Chapter 1: Preamble

This book is intended for use by students who hope to deepen their understanding of C++ and learn about advanced features. It is also useful for C or Java programmers who want to learn C++ and OO style . . . fast. It assumes that the reader knows basic programming including types, type-matching rules, control structures, functions, arrays, pointers, and simple data structures. The material should help you develop a deeper understanding of the implementation of C++, of clean program design and of the features that make C++ a powerful and flexible language.

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 differences are in the use of classes to organize and simplify the code, and the way we do I/O.

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. Note, however, that global constants are OK.

Commandment 3. Test every line of code you write. Turn in screen shots or capture the output from the terminal window. **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.

Commandment 4. Choose a reasonable style and use it consistently. This covers naming, indentation, comments, layout, and every other aspect of programming.

1.2 Style

If you want to be a professional, learn to make your work look professional.

When you work for a company, you will be expected to follow all the style guidelines and coding rules established at that company. They may or may not be the best guidelines and styles. However, following them is not optional. Note: it does not help to tell the boss that he/she is prescribing obsolete or strange customs. As an example, look at NASA's guidelines below.

Example of Local Style Guidelines Following are NASA's rules for developing safety-critical code. Many of them are beyond the scope of this course, but they give an example of local style rules.

1. Restrict all code to very simple control flow constructs; do not use goto statements, setjmp or longjmp constructs, or direct or indirect recursion.
2. Give all loops a fixed upper bound.

3. Do not use dynamic memory allocation after initialization.
4. No function should be longer than what can be printed on a single sheet of paper in a standard format with one line per statement and one line per declaration.
5. The code's assertion density should average to minimally two assertions per function.
6. Declare all data objects at the smallest possible level of scope.
7. Each calling function must check the return value of non-void functions, and each called function must check the validity of all parameters provided by the caller.
8. The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions.
9. Limit pointer use to a single dereference, and do not use function pointers.
10. Compile with all possible warnings active; all warnings should then be addressed before release of the software.

1.2.1 Guidelines and Rules for this Course.

These programming techniques deal with details that are specific to C/C++ programming and are important for success in this course. There is a well-considered reason behind each one, and I expect you to follow them, even if they do not seem "right" to you. In the early weeks of this course, I will help you learn to apply the principles. Your willingness, diligence, and ability to write clean code in my prescribed style will determine my opinion of you and your skills.

1. Use only the **standard C++ language** and its standard template library. Do not use proprietary extensions and language additions such as `clrscr()` and `conio.h`. Also, do not use system commands such as `pause`. Instead, learn how to do these things within the standard language. Your code must compile and run properly on whatever systems the TA and the Professor have.

2. For a compiler, use **g++** (on Linux) or **clang++** (on the Mac). If you turn in work done in visual studio, it may or may not be accepted, because it **must compile and run on my system**.

3. Learn how to include and use a local library, how to create, organize, and compile a multi-module program, and how to compile, link, and run a program from a command shell.

4. Wherever possible, use **symbolic names** in your code. For example, use quoted characters instead of numeric ASCII codes.

5. **Whitespace**. In general, put spaces around your operators and either before or after each parenthesis. Do not write a whole line of code without spaces! Readability matters!

6. **Comments**. Use // and keep comments short. Do not repeat the obvious. Keep everything (code and comments) within 80 columns.

7. **Blank lines**. Use blank lines to group your code into "paragraphs" of related statements. Put a blank line after each paragraph.

   - DO NOT put a blank line after 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.

8. **Simplicity** is good; complexity is bad. If there are two ways to do something, the more straightforward or simpler way is preferred.

9. **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.

10. 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.

11. 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.

12. **Brevity.** Keep all functions short. At worst, a function should fit on one screen. Ideally, the function should be shorter than a dozen lines. Break up long or complex chunks of logic by defining more functions.

13. Limit classes to 20 or fewer members, including both data and function members.

14. Learn to use `const` and use it widely for variables and parameters.

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

16. **Initialize** By the end of the constructor, every class data-members should be initialized and the object should be ready to use. Initialize data members in the class declaration, if possible. Otherwise, initialize them in a constructor-initializer list. If neither is possible, use assignment statements in the body of the constructor.

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. Use camelCase for compound names, not underscores.

3. To be consistent with `Java` usage, the name of a class should start with an Upper case letter (`MyClassName`) and use camel-case after that. CamelCase has an upper case letter for the beginning of every word segment, with lower case letters between them. Variable and function names should always start with a lower case letter: `myVar, myFunction()`.

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

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

6. 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.

7. 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`).

8. Learn to use the names of the various **zero-constants** appropriately. *nullptr* is a pointer to nowhere, `'\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.

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

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 boolean 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 an object or 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

Randomly indented code is unprofessional and inappropriate for a senior or graduate student.

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

   ```
   while ( k < num_items) {
       cin >> ageIn;
       if (inRange( ageIn ) {
           age[k] = ageIn;
           k++;
       }
       else {
           cout << "An invalid age was entered, try again.");
       }
   }
   ```

   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. Some IDE's use this style by default, others can be controlled by setting preferences.

2. The other two generally approved indentation styles are shown below. Please use the style in paragraph 1 above.

Brackets aligned on the left –actually less readable.

```
while ( k < max)
{
    cin >> ageIn;
    if (inRange( ageIn )
    {
        age[k] = ageIn;
        k++;
    }
    else
    {
        cout << "Invalid age.");
    }
}
```

Brackets aligned with the indented code: –rarely used.

```
while ( k < num_items)
    {
    cin >> ageIn;
    if (inRange( ageIn )
        {
        age[k] = ageIn;
        k++;
        }
    else
        {
        cout << "Invalid age.");
        }
    }
```

3. Never let **comments** interrupt the flow of the code-indentation.
4. Break up a long line of code into parts; **do not let it wrap around** from the right side of the page to the left edge of the page on your listings. Confine your lines of code to 80 columns.
5. 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.

1.2.5 Function Definitions

1. If a function is simple and fits entirely on **one line**, write it that way, usually in the class header file. Example:

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

2. Each function definition in the .cpp file definition 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 or postconditions for the function, state them here.

```
// ------------------------------------------------------------------------
// If needed, put description of function and preconditions here.
// Document postconditions after the preconditions.
// Document return values that represent failure or success.
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 and quickly.
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. Be sure that you can see the entire function definition on one screen in your IDE. If the function code has identifiable sections or phases, there is a good chance that each phase should be broken out into a separate function. When a long 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 and other sets of related constants. The name of the constant 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 errorType { sizeOK, TooSmall, TooLarge };
```

   Declare and initialize an errorType variable like this:

```
errorType ercode = sizeOK;
```

3. When using **structured types**, use a class declaration. Please DO NOT use the `struct` declaration or the `struct` keyword this term, even when the class has no explicit function members.

1.3 Procedures for CSCI 4526 / 6626 Projects

1.3.1 Projects

All programs for this course must be done as projects.

1. Suppose your course number is CSCI 1234. Then establish a directory named cs1234 on your disk for all the work in this course.

2. Within the cs1234 directory, create a separate subdirectory for each programming assignment. Name your project directories with the assignment number and your last name: P1-Smith, P2-Smith, P3-Smith, etc. store all the files for the project in the relevant directory: source files, input files, output files, and screen shots.

3. Each programming assignment is one phase of the eventual term project. When you finish one phase, make sure the output for that phase is in the directory, and copy all the code files to the directory for the next phase. This is a very conservative way to ensure that you do not lose important files.

4. I do not use an IDE for making projects, although you may if you wish. Whether working on my own code or yours, I do all compilation from the command line.

5. To submit your work, make a temporary subdirectory called "`P1-SmithSubmit`" Copy the source code, headers, input, and output files, or whatever your instructor requires, into `P1-SmithSubmit`, zip them, and submit according to the procedures established for your course. DO NOT SEND all the files generated by your compiler. They are useless to the Professor and the TA, and they can make a project too large to go through the UNH email system.

6. Warning: the UNH mail system will reject files that are too long or look like executable files to it.

1.3.2 Using the Tools Library

All programs for this course must use the functions `banner()`, `bye()` from the tools library. Many will use `fatal()`, also, for situations where exceptions are not appropriate.

1. From the Canvas site, please download a small library called "tools". It has two files: a source code file `tools.cpp` and a header file `tools.hpp`. The ideal place to put your two tools files is at the top level of your cs4526 directory.

2. To use the tools, put `#include "tools.hpp"` in your main program and in any `.hpp` files you write yourself. Do not include "`tools.cpp`" anywhere but be sure that you compile it separately.

3. 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()*, and *fatal()*. You may also need *flush()*, *DUMPp*, *DUMPv*, *menu()*, *today()*, *oclock()*.

4. Look at the first several lines of `tools.hpp`; note that it includes many useful C and C++ library headers. When you include the tools, you do not need to include these standard header files. Please don't clutter your code with duplicate `#include`s.

5. Personalization. Before you can use the tools, you must put your own name in the `#define NAME` line on line 10 of `tools.hpp`, in place the phrase that appears there. Put your course number (4536 or 6626) on line 11

6. Start each program with a call on *banner()* (or *fbanner()* for output to a file). This will label the top of your output with your name, and the time of execution. End each program with a call on *bye()*, which prints a "Normal termination" message.

7. If you need to abort execution, call *fatal(format, ...)*. The parameters are the same as for printf. This function formats and prints and error comment, then flushes the stream buffers and aborts execution properly.

Chapter 2: Programming for Reliability

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

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 conversions. Thus, it was and still is 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 a 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. Today, the language is in widespread use for low-level work but has never been a good vehicle for large-scale application design.

Python was designed to make scripting easy, useful, and accessible. It serves that purpose admirably, at the cost of eliminating efficiency and the possibility to check for code validity prior to runtime. The language is hugely popular, but not suitable for applications in which runtime failure is unacceptable.

2.1.2 C++ Supports Modeling

The designer of C++. Bjarne. Stroustrup wanted to retain the efficiency and transparency of C, 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. Basic 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. Further, using a diciplined style and following guidelines for clear communication and presentation make a huge difference.

Functions and Methods. In C++, a function name no longer has a single unified definition. Instead, a **function** is a collection of one or more **methods**, possibly defined in multiple classes, that operate on different combinations of parameter types. This is a big improvement because the same method name defined in one class can be defined in others. The burden on a programming team to choose non-conflicting names for every function is gone, function names can be shorter, and programs are more readable

Ideally, all of the methods of a function would have the same (or a very similar) general purpose. For example, a method named `print()` would be expected to print its argument to a file or display it on the screen, and to do so without modifying the argument.

Flexibility. Work on C++ began in the early days of object-oriented programming. The importance of types and type checking was understood, but this led to a style of programming in which the type of everything had to be known at compile time. (This is sometimes called "early binding"). Languages such as Lisp did not do compile-time type checking, they deferred it until run-time (late binding).

C++ operates on a middle ground. While simple code can still be compiled efficiently, an application that relies on late binding can be fully modeled because types are not isolated from each other. Polymorphic class hierarchies can be defined with a base class and several derived subclasses. A derived class inherits data and function members from its base class. Data members no longer have unique types – they simultaneously have all the types above them in the inheritance hierarchy.

Compile-time and Run-time Dispatching. An OO language checks at compile time that the arguments to each function call have the appropriate base types for at least one of the methods of the function. If it is exactly one, that method is used to compile the call. However, for polymorphic types, more than one method may be appropriate, and a final choice of which function method to compile for the call must be deferred until run-time, when the subtype of the argument object can be checked and matched against the parameter lists of the available function methods.

Thus, an OO language uses late binding for polymorphic types and early binding for simple types. This allows a combination of flexibility (where needed) and efficiency (everywhere else) that is not achievable in a language with only late binding.

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++ can be very portable if the programmer designs it with portability in mind and follows 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 (`qsort()` 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 make a huge difference in C . 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.

2.2 Object-Oriented Principles.

The way a language is used is more important in OO programming than the language being used. C++ was designed to support OO programming[1]; 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.[2] Similarly, a non-OO program can be written in C++.

Principles central to object-oriented programming are encapsulation, grouping related data and methods together, and generic or polymorphic functions.

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 from the application data's perspective, as opposed to the traditional functional or procedural point of view. 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 function methods and data. Classes define data structures and the operations that can be performed on them. Access to all of the data and some of the method elements should only be through the defined methods of the class.

Some languages (Ruby, Python) say they are OO because they have classes. However these languages support only the surface syntax of classes, not the semantic properties: privacy, consistency and compile-time type checking. The strength of C++, the classes are both **clearly defined and stable**. They be relied on to implement a stable semantics for every instance of a class. In contrast, in languages like Ruby and Python, a class definition can vary over time.

Encapsulation. The most fundamental OO design principle is that a class should take care of itself. Typically, a class has data members: constants, variables, and internal type declarations. The class also includes constructors (for automatic initialization), destructors (for automatic memory management), and function members that operate on the data members. The OO principle of encapsulation says that *only* a function inside a class should ever change the value of a data member. This is achieved by declaring member variables to be `private`. A member function can then freely use any data member, but outside functions cannot.

Functions that are intended for internal use only are also made `private`. Functions that are part of the function's published interface are made `public`. Finally, constant data members are sometimes made public.

In C++, class relationships can be declared, and the simple guidelines for `public` and `private` visibility are complicated by the introduction of `protected` visibility and of class `friendship`.

[1] C++ is not the first or or only OO language. Earlier OO languages such as SIMULA-67 (from Dahl and Nygaard) and Smalltalk (from Alan Kay) embodied many OO concepts prior to Stroustrup's C++ definition in 1990.

[2] This was how I wrote programs before C++ was invented.

Initialization. One way a class takes care of itself is to define how class objects should be initialized. Initialization is done by *constructors*, which are like functions except that they have no return type. The name of the constructor is the same as the class name. A constructor is called automatically whenever a class object is declared or dynamically allocated. It uses its parameters to initialize the class's data members. A class often has multiple constructors, allowing it to be initialized using various combinations of arguments. A constructor might also dynamically allocate and initialize parts of the class object.

Cleanup. Cleanup, in C++, is done by *destructors*. Each class has exactly one destructor that is called when a class object is explicitly freed or when it goes out of scope at the end of the code block that declares it. The name of the destructor is a tilde (∼) followed by the class name. The primary job of a destructor is to free dynamically allocated parts of the class object.

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. Sometimes, the same symbol is used for purposes that are only distantly related to the original. For example, the + symbol is often used to concatenate strings together. (The result of `"abc" + "xyz"` is `"abcxyz"`.

Generic code enabled the system designers and every programmer to build a very simple and convenient I/O system. Any kind of primitive object can be output using <<. User-defined objects can be output this way by extending the << operator. The generic input operator is >> and can also be extended to new types by a simple definition.

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 also more 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 (like Java compilers) are slower and can give very confusing error comments because program semantics can be more complex.

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 considerably. 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 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 Differences between C and C++

- **Classes.** Structures, as in C are still available in C++, but a C++ `struct` is not the same as a C `struct`. Instead, it is almost exactly like a C++ `class`, with privacy options and methods inside the struct. The single difference between a C++ `struct` and a C++ `class` is that the default level of protection in a C++ `struct` is public, whereas the default in a C++ `class` is private. So which should you use in your programs? Simplify and be explicit. Use class consistently and add the `public` declaration if the members should be public.
- **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.

  - We *extend* a function when you define a method for that function on a new class, and the method is semantically parallel to the original definition.
  - We *override* a function in a base class when we redefine it in a derived class. This is frequently done to tailor the action of a function to each derived class.
  - We *overload* a function when we use the same operator or function name to define an action that is not substantially the same as the original action.

  In this course, please use these words appropriately.

- **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. Chapter 3 gives a comprehensive list and extensive examples of C++ input and output facilities.

- **Templates.** All of the common data structures and simple algorithms have been implemented as *template classes* in the standard library: stacks, queues, sorts, swap, max, min, etc. Using them (instead of writing one's own code) can save a programmer time and hassle. This code has been tested by many people over decades, and is, therefore, safer than anything an individual can write.

- **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 obsolete. It should not be used in new programs and eventually may not work with modern compilers. 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 `std::` in front of every call of any library function or constant.

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

- **Executable declarations.** Declarations can be mixed in with the code. Why is this useful?

  - C++ declarations can trigger file processing and dynamic memory allocation and it is VERY helpful during program construction to precede each major declaration by displaying a message that lets 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 to use for input.

  However, 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.

- **A new kind of *for* loop.** Starting with C++11, an additional kind of `for` loop is available. This loop syntax can be used to iterate through an array or any kind of data structure with multiple elements that is supported by the C++ Template Library (STL). The loop below adds 1 to each component of an array and prints the result. In this code fragment, `ar` is an array of 10 `float`s.

```
for (float f : ar) {
    f += 1.0;
    cout << f;
}
```

Each time around this loop, the name `f` gets bound to the next slot of the array. This makes it possible to both read and modify the array element, as in the above example.

- **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. For example, in C, truth values were written as 0 and 1, In C++, they are `false` and `true`.

- **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 a distinct type that is not the same as type `int` or any other enumeration. Compilers *will* generate error comments and warnings when they are used inappropriately.

- **Type conversions.** C provides "type cast" operators that convert a data object from one type to another. Casts are predefined 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.
  - In *call-by-value*, the value of the argument expression is copied into the parameter variable. Any changes made by the function to its parameter are local. They do not affect the original copy owned by the caller.
  - In *call-by-reference*, the address of the parameter value is passed to the function, and the parameter name becomes an alias for the caller's variable.

  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, while similar in concept, it is not the same as *call-by-reference*, which is fully supported by C++ in addition to all the parameter passing mechanisms in C. In *call-by-reference*, which is denoted by an ampersand (`&`), 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 using a pointer, a `*` with a reference when the variable name 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 large 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.

  Chapter 4 gives extensive explanation and examples of parameter passing mechanisms.

2.4 What is OO?

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

- A *data class* models the state and behavior of a real-world object.
- A *controller class* uses inputs to control a process.
- A *viewer* class presents the state of a computation in a way that can be understood by the user.
- A *business logic* class implements the policies of an organization.

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

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

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

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

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

What class relationships are supported by C++? A program is built from interacting classes.

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

  Instances of class B have *two* types, A and B, and can be used in any context where either type A or type B is accepted.
- An *base class* represents the common parts of more than one subclass.
- The base class is `polymorphic` if it defines at least one virtual function. If all of its functions are non-virtual, it is not polymorphic even if it has two or more derived classes. Polymorphicclasses will be explained fully in a future chapter.
- An *abstract class* is a polymorphic class that characterizes some functionality without implementing it. All implementation is left for the derived classes.
- A class A can give *friendship* to B. This means that A permits B to manipulate its private parts. Friend classes will be explained fully in a future chapter. At this time, it is enough to know that they are used to build data structures (such as linked lists) that require two or more mutually-dependant class declarations.
- A object of class A can sometimes be *cast* to an object of class B.

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

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

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

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

2.5 Header Files and Code Files

Most C++ classes have both a header `.hpp` file and an implementation `.cpp` file. Very simple classes are sometimes defined entirely in the header file.

The purpose of a header file is to provide information about the class interface for inclusion by other modules that use the class. A basic C++ header file has several parts:

- It `#`includes all headers needed for any declaration or function used by the class.
- It may contain `#`define and/or `typedef` statements.
- It may contain other type declarations, especially enumerated types.
- The header file contains the class declaration, which starts with the word `class` and the name of the class. Following that are declarations for the members of the class, both data declarations and method prototypes, enclosed in braces.
- The actual code for the class methods might or might not be inside the class definition. Short method definitions are there, but longer ones are defined separately, usually in an implementation file (`.cpp` , but possibly after the end of the class declaration, using the keyword `inline`.
- The header file must contain the full definition of all inline functions related to the class. Functions that are fully defined *inside* the class are automatically inline. Those `outside` the class declaration must start with the keyword `inline`.
- The header file must *not* contain data declarations *outside* the class declaration except those initialized by a `constexpr` expression.
- Executable code, other than definitions of inline methods, does *not* belong in a `.hpp` file.

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

An *inline* function call is replaced prior to compilation, by the full definition of the function. This always saves execution time. It also saves memory space at run time if the function is very short (one line or two), or if it is called only once.

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

2.6 Class Basics

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

- Members of a `struct` default to public (unprotected) visibility, while `class` members default to private (fully protected).
- Privacy allows a class to encapsulate or “hide” members. This ability is the foundation of the power of object-oriented languages.
- In both a `struct` and a `class`, the default visibility can be overridden by explicitly declaring `public`, `protected`, or `private`.

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

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

```
//-------------------------------------------------------------------
// Documentation for author, date, nature and purpose of class.
//-------------------------------------------------------------------
 #pragma once
 #include "tools.hpp"

  class Mine {
  friend class and friend function declarations, if any;
  private:  // ----------------------------------------------------
      Put all data members here, following the format:
      TypeName  variableName = initializer;          // Comment on purpose of 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) : ctor initializer list {                    // constructor
       initialization actions
    }
    ~Mine() {   ... }                          // destructor
    ostream& print ( ostream& s );             // print function, defined in .cpp file
    Put other interface function prototypes and definitions here.
};
inline ostream& operator<<(ostream& st, Mine& m){ return m.print(st); }

#endif
```

Figure 2.1: The anatomy of a class declaration.

The Form of a Class Declaration Figure 4.1 illustrates the general form of a header (`.hpp`) file and the parts of a typical class declaration.

2.6.1 Data members.

A class normally has two or more data members that represent parts of some real-world object. An individual data member is used by writing an object name followed by a dot and the member name. Data members are normally declared to be private members because privacy protects them from being corrupted, accidentally, by other parts of the program.

- Please declare data members at the top of the class. Although they may be declared anywhere within the class, your program is much easier for me to read if you declare data members before declaring the methods that use them.
- Taken together, the data members define the *state* of the object.
- Read only-access to a private data member can be provided by a "get" method that returns a copy of the member's value. For example, we can provide read-only access to the data member named `name`, by writing this public method: `const string getName(){ return name; }` In this method, the `const` is needed only when the return type is a pointer or structured object; for simple numbers it can be omitted.

- In managing objects, it is important to maintain a consistent and meaningful state at all times. To achieve this, All forms of assignment to a class object or to any of its parts should be controlled and validated by methods that are class members. As much as possible, we want to avoid letting a function outside the class assign values to individual class members one at a time. For this reason, you must avoid defining "set" methods in your classes.
- Some OO books illustrate a coding stysle in which each private data member has a corresponding public "set" method to allow any part of the program to assign a new value to the data member at any time. Do not imitate this style. Public "set" methods are rarely needed and should not normally be defined. Instead, the external function should pass a set of related data values into the class and permit member methods to validate them and store them in its own data members if they make sense.

The Form of a Class Implementation Figure 2.2 shows the skeleton of the corresponding `.cpp` file. It supplies full definitions for member methods that were only prototyped within the class declaration (.hpp). Note that every implementation file starts with an `#include` for the corresponding header file.

2.6.2 Operators, Functions, and Methods

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);
```

```
// 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 2.2: Implementation of the class "Mine".

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

Member methods, related methods, and non-member methods. Methods can be defined globally or as class members[4]. 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 the standard sort() function and extensions to the C++ output operator[5]. The `operator >>` extensions form a special case. Each new method is directly associated with a class, but cannot be inside the class because standard `operator >>` is a global function.

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

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

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

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

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

Static member functions. An object or pointer to an object is needed to call an ordinary member function. However, sometimes it is useful to be able to call a class function before the first object of that class is created. This can be done by making the method `static`. A static method can only use static data[6], that is, data that was allocated and initialized at program-load time. To call a static member function, we use the class name:

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

Global functions. Calls on non-member functions are exactly the same in C and in C++.

2.6.3 Typical Class Methods

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

[3]Polymorphic functions are dispatched at run time.

[4]Almost all C++ methods are member functions.

[5]These cases will be explained later.

[6]A static data member is called a *class variable* because there is only one copy of it per class. All instances of the class share that same static variable.

- The name of a constructor is the same as the name of the class. A constructor method initializes a newly created class object. More than one constructor can be defined for a class as long as each has a different *signature*[7].
- The name of the destructor is a tilde followed by the name of the class (∼`Point`). There are no parameters, and a class can have only one destructor. A destructor method is used to the free dynamic storage occupied by a class object when that object dies. Any parts of an 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. If an object is created by `new`, the destructor must be invoked explicitly by calling `delete`.
- A `print()` method defines the class object's image, that is, how should look on the screen or on paper. Print functions normally have a stream parameter so that the image can be sent to the screen, a file, etc. Name this function simply "print()", with no word added to say the type of the thing it is printing, and define a method for it in every class you write. In your print() method, format your class object so that it will be readable and look good in lists of objects

2.7 Example: Generic Insertion Sort

The purpose of this section is to illustrate interactive input and output in the context of a simple one-class program.. The code 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:

Objectives of Insertion Sort. This is a C++ implementation of insertion sort. The teaching objectives are to illustrate:

- Modular code with a streamlined main function.
- A class with all the normal parts: data, constructor, destructor, and public functions. The MemList class 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.
- Encapsulation: this class takes care of itself. The functions that belong to the MemList class are the only ones that operate on the class's data members.
- The insertion sort algorithm: useful for lists up to about 30 items

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.

2.7.1 Main Program

```
1  //  ------------------------------------------------------------------------
2  //  Main program for the Membership List.                            main.cpp
3  //  Revised June 8, 2021.
4  //  ------------------------------------------------------------------------
5  #include "tools.hpp"
6  #include "memlist.hpp"
7
8  // ------------------------------------------------------------------------
9  //  Read a series of names from the keyboard, store, sort, and display them.
10 int main( void )
11 {   int nMembers;
12     banner();
```

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

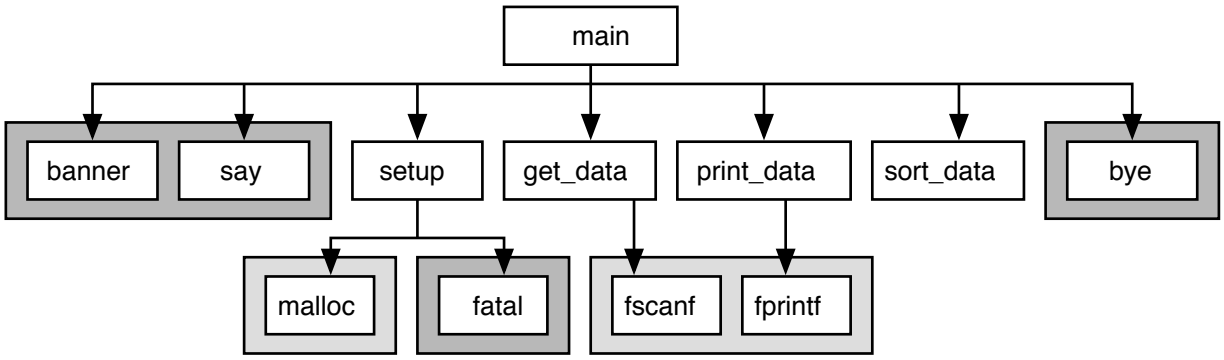

Figure 2.3: The stream type hierarchy.

```
13
14      MemList theData;
15      cout << "Constructed empty MemList" ;
16      cout << "\nEnter names of your club members, period to finish. \n" ;
17      theData.readData( cin );
18
19      cout <<"\n----------------- Beginning to sort.\n";
20      theData.sortData();
21
22      cout <<"\n----------------- Sorted results:\n";
23      theData.printData( cout );
24      bye();
25      return 0;
26  }
```

- A program that is built in modules has a main program plus 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 MemList module and the tools module, so we include the headers for those modules.
- The main program instantiates the MemList object, named "theData".
- The rest of main() (lines 15...26) 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.

2.7.2 The MemData Class Declaration

- Type declarations, #defines, and prototypes are normally found in header files.
- Line 31 includes definitions from the local library, tools, that we will be using throughout the term. Note the use of quotes, not angle brackets for a local library. Note also that we include the header file, not the code file, because we are doing a multi-module compile with a separate link operation.
- Line 32 includes header for the standard C++ string library. We use angle brackets for standard library headers, and write only the library name, omitting the .h or .hpp.
- This header is included here only for emphasis. We do not need to include it because we are including the tools, which includes `<string>`.
- Lines 34–48 define the class MemList, which has three data members, a constructor, a destructor, and three functions.

- The MemList type makes a coherent grouping of all the parts that are needed to manage an array of data. In C++. The data members are declared to be private. The member functions provide access to the data structure and they are the only way to access it.
- Recent versions of C++ permit simple constant initializations of the data members of a class. Here we initialize n, the actual number of names currently stored in the array, and max, the maximum number of names that can be stored there.
- The constructor is an inline function, fully defined in this header file. There is nothing for it to do because all data members are initialized by lines 37–39, so it is defined with the default constructor, which does nothing.
- Likewise, there is nothing for the destructor to do because dynamic allocation is not used in the class. It is defined to be the default destructor, which does nothing.
- The three functions are longer than 2 lines, so they are defined in the .cpp file and only prototyped here.

2.7.3 The MemData Class Definition

- The code file for a module must include its corresponding header file. Normally, it should not `#include` any other header files.
- The readData() function (lines 56. . . 66) reads whitespace-delimited names from the stream and uses them to fill he array with data. We call this from line 17 of main, with cin as the parameter. Therefore, we will be reading input from the keyboard. Look at the output at the end of this section and note that it does not matter whether the user types a space or a newline – they are treated the same way by `>>`.
- The last action in readData() is to store the amount of data received into the MemList's variable. This ensures that any function that processes the data can find out how much data is there. At no time is reference made to global variables or constants; the MemList is self sufficient.
- The printData() function (lines 70 – 75) prints an output header followed by the names in the list, one per line. We call it from line 23 of main, with cout as a parameter. Therefore we will display the output to the screen.

Sorting by insertion. The sortData() function (lines 81. . . 97) implements insertion sort, the most efficient of the simple sorts. In practice, it is reasonable to use insertion sort for small arrays. This is a typical double-loop implementation of the insertion sort algorithm.

- 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.
- 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 87. . . 96) starts with the element at store[1].
- 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 89) from the beginning of the unsorted portion of the array, leaving a hole that does not contain significant data.
- The inner loop (lines 92. . . 96) searches the sorted portion of the array for the correct slot for the newcomer. The for loop is used to prevent overrunning the boundaries of the array. Within the loop body, an if...break, line 92, is used to leave the loop when the correct insertion slot is located.

- To find that position, the loop scans backwards through the array, comparing the newcomer to each element in turn (line 92). If the newcomer is smaller, the other element is moved into the hole (line 93) and the hole index is decremented by the loop (line 90).
- If the newcomer is not smaller, we break out of the inner loop and copy the newcomer into the hole (line 95).

```
50  // ------------------------------------------------------------------------
51  //  Code file for the MemList program.                            memlist.cpp
52  //  Created by Alice Fischer on Aug 23 2014; revised June 8, 2021.
53  #include "memlist.hpp"
54
55  //-------------------------------------------------------------------------
56  // Use sequential access pattern to read mems data from the keyboard.
57  void
58  MemList::readData( istream& infile ){
59      int k=0;  // Declare outside loop because needed after loop.
60      for( ; k<max; ++k ) {
61          infile >> mems[k];
62          if( mems[k][0]=='.' ) break;       // Leave loop if input was a period.
63          cout <<k <<". " <<mems[k] <<endl;  // Debugging output.
64      }
65      n = k;                                 // Save actual # of items read.
66  }
67
68  //-------------------------------------------------------------------------
69  // Print array values, one per line, to the selected stream.
70  void
71  MemList::printData( ostream& outstream ){
72      outstream << n <<" names were entered.\n";
73      for( int k=0; k < n; ++k)
74          outstream << mems[k] <<endl;
75  }
76
77  // ------------------------------------------------------------------------
78  // Generic insertion sort using a MemList.
79  // Sort n values starting at pData->mems by an insertion sort algorithm.
80
81  void
82  MemList::sortData(){
83      int pass;           // Subscript of first unsorted item; begin pass here.
84      int hole;           // Array slot containing no data.
85      string newcomer;    // Data value formerly in hole, now being inserted.
86
87      for ( pass = 1; pass<n; ++pass ) {
88          // Pick up next item and insert into sorted portion of array.
89          newcomer = mems[pass];
90          for ( hole=pass; hole>0; --hole ) {
91              int moveit = hole-1;          // Subscript of slot next to the hole.
92              if( newcomer.compare(mems[moveit]) > 0 ) break; // Right slot found.
93              mems[hole] = mems[moveit];    // Move item up one slot.
94          }
95          mems[hole] = newcomer;
96      }
97  }
```

The output.

```
----------------------------------------------------------------
A. Fischer
CSCI 4526 / 6626
Sun Jun 12 2022 15:21:41
```

```
------------------------------------------------------------
Constructed empty MemList
Enter names of your club members, period to finish.
Andy
0. Andy
Carla Wendy Bill Fran
1. Carla
2. Wendy
3. Bill
4. Fran
.

----------------- Beginning to sort.

----------------- Sorted results:
5 names were entered.
Andy
Bill
Carla
Fran
Wendy

Normal termination.
```

Chapter 3: C++ I/O

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.

This chapter assumes basic knowledge of the C++ input and output. It attempts to help the reader develop more advanced I/O understanding and skills. 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 can be a pain in C++. The commands to do these jobs are referred to as "manipulators" in C++ parlance and are defined in the `<iomanip>` library.

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.

3.1 Streams and Files

A *file* is something that is stored on a memory device. A stream is a data structure within a program that lets us read or write data to a file. The stream data structure contains storage for a block of data and the error flags, cursors, etc. needed to access that data.

Throughout, we will deal exclusively with text files. Please remember that a text file is a sequence of ASCII or Unicode characters – numbers are represented by strings of digits. Basic input for all predefined types is the same and is very easy.

Stream classes are built into C++. They form a class hierarchy whose root is the class `ios`. This class defines flags and functions that are common to all stream classes. To the right of `ios` 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, character-based streams–data is either read or written in strict sequential order. The `iostreams` are derived from both istream and ostream; they permit random-access input and output, such as a database would require. Further to the right are the main stream classes for file-based streams and strings that emulate streams.

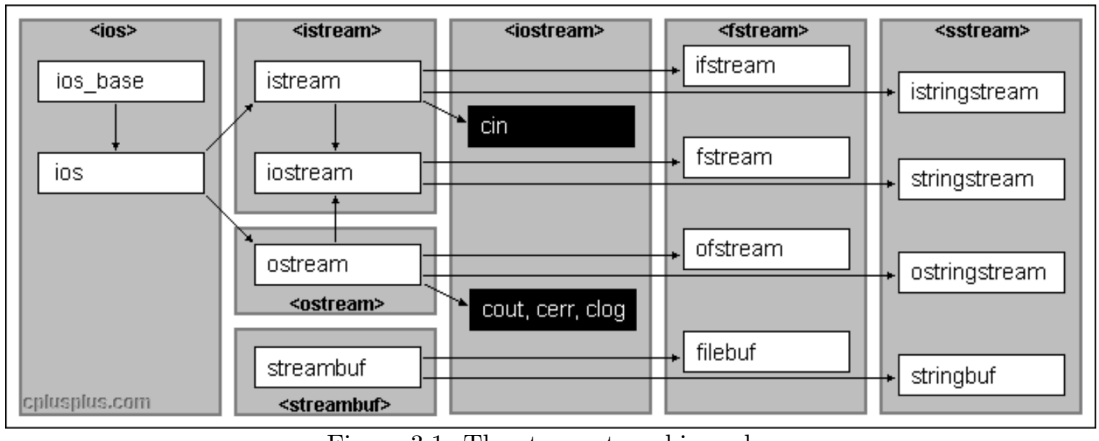

Figure 3.1: The stream type hierarchy.

Each class in the stream hierarchy is a variety of the class to its left. When you call a function that is defined for a class toward the left side, you can use an argument of any class that is connected to it on its right. For example, if `fin` is an `ifstream` (used for an input file) you can call any function defined for class `istream` or class `ios` with `fin` as an argument.

The functions and constants for all these classes you will need this term are defined in four header files:

- `<iomanip>` for format control.
- `<iostream>` for interactive I/O.
- `<fstream>` for file I/O.
- `<sstream>` for strings that emulate streams, used to parse a line of input after reading it.

The remainder of this chapter will cover many details of the C++ I/O system in approximately the same order as the columns in the diagram.

3.1.1 The base class for streams: ios.

Manipulators. `#include <iomanip>;` Or include tools.hpp. A *manipulator* in C++ is a function that affects the internal settings of an stream or an output stream. It modifies the stream and returns that same stream, so that it can be used in a chain of input or output operators. Manipulators are designed to be used in with the I/O operators `>> and <<` on streams. For example: `cout << boolalpha;` sets a flag in the standard output stream to output true and false values as text instead of as integers.

Many manipulators are supported, others could be defined, if desired:

Number-base flags:

- `dec`: Use base-10 number conversion.
- `hex`: Use hexadecimal (base 16) number conversion.
- `oct`: Use octal number conversion

Input manipulators:

- `skipws`: Skip leading whitespace when reading text.
- `noskipws`: Do not skip leading whitespace

Output manipulators The last three require inclusion of `<iomanip>`.

- `endl`: Insert newline and flush the buffer.
- `ends`: Insert a null character.
- `flush`: Flush the stream buffer.
- `boolalpha`: Set the stream to output true and false values as text instead of as integers.
- `noboolalpha`: Set the stream to output true and false values as integers 1 and 0.
- `setfill('.')`: Set the fill character to be a period
- `setprecision(n)`: Output numbers with n places of decimal precision
- `setw(n)`: Set the width of the next number to n.

Floating-point format flags These manipulators set the floatfield flag in the stream:

- `fixed`: Use fixed-point number conversion.
- `scientific`: Use scientific notation for number conversion.

Output alignment flags These manipulators set the adjustfield flag in the stream:

- `left`: Left-align the output.
- `right`: Right-align the output.
- `internal`: Adjust width of field by inserting characters at an internal position.

Manipulators are functions. The C++ standard defines many useful manipulators but misses some that are useful, such as the two that are shown below. The language standard provides manipulators to change numeric output from the default format (like `%g` in C) to fixed point `%f` or scientific `%e` notation. However it does not provide a manipulator to change back to the default `%g` format. This function does the job:

The tools library contains a definition of `general` as a manipulator for `ostreams` This makes it more intuitive for the programmer to format real numbers.

```
// ------------------------------------------------------------------------
ostream& general( ostream& os ){ // Use:  cout <<fixed <<x <<general <<y;
    os.unsetf( ios::floatfield );
    return os;
}
```

The tools also define `flush` as a manipulator for `istreams`, allowing the programmer to empty the keyboard buffer after an input error and leave the buffer in a predictable state: empty.

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

3.1.2 Class iostream

Four instances of this class are defined and opened each time a new process is started up, and before control is transferred to the main() function of the new process.

- `cin`: This is the standard input stream. It is connected to the computer's keyboard.
- `cout`: This is the standard output stream. It is connected to the computer's video screen.
- `cerr`: This stream is used by the system for run-time errors. It, also, is connected to the computer's video screen. Thus, a user may see output from cout and cerr mixed together. You may wish to use the `cerr` stream for debugging output.
- `clog`: The clog stream is used by processes that log transactions. It will not be used in this course.

Buffered and unbuffered streams. When using `cout` or an `ofstream`, information goes into the stream buffer when an output statement is executed. It does not go out to the screen or the file until the buffer is *flushed*. In contrast, something written to `cerr` is not buffered and goes immediately to the screen.

When using `cin`, a line of text stays in the keyboard buffer until the return key is pressed. This allow the user to backspace and correct errors. When return is pressed, the entire line goes from the keyboard buffer to the stream buffer, where it stays until called for by the program. The program can read the entire line or any part of it. The rest will remain in the stream buffer, available for future read operations. In general, newlines in an input file are treated the same as spaces.

Interactive vs. file-based streams. The interactive streams (defined in the class iostream) are pre-declared and pre-opened for every process. Using a file-based stream requires additional code:

- `#include <fstream>;`
- Open the stream.
- Test for successful opening.
- During processing, the program must check for end-of-file in an ifstream.
- Eventually, the file must be closed, either by the program or by the system when the program terminates.

Opening an fstream. There are two ways to open a stream. The preferred way is to use the ifstream or ofstream constructor with a file name parameter. The other way is to declare an instance of a stream, then open it in a separate statement.

```
ifstream fin ( "myin.txt" );     // Declare stream and open for reading.
ofstream fout ( "myout.txt" );   // Declare stream and open for writing.
fin.open( "myfile.in" );         // Alternate way to open a stream.
```

In either case, one *must* test whether the stream is actually open before using it"

```
if (!fin.isopen()) fatal(...);   // Test for unsuccessful open.
```

If you attempt to open a file that does not exist, the operation fails and returns a null pointer. There are several reasons (spelling, pathnames, permissions) why an opening command might fail. A very common beginner's error is to try to use the file without checking. Forgetting to check for opening failure results in confusing error comments and many bewildered students.

Flushing buffered output streams. The output from a buffered stream stays in the stream buffer until it is flushed, which happens under the following circumstances:

- When the buffer is full.
- When the stream is closed.
- When the program outputs an `endl` on any buffered output stream. `endl` is a manipulator that ends the output line, but its semantics are subtly different from a newline character.
- When the program calls `flush`.

Flushing input streams. The built-in `flush` and `fflush` apply only to output streams. Some students are convinced that that `flush` also works on input stems. It does not, but they are confused because of the stream ties. However, after detecting an error in an interactive input stream, it is usually a good idea to flush away whatever remains in the input stream buffer.

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 also specifying append mode, "app". In this case, one must 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 );
```

Stream ties. The stream `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.

Closing an fstream. An fstream may be closed in more than one way:

- Call the `close` function:`fin.close();`
- The fstream's destructor will be run when the fstream goes out of scope.
- When execution ends normally, the fstream will be closed.

Closing an output stream by any of these means will automatically flush its buffer. If the program ends with a call on `exit()`, however, the buffer might not be flushed,

When should you close streams explicitly?

- Always, if you want to develop good, clean, habits that will never mislead you.
- Always, when you are done using a stream well before the end of execution.
- Always, when you are using Valgrind or a similar tool to help you debug.
- In general, it is considered good practice to close a stream as soon as the program is done with it.

Detecting end-of-file. In both C and C++, the end-of-file flag does not become true *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 if there is are one or more newline characters on the end of that line. 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. In general, the test should come immediately `after` the line that reads the input data.

```
if (fin.eof()) break;            // Test for end of file.
```

Error detection and recovery. In a production program, error detection and recovery is often essential. However, this is a large and detailed subject that is usually omitted from classroom instruction for beginners and intermediate students. As a programmer's skill advances, however, these things must be mastered. These are the error detection functions:

```
if (fin.good()) ...  // Test for no errors after a read operation.
if (fin.fail()) ...  // Test for hardware or number conversion error.
if (fin.bad()) ...   // Test for hardware error.
fin.clear();         // Clear the stream's error flags to permit processing to continue.
fin.ignore(1);       // Remove the offending character from the stream.
```

Error recovery is covered in detail, with a full example, at the end of this chapter.

3.2 Input streams: istream and ifstream.

3.2.1 Simple types.

On simple types like numbers, chars, and pointers, input is done in one of two ways:

- Use the *extraction* operator, `>>`. It parses the input stream, extracts characters from it, and does any appropriate number conversion (ASCII to numeric). The result is then stored in a variable.
- You can call `getline()` to read an entire line of input into a char array or a string. If using an array, you must worry about buffer overflow. Then you must parse it yourself, and do the number conversion yourself by calling functions like `atoi` and `strtol`. Input *can* be done this way, however, DO NOT DO IT. It is not necessary and not helpful to reinvent the wheel. Learn to use the `>>` operator as it was intended!

How much was read? After any read operation, the function `gcount()` contains the number of characters actually read and removed from the stream. Saving this information is sometimes useful. The value returned by `gcount()` or `get()` is the number of characters that have been removed from the stream.

Operator >>. The extraction operator is polymorphic. It is predefined for all the built-in types and can be extended to handle any program-defined class. Basic input is quite simple when using it. It skips leading whitespace characters, reads the digits of a number, and stops at the first whitespace or non-numeric input character. 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!

Several operations that use `>>` can be chained (combined in one statement) but those based on `get()` and `getline()` (described below) cannot. A single call on a `get()` or `getline()` function is a complete statement.

Input errors. Operator `>>` works properly if the input is correct. However, if a number is expected, and anything other than a number is in the input stream the stream will signal an error. A bullet-proof program tests the stream status to detect such errors using the strategies explained in Section 3.5. The `clear()` function is used to recover from such errors.

Single character input. When reading a single character, a programmer might with to read only visible inputs, or might need to read whatever character is next, even if it is a whitespace character. In C++, two different functions exist for the two different tasks.

- Use `>>` to read the next non-whitespace char. (It skips leading whitespace.)
- Use `get()` to read the next character. This is useful when a program needs to know exactly what characters are in the input stream, and does not want whitespace skipped or modified. Example: a program that compresses a file.

```
char ch;
ch = cin.get();
get (ch);
```

3.2.2 String input.

The simplest form for a string input command should NEVER be used: `cin >> w1;` . It skips leading whitespace, reads characters starting with the first non-whitespace character, and stops reading at next whitespace. The input characters are stored in `w1`. However, there is no limit on the length of the string that is read. **DO NOT DO THIS!** If you try to read an indefinite-length input into a finite array, and you do not specify a length limit, the input can overflow the boundaries of the array and overlay nearby variables. A program that makes this error is open to exploitation by hackers.

The simplest method of string input is called "string getline" and is used to input characters into variable of type string. Please use these data declarations:

```
ifstream fileIn( "demo.txt")
string input1;
char arr[80];
```

The string input functions, with examples are:

- `istream& getline (istream& is, string& str);`
  - Read an entire line, up to the newline character, from an input stream into a dynamically allocated string of an appropriate length. Remove the newline character from the stream.
  - `getline (cin, str);`
  - `getline (fileIn, input1);`
- `istream& getline (istream& is, string& str, char delim);`
  - Read a line up to the specified delimiting character, from an input stream into a dynamically allocated string of an appropriate length. Remove the delimiter from the stream.
  - `getline (cin, str, char delim);`
  - `getline (fileIn, input1, char ':');`

istream::getline() Do not confuse the string-getline functions with the older C++ istream::getline() functions that read data into a character array. The istream::getline() functions are only safe when used with a length limit. Prototypes and examples are given below.

- `istream& istream::getline (char* arr, arraysize n );`
  - Read an entire line up to the newline character, from the given input stream into the array named `arr`. The second parameter is the length of `arr`. Remove the newline character from the stream.
  - `cin.getline( arr, 80 );`
  - `fileIn.getline( arr, 80, ';' );`
- `istream& istream::getline (char* arr, arraysize n, char delim );`
  - Read a line, up to the specified delimiting character, from the given input stream into the array named `arr`. Remove the delimiter from the stream.
  - `getline( in, str );`
  - `getline(fileIn, input1);`
- `istream& istream::get (char* arr, arraysize n );`
  `istream& istream::get (char* arr, arraysize n, char delim);`
  - Read a line, up to the newline or delimiting character, from the given input stream intothe array arr. Stop after n-1 chars have been read. DO NOT remove the delimiting character from the stream. 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)`.

What should you use? The string-getline() functions are safer and easier to use, and can do almost everything that the istream::getline() functions do. Unless you are doing something unusual, you should use the newer, safer, easier versions. If you truly need the string in an array, not a string variable, extract the contents when the read operation is finished: arr = str.data();

Whitespace Problems. When you are using `>>`, leading whitespace is automatically skipped. However, before reading anything with `get()` or `getline()`, whitespace must be explicitly skipped unless you *want* the whitespace in the input. Use the `ws` manipulator 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.

3.3 Output streams: ostream and ofstream

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. For us, this means that you should use C++ version consistently and avoid `printf()`.

Simple types, pointers, and strings. The table below 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.

However, for file-streams there is one difference: the manipulator flushes the output stream; the character and string do not.

- For output, the *insertion* operator, `<<` inserts values into an output stream.

Type	Function call.
Numeric or char:	`cout <<"Initial:" <<c1 <<" age:" <<k1 <<'\n';`
`char[]` or `char*`	`cout <<word <<"..." <<w2 <<" " <<w3 <<"\n";`
pointer in hexadecimal	`cout <<px1 <<endl;`
integer in hexadecimal	`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 writing the manipulator `hex` or `oct` in the output chain. Once the stream is put into hex or `oct` mode it stays there until changed back by the `dec` manipulator.

Field width, fill, and justification. This table shows how to use the formatting functions for justification, field width, and fill character. Note: you must specify field width in C++ separately for *every* field. However, the justification setting and the fill character stay set until changed.

Style	How To Do It
12 columns, default justification (right).	`cout <<setw(12) <<k1 <<" " <<k2;`
k1 in 12 cols (. fill) then k2 in default width	`cout <<setw(12) <<setfill('.')  <<k1 <<k2 <<endl;`
12 columns, left justified, twice.	`cout <<left <<setw(12) <<k1 <<setw(12) <<k2 <<endl;`
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)	`cout <<y1 <<' ' <<y2 <<endl;`
Change to precision=4	`cout << setprecision(4) <<y1 <<' ' <<y2 <<endl;`
Fixed point, no decimal places	`cout <<fixed <<setprecision(0) <<y1 <<endl;`
Scientific notation, default precision	`cout <<scientific <<y1 <<endl;`
Scientific, 4 significant digits	`cout <<scientific << setprecision(4) <<y1 <<endl;`
Return to default `%g` format	`cout <<general;` // Defined in tools.hpp.

3.4 String streams: sstream.

A `stringstream` (string stream) is a stream whose data is stored as a string in main memory. A string-stream has all the flags, settings, and functions of a normal stream, so when a string-stream is used, all the standard input functions are available.

An `ostringstream` (output string stream) is like a Java StringBuilder. The purpose is to allow a program to build up a line of output, one part at a time, then output it as a single unit. While this is not usually necessary, it is useful in threaded applications to keep the output from each thread coherent and separate from other threads' output.

An `istringstream` (input string stream) allows a program to read a line of data, then parse it using istream input operations. If an error occurs during the parsing, the program can write the whole line to `cerr` or `clog`, providing excellent information for a human who needs to clean up the damaged data.

3.5 Example 1: Files and Error Handling

The focus of this section is basic C++ concepts and tools for input file handling. Topics include using the command line to select an input file, opening a file, end of file detection and input error detection. The code examples do nothing useful; they simply provide examples of using input streams containing text and numbers in a variety of ways. Each example presents a different technique.

3.5.1 Using the command line and string getline().

Command-line arguments allow the programmer to specify runtime parameters, such as a file name, at the time when the command is given to run a program. This is useful when you do not know the name of the input file at compile time. It is an alternative to querying the user for that name.
There are two ways to use file names as command line arguments:

1. The normal way: 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 following brief program illustrates how to use these command line arguments and the simplest way to read file input (so long as you only input lines of text, not numbers).

```
1   // C++ demo program for command-line arguments, an input file, eof, and errors.
2   // A. Fischer, June 2022                                file: IOdemo.cpp
3   #include "tools.hpp"
4
5   int main( int argc, const char* argv[] )
6   {
7       string buf;                              // To hold the lines of input.
8       banner();
9       // Command line must give name of the program followed by 2 file names.
10      if (argc < 2)  fatal( "Usage: eofDemo filename" );
11
12      ifstream instr( argv[1] );  // argv[0] is the name of the program.
13      if (! instr)  fatal( "Cannot open text file %s", argv[1] );
14      cout <<"Stream flags are printed after each read in this order:\n"
15      <<"state = goodbit : failbit : eofbit : badbit \n";
16
17      for(;;){
18          getline( instr, buf );
19          cout <<instr.rdstate() <<" = " <<instr.good() <<":"
20          <<instr.fail() <<":" <<instr.eof() <<":" <<instr.bad()<<": "
21          << buf << endl;
22          if (instr.eof()) break;
23      }
24      // Control comes here at end of file.
25      cout << "--------------------------------------------------\n";
26      instr.close();            // Done reading file; close it.
27      bye();
28  }
```

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

- Line 6 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, or *www.cprogramming.com/tutorial/c/lesson14.html* "Accepting command line arguments in C using argc and argv".
- Line 11 says that the user should write the program's name and one file name on the command line.

- Line 13 picks up the file-name argument and uses it to open an input stream. 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 than working for a while and *then* discovering that the program cannot proceed because of a file-opening error.
- Line 14 tests that the stream was actually opened properly. If not, execution is aborted.
- Line 15. After each line in the file is read, the stream flags will be printed out. This statement provides headings to help you understand the output.
- Line 19 calls a string-getline to read the contents of the file.
- Lines 20–22 print the stream status, flags, and the actual input.
- Line 23 would normally come immediately after the line 19, but here we print out the stream flags before testing them and, potentially, leaving the loop.
- Line 27 closes the file. Closing a stream is considered good style. However, all files will be closed automatically when the program ends.

Reading the stream-status reports.

The file eofDemo.in will be used in this example. It is a normal text file with three normal lines, each ending in a newline character.

The three flags, defined in the `ios` class, are used to record the status of a stream. Their names are: `eofbit`, `failbit`, and `badbit`. They are turned on during a read operation when some exceptional condition is found. They remain set until explicitly cleared.

- Lines 19–24 are the input loop. Line 19 calls `getline()`; the lines after that print the results of the read operation in this order: <stream state>=good : failbit : eofbit : badbit : <Actual input string>
- The value returned by `rdstate()` is a number between 0 and 7, formed by the concatenation of the status bits: fail/eof/bad.
- The `fail()` function returns a true result (which is printed as 1) if no good data was processed on the read operation.
- The eof() function returns true when an end-or-file condition has occurred. There may or may not be good data, also.
- The `bad()` function returns true if a fatal low-level IO error occurred.
- There is no "good" bit. The good() function returns true (which is printed as 1) when none of the three exception flags are turned on. As shown in the output, `eof()` can be true when good data was read, and also when *no* good data was read. When eof happens and there is *no* good data, the `fail()` is true).
- The results after reading the file eofDemo.in are:

```
-----------------------------------------------------------
Stream flags are printed after each read in this order:
state = goodbit : failbit : eofbit : badbit
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
0 = 1:0:0:0: Third line, with a newline on the end.
6 = 0:1:1:0:
-----------------------------------------------
```

  This means that three lines were read (including the newlines on the end) and there were no read errors of any sort. The 4th read was unsuccessful because the third newline was the last char in the file. So the fail flag and end-of-file flag were turned on.

- In contrast, here are the results after reading the file eofDemo2.in; it is an almost identical file, but ends without a newline:

```
-------------------------------------------------------------
Stream flags are printed after each read in this order:
state = goodbit : failbit : eofbit : badbit
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:0:1:0: Third line, without a newline on the end.
-------------------------------------------------
```

Aw see that two lines were read without errors, but the third line turned on the `eofbit` because it hit an end of file before completing the read. When using getline(), the read operation is not complete until a newline character is found.

- The moral? It is normal to *put a newline on the end of every line.* If it is missing, errors can happen.

3.6 Example 2: 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.

It is important 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. The program in this section shows how to detect read errors and handle end-of-file and other stream exception conditions. After detecting an input error, the programmer should clear the bad character or characters out of the stream and reset the stream's error flags.

End-of-file handling interacts with error handling, with the way the line is read: get() or getline() or a numeric read, and with the way the data file ends (with or without a newline character). The following example explores numeric input and errors. The code is given first, then the input file with corresponding output.

```
1  //-----------------------------------------------------------------------------
2  // C++ demo program for error handling while reading numbers.
3  // A. Fischer, June 2022                                  file: IONumeric.cpp
4  #include "tools.hpp"
5  void use_nums( istream& instr );
6
7  int main( int argc, char* argv[] )
8  {
9      banner();
10     // Command line must give name of the program followed by a file name.
11     if (argc < 2)  fatal( "Usage: eofDemo numberfile" );
12
13     ifstream num_stream( argv[1] );
14     if (! num_stream)  fatal( "Cannot open numeric file %s", argv[2] );
15
16     cout <<"\nIOstream flags are printed after each read, in order:\n"
17          <<"state= good() : failbit : eofbit: badbit : $<$actual input$>$ \n";
18
19     use_nums( num_stream );
20     // use_numhex( num_stream );
21     bye();
22 }
23
24 //-----------------------------------------------------------------------------
25 void use_nums( istream& instr ) {
26     cout <<"\nReading numbers.\n";
27     int number;
28     for(;;) {
29         instr >> number;
30         cout <<instr.rdstate()<<"= " <<instr.good() <<":" <<instr.eof()<<":"<<instr.fail()<<":"<<instr.bad
```

```
31          if (instr.good()) cout << number << endl;
32          else if (instr.eof() ) break;
33          else if (instr.fail()) {        // Without these three lines
34              instr.clear();              // an alphabetic character in the input
35              instr.ignore(1);            // stream causes an infinite loop.
36          }
37          else if (instr.bad())           // Abort after an unrecoverable stream error.
38              fatal( "Low-level error while reading input stream." );
39      }
40      cout << "-----------------------------------\n" ;
41  }
```

The file eofNumeric.in.

```
1
278
45abc8
7
```

Output from reading eofNumeric.in.

```
IOstream flags are printed after each read, in order:
state= good() : failbit : eofbit: badbit : $<$actual input$>$

Reading numbers.
0= 1:0:0:0: 1
0= 1:0:0:0: 278
0= 1:0:0:0: 45
4= 0:0:1:0: 4= 0:0:1:0: 4= 0:0:1:0: 0= 1:0:0:0: 8
0= 1:0:0:0: 7
6= 0:1:1:0: -----------------------------------
```

Notes on using getnum().

- Line 19 calls the function that reads the file.
- As with text files, the `eof` and `bad` flags can go on. But with numbers, there is a new possibility: in attempting to read a number, number conversion might fail because of a non-numeric character in the stream. This will cause the `fail` flag to be turned on.
- The file eofNumeric.in (above, on the left) has two fully correct lines followed by one line with non-numeric characters, then another correct line.
- Within the `use_nums` function, Line 30 reads one thing into an `int` variable. Line 31 prints the input if instr.good() is true.
- When `>>` is used for input, and the target variable has a numeric type, the stream is read one character at a time until a non-numeric character is found. Then the prior chars are converted to a binary number and stored in the given variable.
- On the third line of input, the first two characters are read correctly, and the "abc" is left in the stream, unread. The output is `45`.
- The system tries to read the fourth character, 'a', the next time around the read loop. A conversion error occurs because 'a' is not a base-10 digit. The eofbit is not turned on, so control passes to line 33, which detects failure caused by the conversion error.
- Lines 32...38 show how to deal with numeric inputs. The stream is not "good" – this could be caused by an eof or an error. Since eof is normal, we must check for that first, and possibly break out of the input loop.
- Lines 34 and 35 recover from this error. First, the stream's error flags are cleared, putting the stream back into the `good` state. Then the offending character is read and discarded. Control then goes back to the top of the loop, and we try again to read a number.
- After the first error, 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 prints it and goes on to normal completion. Compare the input file to the output: the "abc" just "disappeared", but you can see there were three "failed" read attempts before the final 8 was read successfully.

- **Warning:** 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 an "infinite" `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 input loops this way.

3.6.1 Hexadecimal Input and Output.

Handling hexadecimal data. The line `45abc8` is a legitimate hexadecimal input, but not a correct base-10 input. Using the same program, we could read the same file correctly if we wrote lines 28–31 to use hex :

```
instr >>hex >>number >>dec ;
cout <<instr.rdstate()<<"= " <<instr.good() <<"->"
    <<instr.fail() <<":" <<instr.eof() <<":" <<instr.bad()<<".  ";
if (instr.good()) cout << hex <<number << dec <<number << endl;
```

In this case, all input numbers would be interpreted as hexadecimal and the characters "abcdef" would be legitimate digits.

The output shown below is given in both bases 16 and 40 to make comparisons easy. The results are:

```
Reading numbers.
0= 1->0:0:0.  1        1
0= 1->0:0:0.  278      632
0= 1->0:0:0.  45abc8   4565960
0= 1->0:0:0.  7        7
6= 0->1:1:0.  ------------------------------------
```

Note: whenever you put an input or output stream into hex mode, it is important to put it back into dec mode before leaving the context.

3.7 Formatting and File Handling Demonstration

This example continues to develop basic C++ file-handling tools and syntax. Topics include using a stringstream, reading lines of mixed numeric and alphabetic input and formatting the output in a table.

The program uses two classes and many of the —pl C++ stream, input and output facilities. It incorporates the stream-error detection and recovery techniques of the prior examples and can serve as a model for you in writing correct and robust file-handling programs.

The main program is given first. Following that is a header file for each of the two classes, `Inventory` and `Part`, the input, and the output.

In this application, we do an unusual thing: all the code for the two classes is in the two header files. Modern compilers accept this and do the right thing. Traditionally, though, the function headers are in the header file and the function definitions are in a separate code file. This is still seen as the better way to organize the code. For anything but a very simple class, a separate code file is the best way.

```
 1  // Demo program for mixed numeric/alpha input and formatted output.
 2  // Michael and Alice Fischer,  August 11, 2014  revised June 2021.
 3
 4  #include "inventory.hpp"
 5  #define FILENAME "parts.in"
 6  //--------------------------------------------------------------------
 7  int main( void )
 8  {
 9      banner();
10      Inventory partList( FILENAME );
11      cout << partList;
12      bye();
13      return 0;
14  }
```

The main program. Finally, we have a main program that is brief (according to style guidelines). To make this possible, we introduce a second class (`inventory`) to act as a controller for the application.

- This is a typical main program for C++: it instantiates a controller class delegates all activity to it.
- Line 4 includes the header file for the controller class, `Inventory`.
- Here, the filename is provided on the command line. There are several ways to get a file name into a program:
  1. Make it a command-line argument, as in prior examples. This is the most powerful and flexible method. Use it when there might be more than one input file in your test suite.
  2. Define the filename as a constant above main. This method is OK, but makes it difficult to take different files as inputs, and therefore, difficult to debug a program.
  3. Prompt the user to input a file name. A good idea for beginners. Its shortcomings are that testing becomes slower and clumsier, and that interaction of this sort does not work with automated testing.
  4. Embed it in an open statement somewhere in the code. *Do not do this.* It makes a program non-portable and difficult to maintain. Keep in mind that a Windows pathname does not work on Linux or on a Mac.
- Line 10 instantiates the controller class and line 11 calls on it to do the work.

The Inventory class.

- A "controller" class is in charge of the logic of the application. It instantiates one or more other classes to organize its data and get its task done. It is the only class that should interface with the main program.
- I

```
15  //--------------------------------------------------------------------------
16  //  Implementation file for hardware store parts.           Inventory.hpp
17  //  Created by Alice Fischer, June 19, 20211.
18  //
19  #include "tools.hpp"
20  #include "part.hpp"
21  //--------------------------------------------------------------------------
22  class Inventory {
23  private:
24      ifstream fin;
25      vector<Part> stock;
26
27  public:
28      ~Inventory() =default;
29      //----------------------------------------------------------------------
30      Inventory( ifstream& fin ){
31          bool goodFlag;      // Error code returned from Part constructor.
32          string buf;         // Holds one line of the file.
33
34          cerr << "\nReading inventory from parts input file.\n";
35          for(;;) {
36              fin >> ws;      // Skip prior newline and leading whitespace.
37              getline( fin, buf );
38              cout << buf << endl;          // Echo-print every line.
39              if (fin.good()) {             // Process any good data.
40                  Part p( buf, goodFlag );  // Create one part from buf.
41                  if (goodFlag) stock.push_back( p );
42                  else cerr <<"   Not a good part: \"" <<buf << "\"" <<endl;
43              }
44              if (fin.eof())  return;
45              else if (fin.fail()) {      // Incomplete data.
46                  fin.clear();
47                  cerr <<"Error: input line is incomplete: << buf";
```

```
48              }
49          }
50          fin.close();
51      }
52
53      //-----------------------------------------------------------------------
54      ostream&
55      print( ostream& fout ){
56          cout <<"\nPartname.................. Store Quant Price\n";
57          for (Part p: stock) {
58              fout <<p <<endl;
59          }
60          return fout;
61      }
62  };
63  inline ostream& operator <<(ostream& out, Inventory& p){return p.print(out);}
```

- The tasks of reading and printing the data for a single part are delegated to the Part class. This class is concerned with organizing many Parts into an accessible data structure.
- The first data member is the input stream, which will be opened in the constructor. The second is a data structure to store the validated inputs – we use a `vector<Part>` because we do not know how many parts will be in the inventory.
- Line 28 is an explicit definition of the destructor. The do-nothing default is appropriate because a `Part` does not have any dynamically allocated portions.

The Part constructor (Lines 27 . . . 53)

- Line 31 declares a status variable. It will be passed by reference to the Part constructor. Within the constructor, it will be set to true if the input is fully OK, false otherwise. Since a constructor does not have a normal return value, this is a way for it to pass information back to its caller.
- Lines 32–33 open the input stream and check for proper opening, following the pattern of prior examples.

Reading the file into the inventory vector (Lines 35 . . . 51)

- This constructor is long because it *realizes* the application's data, that is, it uses stream input to bring the data into memory and it stores the data in a structured form. Over half of the lines in this loop are involved with checking for and handling stream errors.
- Lines 37–39 read one complete line from the file into a local temporary, buf. A properly formatted data file has the information for one Part on each line.
- Line 38 starts by skipping any extra whitespace on the prior line and on the beginning of this line. Extra whitespace may or may not exist, but since it is not visible on a printout, it is a good idea to take this precaution; if no whitespace is present in the stream, no harm is done. Operator `>>` does remove leading whitespace, but we cannot use `>>` to read the part name because may contain embedded whitespace.
- Line 39 reads the input and potentially sets error flags. Line 40 echo-prints it. This output helps the user interpret any error comments that might appear. Lines 41–50 check for and handle errors. This logic seems complex, but is probably the shortest and easiest way to do the job properly.
- Line 41 tests for the presence of good input. If found, the input string is sent to the Part constructor for further analysis. If the input is all OK, a part is constructed and pushed onto the end of the stock vector.
- It is possible to have input *and* an end-of-file. Line 46 leaves the loop if an eof was found.
- There is only one possibility left: an input error. Line 48 clears the stream's error flags and line 49 informs the user of the error. The error comment will appear below the input line that caused it.
- When eof happens, the file is closed (lines 46, 52).

Printing the inventory array. [Lines 56...61]

- A foreach loop is the easiest way to print the contents of any standard template data structure.
- This print function returns an `ostream&` so that it can be easily used in a definition of a new method for `operator <<` (line 66) which is called from main() on line 10.
- Line 60 uses the `operator <<` method defined in the Part class.

The operator extension. [Line 63]

- Line 63 defines an additional method for the `<<` operator. This permits a programmer to use this `<<` to print an instance of the Inventory class.
- This is a global function method. It cannot access the private members of Part, but it can (and does) call the public `Part::print()` method.

3.7.1 Testing the Code

An input file with no errors. In the input file, each data set should start with a part description, terminated by a comma and followed by two integers and a price. A program should not depend on the number of spaces before or after each numeric field. This is the fully-correct input file used for testing. Please note that it *does* end in a newline character.

```
claw hammer, 3  28 9.99
claw hammer, 5  3  8.95
long nosed pliers, 5 57 2.38
pliers, 3 31 2.38
roofing nails-1 lb, 5 15 1.59
```

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

```
 Reading inventory from parts input file.
claw hammer, 3  28 9.99
claw hammer, 5  3  8.95
long nosed pliers, 5 57 2.38
pliers, 3 31 2.38
roofing nails-1 lb, 5  15 1.59
roofing nails-1 lb, 5  15 1.59

Partname..................  Store Quant Price
claw hammer..............   3     28    9.99
claw hammer..............   5      3    8.95
long nosed pliers........   5     57    2.38
pliers...................   3     31    2.38
roofing nails-1 lb.......   5     15    1.59
```

A similar input file with errors. In the input file, the third line is incomplete and produces an error comment. This is the fully-correct input file used for testing. Please note that it *does NOT* end in a newline character, an error, so the last line is missing from the final listing.

Output from the file with errors.

```
Reading inventory from parts input file.
 claw hammer, 3  28 9.99
 claw hammer, 5  3
 Not a good part: "claw hammer, 5  3  "
 long nosed pliers, 5 57 2.38
 31 2.38
 Not a good part: "31 2.38 "
```

```
roofing nails-1 lb, 5  15 1.59

Partname..................  Store Quant Price
claw hammer..............   3     28    9.99
long nosed pliers........   5     57    2.38
```

The Part class. A data class concerns itself with one object. Multiple objects must be handled by the class that instantiates the data class. It contains all of the code that deals with the properties and details of one object, in this case, a part: name, store code, quantity at that store, and price.

```
66  //-------------------------------------------------------------------------
67  //  Header file for hardware store parts.                          Part.hpp
68  //  Created by Alice Fischer on Mon Dec 29 2003.
69  //
70  #include "tools.hpp"
71  //-------------------------------------------------------------------------
72  class Part {
73  private:
74      string partName;
75      int storeCode;
76      int quantity;
77      float price;
78  public:
79      ~Part() =default;
80      //-------------------------------------------------------------------------
81      // Initialize Part with data from the buffer.
82      Part( string buf, bool& isValid ){
83          stringstream ssin( buf );   // Wrap a stream around input buffer.
84          getline(ssin, partName, ',' );
85          ssin >>storeCode >>quantity >>price;
86          isValid = !ssin.fail() ;
87      }
88
89      //-------------------------------------------------------------------
90      ostream&
91      print( ostream& fout ){
92          fout <<left  <<setw(25) <<setfill('.') <<partName <<"  ";
93          fout         <<setw(3)  <<setfill(' ') <<storeCode;
94          fout <<right <<setw(5)                 <<quantity;
95          fout <<fixed <<setw(8)  <<setprecision(2) <<price;
96          return fout;
97      }
98  };
99  inline ostream& operator <<(ostream& out, Part& p){ return p.print(out ); }
```

The Part constructor.

This function parses the input line for a single part and attempts to find gross errors. It does not try to validate any of the fields, although a real program would probably do that. If an error is discovered while reading a data set, the entire data set is skipped and an error comment is printed.

- The first parameter to the constructor is a string that contains one line of input. It was read by the Inventory class and passed to the Part class for parsing and necessary error checking. The easiest way to parse a line of input is to wrap an istringstream around a string. (Line 83)
- Any of the C++ input functions can be used on a stringstream, and they behave the same way as on an istream. Here we use `getline()` and `>>` to parse the string. No `ws` is needed here because Inventory did it.
- We use the 3-parameter form of `getline()` to read the partname up to the comma; the third parameter is the character that delimits a field, in this case, a comma. The comma is removed from the stream but not stored in the partname. Instead, a null terminator is stored in the partname.

- Operator >> is used three times to read the three numbers. It does several useful things:
  - Skip leading whitespace
  - Read a whitespace-delimited numeric field
  - Perform character-to-binary number conversion
  - Store the resulting number in the specified class member.
- If any one of the four inputs is missing from `buf`, the fail flag will be set. The last line (line 86) in this constructor tests for failure and stores the result in `isValid`, the reference parameter, to be sent back to the caller.

The `Part::print()` function.

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.

C++ I/O does not use formats; we accomplish the same goals with stream manipulators. A series of directives control the alignment, field width, and fill character. Lines 92...95 each format and output one data members of the Part. (It is easier to use C formats!)

- To print data in neat columns, the width of each column must be set individually using `setw()`. This produces lengthy code; an old-fashioned C format was much more concise. In this function, the code to set up and print each field of the line is written on a separate line.
- Right justification within the field width is the default. This is appropriate for most numbers but is usually not good for alphabetic information. We set the stream to left justification (line 94) before printing the part name. It will remain left-justified until set to something else, as seen on line 96. Thus, the store code will also be printed with left justification.
- The default fill character is a space, but once that is changed, it stays changed until the fill character is reset to a space. For example, line 94 sets the fill character to `'.'`, so dot fill is used for the first column. Line 95 resets the filler to `' '`, so spaces are used to fill the other three columns.
- The price is a floating point number. Since the default format (%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. (Line 97)
- The return statement on line 96 makes the definition of the operator extension easier. That method must return an `ostream&` because it must conform to the prototype of the `<< operator`.

Files with no data.

The result of running the program on an empty file is not ideal, but it is also not misleading:

```
Reading inventory from parts input file.

Partname..... Store Quant Price
```