Practical C++ Programming Teacher's Guide

Introduction

This guide is designed to help with the classroom presentation of the material in *Pracctical C++ Programming.* It contains a set of *teacher's notes* for each chapter which give you information about the key concepts covered in the chapter as well some ideas for in-class demonstration.

The *live demonstration* section alrets you to the live programs available for demonsration in class.

There is also a set of *review questions* for each chapter with answers. (The “Quiz” document contains these questions without answers.)

Hopefully this guide will make your teaching of C++ a little easier.

Table of Contents

Chapter 1: *What is C++?*

Profanity is the one language that all programmers understand.
-- Anon.

Teacher's Notes

This chapter is designed to give students some idea of where C++ fits in the world of programing.

You might want to take this opportunity to tell your class about your personal experience with C++. For example, when did you first encounter the language, what where you doing with it, how did it make your life better?

This is also the time to acquaint the students with what the course is going to cover and what is expected of them.

Good style and programming practices are stressed in this book. I've gone through a lot of other people's code and am a fanatic about creating simple, readable programs.

I grade 60% on style and 40% on function. Style covers things like, "Is the program readable? Is it simple and easy to understand? Are there comments at the beginning of each function? Are there comments after each variable declaration?" and so on.

Functionality covers whether or not the program works and works efficiently.

Finally this is the time to acquaint the students with the local computing facilities. They should know where the machines are and where to go for help.

Review Questions

1. Define "class."

   A set of data and the functions that work on that data.

2. Where are the computers that are to be used for homework for this course?
3. What compiler are we using for this course?

Chapter 2: *The Basics of Program Writing*

The first and most important thing of all, at least for writers today, is to strip language clean, to lay it bare down to the bone.

— Ernest Hemingway

Teacher's Notes

In this chapter we give students an idea of what a programming language is. We try to give students an idea of the work done to translate a high-level programing language into an executable program.

Finally, we have a section that describes in extreme detail the steps needed to run a compiler. Four specific compilers, Borland-C++, Microsoft Visual C++ .NET, GNU's g++, and a generic UNIX CC compiler, are described.

Computers require precise instructions in order to work. We start by introducing the students to a language they probably already know: English. Even with English, precise instructions are hard to create.

Programming languages have evolved over the years. In the beginning everyone programed in machine language. This evolved through assembly language, higher level language, the C language, to the C++ language. This chapter gives a brief description of each of these stages.

Next the students are introduced to the tools used to create programs. At this point I suggest that you tell the students the absolute minimum needed to actually use the tools and no more. After that they can either learn by reading the manual or you can give little ten minute mini-lectures at the beginning of future classes.

But for now they need to know only how to use the editor, compiler, linker, and the make utility. If they are programming in DOS or Microsoft Windows, a short tour of the various components of the Integrated Development Environment is in order.

At this point it would be good to demonstrate creating a program. I suggest that you deliberately include a mistake. This gives you a chance to show the students what an error message looks like and how to correct it.

When the chapter is finished the students should be able to type in a program and get it to run. The program won't be understandable to the student at this point, but it will be enough to get something to run.

Live Demonstration

Slide 13 *hello/hello.cpp*

Classroom Presentation Suggestions

Before the class, go through the slides and remove any "Compiling the program using ...." slides that do not pertain to your local setting.

If you are using a commerical compiler (Borland, Microsoft), check the slides to make sure that informatioin them is still current.

Starting with the slide on Construction Tools (Slide 9) you may want to demonstrate the creation of an actual program. Show how to edit, compile, get an error, edit again, compile, and run a program.

After this lecture, the class should adjourn to the computer lab where they will type in their first program under the watchful eye of the instructor. This is the time they will need the most hand-holding as they will make a large number of simple errors trying to get the compiler to run.

Review Questions

1. Define "Machine Language."

   A language consisting of a series of numbers. These numbers represent the actual instructions used by the computer. Easy for computers to understand, but very difficult for humans.

2. Define "Assembly Language."

   A language in which a single instruction translates directly into a single machine instruction.

3. Define "source code."

   The high level code written by the programmer. In a high-level language, the source code is usually machine independent.

4. Define "object code."

   The source code after it has been translated into machine language.

5. Define "library."

   A collection of generally useful procedures available to the programmer

6. Define "linker."

   A program that combines one or more object files with a set of libraries and produces an executable program.

7. Define "executable program."

   A machine dependent file that contains all the instructions necessary to perform a task.

8. How is assembly language different from machine language?

   Machine language is solely numbers. Assembly language uses words to represent those numbers.

9. What does a compiler do?

   It takes a source file and transforms it into an object file. Note: many "compilers" are actually wrappers that run a compiler and a linker.

10. How is a compiler different from an assembler?

    One assembly instruction translates to one machine language instruction. Assembly language is a low-level language. A compiler translates one statement into many instructions.

    Compilers are also machine-independent, while assemblers are machine-dependent.

11. What is the extension for source files on our machine?

    The extension ".cpp". Although .C and .cc are sometimes used on UNIX / Linux.

12. What type of program is used to create "source code."

    A text editor. Note: many Microsoft Windows compilers supply an Integrated Development Environment (IDE) that contains a text editor (among other things). In spite of all the wrapping, however, it's still a text editor.

13. What is the extension for object files on our machine?

    DOS/Windows uses ".OBJ". UNIX uses ".o".

14. What type of programs are used to produce object files?

    Compilers.

15. What is the extension for the executable programs?

    DOS/Windows uses ".EXE". UNIX uses no extension.

16. What type of files go into creating an executable program?

    Programs start out as "source." They get translated into "object" files by the compiler. These are combined with "libraries" by the linker into an "executable program."

17. What type of program is used to create an executable program?

    The compiler starts the process, but it produces object files. The linker is the program that actually produces the executable programs. Note: the actual linking step may be hidden since the UNIX CC wrapper and the DOS Integrated Development Environments run the linker behind your back.

18. How do you access the on-line help system for your compiler?

    Most DOS/Windows Integrated Development Environments have a built-in help system. On UNIX you can use the "man" command.

Chapter 3: *Style*

There is no programming language, no matter how structured, that will prevent programmers from writing bad programs.

- L. Flon

It is the nobility of their style which will make our writers of 1840 unreadable forty years from now.

- Stendhal

Teacher's Notes

As a professional programmer I have had to maintain a lot of code. Most of it in very poor condition. Hopefully, by teaching commenting and style early we can convince future programmers to write well commented, easy to read programs.

In this chapter we describe how to comment programs. It may seem strange to learn how to comment before we know how to program, but the comments are the most important part of the program.

That last statement deserves repeating, *Comments are the most important part of the program. I grade homework 60% on comments and 40% on code.*

At this point it may seem that I'm somewhat of a fanatic on commenting. I am. You get this way after spending years going through uncommented code while trying to maintain it.

The purpose of this chapter is to convince your students to put copious comments in the code.

Comments should be written as an integral part of the programming process. Time and time again I've heard, "Here's my homework, but I didn't have time to put in the comments."

My answer, "You should put in the comments first. I don't mind too much if you put in the comments and don't have enough time to do the code, but don't leave out the comments — ever!"

When I was grading homework I had a rubber stamp made up:

If you teach your students one thing it should be to make their programs clear and easy to understand. Comments are an essential part of that process.

Classroom Presentation Suggestions

There should be a style sheet for the class. This can be created by the class or supplied by the instructor.

It should include a list of what you as a teacher want for the heading comments of each assignment. Suggestions include:

- The student's name
- The assignment number
- Student ID number
- All the other stuff detailed in this chapter

Review Questions

1. Why are comments *required* in a program?

   Comments provide the programmer with a clear, easy-to-read description of what the program does. They are required because uncommented programs are hard to understand (sometimes even the person who wrote them can’t understand them). Uncommented programs are also extremely difficult to maintain.

   As the age and complexity of software programs skyrockets, maintenance (and comments) becomes much more important. (

2. Why must you write comments before or while you write the program, never after?

   Because at the time you write the code you know what you are doing. If you leave the commenting until later, you may forget and have to guess.

3. Which is better: 1) The underscore method of writing names, such as: `this_is_a_var`, or 2) The upper/lower case method: `ThisIsATest`?

   I like `this_is_a_var` myself because you can run your program through a spelling checker. I think it is more readable. Also it allows you to use lower case only for variables and upper case only for constants.

   Some people think that `ThisIsATest` is more readable.

   This is a religious issue and has no right answer.

Chapter 4: *Basic Declarations and Expressions*

A journey of a thousand miles must begin with a single step.
— Lao-zi

If carpenters made buildings the way programmers make programs, the first woodpecker to come along would destroy all of civilization.
— Anon.

Teacher's Notes

In this chapter students are introduced to the concept of a program. Programs consist of three parts: comments, data, and code.

We've already covered comments. They are the most important part of a program and are used to document the things that the program does. They are written for human beings like maintenance programmers and C++ instructors.

To simplify things as much as possible we present only one type of data, simple variables declared globally. Later we'll see how we can declare complex variables such as structures and classes. We'll also look at the difference between local and global variables. But for now we just want to get the concept of a variable across.

Similarly, the code part has also been simplified. Our programs consist of only one function, `main`.

We also limit ourselves to simple expressions and assignment statements, which, along with the `std::cout` statement are all we need to write programs. True, they are very limited programs, but they still give your students a chance to make a lot of mistakes.

Because the students know next to nothing about programming at this point, we use the "trust me" method of teaching several key elements.

For example, the executable part of the program begins with:

```
int main()
{
```

Rather than confuse the student with what this actually does, we say "trust me, it works". Actually, we are declaring the beginning of the function `main`. This function is just like any other C++ function except that the special name causes C++ to call it first. All other functions are called directly or indirectly from `main`.

At the end of the program we have another "trust me." We end programs with:

```
        return(0);
    }
```

Actually, this returns a value of zero to the caller (the operating system). The 0 indicates a successful execution. If an error occurs we would return a positive number. The bigger the number, the more severe the error.

Live Demonstration

Slide 14 *tterm/tterm.cpp*

Classroom Presentation Suggestions

Keep it simple. The students are about to write their first programs, and they will tend to have a lot of very basic problems.

Review Questions

1. Name and define the three elements of a program.

   Functions, data declarations, and comments.

2. Why are comments required in a program?

   Without them programs are next to impossible to debug, maintain, or enhance.

3. What is the purpose of the `return(0);` near the end of each program?

   It returns a status code of `0` (good return) to the operating system.

4. What punctuation character signals the end of each statement?

   A semicolon (;).

5. What statement to you use to print something to the screen?

   `std::cout` statement.

6. What are the five simple C++ operators?

   Multiplication (*), division (/), addition (+), subtraction (-), and modulus (%).

7. Evaluate the following C++ expressions:

   a. `5 + 3 / 2 + 1`

      7

   b. `(5 + 3) / ( 2 + 1)`

      2

   c. `8 % (3 * 4) + 8 / 3 - 9`

      1

d. `8 / (3 * 4 + 8 / 3 - 9)`

______1

e. `4 + 9 * 6 - 8`

______50

f. `(11 % 7) / 3`

______1

8. What will the following statements print? (Assume i=5 and j=7.)

a. `std::cout << i << '\n';`

______5

b. `std::cout << "i" << '\n';`

______i

c. `std::cout << i / j << '\n';`

______0

d. `std::cout << "i=" << i;`

______i=5

Warning: There is no newline at the end of this statement, so the next std::cout output will run right up against the 5. (Example: `i=5Hello World`.)

e. `std::cout << "i=" << i << '\n';`

i=5

______Notice that the newline is added, so this will be a complete line.

9. Turn the equation $d = \frac{1}{2} \cdot g t^2$ into a C++ statement.

```
d = 1.0 / 2.0 * g * t * t;
```

or using better variable names:

```
distance = 1.0 / 2.0 * GRAVITY * time * time;
```

10. What does it mean when you see the message “Warning: Null effect”?

A statement is correctly written, but useless. For example, a statement that computes a value then throws it away.

11. Define “variable.”

A named area of memory used to store values.

12. Define "variable declaration."

A C++ statement that describes a variable. It is used to tell C++ what variables we are going to use in our program.

13. What are the three purposes for a variable declaration?

The type of the variable (**int**), the name of the variable (`total count`), and a comment describing the variable.

14. The FORTRAN language (among others) would automatically declare a variable the first time it was used. Variables that began with A-H,O-Z where automatically declared float and variables that begin with I-M were automatically declared int. Discuss the advantages and disadvantages of this "feature." (Note: FORTRAN was invented before lowercase was widely used in computes, so all variables were written in uppercase only.)

Advantages

You don't have to declare variables.

Disadvantages

You got names like `KOUNT` to force the count variable to be an integer type. Misspelling of variable names could not be caught by the compiler.

Note: When I was writing FORTRAN programs I wanted to force the compiler to require that all variable names be declared. I did this by the statement:

```
IMPLICIT COMPLEX (A-Z)
```

This told the compiler that any variable that was not declared was to be considered a complex variable. Because complex variables required a special syntax and because I didn't use them, any misspelled variable name showed up as a compiler error:

```
Misuse of a complex variable.
```

15. Define `std::cout` and use it in a C++ statement.

`std::cout` is the C++ class that is used for writing data to the screen. Example:

```
std::cout << "Hello World\n";
```

16. Define "assignment statement" and give an example.

Assignment statements take expressions and put them in variables. Example:

```
result = 45 / 36;
```

17. What is the difference between a real and an integer variable?

Integer variables can only hold whole number or integers such as 1, 5, 98, and -42. Real or floating point variables can hold fractional numbers such as 5.3, 8.6, and 45.2. Note: Any number with a decimal point is floating point, even if it looks like 5.0.

18. Give an example of an integer division statement and a floating point division statement?

```
i_var = 12 / 5;     // Integer Division
```

```
f_var = 12.0 / 5.0; // Floating point division
```

19. The same operator '/' is used for both floating point and integer divides. How does the compiler know which one to use?

    The compiler checks both sides of the division to see if either the divisor or the dividend is a floating point number. If either one is floating point, then a floating point divide is performed. If both operands are integers then an integer divide is done.

20. A thermos keeps hot things hot and cold things cold. How does it know which one to do?

21. The standard character set, ASCII, handles 128 characters. Only 95 of these can be typed in using the keyboard. What does a C++ programmer use to specify the none printing characters?

    The backslash character (called the escape character sometimes) is used to indicate a special character. For example "\b" is used to indicate the non-printing "backspace" character.

22. Define "character variable" and show how a character variable can be declared and used.

    A variable that can hold a single character.

```
char a_char; // Sample character declaration
//...
a_char = 'x';
```

    Characters may also be used as "very short integers." That is, numbers in the range of 0-127. Example:

```
a_char = 5;
```

23. What are the possible value of a boolean (**bool**) variable?

    true and **false**.

    (Do NOT use "True", "False", "TRUE", or "FALSE". If you see these they are legacy constants for ancient code.)

Advanced Questions

The answers to these may require resources outside the scope of this book.

24. What is the value of `4.5 % 3.2`?
25. Why are real numbers called floating point numbers?
26. What "floats" in a floating point number?
27. What is the biggest integer variable that you can have on your machine?
28. What is the biggest floating point variable that you can have on your machine?
29. Why are integer variables exact and floating point variables inexact?

Chapter 5: *Arrays, Qualifiers, and Reading Numbers*

That mysterious independent variable of political calculations, Public Opinion.
— Thomas Henry Huxley

Teacher's Notes

In the previous chapter students were introduced to the simple variable and the output statement. In this chapter we expand on that knowledge by introducing them to the array type and the input statement `std::cin`.

C++ has two basic types of strings. The first, the *C++ Style String* is a class found in the *string* header file:

```
#include <string>
std::string name = "Steve Oualline";
```

The other type is the *C Style String* which is created from an array of characters.

```
char name[50] = "Steve Oualline";
```

C++ style strings are introduced first since they are simpler to use and less error prone than C style strings.

One of the things new to this edition is the introduction of safety code. Starting with the section *Bounds Errors* (page 54) the assert statement explained. Since I wrote the first edition, I have encounter a huge mass of badly written code. It was only the extensive use of asserts which made it possible to debug the thing. These statements prevented array boundary errors which could cause random problems in the code.

The use of error checking code and asserts should be highly encouraged.

The book discusses C Style strings because there are a lot of programs out there that use them. Your students should avoid them because they force you to handle your own storage. (C++ Style strings automatically allocate their own storage.)

This chapter introduces the student to all the modifiers such as **short**, **long**, and **register**. Some, like **short** and **double**, the student can use immediately. Some, like **register**, will be discussed in later chapters. A few, like **volatile**, are so advanced that this course does not cover them at all.

At this stage of the game, reference declarations are not useful. In fact putting them in a program tends to confuse things. However, you will find them very useful when we start using them to declare function parameters later in the book.

All of the modifiers have been included in this chapter for completeness.

C++ contains a large number of shortcut operators. This makes programming more compact and concise. Unfortunately, the overuse of these operators can cause problems.

Verbose programs that work are much better than terse programs that do not.

I believe that the goal of programming is not compactness, but clarity and a working program. For that reason I don't allow my students to use shortcut operators such as ++ and -- inside other statements.

Side effects such as those caused by ++ and -- inside other statements can easily cause trouble. What's worse, the type of trouble they cause is difficult to detect. Avoid side effects.

Live Demonstration

Slide 3 *five/five.cpp*

Slide 24 *name2/name2.cpp*

Slide 26 *len/len.cpp*

Classroom Presentation Suggestions

On Slide 20 we ask the question "Are all 'C Style strings' 'arrays of characters'?" The answer is "Yes." A string is a special form of an array of characters. A string starts at the first element of the array and continues until the end-of-string character ('\0').

Note: The declaration

```
    char data[10];
```

defines a 10-character array. As an array it holds 10 characters, no more, no less. It can hold strings of length 0 to 9. Why not 10? Because we need one location to store the end-of-string character.

Are all "character arrays" "C Style strings"?

No. Consider the following character array:

```
        char data[4] = {'A', 'B', 'C', 'D'};
```

In this case we have an array containing four characters. It does not contain a string. There is no end-of-string character.

How can the compiler tell the difference between a character array and a string? It can't. If we tried to treat `data` as a string, the compiler would let us. We would, however, get strange results.

Question on Slide 9 of the slides

The variable `width` is undefined because it is in the middle of a comment. Check out the comment end on the previous line.

Question on Slide 18 of the slides

The question is, why does `array[2,4]`print out a funny value? C++ uses the syntax `array[2][4]` to select a single element of the array.

When C++ sees the expression `2,4` it uses the comma operator (see Chapter 28, *C++'s Dustier Corners*) to evaluate the expression. The result is 4. So `array[2,4]` is the same as `array[4]`. This, unfortunately, is a pointer and C++ prints it as a pointer, which results in the strange printout.

Question on Slide 32 of the slides

Answer "c." I'm a great believer in practical programming and follow the adage. "If you don't write stupid code, you don't have to answer stupid questions."

The "real" answer to this question, according to the ANSI draft standard, is that there are three flavors of characters, **char**, **unsigned char**, and **signed char**, and compilers should generate a warning if you try to mix flavors.

Many real compilers don't follow the standard. Some generate signed characters, some unsigned, and the Borland compiler has a switch that allows you to specify signed or unsigned as the default.

Side Effects

If possible, include a story from your programming experience titled "How I found a nasty bug caused by a side effect and why I want to strangle the programmer who put it in."

Review Questions

1. What is the difference between an array and a simple variable?

   A simple variable holds only one value. Arrays hold many values.

2. What is the number of the last element of the following array?

   ```
   int test[5];
   ```

   "4" (The elements go from 0 to 4).

3. What's the header file that's used to bring in the data definitions for C++ strings?

   ```
   #include <string>
   ```

4. How do you concatenate two C++ style string?

   Use the "+" operator.

5. What is the difference between a C style string and an array of characters?

   Arrays of characters are fixed length. C++ makes no restrictions on what can be placed in any array of characters.

   Strings are stored in arrays of characters. The end of a string is indicated by an end of string character ('\0'). Because strings use this marker strings can be any length up to the size of the array they are stored in (—1 for the end of string marker).

6. Why must we use `std::strcpy` to assign one C style string to another?

   C++ does not have an array assignment operator, so it's impossible to assign one character array to another.

7. We define a character array whose job is to hold a string. What is the difference between the size of the array and the length of a C style string?

   The size of the array is fixed and set to the size declared at compile time. The length of a string can change depending on where we place the end of string character.

8. Can the size of any array change during the execution of a program?

   No.

9. Can the length of a string variable change during the execution of a program?

   Yes.

10. What happens if you try to read a C++ style string using `std::cin` and the `>>` operator? (Try it!) What about the C style strings?

    The input line, up to the first space or other whitespace character, is read into the string.

11. How many elements are allocated for the multiple dimension array:

    ```
    int four_dimensions[25][86][48][96];
    ```

    Bonus question: Why will you probably never see a declaration like this in a real program?

    This array defines 3,863,808 elements. (25*86*48*96) On a machine that uses 4 bytes per integer that's 15,455,232 bytes or 15MB. That's a lot of memory for a single array.

12. Define the following:

long int	**short int**	**unsigned**	**signed**
float	**double**	**register**	**auto**
volatile	**const**	**reference**	**extern**

long int a signed integer that may hold more data than an ordinary integer.

short int a signed integer that may hold less data than an ordinary integer.

unsigned	variable that may hold only positive value. By sacrificing the sign bit, the number can hold twice as many positive values.
signed	a number that may hold positive and negative values. Since this is the default for integers and floating point variables, it is rarely used.
float	a variable type that can hold real or fractional values.
double	a real variable type that has more range and precision than “float.”
register	a suggestion from the programmer to the compiler indicating that this variable is used a lot so it would be a good idea to put it in a machine register.
auto	a variable that is automatically allocated from the stack.
volatile	a special variable whose value may be changed by things outside the program. Used for things like memory mapped I/O.
const	a named value that cannot be changed.
reference	an alias or second name for an already declared variable.
extern	a variable that is defined in another file. Actually, C++ won’t get mad at you if you define it in the current file.

13. Why must you use the notation `static_cast<int>(very_short)` to write a very short number (a.k.a. a character variable)? What would happen if you wrote it out without the `int` wrapper?

    The `static_cast<int>(x)` notation changes the character variable’s type from character to integer for the purpose of the write. If the `int` wrapper is left off, the C++ would write the character variable out not as a very short integer but as a single character. After all, C++ does not know if you are using a character variable to hold a character or a very short int.

14. Define side effect.

    A side effect is an operation that occurs in addition to the main operation specified in a statement.

15. Why are side effects bad things?

    The confuse the code and can easily cause unexpected things to happen. Using them can make the code dependent on things like the evaluation order of operands. They save space in the source code at the expense of clarity and reliability.

Chapter 6: *Decision and Control Statements*

Once a decision was made, I did not worry about it afterward.
— Harry Truman

Teacher's Notes

Student are now introduced to the fact that computers cannot only do computitions, but can also make decisions.

Decision statements for the most part are fairly straight-forward and fairly simple, as are the relation operators.

A conditional statement works on the single statement that follows it. Multiple statements my be grouped together by curly braces ({}) and treated as a single statement.

The **if/else** statement is ambiguous as we can see in Slide 7. One of the things emphasized in this book is practical programming, and practically speaking you don't have to worry about language weirdness if you avoid it.

Note: For the purist in your class, the ambiguity is resolved by placing the **else** clause with the nearest **if**. (Answer b).

Then show the students how to use decision and control statements by writing a program to print out a series of Fibonacci numbers.

Finally, teach the student the dangers of using = inside an **if** or **while** statement. The one mistake you will make, and your students will make, is putting = where you meant to put ==. Emphasize this at every opportunity.

I remember after one class a student came up to me and said, "Steve, I have to admit that I thought you were a little crazy going on and on about = vs. ==. That was until yesterday. You see I just wrote my first C program for work and guess what mistake I made."

Live Demonstration

Slide 12 *fib/fib.cpp*

Slide 14 *total/total.cpp*

Slide 16*balance/balance.cpp*

Review Questions

1. Define the following:

   Branching Statements

Looping Statements

if statement

else clause

relational operator

logical operator

while statement

Fibonacci number

break statement

Branching Statements

change the control flow of a program by executing one or more different branches of the code.

Looping Statements

statements that cause the flow of the program to repeat or loop over a section of code.

if statement

a statement conditionally executes a block of code.

else clause

a clause at the end of an if statement that defines a section of code to be executed if the condition in the if is not true.

relational operator

operators such as ==, !=, and <, that compare two values. These operators are used to test the relationship between two values.

logical operators

Operators perform the logical functions “and,” “or,” and “not”.

while statement

a looping statement that will repeatedly execute a section of code until its condition becomes false.

Fibonacci number

a number in a series that starts out with “1 1”. Each element of the series from here on is the sum of the previous two numbers.

break statement

a statement that when executed exits a loop.

Chapter 7: *The Programming Process*

It's just a simple matter of programming.
— Any boss who has never written a program.

Teacher's Notes

At this point our students know enough to write programs. Very simple programs, but real, useful programs.

In this chapter we try to introduce students to the entire software life cycle. One of the limitations of the classroom is that the life cycle of a program is about a week long. The student gets his assignment, does it, turns it in, and never touches it again.

In the real world programs stay around for much longer. Programs can easily remain around for years. There are also a number of things you must do to create a program, other than writing the program. These include the specification, the design, and the revision process.

We then take the students through a sample of the programming process using a simple calculator.

Very simple debugging is introduced. This consists of putting diagnostic std::cout statements in strategic places throughout the program. One trick I use is to put "##" at the beginning of each diagnostic statement. It makes it easy to spot the diagnostic output. In addition when we find the problem, it's easy to go through the program with the editor and delete all the ## statements.

Most programmers don't actually create new programs. Most programmers revise and update existing programs. Most of these programs are poorly commented if they are commented at all. Thus we have a section on *Electronic Archaeology*, the art of digging through someone else's code to figure out amazing things, such as what the heck the code does.

Live Demonstration

Slide 19 *calc3/calc3.cpp*

Slide 27 *guess/bad.cpp*

Slide 28 *guess/good.cpp*

Classroom Presentation Suggestions

When presenting the first slides you might want to relate it to some of the programs you've written.

There are five slides for different versions of the *Makefile*. You will need to select the one appropriate for your class.

Note: I suggest that you require your students to include a "Specification" and "Test Plan" for all assignments from this point on.

After the students see the slide containing the debugging output (Slide 17), put up the prototype (Slide 8) and let the students spot the problem.

Again, we emphasize the difference between = and ==. This is the most common mistake people make and one of the most difficult to detect.

The section on *Electronic Archaeology* is designed to teach the students how to go through strange code. A typical program is presented on Slide 79. Try running it. See what it does. Let the students go through it and try to figure out what is going on. Mark up the slide and add comments describing what the students discover.

Note: Anyone who uses lowercase L (l) or uppercase O (O) as a variable name should be shot. It's very hard to tell the difference between the letter "l" and the number "1." Also the letter "O" looks a lot like the number "0."

Review Questions

1. Describe the various steps of the programming process:
    a. "Requirements"
    b. "Specification"
    c. "Code Design"
    d. "Coding"
    e. "Testing"
    f. "Debugging"
    g. "Release"
    h. "Maintenance"
    i. "Revision and updating"

Requirements

The requirement document describes, in very general terms, what is wanted.

Specification

A description of what the program does.

Code design

The programmer does an overall design of the program.

Coding

Writing the program.

Testing

A description of how to test the program.

Debugging

Locating and removing errors in the program.

Release

The program is packaged, documented, and sent out into the world to be used.

Maintenance

Correction of bugs in released software.

Revision and updating

Adding new features to the program and going through the process again.

2. Why do we put each program in its own directory?

   To keep all the files for this program together in one place and to have a place that isn't cluttered with irrelevant files.

3. What is "Fast Prototyping?"

   Writing a very small first version of the program that allows the programmer to examine and debug a small program before making it into a large program.

4. What is the *make* utility and how is it used?

   The *make* utility is a programmer's assistant that decides what commands need to be run to produce a program and runs them. To use this program, you create a description of how your program is built in a file called *Makefile* and type make on most systems. If you are using Microsoft C++, you need to type *nmake*.

5. Why is a "Test Plan" necessary?

   If a bug is found, it gives the programmer the steps needed to reproduce it. Also when modifications are made it serves as a guide to regression testing.

6. What is "Electronic Archeology?"

   The art of going through someone else's code to figure out amazing things like how and why it runs.

7. What tools are available to aid you in going through some one else's code?

   This varies from system to system. Most of the common ones are the text editor and *grep*.

8. What is *grep* and how would you use it to find all references to the variable `total_count`?

   The utility *grep* searches for words (actually strings) in a set of files. To find everywhere `total_count` is used on DOS execute the command:

```
grep "total_count" *.cpp *.h
```

   or on UNIX type:

```
grep "total_count" *.cc *.h
```

Chapter 8: *More Control Statements*

Grammar, which knows how to control even kings...
— Molière

Teacher's Notes

Here's where we teach students about the remainder of the control statements. These include **for**, **switch**, and **break**.

The **for** statement you find in other languages is nothing like the for statement in C++. It allows the programmer to explicitly control the way the control variable moves through the data. For example, in the STL chapter we will use something like:

```
for (cur_item = list.begin(); cur_item != list.end();
     ++cur_item)
    // ...
```

to loop through a linked list.

Some languages prohibit you from changing the control variable inside of a loop. C++ contains no such limitation. You can change the control variable at will. The figure on Slide 4 is very good illustration of how C++ treats the for statement. It's merely a repackaging of statements that could easily go into a while statement.

The **switch** statement is somewhat complex. There are several style rules we always follow to make our code easier to read and more reliable. These are:

- Always end each **case** statement with **break** or `// Fall Through`.
- Always put a **break** at the end of the last **case**.
- Always include a **default** case even if it is only:

```
default:
    // Do nothing
    break;
```

Finally we introduce the **switch**, **break**, and **continue** statements. These statements can be tricky. The **break** statement exits a **switch** or looping statement (**for**, **while**). The **continue** statements starts a loop over from the top. It works only on loops not on switches.

Live Demonstration

Slide 5 *cent/cent.cpp*

Slide 6 *seven/seven.cpp*

Slide 10 c*alc-fig/calc.cpp*

Classroom Presentation Suggestions

Emphasis should be placed on the fact that **for** is merely a repackaging of the **while** loop. This is illustrated by the figure on Slide 5.

If possible, show the programs on Slide 5 and Slide 6 live. Single step through them to show the students what happened.

Many people believe that the **default** case of a **switch** statement must be the last statement. This is not true, the **default** may be first, last, or anywhere.

Starting with Slide 12 we begin to teach some of the style rules used with **switch** statements. We first present a code fragment that doesn't follow the rules, followed by a corrected fragment. Emphasize that these rules make your code clearer, easier to maintain and more reliable.

For example, the code on Slide 14 contains a problem. We've just added "case 3." We didn't have a break at the end of "case 2" on the previous page. Normally this would be OK as it was the last case in the statement. But we've added "case 3" and caused trouble at "case 2." In this example, if control is 2 the output is:

```
Working
Closing down
```

This is not what the programmer wanted.

Finally we present **switch**, **break**, and **continue**. The **break** statement works inside of loops and **switch** statements. The continue statement works inside loops only.

In our diagram we have **continue** inside a **switch**. This works because the **switch** is inside a loop. The **continue** doesn't care about the **switch**, it cares about the loop.

If possible, step through the program on Slide 17 to illustrate the paths outlined in the diagram.

Review Questions

9. Why do C++ programs count to five by saying "0, 1, 2, 3, 4?" Why should you learn to count that way?

   C++ uses "zero based" counting. For example, an array with 5 elements uses the numbers 0, 1, 2, 3, 4 to index the array.

10. Write a for statement to zero the following array:

```
int data[10];
```

   (Did you call your index "i"? If so go back and give it a real name.)

```
for (data_index = 0; data_index < 10; ++data_index)
    data[data_index] = 0;
```

11. Why do we end each **case** with **break** or `// Fall through`?

    Without the `// Fall through` we can't tell whether or not the programmer who wrote his code intended for the program to fall through to the next **case** or forgot to put in the break. The `// Fall through` statement tells everyone, "Yes, you are supposed to fall through to the next **case** here."

12. Why do we always put a **break** at the end of each **switch**?

    Because some day we might add another case onto the end of the **switch** and forget to go back and put in the break that was missing. Also, if we always put in the **break**, we don't have to think about whether or not we can safely omit it.

13. What will **continue** do inside of a **while**?

    It causes the program to jump to the top of the loop.

14. What will **continue** do inside of a **for**?

    It causes the program to update the loop counter and jump to the top of the loop.

15. What will **continue** do inside of a **switch**? (This is a trick question.)

    This is a trick question because you can't put a **continue** inside a **switch** statement. The **continue** statement only works inside a loop such as a **for** or while loop.

    The only way you can put a **continue** inside a **switch** is to put the **switch** inside a loop. This will put the **continue** inside a loop. When executed the continue will start at the top of the loop.

16. What will **break** do inside of a **while**?

    It causes the program to exit the loop.

17. What will **break** do inside of a **for**?

    It causes the program to exit the loop.

18. What will break do inside of a switch? (This is not a trick question.)

    It causes the program to exit the switch.

19. Discuss the pros and cons of putting the default statement at the end of every **switch**. Is there a case where putting the default statement somewhere else might be useful?

    Advantage: If you always put the default at the end of the **switch**, you can always find it easily.

    Disadvantage: Suppose you want the fifth case of a ten case **switch** statement to fall through to the default case. In this case the "natural" place for the default statement is just after the fifth case, not the end.

Chapter 9: *Variable Scope and Functions*

But in the gross and scope of my opinion
This bodes some strange eruption to our state.
— Shakespeare *Hamlet*, Act I, Scene I

Teacher's Notes

Note: This is one of the longer chapters. You might want to spend extra time on it.

So far we've been using only global variables. That's because they're simple and we didn't want to confuse the student with a lot of extra material.

In this chapter we introduce students to local variables. First familiarize the students with the concepts of when variables are created and destroyed. This will prepare them for classes where variable creation and destruction my result in a function call.

Note: The term class used to refer to a variable's class has nothing to do with the C++ keyword class.

Up to now, our programs have been in one big section. At this point we start dividing up the programs into functions.

We've been using one function all along called `main`. Except for the special name, this function is defined like any other. It has no parameters and returns an integer value. The only tricky part is that because its name is main, it is "called" by running the program and returns a status to the operating systems.

The old C language provides the program with two types of function parameters: "call by value" and "array." (Pointers are used to get around the limitations of "call by value.")

C++ provides you with lots of new parameter and return types. These include constant and reference parameters and return values.

Next there is a quick overview of structure programming (top-down programming) and bottom-up programming. One thing to emphasize is that there have been a lot of different programming methodologies introduced over the years. Each has its own band of fanatics who support it above all others. Each has proved that although it is useful, it is not the ultimate programming system.

The current design fad is called "Object Oriented Design." We get into this later when we get into classes. "Object Oriented Design" is not the ultimate programming system. This too will pass.

Ideally, what you should teach your students is that there are a lot of programming techniques out there and the programmer should pick the one that allows him to best organize his program. Don't blindly follow the rules if there is some way to make your program clearer and easier to maintain.

Finally, we introduce the tricky subject of recursion. The idea of calling a function from within itself is foreign to some people. But I've programmed in LISP using no looping statements, and you'd be amazed at what you can do recursively when you're forced to.

Recursion is very simple actually. You just need to remember the two rules:

1. It must have an ending point.
2. It must make the problem simpler.

Live Demonstration

Slide 3 *scope/scope.cpp*

Slide 6 *perm/perm.cpp*

Slide 25 *length/length.cpp*

Classroom Presentation Suggestions

Use the debugger to step through the program on Slide 3 and set watch points for the variables `global`, `local`, and `very_local`. If you do this right, you will start out with our "watch" window looking like:

```
global          0
local           <variable out of scope>
very_local      <variable out of scope>
```

(Some debuggers don't allow you to insert a watch for a variable that is not currently active. In this case you need to single-step to the middle of the program, set watches on all the variables and restart. This will prepare your program for the students.)

Step through the program on Slide 6. Emphasize the fact that the statement

```
int temporary = 1;
```

constructs and initializes the variable. This will come in handy when classes are discussed. Also, the variable is destroyed at the end of the block. Using the words "construction" and "destruction" will help relate this material to the class constructor and destructor member functions.

When showing the sample "triangle" function on Slide 12, ask the students if they can spot any other function on the page. The other function is called `main` and, except for its name, is an ordinary function.

Single-step through the "triangle" program on Slide 12 and relate each of the actions to the items listed in Slide 13. C++ does a lot behind people's back, so get the students used to the idea that the compiler does all sort of things that are not obvious from the code.

You might want to point out that other languages have functions (which return values) and subroutines or procedures (which do not). C++ makes a single "function" syntax do the work of both.

Call by Value Demonstration

The general idea is to have the members of the class execute by hand a simple program to compute the area of a rectangle. Assign one student the job of `funct` function. This "code" is:

```
void funct(int j) {
     j++;
}
```

Have the student write down his code on the right side of the board.

Another student is given the job of `main`. His code is:

```
int main()
{
    int my_j = 20;

    func(j);
    cout << “my_j is “ << my_j<< ‘\n’;
    return (0);
}
```

The `main` student hand executes his code. When he comes to the `funct` call he should write 20 on a slip of paper and pass it to the funct student.

Point out that the information is passed one way from `main` to `funct`. Once the `funct` student gets the note, he can do anything with it. He can alter the values, eat it, anything. The only thing he can’t do with it is pass it back to the `main` student.

Reference Values

Set up the `main` and `funct` students as before. The `area` student’s code is now:

```
int area(int &j) {
     j++;
}
```

With references the parameter passing is different. Have the `main` student create two boxes labeled `my_width` and `my_height` in the middle of the board. The labels should be on the left toward the `main` side of the board.

When the call is made, the `area` student puts the labels `width` and `height` on the right side of the boxes. This means that each box will have two names. There is only one box.

It should be noted that if the function `funct` changes anything in the box, the change will be reflected in the `main` program. That’s because there is only one box with two labels.

The process of adding a label `j` to the box already labeled `my_j` is called binding and is automatically performed by C++ during a function call.

Parameter Type Summary

C++ has a lot of different ways to pass parameters. Slide 18 lists the various types. At this point lots of examples are useful.

One question that keeps cropping up: “Aren’t reference parameters just like C pointer variables?”

Answer: There are two differences: Pointer variables can point to an array or a single variable. You can’t tell which from inside the function. Reference parameters can only reference one variable. Secondly, we’ve studied reference parameters, and we haven’t gotten to pointer variables yet.

Programming Techniques

There are many different programming techniques. Which one you use depends on your experience, and the type of problem you have to solve. This section is designed to give the students a little introduction to various techniques.

One way of showing how the various techniques work is to design and implement a program line. Explain how you are analyzing the problem, designing the code, and implementing the solution.

Recursion

Emphasize the two rules of recursion. There are lots of problems that are normally solved with loops that can be solved with recursion.

For example, summing the elements in an array. The book contains a recursive algorithm to do this.

Review Questions

1. Define “Variable Scope.”

   The area of the program where the variable is valid.

2. Define “Variable Storage Class.”

   A Variable’s Storage Class defines how long the variable exists. It may be permanent or temporary.

3. Why are hidden variables a bad idea?

   They create confusion and readability problems.

4. What is the scope of a function’s parameters?

   The scope of function parameters are limited to the body of the function.

5. What is the class of a function’s parameters?

   Temporary.

6. The keyword **static** changes a local variable’s storage class from __________ to __________.

   “Temporary” to “permanent.”

7. Why are all global variables permanent?

   A better question might be: “How could we possibly make them temporary?” Temporary variables are created and destroyed as needed. Global variables are always “needed” so there is no way of creating and destroying them.

8. When is the following variable created, initialized, and destroyed?

```
int funct(void) {
    int var = 0     // The variable in question
    // ......
```

   The variable is created when the function is called. It is initialized every time the function starts up and it is destroyed at the end of the function. It is a temporary variable.

9. When is the following variable created, initialized, and destroyed?

```
int funct(void) {
    static int var = 0     // The variable in question
    // ......
```

The variable is created when the program starts. It is initialized once when the program begins and it is destroyed when the program is done. It is a permanent variable.

10. When is the following variable created, initialized, and destroyed?

```
int var = 0     // The variable in question
int funct(void) {
    // ......
```

The variable is created when the program starts. It is initialized once when the program begins and it is destroyed when the program is done. It is a permanent variable.

11. Define **namespace**.

A namespace is a way of collecting similar symbols together in a group. Typically all the symbols for a module, library, or other unit of programming will have the same namespace.

12. What is the name of the namespace we've already used in this book for **cin** and **cout**?

std.

13. Define “reference.”

A reference is a variable that is really another name or alias for an existing variable.

14. What is binding and when does it occur?

Binding is the act of associating a reference with the variable that it aliases.

Binding of reference variables occurs when they are declared. They must be declared with an initialization statement. Reference parameters are bound when the function is called. References may also be used for the return type of functions. In this case the return statement does the binding.

15. What is the difference, if any, between the type of values returned by the following two functions:

```
int func1(void)
const int funct2(void)
```

In actual practice there is no difference between the two declarations.

16. What is the difference, if any, between the type of values returned by the following two functions:

```
int &fun1(void)
const int &fun2(void)
```

The first one returns a reference to an integer variable (or array element) that can be modified. The second one returns a reference to an integer that cannot be modified.

17. Which parameters in the following function can be modified inside the function:

```
void fun(int one, const int two, int &three, const int
&four);
```

Parameters "one" and "three" can be modified.

18. In the previous question, which parameters, when changed, will result in changes made to the caller's variables?

If you change parameter "three" the results will be passed back to the caller.

19. Which parameters in the following function can be modified inside the function:

```
void fun(int one[], const int two[]);
```

Parameter "one" can be modified inside the function.

20. In the previous question, which parameters, when changed, will result in changes made to the caller's variables?

Changes made to parameter "one" will be passed back to the caller.

21. Define "Dangling Reference."

A reference to a variable that has been destroyed.

22. Why are dangling references a bad thing?

Because the reference is to something that doesn't exist, using it can cause unexpected trouble.

23. Can we overload the square function using the following two function definitions?

```
int square(int value);
float square(float value);
```

Yes.

24. Can we overload the square function using the following two function definitions?

```
int square(int value);
float square(int value);
```

No. The parameters to the function must be different in order to overload it. In this case both functions take a single integer as a parameter.

25. Define "default parameters."

Parameters whose value you don't have to define when you call the function. If you don't define them, then default values are used.

26. Why must default parameters occur at the end of a parameter list?

C++ has no way of knowing parameters are missing from the beginning or middle of a parameter list. (Page 136.)

27. What does the keyword **inline** do?

It recommends to the compiler that the code for this function be generated "inline" without the overhead of the code needed to call a function and return from it. It is used only for short functions since the entire body of the function is placed **inline** where the function is called.

28. What programming technique was used to solve the "calculator problem" of Chapter 7, *The Programming Process*?

"Top down programming." Mostly.

29. Define "Structured Programming."

A programing technique where you define a basic structure for your program, and then refine it until you get a program.

30. Describe "Top down programming."

Another term for "structured programming."

31. Describe "Bottom up programming."

A programming technique where you start by building small, well tested functions, then combine them in higher level functions, until you have a working program.

32. Define "recursion."

In programming, this means that a function calls itself. A more general term is something that is defined in terms of itself.

33. What are the two rules you must follow to make a recursive function?

1) There must be an ending point. 2) Each stage of the function must make the problem simpler.

Chapter 10: *The C++ Preprocessor*

The speech of man is like embroidered tapestries, since like them this has to be extended in order to display its patterns, but when it is rolled up it conceals and distorts them.
— Themistocles

Teacher's Notes

The preprocessor is just one specialized text editor. The key to learning it is the fact that it is a very simple program whose syntax is completely different from C++'s syntax.

This chapter contains the greatest number of mixed up programs per page of any chapter. Preprocessor errors are more difficult to detect than normal errors. It's also easier to make errors that will slip past the compiler. For that reason this chapter contains many broken programs.

The rules developed at the end of this chapter were created to help eliminate most of the risk in using the preprocessor. By using them you will avoid many of the C++ preprocessor's surprises.

Classroom Presentation Suggestions

Most of the material presented in this chapter can be summarized as:

- This is a preprocessor directive
- This is what it generates

Sometimes what's generated is not what's expected.

To present this material, run the sample programs through the preprocessor and show the two version (before and after) to the class.

Review Questions

1. Define **#define**.

   <u>The #define statement is used to create, or define a preprocessor macro</u>.

2. What is the difference between a simple **#define** macro and a **const** declaration?

   <u>The **#define** statement defines a text replacement. For example `#define FOO 1+2` defines FOO as literally "`1+2`", not `3`. This makes #define statements tricky to use. The const statement is easier to use and has better error checking.</u>

3. When would you use conditional compilation?

One example is when you want to have a debug and production version of your program. Conditional compilation can also allow you to have a crippled demonstration version and a full-featured production version.

4. 4. Would you ever use conditional compilation for something other than debugging?

See previous answer. Also you might want to put machine dependences in conditional sections.

5. What is the difference between a parameterized macro and a simple macro?

A parameterized macro takes arguments, a simple macro does not.

6. What is the difference between a macro and a normal function?

A macro is just a text substitution command to the preprocessor. A function is a set of instructions to be executed by the program.

7. What is the difference between a macro and an **inline** function?

Both are expanded **inline**, however a macro is expanded by the preprocessor and the **inline** function by the compiler. This means that the **inline** function uses the C++ syntax and has better error checking. The "**inline**" keyword is also a suggestion so that the compiler may generate a real function if needed.

8. Can the C++ preprocessor be run on things other than C++ code? If so what?

Yes. After all, it's just a specialized text editor. Some non-C++ things include generating makefiles from templates (the *imake* does this), using the preprocessor on assembly code and other things.

9. Why do we not want to use macros to define things like:

```
#define BEGIN {
#define END }
```

This makes our C++ code look like PASCAL. C++ code is a whole lot easier to understand and debug if it looks like C++ code.

Chapter 11: *Bit Operations*

To be or not to be, that is the question
— Shakespeare on boolean algebra

Teacher's Notes

This chapter presents the various bit operators. This chapter is a complete unit in itself and can be safely skipped.

Bitmapped operators have limited uses. For example, they are hardly used at all for business programming. For people doing I/O drivers, graphics, and other low level programming, bit operators are extremely valuable.

There are two practical uses for bit operators. First, we show how to put eight single bit flags in one byte. Secondly, we demonstrate simple bitmapped graphics.

You will probably want to augment the material in this chapter with examples relevant to your class.

Live Demonstration

Slide 6 *and/and.cpp*

Slide 19 *graph/graph.cpp*

Slide 21 *loop/loop.cpp*

Classroom Presentation Suggestions

The basic operators are fairly straight forward, except for the right shift operator. The value for the sign bit changes depending on the type of the data. This is shown by the table on Slide 10.

Beginning with Slide 14 we show how to define, set, test, and clear single bits in a byte. At this point, it might be useful to put a couple of bytes on the board and show how the operators affect them.

On Slide 15 we demonstrate how bit operators are useful in creating bitmapped graphics. If possible, connect this with real bitmapped graphic devices available to the class.

Review Questions

10. Describe the difference between bitwise and (&) and logical and (&&). Give an example of each.

    Bitwise and (&) works on each bit in the two operands independently. Logical and (&&) works on the two operands as a whole. While the result of bitwise and can be any number, the result of a logical and must be a 0 or 1.

```
bit_result = 0x1234 & 0xAAAA; // Bitwise and
logical_result = (a < b) && (c < d); // Logical and
```

11. Describe the difference between bitwise or (|) and logical or (||).

Bitwise or (|) works on each bit in the two operands independently. Logical or (||) works on the two operands as a whole. While the result of bitwise or can be any number, the result of a logical or must be a 0 or 1.

```
bit_result = 0x1234 | 0xAAAA; // Bitwise or
logical_result = (a < b) || (c < d); // Logical or
```

12. If you didn't have an *exclusive or* operator (^) how would you create one out of the other operators?

```
unsigned int exclusive_or(const int i1, const int i2) {
    return ((i1 | i2) & ~(i1 & i2));
}
```

In other words, take the bitwise or of both numbers, then remove the bits that are set in both numbers. (Bitwise and is used to find these numbers.)

13. How would you test to see if any one of a set of bits is on? (This can be done without using the logical and (&&) operation.)

When all the bits are clear, the number is zero. So we can test to see if any bit is set by using the code:

```
any_set = data != 0;
```

14. How would we rewrite the `set_bit` function if we used an array of **short int** instead of an array of **char**.

```
// Assumes 16 bit short integers
unsigned short int graphic[X_SIZE / 16][Y_SIZE];
//...
inline void set_bit(const int x, const int y)
{
    graphcs[x/16][y] |= 0x8000 >> (x % 16);
}
```

Chapter 12: *Advanced Types*

Total grandeur of a total edifice,
Chosen by an inquisitor of structures.
— Wallace Stevens

Teacher’s Notes

Structures and unions represent the most advanced types available to the C programmer. A structure is a collection of data. Add in the functions that operate on that data and you get a C++ class. As a matter of fact, some C++ compilers consider a structure to be a very limited form of class.

Classroom Presentation Suggestions

A *structure* can be thought of as a box subdivided into many different named compartments. A *union* is a box with many different labels slapped on a single compartment.

For the graphics example, if possible present the students with a program to draw a bitmapped graphic line on a graphics device available to the class.

Review Questions

1. Define Structure.

   A data type that can hold several different values. Unlike an array, these values don’t have to have the same type. Each item or field in the data structure has it own name.

2. Define Union.

   A data type similar to a structure. But unlike a structure where each field has its own storage, in a union all the fields occupy the same memory.

3. How can we tell what type of data is currently stored in a union?

   There is no way to do this with just the information in the union. One way of getting around this restriction is to keep type information around in another variable.

4. Define **typedef**.

   A statement much like a variable declaration, but instead of define a new variable it uses the typedef keyword and defines a new data type.

5. Which is better **typedef** or **#define?** Why?

The **#define** statement is a preprocessor statement. The preprocessor uses a different syntax from the C++ language. Therefore using **#define** prevents the C++ compiler from doing syntax checking on the statement.

C++ knows about **typedef** statements and can check them to make sure their syntax is correct. Also consider the following:

```
#define CHAR_PTR char *
typedef char *char_ptr;

CHAR_PTR ptr1, ptr2;
char_ptr ptr3, ptr4;
```

This obviously declares four pointer variables. Unfortunately, the obvious is wrong. The variable “ptr2” is a character, not a character pointer. That’s because the line it is declared, when expanded, look like:

```
char * ptr2, ptr2;
```

6. Define **enum** type.

A data type for variables that hold a limited number of named values. Each of these values is listed (or enumerated) in the **enum** statement that defines the type.

7. List two reasons that **enum** types are better than defining a series of constants.

The **enum** statement automatically assigns values to the various values for the enumerated type. Also the C++ compiler will prevent the programmer from assigning the wrong enum value to the wrong variable.

8. Define “bit field.”

Also known as a packed structure. In this type of structure we limit the size of the fields so that multiple fields can be packed into a single word.

9. What are the advantages and disadvantages of using bit fields.

Advantage: Much more compact. Disadvantage: Packed structures are slower than normal structures. They are also more difficult to construct since the programer must define the number of bits to use for each field.

Chapter 13: *Simple Classes*

She thinks that even up in heaven
Her class lies late and snores,
— Cyril Connolly

Teacher's Notes

In this chapter we define a simple *stack*. The first version uses procedures and a structure.

The second version uses a **class**. The class version of the stack is very similar to the procedure/structure version of the stack, except that the procedures (member functions) and structures are integrated. That means that you don't have to pass the structure as the first parameter to each procedure.

Classes are also known as objects. C++ makes object design easier by providing the programmer with the ability to control how the class is created, destroyed, and copied. This is done through the use of the constructor, destructor, and copy constructor.

Remember the big four functions:

- The default constructor
- The destructor
- The copy constructor
- The assignment operator

These functions will automatically be called by C++ as needed. They will also be automatically generated if the programmer does not specify them.

As we will see at the end of the chapter, it's work to force C++ not to generate them.

The big four are the key to object management. Students must know when these four procedures are called automatically.

Live Demonstration

Slide 278 *stack_c/stack_c2.cpp*

Classroom Presentation Suggestions

When defining the stack, put a sample use on the board. Push a few numbers on the stack by writing them on the board. Pop them off and erase the numbers as they are popped.

The chapter is designed to relate the *procedure* version of the stack to the *class* version.

Slide 284 shows the various member functions that are automatically called by C++. It's important to let the students know what goes on behind their back.

Slide 285 lists the "big four." This is the list of the member functions that are automatically generated and automatically called. Every class should define these functions or explicitly tell the user that the default is to be used.

Review Questions

1. Define "class"

   Data and the operations that are performed on it. A class is like a structure except that it has added functions and access protection.

2. Define the "big four" member functions.

   Constructor, Destructor, Copy Constructor, Assignment operator.

3. What are the hidden functions called in the following code:

```
void use_stack(stack &local_stack)
{
    local_stack.push(9);
    local_stack.push(10);
}
```

Answer:

```
int main()
{
    stack a_stack;     // Generate a default stack

    a_stack.push(1);
    a_stack.push(2);

    use_stack(a_stack);

    cout << a_stack.pop() << '\n';
    return (0);
}
Be careful, this is not the same code as presented in the
slides.
void use_stack(stack &local_stack)
{
    local_stack.push(9);
    local_stack.push(10);
}


int main()
{
    stack a_stack;     // Generate a default stack

    a_stack.push(1);
    a_stack.push(2);
```

```
        use_stack(a_stack);

        cout << a_stack.pop() << '\n';
        return (0);
    }
```

4. Why can you overload the constructor and not overload the destructor?

   How would you call it? Since there is no way to explicitly call the destructor, there is no way you can pass it parameters.

5. Which of the big four are generated automatically?

   All of them: default constructor, destructor, copy constructor, and assignment operator.

6. How do you prevent each of them from being generated automatically if you don't explicitly define one.?

   Default constructor -- Declare any parametrized constructor.

   Destructor -- Every class must have a destructor, so there is no way to prevent one form being automatically generated.

   Copy constructor -- Declare it as a private member function.

   Assignment operator -- Declare it as a private member function.

Chapter 14: *More on Classes*

This method is, to define as the number of a class the class of all classes similar to the given class.

— Bertand Russell
Principles of Mathematics
part II, chapter 11, section iii, 1903

Teacher's Notes

This chapter describes some miscellaneous features. Each of these features pretty much stand on their own.

Review Questions

7. Define "friend."

   A friend of a class is a function or class that has access to the private data of that class.

8. Describe an example when you would use a "friend" function.

   One example would be if you wanted to create a function to compare two classes. Another example you will see later is that the linked list class needs access to the private data of the linked list element class.

9. Define "constant member functions."

   Member functions in a class that can be called even in a constant instance of the class.

10. Define "constant member variable."

    A data member of a class that is initialized at construction time and cannot be changed during the lifetime of the class.

11. How do you initialize the constant member variables of a class?

    The initialization of constant data members comes after the function declaration and before the first curly brace. For example:

```
public:
    stack(void) : stack_size(1024) {
```

12. Define "static" member variables.

    A member variable that is created once for the class instead of being created once per instance of the class. A static member belongs to the class, not a instance of that class.

13. Why are you able to access static member variables using the `class::var` syntax you cannot use the same syntax for ordinary member variables?

    Since the static member variable belongs to the class, you can access it by using the `class::var` notation. Ordinary member variables belong to a specific instance of the class. If you tried to access them using the class::var notation, C++ would be unable to determine which instance they belonged to.

14. Define "static member function."

    A function that is declared static and can only access the static data members of a class.

15. Why are static member functions prohibited from accessing ordinary member variables?

    Static member functions belong to the class, not a specific instance of the class. If they tried to access an ordinary member variable, C++ wouldn't know which instance of the class to use.

16. Why can they access static member variables?

    Because static member variables are not associated with a instance of the class so C++ doesn't have to figure out which instance to use.

Chapter 15: *Simple Pointers*

The choice of a point of view is the initial act of culture.
— Ortega y Gasset

Teacher's Notes

"There are things and there are pointers to things." That's probably the second most important thing you can teach your students. (The first is the difference between = and ==.)

In this chapter you will ask your students the same question over and over again: "What is 'x'? Is it a horse? Is it an apple? No! It's a pointer. So we must do blah blah to it so ..."

Also you will ask "What is 'y'? Is it a horse? Is is an apple? No! It's a thing, so we can ..."

Teaching students the difference between a thing and a pointer to thing is the first step to understanding complex data structures. We'll hit these a little later on.

C++ blurs the distinction between things and pointers to things when we talk about arrays. This is because C++ will automatically turn an array into a pointer and a pointer into an array. This can cause confusion.

This automatic changing is also why array parameters are treated differently in C++. See Table 9-2 for a listing of parameter types.

Live Demonstration

Slide 317 *split/split.cpp*

Slide 318 *tmp-name/tmp-name.cpp*

Slide 322 *print/print.cpp*

Classroom Presentation Suggestions

Always put `_ptr` at the end of your pointer variables in all examples.

Slide 10

Go through this program line by line and draw on the board a graphic representation of what's happening.

Slide 13

Suggested dialog:

Teacher: "What is `first_ptr`?"

Student: “A pointer to **const** data.”

Teacher: “Can we change it?”

Student: “Yes: It’s a pointer.”

Teacher: “What does it point to?”

Student: “A **const** character.”

Teacher: “Can we change that?”

Student: “No. It’s a constant.”

Repeat this series of questions for `second_ptr` and `third_ptr`.

Slide 24

The problem with this program is documented in the book. What’s different about this question is that after you fix it, you can demonstrate another pointer problem by trying to allocate two temporary names. See the book for details.

Slide 28

If you are on a UNIX system, the shell expands wildcards (*, ?, etc.). You can show this using the echo command. DOS does not expand wildcards.

Review Questions

1. Define “pointer.”

   Pointers, or address variables contain the address of other variables. In other words, they point to data. (Page 221.)

2. How is a pointer different from a thing?

   A thing (structure, class, or simple variable) can be large or small. It can hold one item or many. Pointers come in one size.

3. How many pointers can point to a single thing?

   As many as you want.

4. What happens if you assign a pointer NULL and then try and dereference it?

   You get garbage or crash your program. On DOS/Windows systems, 0 is a legal address and you can get the value of the data (an interrupt vector) that’s stored there. On most UNIX systems it will cause a “segmentation violation” error which will crash your program.

   Try it on your system and see what happens.

5. Are pointers more efficient than arrays?

   My opinion, in most cases with a smart compiler, no. In some cases it is more efficient to use pointers, but not many.

6. Are pointers more risky to use than arrays?

    In my opinion, yes. It is very easy to assign the wrong value to a pointer and to trash memory. It is harder to assign the wrong value to an array index and trash memory.

7. Which of the preceding two questions is the more important?

    I believe that a working program is better than a fast program, so I favor limiting risk at the expense of speed.

Chapter 16: *File Input/Output*

I the heir of all the ages, in the foremost files of time.
— Tennyson

Teacher's Notes

We are now ready to explore how to read and write files. About half of this chapter is devoted to explaining some of the I/O functions and classes.

The end-of-line puzzle is extensively discussed. The Macintosh, the PC, and UNIX all use a different end of line scheme. Since C was first implemented on UNIX, all I/O systems must edit their character input so the end of line conforms to the UNIX "standard." The editing is detailed in

Table 16- 1 I/O Editing

	UNIX	**Windows MS-DOS**	**Apple**
End-of-Line	<line feed>	<carriage return> <line feed>	<carriage return>
Input Trans.	None	Delete <carriage return>	Change <carriage return> to <line fed>
Output Trans.	None	Insert <carriage return>	Change <line feed> to <carriage return>

On UNIX a read is a read. On DOS, a read edits the input if it's an ASCII file and doesn't if it's a binary file. Same thing on the Apple systems. Be aware of this problem.

Finally there is a section on file design.

This chapter by no means covers all the things that one can do with a file. It is designed to give the student a good base, but leaves out things like advanced formatting and random access. These can be learned from reading the reference manual or taking an advanced programming class.

Live Demonstrations

Slide 5 *read/read.cpp*

Slide 28 *copy2/copy2.cpp*

Slide 34 *two/two.cpp*

Classroom Presentation Suggestion

This chapter consists mostly of short definitions and small examples. These pretty much stand by themselves.

The sample file presented on Slide 30 is pretty bare. The idea is that as you refine the file structure you write the additional information into the slide.

Review Questions

1. There are three different I/O packages described in this chapter: C++ streams, raw I/O, and C standard I/O. Describe the advantages and disadvantages of each.

   C++ I/O streams library. Advantages: Much better type checking than any other system. Disadvantages: Changing the size, precision, or format of the output requires outputting of clunky I/O manipulators or the calling of special member functions.

   Raw I/O system. Advantages: Able to handle large amount of raw I/O in a single bound. Disadvantages: Very slow for small amounts of data and no formatting capability.

   C Style `printf/scanf`. Advantages: Able to do string I/O well (`std::sscanf`). Disadvantages: You can easily get the parameter types and the conversion characters mixed up resulting in strange output. The end of line handling in the "`fscanf`" call is so poor that this function is almost impossible to use.

2. Define `std::cin`, `std::cout`, and `std::cerr`.

   `std::cin` -- console input. `std::cout` -- console output. `std::cerr` -- console error log. These three I/O streams are automatically opened for every program.

3. Define `ifstream`.

   The class that's used for defining a file reading variable.

4. Define `ofstream`.

   The class that's used for defining a file writing variable.

5. How can we tell if a write worked?

   For the `std::ofstream` and `std::ostream` classes, if the member function `bad` returns true (non-zero), the write failed.

6. How can we tell if a file exists?

   Try to open it with an `std::ifstream`. If the member function `bad` returns true (non-zero), then the file does not exist or cannot be read.

   Note: There is a standard function called `access` that checks for the existence of a file. This function has not been covered in the book.

7. What's the difference between `std::getline` and the >> string operation?

   The `std::getline` member function reads an entire line. The >> string operation reads a string up to the first whitespace character.

8. What does `sizeof` do?

   Returns the number of bytes in its argument.

9. Write a C++ program fragment to write out a variable containing the value "3" as "0003".

```
std::cout << std::setw(4) << std::setfill('0') << 3 << '\n';
// Clean up the code for those to come
std::cout << std::setw(0) << std::setfill(' ');
```

10. Why are graphic images almost always stored as binary files?

    Graphics files contain large amounts of data. Binary files are more compact than ASCII files.

11. One type of image that's stored as an ASCII file is an X icon. Why is it ASCII?

    The X Windows system is designed to be portable and ASCII files are portable. Although X icons are graphics, they are small graphics so the ASCII doesn't take up that much more additional space.

12. What's the difference between 0 and '0'?

    The first 0 is a number. The second '0' is a character.

13. What type of end of line is used on your machine?

    Apple, carriage return (\r). DOS/Window carriage return/line feed (\r\n). UNIX line feed (\n).

14. Someone took a text file from DOS and moved it to UNIX. When they tried to edit it a lot of control-M's (^M) appeared at the end of each line. Why?

    Control ^M is the carriage return character. DOS files uses carriage return/ line feed to end lines. UNIX uses just line feed, so it thinks that the carriage return is part of the file and it shows up at the end of the line as ^M.

15. Why is buffering a good idea?

    Operating system calls take time. If we had to call the operating system for every character output, our program would be very slow.

16. When is it not a good idea?

    When dealing with large amounts of data.

17. Why is the C style `std::printf` more risky to use than the C++ style `std::cout`?

    C++ automatically checks the variable types when using a `std::cout` statement. It does not do so for a `std::printf` statement.

Chapter 17: *Debugging and Optimization*

Bloody instructions which, being learned, return to plague the inventor.
— Shakespeare on debugging.

Teacher's Notes

Everyone has their own way of debugging. Some people are good at sitting down and analyzing their programs while others need to use the interactive debugger. In this chapter we present the students with a variety of techniques. Let them decide which ones are useful.

One point should be emphasized, however: It is a good idea to design in debug code.

With the advent of faster processors optimization is growing less and less important. We describe some simple optimization techniques in this chapter, enough to help the student avoid doing anything really stupid in his program.

Live Demonstrations

Slide 4 *seven/seven.cpp*

Slide 7 *search/search0.cpp*

Slide 13 *search/search1.cpp*

Slide 15 *search/search2.cpp*

Slide 18 s*earch/search3.cpp*

Slide 21 *debug/flush.cpp*

Slide 22 *debug/flush2.cpp*

Classroom Presentation Suggestions

The debugging section is designed to be done interactively. We've presented a very broken program and document the path needed to fix all the errors. Slides of the debugging sessions have been produced in case you can't run the debugger interactively.

Review Questions

1. Describe useful debugging aides that might be built into a program.
2. Describe your hardest bug and what you did to fix it.
3. What happens when you try to divide by 0?

   This is somewhat operating system dependent and sometimes compiler dependent. Try it. On UNIX systems you get a message “Floating exception” (even for an integer divide!)

4. What happens when you attempt to dereference the NULL pointer?

   On some systems, nothing. On others you get an error. Most UNIX systems will write the message “Segmentation Violation” and abort the program.

5. What happens when you modify the data pointed to by the NULL pointer?

   Same thing.

6. What happens when you write a program that does infinite recursion?

   You get a stack overflow. It is important to turn on stack checking for the DOS/Windows compilers or this problem may not be caught.

7. Define `std::flush`..

   A function of the class `std::ostream` that forces the output buffer to be emptied.

8. Define “Confessional Method of Debugging.”

   A method of debugging where the programmer explains his program.

9. Define “Loop ordering.”

   Changing the order in which statements in a series of loops are executed to reduce the computations involved.

10. Define “Reduction in Strength.”

    The art of substituting cheap operations for expensive ones.

11. Write a program that divides an integer variable by 10 a thousand times and time how long it takes. How long does it take to divide the variable by 16?

Chapter 18: *Operator Overloading*

Overloaded, undermanned, ment to flounder, we
Euchred God Almighty's storm, bluffed the Eternal Sea!
— Kipling

Teacher's Notes

One of C++'s great features is its extensibility. As we've seen in the section on classes, you can define objects and have them manage their own data structures. Now we learn how to define how the various C++ operators that work on these data structures.

For this chapter we present a fixed point class. This class was chosen because it's simple and it demonstrates all of the simple operator overloading functions.

One problem with operator overloading: "diarrhea of the definition." Once you start overloading, you will need to define a complete set of operators. That's a lot of operators as we can see by the size of our complex data type.

After presenting the slides, ask your students to list on the board all the various operators that should be overloaded for a fixed point class. (fixed_pt + fixed_pt, fixed_pt + double, double + fixed_pt, fixed_pt + integer, integer + fixed_pt, ...) -- and that's just for the addition operator. See how many you can come up with.

Live Demonstration

Style 20 *equal/equal.cpp*

Review Questions

1. Define "overloading."

   Using the same function or operator for more than one purpose. Two functions with the same name but different parameters are said to overload the function name.

2. Define "operator overloading."

   Defining procedures for C++ operators to handle additional data types.

3. What's the difference between x++ and ++x?

   The first one increments a variable and returns the value of the variable before incrementation. The second increments a variable and returns the value of the variable after incrementing. The first is more expensive than the second to perform.

4. How do you tell them apart when defining the operator functions?

   The postfix version of the increment (x++) takes two parameters. The second is an dummy integer.

5. What is casting?

   An operation that changes the type of one variable to another.

6. Why should you not overload the casting operators? What should you use instead? Why?

   By defining casting operators you let C++ automatically change the type of a class to something else when needed. The trouble is that C++ may do this when you don’t want it done.

   You can define an ordinary member function to do the conversion. Then you can explicitly control the conversion.

Chapter 19: *Floating Point*

1 is equal to 2 for sufficiently large values of 1.
— Anonymous.

.

Teacher's Notes

I feel that floating point calculations are one of the most overlooked parts of programming. Few if any programming books talk about floating point errors. In order to properly use floating point you need to know the basics of how floating point arithmetic is done as well as understanding its limitations.

This chapter only covers the basics. There is an entire science, Numerical Analysis, devoted to studying the behavior of numeric calculations and devising methods to minimize the impact of floating point errors.

This chapter does serve as a brief introduction to the subject.

Classroom Presentation Suggestions

Lock the computer away and do everything on the board.

Review Questions

1. Define "Overflow."

   <u>Overflow occurs when the results of an operation are too big for the number format to handle.</u>

2. Can you have overflow with integer calculations?

   <u>Yes. Just add "maxint" + "maxint" on your system.</u>

3. Define "Underflow."

   <u>Underflow occurs when the result of a calculation is too close to zero to be represented by the number format used.</u>

4. Can you have underflow in integer calculations?

   <u>No. The closest that an integer calculation can get to zero (and not be zero) is 1. This can easily be represented by the integer format.</u>

5. What's the biggest floating point number that can be represented on your machine.
6. On your pocket calculator what does 1-(1/3)-(1/3)-(1/3) equal?
7. Why do you never want to use floating point for money?

   <u>Floating point is not exact. The IRS is.</u>

Chapter 20: *Advanced Pointers*

A race that binds
Its body in chains and calls them Liberty,
And calls each fresh link progress.
— Robert Buchanan

Teacher's Notes

Pointers along with the **new** and **delete** operators allow the C++ program to design complex and intricate data structures. This chapter covers some of the basic data structures such as linked list, double linked list, and trees. This is merely an introduction; a full discussion of data structures can easily take an entire course.

Live Demonstration

Slide 21 *words/words.cpp*

Classroom Presentation Suggestions

On Slide 6 we ask "What would happen if we didn't **free** the memory?" The answer is a memory leak. This is a major problem with C++ programs. As someone once put it, "C++ was designed to leak memory."'

Trees are actually a very simple data structure once you understand them. Slide 17 can be used to demonstrate by hand how to find an element in the tree as well as how to insert an element into the tree.

Printing a tree is the easiest of all. Most people when confronted with printing a tree won't see the algorithm. But it's simple and it works.

Review Questions

1. Define "Linked List."

   A data structure consisting of a list of nodes. Each node has a pointer, or link that points to the next node in the list. See Figure 20-3.

2. What would you use a linked list for?

   It is useful for the storage of numbers or other data when you don't know how many elements you are going to need ahead of time, and you will be accessing the data in order. One example is the line numbers for a symbol in a cross reference program.

3. Define “Double-linked List.”

   A linked list with both forward and backwards links.

4. What can a double-linked list be used for that a linked list cannot.

   One example is the storage of points in a square, polygon, or other graphic object. This structure allows you to define a “current point” and move forward or backward in the graphic easily.

5. Define “binary tree.”

   A data structure that starts with a top level “root” node and then branches out into two sub-nodes.

6. What are trees good for?

   Symbol tables.

7. How would you delete an element from a tree?

   It’s not easy. See the tree.cpp program on the answer disk.

Chapter 21: *Advanced Classes*

The ruling ideas of each age have ever been the ideas of its ruling class.
— Karl Marx
Manifesto of the Communist Party

Teacher's Notes

Derived classes are the base of object-oriented design. They allow you to define a base class and extend it in a way that still allows you to use it as the base class.

One of the key points is that when you pass a derived class to a function that uses a base class as a parameter (Slide 5) the derived class is treated as if it were a base class.

The reason that the program on Slide 21 fails is because it calls a virtual function. Remember that the destructors are called in the order derived, base. So the derived destructor is called -- the derived calls is gone – the base class destructor is called – it calls the virtual function clear – the derived class is long gone – so the system fails. Lead the students to the answer on this one by talking about the destructor calling order.

Live Demonstrations

Slide 8 *virt/virt.cpp*

Slide 21 *blow/blow.cpp*

Review Questions

1. Define "derived class."

   A class made by building onto an existing class.

2. Define "base class."

   The class that a derived class builds on.

3. Can a class be both derived and base?

   Yes. Consider the following class structure:

```
class a {};
class b : public a {};
class c : public b {};
```

Class “b” is derived from “a.” It is a base class for “c.”

4. Define “virtual function.”

   A member function in a base class that may be overridden by a derived class. In other words, when the member function is called in the base class, C++ will execute the derived class’s version of the function if present.

5. Define “pure virtual function.”

   A base class member function that must be defined in the derived class.

6. Describe how to implement virtual functions using function pointers. (This is what C++ does internally.)

   The code for the base class would look like:

```
class base {
    public:
        void (*funct_ptr)(void); // Pointer to a function

         void base_version_of_funct(void) {...};

        base(void) {
            funct_ptr = base_version_of_funct;
        }
        void call_funct(void) {
            (*funct_ptr)();
        }
};

class derived {
    public:
        void derived_version_of_funct(void) {...}

        derived(void) {
            // ... base gets constructed first
            // Now override the function pointer
            funct_ptr = derived_version_of_funct;
         }
};
```

7. Define “abstract class.”

   A class with one or more pure virtual functions in it.

8. Why can’t you use an abstract class to define a variable?

   The pure virtual functions say that to C++, “these functions must be defined by the derived class.” When you use an abstract class to define a variable, you tell C++, “This is the variable, there is no derived class.” C++ gets upset and asks, “so who is going to define the pure virtual functions then?”

Chapter 22: *Exceptions*

How glorious it is — and also how painful — to be an exception.
— Alfred De Musset

Teacher's Notes

The old C way of handling an error was to have each function return an error code to the caller. The caller had to check each function call and return an error code up to the next level.

With exceptions, the error handling interface is much easier. The C++ **throw** command creates an expection that's handled by the **catch** block. The nice thing only the functions that does the **throw** and the one that catches it need to worry about the exception. All the other functions on the call stack don't have to worry about problems and don't have to pass error codes up the call stack.

Note: Because C++ was build on C, there's still a lot of C style error handling built into th esystem. (For example, the raw read and write calls return error codes not exceptions.)

Review Questions

1. Define "exception."

   <u>An unusual emergency type programming condition that must be handled outside the normal flow of the program.</u>

2. Define "try."

   <u>A C++ keyword indicating a block of code in which an exception might occur.</u>

3. Define "catch."

   <u>A C++ keyword indicating a section of code designed to handle an exception.</u>

4. Define "throw."

   <u>A C++ keyword used to cause an exception.</u>

Chapter 23: *Modular Programming*

Many hands make light work.

— John Heywood

Teacher's Notes

Our programs have now grown large enough so that it's time to divide them up into modules. This chapter will teach the students how to divide up the work into smaller pieces.

Live Demonstration

Whole chapter: *ia/<all>*

Review Questions

1. Define "header file."

   Also called an "include file." This is a file that contains shared information that is designed to be included in multiple code files.

2. What should go in a header file? Why?

   Constants, data structures, class definitions, inline procedures, extern specifications. Why? Because these elements are shared between modules in a program.

3. What should not go in a header file? Why?

   Code and private data. Because there is no reason this information should be shared.

4. Define "extern."

   A modifier that indicates that the variable or function it modifies may be defined in another file.

5. Define "static." (Warning: It's used for a lot of things.)

   Global variable -- This variable is not used outside this file.

   Function -- This function is not used outside this file.

   Local variable -- This variable is permanent.

   Class member variable -- This variable belongs to the class and not an instance of a class.

Class member function -- The function belongs to the class and not an instance of the class. The member function can only access static member data.

6. Define “infinite array.”

A data structure that looks like any array but can hold an indefinite number of values.

Chapter 24: *Templates*

Thou cunning'st pattern of excelling nature,.
— Shakesphere, Othello Act V

Teacher's Notes

Templates are just a fancy way of automatically generating code. However, at the time of this writing, only the template syntax has been defined. The implementation is currently up in the air. So different compiler makers have defined different ways of working things.

You may want to spend some time finding out how your compiler implements templates. Be prepared to take some time; I've seen some implementations that were very complex and difficult to use.

Review Questions

1. Define "template."

   A generic procedure or class that can be used to generate the real procedure or class at compile time.

2. When does a template cause code to be generated?

   For functions, when the function is used for the first time. For a class, when the first variable of that class is declared.

3. What is a function specialization?

   The defining of a version of a template function that overrides the generic definition.

4. What is a class specialization?

   The defining of a version of a class template's member function that overrides the generic definition.

5. Define "export"?

   Indicates a template function (or member function) which will be used in another C++ file. Exported template definitions must be compiled before use.

6. Name a compiler which implements the "export" directive correctly.

   As of this date, I have yet to find one.

Chapter 25: *Standard Template Library*

Goodness and evil never share the same road, just as ice and charcoal never share the same container.
— Chinese proverb

Teacher's Notes

When I started programming in C, I had to spend a lot of time time creating, designing, and maintaining data structures like trees and linked lists. Now that the STL has come along, I no longer have to deal with my own data structures. It hides all the details.

The STL contains many different types of containers. These differ in their capibilities and the speed with which they perform their operations.

This chapter provides only an overview of the STL. If you want to go into more detail, check out the semi-official documentation at http://www.sgi.com/tech/stl/index.html.

It should be noted that the program presented in this chapter (class/class.cpp) does not make maximum use of the STL algorithms. You may want to challenge the class to see how they can improve this program.

Review Questions

1. Define "STL Container."

   A data structure used to hold other objects.

2. What are the differences between a STL vector and a C++ array?

   A STL vector can grow and shrink. You can also insert and delete elements in the middle of a vector. Also if you use the "at" member function of a vector, the vector will do bounds checking of indicies.

   The C++ array is much faster for fixed size arrays.

3. How do you check to see if an iterator has reached the end of a container?

   Use the phrase:

   ```
   if (iterator != container.end())
   ```

   Note: You can't use < or > with most iterators because the iterators are unordered. Thus < and > are meaningless. (Random access iterators are ordered so < and > apply.)

4. What's stored in a map?

Two items, a key and a value. These collectively are know as a pair. The STL function `pair` takes a key and value and constructs an item that can be put in a map. Also `map::iterator` points to such a `pair`.

Chapter 26: *Program Design*

If carpenters made houses the way programmers design programs, the first woodpecker to come along would destroy all of civilization

— Traditional computer proverb

Teacher's Notes

99% of design is thinking about things and common sense. Teaching this chapter involves asking the questions "Why did they do things that way?" and "Is there a system that can be used to make things better?"

This chapter gives you a chance to tell the students about your programming expierence. Talk about how a major piece of software was designed and lead the class through a discussion of the benefits and drawbacks to such a system.

Review Questions

1. Name and define three design goals that should be a part of any program's design.

    Reliablity – The program should work.

    Economy – The program should be cheap to make.

    Ease of Use – The program should be as easy to use as possible.

2. What are other design factors that might be considered.

    Sellibility – Commericial programs must have some feature that gives them the ability to be sold at a profit..

3. MS/DOS was designed for a 640K PC with 360K floppies as it's primary storage device. Did the design of the operating system make it easy to expand the system for today's modern computers?

    No. MS/DOS was full of silly deisgn limitations. It seems that each major release contained some features that were designed to get around limitations of the previous version. One of the biggest limitation was the 640K limit on memory. This design limit was never fully eliminated until the addition of the Windows operating system.

4. What makes a good module?

    A module should be designed to do one thing well. It should provide as simple an interface to the outside world as possible.

5. What is information hiding?

    I'll tell you enough to do the job and not one bit more. I'll take care of the details.

6. Why are global variables considered bad?

    They expose information to the world (and thus do not promote information hiding). Also since they can be used and modified anywhere in the program, they add complexity to the program.

7. Are all global variables bad?

    No. If the design can be made simpler by using a global variable, you should use one. The cases in which one should be used are few. One example, might be a master "I should keep running" variable. When this variable is turned off, the first procedure to notice it should clean up the system and exit.

8. One of the problems with C++ is that a class must expose implementation information to modules using that class. Where is this implementation information located?

    In the header file that defines the class, the implementation information is the stuff marked **private**.

Chapter 27: *Putting It All Together*

For there isn't a job on the top of the earth the beggar don't know, nor do.
— Kipling

Teacher's Notes

In this chapter we produce a real program. The idea is to use all that we've learned in making something real. We follow the development of the program from design to the first implementation.

The program chosen was designed to be useful to the widest possible audience. For this book the audience is C++ programmers. If your class is more specialized, you may want to create your own programs. For example, if you are teaching a bunch of accountants, you might want to create a bookkeeping program.

The program, although simple, does demonstrate most of the major concepts covered in this book.

Classroom Presentation Suggestions

One thing to emphasize is that this program was designed for expendability.

For example, what would it take to add another statistic? What has to be changed in the program? Turns out very little. You need to define another derived class for the statistic and add it to `stat_list`, but that's it.

Live Demonstration

Whole chapter: *stat/<all>*

Review Questions

1. Describe the programming process used to create this program. Explain what worked and what didn't. If something didn't work, what can be done to fix the problem?

Chapter 28: *From C to C++*

No distinction so little excites envy as that which is derived from ancestors by a long descent.
— François De Saliganc De La Mothe Fénelon

Teacher's Notes

Many organizations are porting their C code to C++. As a matter of fact, there are probably some of your students who are trying to port their skills from C to C++. This chapter presents a lot of lessons learned by porting a lot of code.

Live Demonstration

Slide 10 *setjmp/setjmp.cpp*

Review Questions

1. Define "K&R Style Functions."

   A old style C function header where the type information is defined in separate declarations following the parameter list.

2. Define `malloc`.

   A C version of **new**.

3. Define `free`.

   A C version of **delete**.

4. Why should you never use `malloc` on a class?

   Because it doesn't call the constructor.

5. Why should you never use `free` on a class?

   Because it doesn't call the constructor.

6. How do you initialize a structure to be all zeros? How do you initialize a class to be all zeros?

   Structure: use the function `memset`. Class: put the code to zero the class in the constructor.

7. Define `setjmp` and `longjmp`.

   These are the C style exception handlers. They allow the program to mark a place in the code (`setjmp`) and return to it from inside a series of functions. (`longjmp`.)

8. What are the dangers associated with using these functions?

   The `longjmp` function can jump up several levels of function calls, but does not call the destructor of any of the local variables that are destroyed.

Chapter 29: *C++'s Dustier Corners*

There be of them that have left a name behind them.
— Ecclesiaticus XLIV, 1

Teacher's Notes

This chapters describes some of the more obscure features of C++. Some of these features I use about once every two years. Others I use less frequently. At the end there are"vampire features," that is, features that have never seen the light of day. These features are defined, but I have yet to find a compiler that implements them.

At this point it the class you can describe any cases you've encounter where the contstructs were used.

Chapter 29: *Programming Adages*

Second thoughts are ever wiser.

— Euripides

Teacher's Notes

This chapter pretty much speaks for itself. It is a collection of wise sayings, created by painful programming experiences, collected over the years.

Live Demonstration

Slide 13 *not2/not2.cpp*

Supplement: *From C to C++*

New nobility is but the act of power, but ancient nobility is the act of time.
— Francis Bacon

Teacher's Notes

This supplement is for those of you who are teaching C++ to C programmers.

One of the major problems with people who already know C is that they know how to program. Thus they have a tendency to write C code even when using a C++ compiler and avoid using the new features of the language.

Your job is to help ease them into writing C++ code. A good start is to outlaw the use of the `printf` statement.[1] Also watch out for people who use the pointer parameters where reference parameters can be used:

```
int funct(int *param) // Not good
int funct(int &param) // Better
```

We start with a short lecture on style. As a professional programmer I've had to maintain a lot of code. Most of this code is in very poor condition. Hopefully by teaching commenting and style early we can convince future programmers to write well commented, easy to read programs.

In this chapter we describe how to comment programs. It may seem strange to learn how to comment before we know how to program, but the comments are the most important part of the program.

That last deserves repeating, "Comments are the most important part of the program." I grade homework 60% on comments and 40% on code.

At this point it may seem that I'm somewhat of a fanatic on commenting. I am. You get this way after spending years of going through uncommented code while trying to maintain it.

The purpose of this chapter is to convince your students to put copious comments in the code.

Comments should be written as an integral part of the programming process. Time and time again I've heard, "Here's homework, but I didn't have to put the comments in."

My answer, "You should put the comments in first. I don't mind too much if you put in the comments and don't have enough time to do the code, but don't leave out the comments — ever!"

When I was grading homework, I had a rubber stamp made up:

Comment!

1 (`sprintf` is considered OK because it is clearer and easier to use than the C++ version.)

If you teach your students one thing it should be to make their programs clear and easy to understand. Comments are an integral part of that process.

Finally, we go through a list of the features that are new to C++. The two main things we teach in this chapter are the basics of the new I/O system and the extensive new parameter passing mechanisms.

Classroom Presentation Suggestions

There should be a style sheet for the class. This can be created by the class or supplied by the instructor.

It should include a list of what you as a teacher want for the heading comments of each assignment. Suggestions include:

- The Student’s name
- The assignment number
- Student ID number
- And all the other stuff detailed in this chapter.

Call by Value Demonstration

The general idea is to have the members of the class execute by hand a simple program to compute the area of a rectangle. Assign one student the job of the `funct` function. His “code” is:

```
void funct(int j) {
     j++;
}
```

Have him write down his code on the right side of the board.

Another student is given the job of `main`. His code is:

```
int main()
{
    int my_j = 20;

    func(j);
    cout << “my_j is “ << my_j<< ‘\n’;
    return (0);
}
```

The `main` student hand executes his code. When he comes to the `funt` call he should write 20 on a slip of paper and pass it to the `funt` student.

Point out that the information is passed one way from `main` to `funct`. Once the funct student gets the note, he can do anything with it. He can alter the values, eat it, anything. The only thing he can’t do with it is pass it back to the main student.

Reference Values

Set up the `main` and `funct` students as before. The area student code is now:

```
int area(int &j) {
     j++;
}
```

With references, the parameter passing is different. Have the main student create two boxes labeled `my_width` and `my_height` in the middle of the board. The labels should be on the left toward the main side of the board.

When the call is made, the area student puts the labels `width` and `height` on the right side of the boxes. This means that each box will have two names. There is only one box.

It should be noted that if the function funct changes anything in the box, the change will be reflected in the main program. That's because there is only one box with two labels.

The process of adding a label `j` to the box already labeled `my_j` is called binding, and it is automatically performed by C++ during a function call.

Parameter Type Summary

C++ has a lot of different ways to pass parameters. Slide 60 lists the various types. At this point lots of examples are useful.

One question that keeps cropping up: "Aren't reference parameters just like C pointer variables?"

Answer: There are two differences: Pointer variables can point an array or a single variable. You can't tell which from inside the function. Reference parameters can only reference one variable. Secondly, we've studied reference parameters, and we haven't gotten to pointer variables yet.

Where to go from here

Go through Chapter 7, *The Programming Process*, then continue on with Chapter 13, *Simple Classes* followed by the rest of the book.

Review Questions

1. Why are comments *required* in a program?

   <u>Comments provide the programmer with a clear, easy-to-read description of that the program does. They are required because uncommented programs are hard to understand (sometimes even the person who wrote them can't understand them) Also uncommented extremely difficult to maintain.</u>

   <u>As the age and complexity of software programs skyrockets, maintenance (and comments) becomes much more important.</u>

2. Why must you write comments before or while you write the program, never after?

   <u>Because at the time you write the code you know what you are doing. If you leave the commenting till later, you may forget and have to guess.</u>

3. Which is better: 1) The underscore method of writing names example: `this_is_a_var`, or the upper/lower case method: `ThisIsATest`? (Note this is a religious issue and has no right answer.)

   I like `this_is_a_var` myself because you can run your program through a spelling checker. I think it is more readable. Also it allows you to use lower case only for variables and upper case only for constants.

   Some people think that `ThisIsATest` is more readable.

4. Define std::cout and use it in a C++ statement.

   "cout" is the C++ class that is used for writing data to the screen. Example:

   ```
   std::cout << "Hello World\n";
   ```

5. Define the following:

long int	**short int**	**unsigned**	**signed**
float	**double**	**register**	**auto**
volatile	**const**	**reference**	**extern**

   "long int" a signed integer that may hold more data than an ordinary integer.

   "short int" a signed integer that may hold less data than an ordinary integer.

   "unsigned" variable that may hold only positive value. By sacrificing the sign bit, the number can hold twice as many positive values.

   "signed" a number that hold positive and negative values. Since this is the default for integers and floating point variables, it is rarely used.

   "float" a variable type that can hold real or fractional values.

   "double" a real variable type that has more range and precision than "float."

   "register" a suggestion from the programmer to the compiler indicating that this variable is used a lot so it would be a good idea to put it in a machine register.

   "auto" a variable that is automatically allocated from the stack.

   "volatile" a special variable who's value may be changed by things outside the program. Used for things like memory mapped I/O.

   "const" a named value that can not be changed.

   "reference" an alias or second name for an already declared variable.

   "extern" a variable that is defined in another file. Actually C++ won't get mad at you if you define it in the current file.

6. Why must you use the notation `static_cast<int>(very_short)` to write a very short number (aka. a character variable)? What would happen if you wrote it out without the `int` wrapper?

   The int(x) notation changes the character variable's type from character to integer for the purpose of the write. If the int wrapper is left off, the C++ would write the character variable out not as a very short integer but as a single character. After all, C++ does not know if you are using a character variable to hold a character or a very short int.

7. Define side effect.

   A side effect is an operation that occurs in addition to the main operation specified in a statement.

8. Why are side effects bad things.

   The confuse the code and can easily cause unexpected things to happen. Using them can make the code dependent on things like the evaluation order of operands. They save space in the source code at the expense of clarity and reliability.

9. When is the following variable created, initialized and destroyed?

```
int funct(void) {
    int var = 0     // The variable in question
    // ......
```

   The variable is created when the function is called. It is initialized every time the function starts up and it is destroyed at the end of the function. It is a temporary variable.

10. When is the following variable created, initialized and destroyed?

```
int funct(void) {
    static int var = 0     // The variable in question
    // ......
```

   The variable is created when the program starts. It is initialized once when the program begins and it is destroyed when the program is done. It is a permanent variable.

11. When is the following variable created, initialized and destroyed?

```
int var = 0     // The variable in question
int funct(void) {
    // ......
```

   The variable is created when the program starts. It is initialized once when the program begins and it is destroyed when the program is done. It is a permanent variable.

12. Define "reference"?

   A reference is a variable that is really another name or alias for an existing variable.

13. What is binding and when does it occur?

   Binding is the act of associating a reference with the variable that it aliases.

Binding of reference variables occurs when they are declared. They must be declared with an initialization statement. Reference parameters are bound when the function is called. References may also be used for the return type of functions. In this case the return statement does the binding.

14. What is the difference, if any, between the type of values returned by the following two functions:

```
int func1(void)
const int funct2(void)
```

In actual practice there is no difference between the two declarations.

15. What is the difference, if any, between the type of values returned by the following two functions:

```
int &fun1(void)
const int &fun2(void)
```

The first one returns a reference to an integer variable (or array element) that can be modified. The second one returns a reference to an integer that can not be modified.

16. Which parameters in the following function can be modified inside the function:

```
void fun(int one, const int two, int &three,
         const int &four);
```

Parameters “one”, “three” can be modified.

17. In the previous question, which parameters when changed will result in changes made to the caller’s variables?

If you change parameter “three” the results will be passed back to the caller.

18. Which parameters in the following function can be modified inside the function:

```
void fun(int one[], const int two[]);
```

Parameter “one” can be modified inside the function.

19. In the previous question, which parameters when changed will result in changes made to the caller’s variables?

Changes made to parameter “one” will be passed back to the caller.

16. Define “Dangling Reference.”

A reference to a variable that has been destroyed.

17. Why are dangling references a bad thing?

Because the reference is to something that doesn’t exist, using it can cause unexpected trouble.

18. Can we overload the square function using the following two function definitions?

```
int square(int value);
float square(float value);
```

Yes.

19. Can we overload the square function using the following two function definitions?

```
int square(int value);
float square(int value);
```

No. The parameters to the function must be different in order to overload it. In this case both functions take a single integer as a parameter.

20. Define “default parameters.”

Parameters who’s value you don’t have to define when you call the function. If you don’t define them, then default values are used.

C++ has no way to know parameters are missing from the beginning or middle of a parameter list.

21. What does the keyword **inline** do?

It recommends to the compiler that the code for this function be generated “inline” without the overhead of the code needed to call a function and return from it. It is used only for short function since the entire body of the function is place inline where the function is called.

22. What programming technique was used to solve the “calculator problem” of Chapter 7, The Programming Process.

“Top down programming.” Mostly.