the same grandparent from two parents. If the grandparent has data members, are two copies of each inherited? Virtual inheritance exists to prevent this. When used in this way, the word `virtual` has nothing to do with virtual functions.

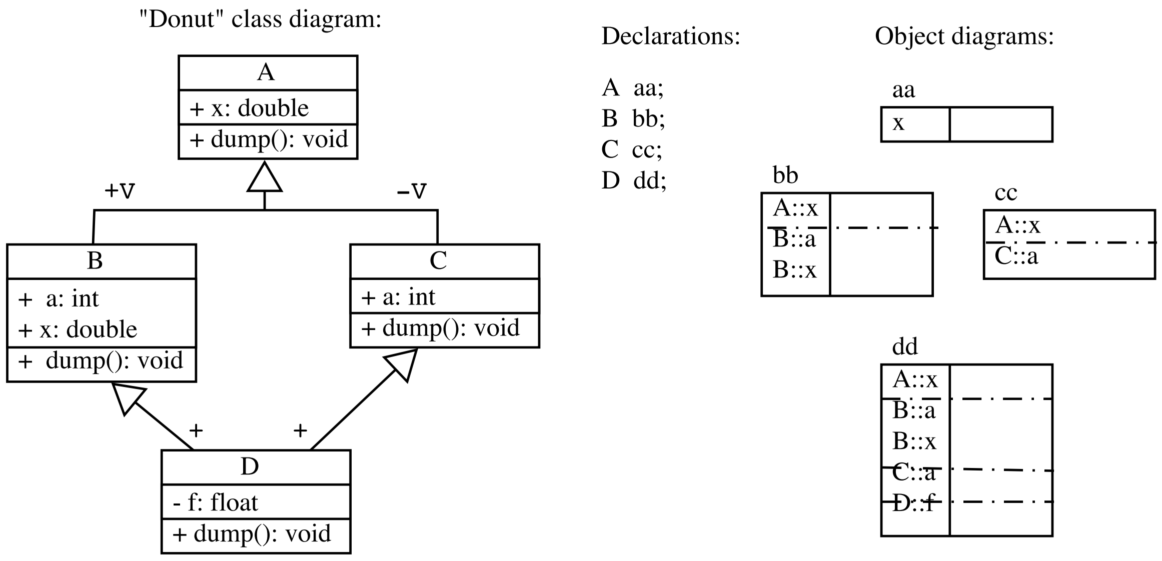


Figure 16.5: Double (donut) inheritance.

16.6.1 Virtual Inheritance

The keyword `virtual` can be used in a derivation declaration to prevent inheriting the same members from two parents. It is only relevant when multiple inheritance will be used, and when there is more than one path through the UML diagram from an ancestor class that has data members to some derived class. A class may have both virtual and nonvirtual base classes. When virtual derivation is used, an object of a derived class will have exactly one copy of the members of each ancestor. If derivation is not virtual, each derivation path will produce its own copy of any common ancestor, resulting in two data members with the same name in the derived-class object.

The simplest situation that illustrates the rules for virtual derivation is a class hierarchy in the shape of a "donut". The following set of four classes creates the "donut" shown in Figure 16.6. Each class has either one or two data members, a constructor, and a dump function. Data is made public to make it easier to show what is going on.

The data diagrams on the right show how storage would be allocated for objects of each of the four classes. The data member of A is inherited by classes B and C and becomes the first member of objects bb and cc. class D inherits all of the data members of B and all of the data members of C. Nonetheless, dd has only one sub-object of class A, because virtual derivation was used to derive B and C from A.

Naming rules. Two rules govern the meaning of a name in a donut situation:

- A class can inherit two members with the same name from different parent classes. When this happens, the name is *ambiguous* and you must use the :: to denote which one you want. For example, within class D, you would write B::a to refer to the member named 'a' inherited from B or C::a for the member inherited from C.
- If there are more than three levels in an inheritance hierarchy, the virtual property must be redeclared at each level that has multiple inheritance.

```
151  // ------------------------------------------------- File:  "donut.hpp"
152  #ifndef DONUT_H
153  #define DONUT_H
154  #include <iostream>
155  #include <iomanip>
156  using namespace std;
157  class A { //---------------------------------------- Grandparent Class
158      public:
159          double x;
160          A(): x(11.1) {}
161          virtual ~A(){};
162          virtual void dump(){ cerr <<"    A::x = " << x <<"\n"; }
163  };
164
165  class B: virtual public A { //---------------------- First Parent of D
166      public:
167          int a;
168          double x;
169          B(): a(20), x(22.2) {}
170          virtual ~B(){};
171          virtual void dump(){
172              A::dump();
173              cerr <<"    B::x = " << x <<"   "<<"B::a = " << a <<"\n";
174          }
175  };
176
177  class C: virtual private A { //--------------------- Second Parent of D
178      public:
179          int a;
180          C(): a(30) {}
181          virtual ~C(){};
182          virtual void dump(){ A::dump(); cerr <<"    C::a = " << a <<"\n"; }
183  };
184
185  class D: public B, public C { //---------------------- Grandchild Class
186          float f;
187      public:
188          D(): f(44.4) {}
189          void dump(){
190              A::dump();
191              B::dump();
192              C::dump();
193              cerr <<"    D::f = " << f <<"\n";
194          }
195  };
196  #endif
```

- A is a parent class of B, and both define a member named x. In classes A and C, only A::x is visible, so writing x in these contexts will always mean A::x. For functions in classes B and D, both A::x and B::x are visible, but B::x *dominates* A::x because it is “closer” to these classes. The dominant member will be used when x is written without the double colon. To refer to the non-dominant member, the full name A::x must be used.

The following brief program shows how the naming works and how visibility interacts with private inheritance. Two lines are commented out because they caused compilation errors:

- Line 204 produced this compiler error:

  ```
  double A::x  is inaccessible within this context in ‘C’ due to private inheritance.
  ```

- Line 210 was ambiguous. Neither B::a nor C::a is dominant here because the two classes B and C are equally close ancestors of D. To use either one in class D, we must qualify the member name as in lines 208 and 209.

```
// ------------------------------------------------------- file: naming.cpp
#include "donut.hpp"
int main( void )
{
    C cc;                              // Object has one base class.
    D dd;                              // Object has three base classes.
    cout <<" &cc.a=    " <<&cc.a <<"  value= " <<cc.a <<endl;
    //cout <<" cc.A::x= " <<&cc.A::x <<"  cc.x=" <<cc.x <<endl;

    cout <<" &dd.x=    " <<&dd.x    <<"  value= " <<dd.x <<endl;
    cout <<" &dd.A::x= " <<&dd.A::x <<"  value= " <<dd.A::x <<endl;
    cout <<" &dd.B::a= " <<&dd.B::a <<"  value= " <<dd.B::a <<endl;
    cout <<" &dd.C::a= " <<&dd.C::a <<"  value= " <<dd.C::a <<endl;
    //cout <<" &dd.a= "  <<&dd.a    <<"  value= " <<dd.a <<endl;
}
```

Here is the output:

```
&cc.a=    0xbffff864  value= 30
&dd.x=    0xbffff850  value= 22.2
&dd.A::x= 0xbffff840  value= 11.1
&dd.B::a= 0xbffff84c  value= 20
&dd.C::a= 0xbffff834  value= 30
```

16.6.2 Dynamic Casts on the Donut

The final program in this chapter demonstrates dynamic casts and polymorphism in the context of our donut-shaped derivation tree.

```
#include "donut.hpp"
// ----------------------------------------------------------- File:  "donut.cpp"
int main( void )
{
    A aa;   // Data member: x
    C cc;   // Data members: A::x, C::a
    D dd;   // Data members: A::x, B::a, B::x, C::a, D::f

    cerr <<"Dumping aa\n";  aa.dump();
    cerr <<"Dumping cc\n";  cc.dump();
    cerr <<"Dumping the B part of dd\n";  dynamic_cast<B*>(&dd)->dump();

    // Upward casts. ----------------------------------------------------------------
    B* bp = &dd;                          // Explicit Low->high pointer cast not needed.
    B& br = dd;                           // Explicit Low->high reference cast not needed.
    cerr <<"Dumping br\n";  br.dump();  // Use the reference variable to dump.

    A* ap = dynamic_cast<A*>(bp);       // Upward cast IS OK if derivation is public.
    cerr <<"Dumping ap\n";  ap->dump(); // Show result of dynamic cast.
    //dynamic_cast<A*>(&cc)->dump();    // Upward cast NOT OK if derivation is private.

    // Downward casts. --------------------------------------------------------------
    cerr <<"Dumping dp after up and down casts, D->B->A->D .\n";
    D* dp = dynamic_cast<D*>(bp);     // Can dynamic_cast downward where type is true.
    dp->dump();                       // Start with D and return to a D.

    cerr <<"Dumping cp after up and down casts, D->B->A->C .\n";
    C* cp = dynamic_cast<C*>(ap);     // Can dynamic_cast downward where type is true.
    cp->dump();                       // A D object has all the parts of a C object.

    cerr <<"Dumping after down casts, A->C .\n";
    ap = &aa;
    //cp = dynamic_cast<C*>(&aa);     // Cannot dynamic_cast down to wrong type.
    cp = dynamic_cast<C*>(ap);        // Cannot dynamic_cast down to wrong type.
    cerr <<"No exception was thrown.\n";
    cp->dump();
}
```

Polymorphism. The file `donut.hpp` defines a base class named A with two derived classes B, and C, and a class D derived from both B and C. We can create a donut diagram like this with or without polymorphism; a class only becomes polymorphic when it has virtual functions. In this example, Classs A, B, and C are polymorphic because they have virtual `dump()` functions. This forces us to also define virtual destructors.

Class D is the end of the derivation chain and the last class in the polymorphic family of classes. It is last because its derivation is not virtual and its functions are not virtual. Because these properties end in class D, it should not be used for further derivation.

A class is polymorphic if it has even one virtual function. If a class is polymorphic or is derived from a polymorphic class, the true type of every class object must be stored as part of the object at run time. I call this a "type tag". Whenever a virtual function is called, this type tag is used to select the most appropriate method for the function. If class is not part of a polymorphic family, no run-time type tag is attached to its objects, and no run-time function dispatching happens.

Virtual derivation

- Lines 216 through 222 create and print three variables. The output is:

```
Dumping aa
    A::x = 11.1
Dumping cc
    A::x = 11.1
    C::a = 30
Dumping the B part of dd
    A::x = 11.1
    B::x = 22.2   B::a = 20
```

- In this example, both B and C are derived virtually from A. If we omit the "virtual" from either one of the lines 163 and 174, or from both, we get this error comment.

```
donut.hpp: In method ‘void D::dump()’:
donut.hpp:37: cannot convert a pointer of type ‘D’ to a pointer of type ‘A’
donut.hpp:37: because ‘A’ is an ambiguous base class
```

Upward dynamic casts.

- Lines 225 and 226 perform implicit upward dynamic pointer and reference casts from class D to class B. Line 227 shows how to use the reference variable. The output is:

```
Dumping br
    A::x = 11.1
    B::x = 22.2   B::a = 20
```

- On line 222, an explicit cast was used because it was simplest. I tried an implicit cast there but it had the wrong prececence in relation to the `->` operator: `(B*)(&dd)->dump()`
- Line 229 shows another explicit dynamic cast, and line 230 prints the result. The output is:

```
Dumping ap
    A::x = 11.1
```

- Line 231 is commented out because it caused a privacy error:

```
Error in function ‘int main()’ of donut.cpp:
donut.cpp:20: dynamic_cast from ‘C’ to private base class ‘A’
A* ap = dynamic_cast<A*>(&cc);
```

Polymorphic downward dynamic casts.

- Lines 235, 239, 244, and 245 do explicit downward dynamic casts. If Classes A, B, and C did *not* have virtual functions, all of these lines would cause compile-time errors. The dynamic down-cast uses the run-time type information stored with every polymorphic object. But when a class is not polymorphic, the type tag is not there, and a dynamic down-cast cannot be done. Here is the error comment:

```
donut.cpp:24: cannot dynamic_cast ‘ap’ (of type ‘class A *’) to type ‘class C *’
```

- Because donut.hpp *does* defines a polymorphic class, lines 235, 239, 244, and 245 compile. The first of these lines is fine: we started with an object of type D, up-cast it, then down-cast it again. On line 235, we finished with class D, where we started. Clearly, every step in this casting-process was meaningful. The output is not surprising:

  ```
  Dumping dp after up and down casts, D->B->A->D .
      A::x = 11.1
      A::x = 11.1
      B::x = 22.2   B::a = 20
      A::x = 11.1
      C::a = 30
      D::f = 44.4
  ```

- On Line 239, we downcast the same object to class C, which is also meaningful, since every D object contains a C object as part of itself. Even though the program now thinks it has a C object, the object itself retains its true type identity, and when we dump it, we get class D's version of dump, just as we did on line 236.

  ```
  Dumping cp after up and down casts, D->B->A->C .
      A::x = 11.1
      A::x = 11.1
      B::x = 22.2   B::a = 20
      A::x = 11.1
      C::a = 30
      D::f = 44.4
  ```

- Line 244 is commented out because it gives a warning message:

  ```
  donut.cpp:33: warning: dynamic_cast of 'class A aa' to 'class C *' can never succeed
  ```

  This warning happens because aa is an object, not a pointer, and we know that it does not have all the members that a C object needs. No casting magic can create the missing parts.

- On line 245, we down-cast an A* instead of an A&. There is no warning here, because the compiler cannot predict the actual type of an object that an A* pointer might point at. The result is a run-time malfunction:

  ```
  Dumping after down casts, A->C .
  No exception was thrown.
  Bus error
  ```

  The Schildt text says that this downcast should cause an exception to be thrown, and since this program does not attempt to catch exceptions, the program should be terminated. (Exceptions are covered in Chapter 17.) Clearly, since control reached line 246, this did not happen. The compiled code did not check for an illegal downcast; it performed it. The result was a bus error (segmentation error on a second machine) when the print function tried to access a member of the object that never existed.

Chapter 17: Exceptions

From the board game MONOPOLY, the rule to follow when your man lands on the “illegal” square:

Go to jail. Go directly to jail, do not pass GO and do not collect $200.

17.1 Handling Errors in a Program

Function calls, loops, conditionals, switches, and breaks permit the programmer to control and direct the sequence of evaluation of a program and modify the default order of execution, which is sequential. These statements permit controlled, local perturbations in the order of execution. Almost all situations that arise in programming can be handled well using some combination of these statements.

The goto statement is also supported in C, but its use is discouraged because it makes uncontrolled, non-local changes in execution sequence. Use of a goto is even worse in C++; if it is used to jump around the normal block entry or exit code, the normal construction, initialization, and deletion of objects can be short-circuited, potentially leaving the memory in an inconsistent state. Therefore, this statement should simply not be used.

There are some situations in which the ordinary structured control statements do not work well. These involve unusual situations, often caused by hardware or input errors, which prevent the program from making further fruitful progress. In C++, a failure during execution of a constructor is one situation that calls for use of exceptions.

In the old days, these situations would be handled by a variety of strategies:

1. Do nothing. Don’t check for errors; hope they don’t happen. Let the program crash or produce garbage answers if an error happens.

2. The program could identify the error and call an error function. However, this defers the problem instead of solving it. The error function still must do something about the error.

3. Identify the error, print an error comment, and call exit(). (This is equivalent to using the fatal() function in the toolspp library.) However, aborting execution is not permissible in many real-life situations. For example, aborting execution of a program that handles bank accounts could leave those accounts in an inconsistent or incorrect state.

4. One could use assert() to check for errors. This is similar to option (1) but worse because it gives no opportunity to print out information about the error or its cause.

5. The function being executed could return with an error code. The function that called it would need to check for that error code and return to *its* caller with an error code, and so on, until control returned to the level at which the error could be handled.

   This method usually works but distorts the logic of the program and clutters it with a large amount of code that is irrelevant 99% of the time. Using this technique discourages use of functions and modular code.

6. A long-distance goto could be used to return to the top level. Using this strategy, any information about the error would have to be stored in global variables before executing the goto. This is like programming in BASIC. Use of this control pattern destroys the modularity that C programs can otherwise achieve. It is not recommended.

17.1.1 What an Exception Handler Can Do

An exception handler provides an additional control option that is designed for dealing with unusual situations (exceptions) in which the ordinary structured control statements do not work well. (Exceptions can be used in non-error-conditions, but should not be used that way.) An exception handler lets control go ...

- From the level at which an error is discovered, often deep within the code, at a stage when several functions have been called and have not yet returned ...
- Carrying as much information as necessary about the error ...
- To the level at which the error can be handled, often in the main function or a high-level function that handles the overall progress of the program or operator interaction...
- And back into normal execution, if that is appropriate.

An exception handler and the exceptions it can handle are defined at a high level. When an exception condition is identified at a lower level, an exception object is created and "thrown" upward. This causes immediate exit from the originating function, with proper clean-up of its stack frame. The function that called it then "has the ball" and can either catch the exception or ignore it (ignoring the exception causes an immediate return to *its* caller). Thus, control passes backward along the chain of function calls until some function in the chain-of-command "catches" the exception. All stack frames in this chain are deleted and the relevant destructors are run. The programmer should be careful, therefore, not to allow exceptions to be thrown when some objects are half-constructed.

Implementing a compiler and run-time-system that can do this efficiently is a hard problem. Exceptions can come in many types and each type of exception must have a matching handler. An exception is an object that may have as many fields as necessary to contain all the important facts about where and why the exception was thrown. This object must outlive the function that created it (it must be created in dynamic storage, not on the stack) and it must carry an identifying type-tag, so it can be matched to the appropriate exception handler. An uncaught exception will abort a process.

What next? After catching an exception, there are several options for handling it:

1. The catcher can clean up the data and execute return with some appropriate value.
2. The catcher can clean up the data and call the containing function recursively.
3. The catcher can clean up the data and continue from the line that follows the exception handler (not the line that follows the function call that caused the exception).
4. The catcher can comment, get more information from the operator, and carry on in either of the two preceding ways.
5. The catcher can abort the process or return to its caller, as in C.
6. The catcher can fix some data fields and rethrow the same exception or some other exception.

17.2 Defining Exceptions in C++

One major principle applies to the use of exceptions: they are not a substitute for proper use of `else` or `while`, so the function that throws an exception should not be the function that catches it. Exceptions are intended for global, not local, use. Their proper use is to enable control and information to pass, directly, across classes and through a chain of function calls.

17.2.1 Defining an Exception Type

An exception is an object and its type is defined like any class. It can (but usually does not) have public, protected, and private parts. The exception class definition can be global or it can be contained within the definition of another class. Related exceptions can be created by derivation. For example, the Bad, BadSuit, and

BadSpot classes, below, define three exception types that might be used in a program that plays an interactive card game and reads input from the keyboard. The class Bad is a base class for the others and defines the functionality common to all three classes. The UML diagram in Figure17.1 shows the derivation hierarchy. The purple circle with a V marks a virtual function.

```
//===========================================================================
// Exception Classes for Playing Card Errors.                     file: bad.hpp
// Exception demonstration program:  March 2009
// ---------------------------------------------- Error reading input stream.
#pragma once;
#include "tools.hpp"

class Bad {
public:
    char spot;
    char suit;
    //---------------------------------------------- Suit and Spot both wrong.
    Bad (char n, char s) : spot(n),  suit(s) {};
    virtual void print(){
        cerr <<"  Both spot value and suit are wrong\n"
        <<"  Legal spot values are 2..9, T, J, Q, K, A\n"
        <<"  Legal suits are H D C S\n";
        pr();
    }
    void pr(){
        cerr <<"  You entered "<<spot <<" of " <<suit
        <<". Please reenter. \n";
    }
};

// ---------------------------------------------------- Only the suit is wrong.
class  BadSuit : public Bad {
public:
    BadSuit (char n, char s) : Bad(n, s) {}
    virtual void print(){
        cerr <<"  Legal suits are H D C S\n";
        pr();
    }
};

//--------------------------------------------- Only the spot value is wrong.
class  BadSpot : public Bad {
public:
    BadSpot (char n, char s) : Bad(n, s) {}
    virtual void print(){
        cerr <<"  Legal spot values are 2..9, T, J, Q, K, A\n";
        pr();
    }
};
```

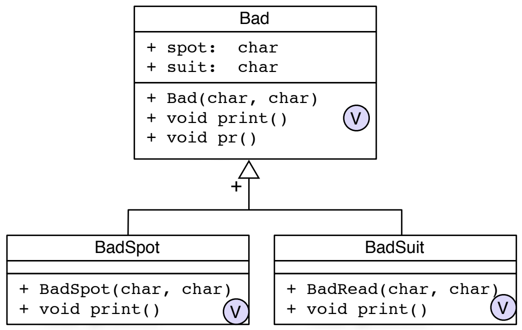


Figure 17.1: UML diagram for the exception classes.

17.2.2 Exception Specifications

In Java, every class must declare the exceptions that might happen within it, unless they are also caught within it. This requirement is confusing to beginners, but it helps make Java programs more predictable and it leads to better-informed error handling. In C++, a function *may* declare a list of potential exceptions, or declare that it does not throw exceptions. However, such a declaration is not required. Since an uncaught exception will terminate a program, this can be an important kind of documentation. Note: this is a new feature of C++ and may not be supported by all compilers.

The exception specification follows the parameter list in a prototype and precedes the semicolon. For example, suppose a fuction named `divide(int, int)` could throw two kinds of exceptions. The declaration would be:

```
int divide (int, int) throw (Zerodivide, MyException);
```

We can also declare that a fuction does not throw exceptions at all. The syntax is:

```
int safeFunction (int, int) throw ();
```

Technically, this declaration is legal even if some function called by safeFunction throws exceptions. However, since we can't know (let alone control) which library functions throw exceptions, this kind of declaration has questionable value.

17.2.3 Playing card demo.

Next is a simple class that models a playing card. We will use it to demonstrate throwing and catching exceptions. The constructor for this class takes input from the keyboard, a notoriously error-prone data source. Each card is represented by two chars representing the face value (spot) and the suit of the card. Spaces and capitalization are not important and can be used freely. After reading the two chars, the Card constructor validates them and (possibly) throws an exception. The nature of the input error is encoded in the name of the exception, enabling the program to use virtual functions to display a specific and appropriate error comment each time.

```
//===========================================================================
//  A playing card class and related exception classes.
//  Alice E. Fischer, June 16, 1998                                 cards.hpp
//===========================================================================
#pragma once;
#include "tools.hpp"
#include "bad.hpp"

enum SuitType{ spades, hearts, diamonds, clubs, bad };

//===========================================================================
// This is the main data class; it represents one playing card.
//
class  Card {
    int         spot_;
    SuitType    suit_;
public:
    Card (istream& sin) throw (Bad, BadSpot, BadSuit);
    int         spot(){ return spot_; }
    SuitType    suit(){ return suit_; }
    ostream&    print(ostream&);
    static void instructions(ostream&, int n);
};
```

17.2.4 Throwing an Exception

The `throw` statement is used to construct an exception and propagate it. One type of exception is commonly used without definition; a function may throw a literal string (type `string` or `char*`). Throwing an exception causes control to pass backwards through the chain of function calls to the nearest previous catch clause that handles that particular type of exception. The destructors will be run for all objects in the stack frames between the throw and the catch.

```
 68   //=========================================================================
 69   //  Functions and constants for the Card class.
 70   //  Alice E. Fischer, June 16, 1998                                cards.cpp
 71   //=========================================================================
 72   #include "cards.hpp"
 73   const char* suitlabels[5] = {"spades", "hearts", "diamonds", "clubs", "bad"};
 74
 75   const char* spotlabels[16] = {
 76       "bad","Ace","2","3","4","5","6","7","8","9",
 77       "10","Jack","Queen","King","Ace"
 78   };
 79
 80   // -----------------------------------------------------------------------
 81   void Card::instructions( ostream& out, int n) {
 82       out << "Please enter " << n << " cards.\n"
 83           << "Spot codes are 2..9, T, J, Q, K, A \n"
 84           << "Suit codes are S H D C \n";
 85   }
 86
 87   // -----------------------------------------------------------------------
 88   Card::Card (istream& sin) throw (Bad, BadSpot, BadSuit) {
 89       char inspot, insuit;
 90       sin >> inspot >> insuit;
 91       if (!sin.good()) throw "Low level read error\n";
 92       if (inspot >=’2’ && inspot<=’9’) spot_ = inspot - ’0’;
 93       else switch( toupper(inspot) ){
 94           case ’T’: spot_ = 10; break;
 95           case ’J’: spot_ = 11; break;
 96           case ’Q’: spot_ = 12; break;
 97           case ’K’: spot_ = 13; break;
 98           case ’A’: spot_ = 1; break;
 99           default : spot_ = 0;
100       };
101       switch( toupper(insuit) ){
102           case ’S’: suit_ = spades;   break;
103           case ’H’: suit_ = hearts;   break;
104           case ’D’: suit_ = diamonds; break;
105           case ’C’: suit_ = clubs;    break;
106           default : suit_ = bad;
107       };
108       if (spot_ == 0 && suit_ == bad) throw Bad(inspot, insuit);
109       if (spot_ == 0)   throw BadSpot(inspot, insuit);
110       if (suit_ == bad) throw BadSuit(inspot, insuit);
111   }
112
113   // -----------------------------------------------------------------------
114   ostream&
115   Card::print(ostream& sout) {
116       return sout <<spotlabels[spot_] <<" of " <<suitlabels[suit_] <<endl;
117   }
```

In this program, line 91 throws a string exception. This will be caught by the general exception handler on line 153. Lines 108–10 throw exceptions from the Bad hierarchy. These will be caught on line 146. The names of these exceptions are announced on line 88. (This is a worthy but optional form of program documentation. It is not clear how much checking, if any, compiler does with these declarations.)

17.2.5 Catching an Exception

```
118  //==============================================================================
119  //  Exception handling demonstration.
120  //  Alice E. Fischer, June 16, 1998                                     except.cpp
121  //==============================================================================
122  #include "tools.hpp"
123  #include "cards.hpp"
124  #include "bad.hpp"
125  #define NCARDS 3
126
127  int main( void )
128  {
129      Card* hand[NCARDS];
130      int k;
131      bool success;
132      Card::instructions( cout, NCARDS );
133      //----------------------------------------- Main loop that reads all cards.
134      for (k=0; k<NCARDS; ){
135          success = false;         // Will not be changed if an exception happens.
136          //------------------------------ Here is the single line of active code.
137          try {
138              cout << "\nEnter a card (spot code, suit code): " ;
139              hand[k] = new Card(cin);    //---------------------- Input one card.
140              success = true;             //-- No exception - we have a good card.
141              cout << "  Card successfully entered into hand: ";
142              hand[k]->print(cout);
143              ++k;
144          }
145          //----------------- Check for the three application-specific exceptions.
146          catch (Bad& bs) { bs.print(); }          // Will catch all 3 Bad errors.
147
148          //------------------- Now check for general exceptions thrown by system.
149          catch (bad_alloc bs) {          //-------------- Catch a malloc failure.
150              cerr << "  Allocation error for card #" <<k <<".\n";
151              return 1;
152          }
153          catch (...) {                   //--------------- Catch everything else.
154              cerr << "  Last-ditch effort to catch exceptions.\n";
155          }
156          // ----------------- Control comes here after the try/catch is finished.
157          if(!success) delete hand[k];       // ----- Delete the half-made object.
158      }
159      cout << "\nHand is complete:" << endl;
160      for (k = 0; k < NCARDS; ++k) { hand[k]->print( cout ); }
161  }
```

Exceptions are caught and processed by exception handlers. A handler is defined like an ordinary function; the type of its parameter is the type of exception that it will catch.

Code that may generate exceptions (at any nesting level) and wishes to catch those exceptions must be enclosed in a `try` block (lines 137..144). The try block can contain multiple statements of any and all sorts. The exception handlers are written in `catch` blocks that immediately follow the `try` block (lines 146..155). Several things should be noted here:

1. The fields inside an exception object may be used according to the normal rules of public and/or private access. The data can be public because there is no need to protect it.

2. If the handler does not need to access the information in the exception, the parameter name may be omitted.

3. The order in which the handlers are written is important; the general case must come after all related specific cases.

4. A base-class exception handler will catch exceptions of all derived types. When an exception of a derived type is caught, the fields that belong to the derived type are not "visible" to the handler. If the exception is rethrown, all fields (including the fields of the derived type) are part of the rethrown object.

5. An exception handler whose parameter is . . . will catch all exceptions.

In this example, line 139 calls the Card constructor, which can throw exceptions from the Bad class and its derived classes. The handler for those exceptions is on line 146. This line will catch all three kinds of exceptions and process them by calling the print function in the exception class.

Processing the exception. When an exception is caught by line 146, we call the `bs.print()` to process it. Note that the print function is virtual in the Bad class hierarchy, and has three defining methods. When `bs.print()` is called, the system will inspect the run-time type-tag attached to `bs` to find out which class or subclass constructed this particular exception. Then the `print()` function of the matching class will be called. This is how we get three different kinds of output from one catch clause. (Note the first, second, and third blocks of error comments in the output transcript, below.) If `Bad::print()` were not virtual, the base-class `print()` function would always be used.

Output The output that follows shows two sample runs of this program with different exception handlers active. The first output is from the program as shown:

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter a card (spot code, suit code): mn
  Both spot value and suit are wrong
  Legal spot values are 2..9, T, J, Q, K, A
  Legal suits are H D C S
  You entered m of n. Please reenter.

Enter a card (spot code, suit code): 2m
  Legal suits are H D C S
  You entered 2 of m. Please reenter.

Enter a card (spot code, suit code): 1s
  Legal spot values are 2..9, T, J, Q, K, A
  You entered 1 of s. Please reenter.

Enter a card (spot code, suit code): 2s
  Card successfully entered into hand: 2 of spades

Enter a card (spot code, suit code): kh
  Card successfully entered into hand: King of hearts

Enter a card (spot code, suit code): JC
  Card successfully entered into hand: Jack of clubs

Hand is complete:
2 of spades
King of hearts
Jack of clubs
```

The following output was produced after commenting out the first catch clause (line 146).

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter a card (spot code, suit code): mn
  Last-ditch effort to catch exceptions.

Enter a card (spot code, suit code): 3s
   Card successfully entered into hand: ... and so on.
```

17.2.6 Built-in Exceptions

Simple objects like iintegers and strings can also be thrown and caught. In addition, `C++` has about a dozen built-in exceptions (see Schildt, pages 922-924). One of these, `bad_alloc` changes the way we write programs.

In C, it was necessary to check the result of every call on `malloc()`, to find out whether the system was able to fulfill the request. In C++, such a check is not necessary or helpful. If there is an allocation failure, the run-time system will throw a `bad_alloc` exception and control will not return to the failed call. If your program does not have a handler for such exceptions, and one occurs, the program will be terminated.

The `bad_cast` exception is used when a program executes an invalid downward dynamic cast, and `bad_exception` is thrown when a function violates its own exception specification.

One group of standard exceptions called `runtime_errors` are beyond the programmer's control and are used for mistakes in library functions or the run-time system. These include `overflow_error`, `range_error`, and `underflow_error`.

A final group of exceptions called `logic_errors` are defined by the standard but, so far as I know, not used by the system. They seem to be intended for use by any program when a run-time error is discovered. These include `domain_error`, `invalid_argument`, `length_error`, and `out_of_range`.

17.2.7 Summary

In the hands of an expert, exception handlers can greatly simplify error handling in a large application. However, they are not easy to use and using them without understanding is likely to lead to errors that are difficult to diagnose and cure. They are also difficult for a compiler to handle and force the compiler to interact with the host system in awkward ways. Virtual print functions are a powerful and simple tool to produce good error comments in situations where multiple faults can occur. Be aware, though, that processing a call on a virtual function takes more time than processing a non-virtual function in the same class.

Chapter 18: Design Patterns

Design patterns are elegant, adaptable, and reusable solutions to everyday software development problems. Each pattern includes a description of a commonly occuring type of problem, a design for a set of classes and class relationships that solve that problem, and reasons why the given solution is wise.

18.1 Definitions and General OO Principles

18.1.1 Definitions

1. Subclass: X is a subclass of Y if X is derived from Y directly or indirectly.
2. Collaboration: two or more objects that participate in a client/server relationship in order to provide a service.
3. Coupling: A dependency between program elements (such as classes) typically resulting from collaboration between them to provide a service. Classes X and Y are coupled if. . .
   - X has a function with parameter or local variable of class Y.
   - X has a data member that points at something of class Y.
   - X is a subclass of Y.
   - X implements an interface for class Y (Y gives friendship to X).

   [Example:] If the Key class calculates the hash-table index, it must know the length of the hash table. But this couples two classes that would not otherwise be coupled.
4. Cohesion: This is a measure of how strongly related and focused the responsibilities of a class are. A class with high cohesion is a "specialist" with narrow power.
5. System event: A high-level event generated by an external actor; an external input event. For each system event, there is a corresponding operation. For example, when a word-processor user hits the "spell check" button, he is generating a system event indication "perform spell check".
6. Use case: The sequence of events and actions that occur when a user participates in a dialog with a system during a meaningful process.

18.2 General OO Principles

1. Encapsulation. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another.
2. Narrow interface. Keep the interface (set of public functions) as simple as possible; include only those functions that are of direct interest to client classes. Utility functions that are used only to implement the interface should be kept private. [Why] This minimizes the chance for information to leak out of the class or for a function to be used inappropriately.
3. Delegation: a class that is called upon to perform a task often delegates that task (or part of it) to one of its members who is an expert. [Example:] HashTable::find selects one list and delegates the searching task to List::find.

18.3 Patterns

A pattern is a design issue or communication problem... with a solution based on class structure... and guidance on how to apply the solution in a variety of contexts.

18.3.1 GRASP: General Responsibility Assignment Software Patterns

- High cohesion is desirable. [Why?] It makes a class easier to comprehend, easier to maintain, and easier to reuse. The class will also be less affected by change in other classes. [Example:] Cohesion is low if a HashTable class contains code to extract fields from a data record. Cohesion is low if the data class computes a hash index. Cohesion is high if each class does part of the process.
- Low coupling is desirable. Which class should be given responsibility for a task? [A] Assign a responsibility so that its placement does not increase coupling. [Why?] High coupling makes a class harder to understand, harder to reuse, and harder to maintain because changes in related classes force local changes.
- Expert. Who should do what? [A] Each class should do for itself actions that involve its data members. Each class should "take care of" itself and handle its own emergencies. [Why] This minimizes coupling.
- Creator. Who should create (allocate) an object? [A] The class that composes, aggregates or contains it. [Why] This minimizes coupling.

  Who should delete (deallocate) an object? [A] The class that created it. [Why] To minimize confusion, and because, often, nothing else is possible.
- Don't Talk to Strangers. That is, don't "send messages" to objects that are not close to you. Non-strangers are:
  - your own data members.
  - elements of a collection which is one of your own data members.
  - a parameter of the current function.
  - a locally-created object.
  - `this` (but using `this` is rarely the right thing to do).

  Delegate the operation [Why] This is a generalization of the old rule, don't use globals. It maximizes the locality of every reference and avoids unnecessary coupling between classes.

  [Example] Don't deal directly with a component of one of your own members. Suppose a hardware store class composes an object of class Inventory, and the Inventory is a flex-array of Item pointers, as shown below. The retail store would be talking to a stranger if its sell function called the sell function in the Item class directly, like this: `Inv.find(currentKey)->sell(5);` The preferred design is to work through the intermediate class. That is, Inventory should provide a function such as `sell(key, int)` that can be called by RetailStore, and `Inventory::sell(key, int)` should call `Inventory::find(key)` followed by `Item::sell(int)`. Evaluation: In the first design, a change in the Item class might force a change in both RetailStore and Inventory. Using the second design, only Inventory is affected.

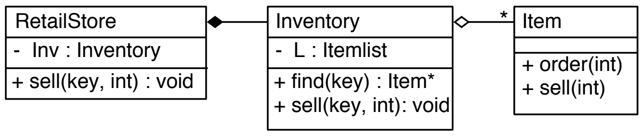


Figure 18.1: Don't talk to strangers.

18.4 More Complex Design Patterns

- Adapter. Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires. [Solution:] Define an adapter class that can add, subtract, or override functionality, where necessary. There are two ways to do this; on the left is a class adapter, on the right an object adapter.

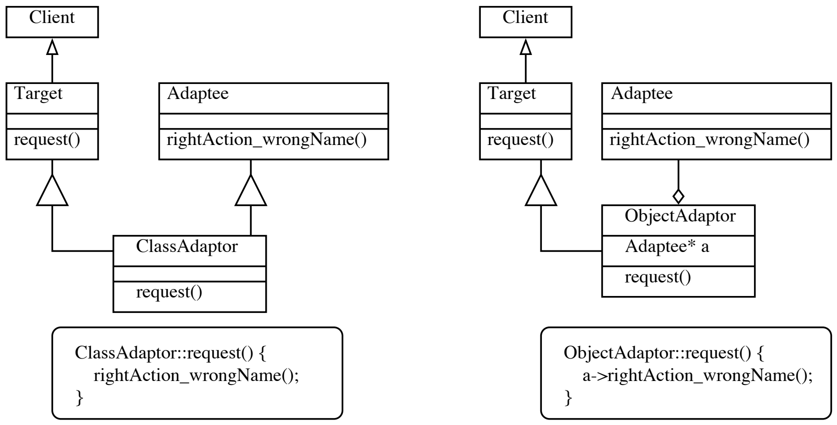


Figure 18.2: Two kinds of adapters.

- Indirection. This pattern is used to decouple the application from the implementation where an implementation depends on the interface of some low-level device. [Why] To make the application stable, even if the device changes.

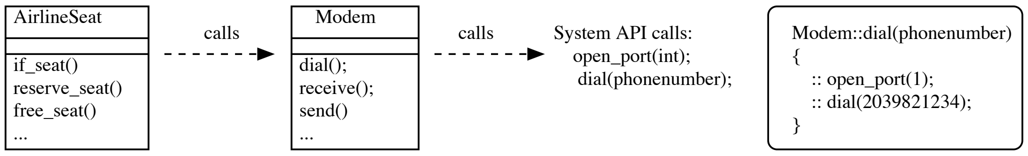
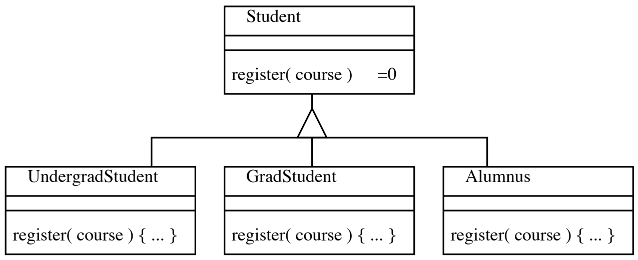


Figure 18.3: Indirect coupling to an unstable interface.

- Proxy. This pattern is like Indirection, and is used when direct access to a component is not desired or possible. What to do? [Solution:] Provide a placeholder that represents the inaccessible component to control access to it and interact with it. The placeholder is a local software class. Give it responsibility for communicating with the real component. [Special cases:] Device proxy, remote proxy. In Remote Proxy, the system must communicate with an object in another address space.


Student
register(course) =0
UndergradStudent
register(course) { ... }
GradStudent
register(course) { ... }
Alumnus
register(course) { ... }
Figure 18.4: Polymorphic implementation of an abstraction.

- Polymorphism: In an application where the abstraction has more than one implementation, define an abstract base class and one or more subclasses. Let the subclasses implement the abstract operations. [Why] to decouple the implementation from the abstraction and allow multiple implementations to be introduced, as needed.
- Controller: Who should be responsible for handling a system event? [A] A controller class. The controller should coordinate the work that needs to be done and keep track of the state of the interaction. It should delegate all other work to other classes.

  Factors such as the number of events to be handled, cohesion and coupling should be used to decide among the three kinds of controllers described below and to decide how many controllers there should be. A controller class represents one of the following choices:

  – The overall application, business, or organization (facade controller).

– Something in the real world that is active that might be involved in the task (role controller). [Example:] a menu handler.
– An artificial handler of all system events involved in a given use case (use-case controller). [Example:] A retail system might have separate controllers for BuyItem and ReturnItem.

- Bridge: This pattern is a generalization of the Indirection pattern, used when both the application class and the implementation class are (or might be) polymorphic. The bridge decouples the application from the polymorphic implementation, greatly reducing the amount of code that must be written, and making the application much easier to port to different implementation environments. In the diagram below, we show that there might be several kinds of windows, and the application might be implemented on two operating systems. The bridge provides a uniform pattern for doing the job.

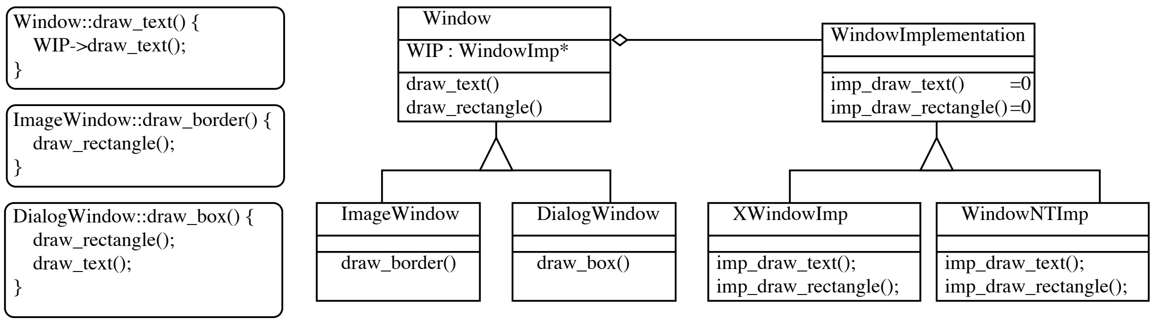


Figure 18.5: Bridging the implementation gap.

- Subject-Observer or Publish-Subscribe: Your application program has many classes and many objects of some of those classes. You need to maintain consistency among the objects so that when the state of one changes, its dependents are automatically notified. You do not want to maintain this consistency by using tight coupling among the classes.

  [Example:] An OO spreadsheet application contains a data object, several presentation "views" of the data, and some graphs based on the data. These are separate objects. But when the data changes, the other objects should automatically change.

  [Solution:] In the following discussion, the SpreadsheetData class is the subject, the views and graphs are the observers. The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects from its list. Observer objects may be added to this list, as needed, and all will be notified when the subject (SpreadsheetData) changes. We derive a concrete subject class (SpreadsheetData) from the Spreadsheet class. It will communicate with the observers through a `get_state()` function, hat returns a copy of its state.

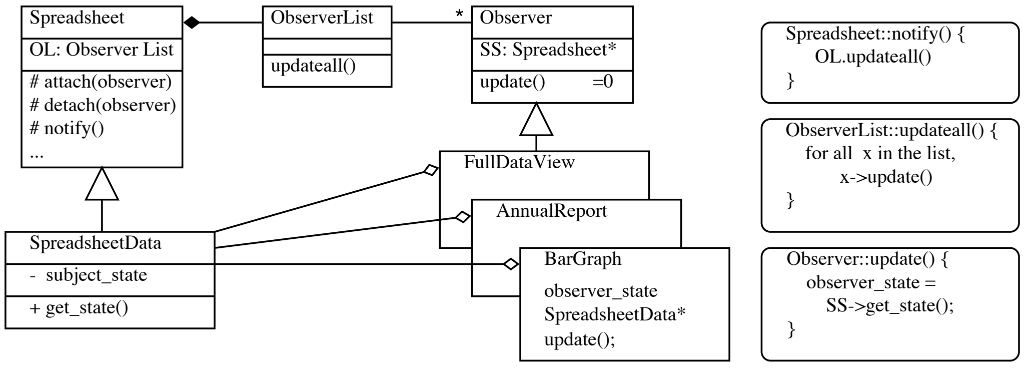


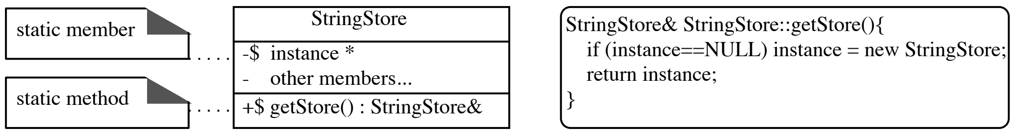
Figure 18.6: One subject with three observers

The ObserverList class defines an updating interface for objects that should be notified of changes in a subject. The Observer class provides an abstract public function called update() which will be called by

ObserverList whenever updateall() is called. This abstract function must be implemented in each concrete observer class.

When the state of the SpreadsheetData subject changes, it executes its inherited notify() function, which calls `ObserverList::updateall()`, which notifies all of the observers. Each one, in turn, executes its update function, which calls the subject's get_state function. Changes can then be made locally that reflect the change in the subject's state.

- Singleton: Suppose you need exactly one instance of a class, and objects in all parts of the application need a single point of access to that instance. [Solution:] A single object may be made available to all objects of class C by making the singleton a static member of class C. A class method can be defined that returns a reference to the singleton if access is needed outside its defining class.



```
StringStore& StringStore::getStore(){
    if (instance==NULL) instance = new StringStore;
    return instance;
}
```

Figure 18.7: How to create a singleton.

[Example] Suppose there were several parts of a program that could use a StringStore. We might define StringStore as a singleton class. The `StringStore::put` function would be made static and would become a global access point to the class, while maintaining full protection for the class members.

More patterns? The seven patterns presented here are some of the earliest and most useful that have been developed. But many, many more design patterns have been identified and published. A professional working in either C++ or Java would do well to study some of the available literature.