Instructor's Guide
for
Programming
Principles and Practice using C++

Bjarne Stroustrup
Texas A&M University
http://www.research.att.com/~bs
bs@cs.tamu.edu

Abstract

This is a grab-bag of observations and information that might be helpful if you run a course based on "Programming: Principles and Practice using C++".

Assumptions

I assume that you are a professor, lecturer, instructor, teaching assistant, or whatever teaching or about to teach a course based on "Programming: Principles and Practice using C++." I assume that you have read (at least) the book's preface and Chapter 0 "Notes to the reader." If you have not, please do so before proceeding.

I assume that your students have never programmed before, have a weak programming background, or have programmed in a language different from C++. I mostly address issues related to the first two groups.

After the general information, I have comments relating to individual chapters. I write those comments to be read while planning a lecture based on that chapter. My slides are available (**www.stroustrup.com/Programming**); if you use them, I hope you'll suggest improvements.

Feedback and descriptions of teaching experiences are most welcome.

Note: These are notes, just notes. Don't expect book quality copyediting. Don't expect completeness. Do expect updates based on further experience and more time spent on these notes. I plan to keep updating this guide (available on the support site).

General

The greatest obstacle to learning to write good programs is a firm belief that

- it is among the most difficult of technical skills
- it is a skill that requires some rare talent
- it is done by socially inept young men in total isolation, mostly by night
- it is mostly about building violent video games
- it is a skill that requires mastery of advanced mathematics
- it is an activity completely different from everyday ways of thinking
- it is something that doesn't help people

As a professor, teacher, instructor, teaching assistant, etc. it is your most important task to minimize the impact of these myths.

One thing we have done, which appears to be successful, is to take a couple of minutes at the start of some lectures (maybe every third or fourth) to briefly tell about some interesting application of programming (Chapter 1 gives an idea of what we find interesting). The point is to make the otherwise dry programming material appear relevant by pointing to where it applies. We consider it important that these comments are brief and not too preachy. Unless you fell something is exciting, don't try to tell students that it is. Students are experts at detecting forced enthusiasm. We typically base these brief presentations/explanations of our personal experience: something we saw on a trip, found in the news, or read in a magazine. A photo or two (or an actual gadget, e.g. an iPod, a cell phone, a watch) can be important as a prop. Please note that not every student find the same type of news/use interesting. A new development in video games may be exciting to some students, but can also reinforce a negative image of programming to many students who are not quite sure whether programming/CS/IT is for them. We find biological and biomedical applications interest that second group of students disproportionately. It is essential to provide a great variety of applications or the students nod off and retain a narrow and generally unimaginative view of software. Wherever possible, try to draw a direct connection from the example to the code: "Google uses techniques just like the STL" is a good example for the STL chapters, the iPod interface for the GUI chapter, and an airplane for the access to hardware and correctness sections, etc.

It is essential that students get the idea that writing and running small programs are part of "reading a chapter". We find that for many freshmen, some material in the first few chapters seems too easy to take seriously and some parts too esoteric to bother with. The students often come from a background where they have either done a fair bit of programming and think they know everything (at least everything that anyone could possibly tell them during the first month). That view may even make them skip early lectures. Others have done no programming and think that they can learn everything from reading a book. In either case, they have the impression from high school that they know what is important and what is not. Some of the best students also underestimate the time and effort they need to spend – often they were doing really well in high school without much effort. Reality tends to be different. The sooner this reality hits home the better.

Of all the things that correlate with success in this course, "spending the time" is the most important; not previous programming experience, previous grades, or brainpower (as far as we can tell). The drills are there to get people a minimal acquaintance with reality, but attending the lectures is essential, and doing some exercises really matter. We use "Labs" and teaching assistants to ensure the latter. Studying in small groups – preferably heterogonous (in terms of programming background) – groups is another indicator of success. So please try very hard to for each chapter get every student to:

1. Do the "Drill." It is good to have an instructor, TA, or experienced student accessible to help with "small practical problems" that can spring up unpredictably.
2. Read the "Review" and answer at least a few of the questions. This may require re-reading some part of the text. Good students will do this naturally. Some of the students who need it most will need "prodding."
3. Do a couple or more exercises. The exercises are roughly ordered according to difficulty (easiest first) and later exercises sometimes build on earlier ones.

Beware of geeky know-it-alls who try to convince other students that they are smart by referring back to their high-school wisdom or what they found in web-forums. The most damaging "bright idea" is to convince other students that arrays and pointers are much better than vectors and strings, thus diverting genuine naïve students into the problems of range errors, fixed size allocations, **strcpy()** use, etc. long before they are ready for that.

It have been suggested that we write for "elite students at elite institutions". That is of course very complimentary to TAMU's freshmen, but not realistic. We are not among the most selective colleges and the students were not all as highly motivated for this course as we might have liked. In fact, the first several hundred students were mostly EE and Computer Engineering students who did not see software as central to their studies (and the course is compulsory). Only later, after seeing very positive results compared with the CS majors taking a traditional first CS course, did we include CS freshmen.

If you are teaching a large class, not everyone will pass/succeed. In that case you have a choice which in its crudest for is: slow down to help the weaker students or keep up the pace and lose them. The urge and pressure is typically to slow down and help. By all means help –and supply extra help through teaching assistants if you can – but don't slow down. Doing so would not be fair to the smartest, best prepared, and hardest working students – you'll lose them to boredom and lack of challenge. If you have to lose/fail someone, let it be someone that will never become a good software developer or computer scientist anyway; not your potential star students. Computer Science and software development are losing students to fields of study perceived to be more challenging. Please don't add to that problem.

We strongly believe that students live up (or down) to the expectations made of them, and this course seems to bear it out. Surveys show the students in this course work 25% more hours and report 25% more positive opinion than the average freshman course (incidentally, that was the highest satisfaction score in the college of engineering). This course is no harder than average freshman biology, physics, or math freshman course,

given an ordinary US high school background (where the preparation tends to be better in more established academic fields, such as biology, than in CS).

It has been suggested that the course has succeeded "because it was taught by Bjarne Stroustrup." That again is very complimentary, but not realistic. The freshmen are not awed by reputation and many are initially (sadly) far more interested in grades than actually learning serious technical stuff. Anyway, for the last 4 semesters the course has been taught by Pete Petersen, Walter Daugherity, and Ronnie Ward; they are very experienced teachers of undergraduates; I have just done the odd "guest lecture."

Initially the course was taught with two lectures a week plus TA-run "Labs". That was not enough time and we had to cut corners. Later, the course was expanded to three lectures a week plus Labs and that works better. The key difference was that we could devote more in-class time to "talking through" examples and to add more examples from the book. Most likely, the strongest students didn't need that (the information is in the chapters), but most students did better. There is still far more information in the book than can be presented in the lectures. We use about 2 minutes per slide. If that is your speed also, you'll find that you often have to skip a few slides to shoehorn one our lectures into a 50-minute or 60-minute slot. We do that also.

An experiment of running the course over two semesters with two lectures a week has started. I don't expect success because the time between starting and the time "interesting" examples are within the students' grasp become too long. Let's hope that I'm wrong about that.

I have attached Chapter Reviews, which I decided should not be in the book, but could be useful for instructors. The review questions seem more useful to the students than the reviews, which are basically answers (and therefore liked for the wrong reason).

We always manage to cover Chapters 1-22 in a 15-week semester. That leaves a few days for review sessions and room for the essential group project during the last three (or so) weeks (concurrently with the presentation of the last chapters). Chapters 22-27 really don't have a definite sequence. You can read one if and whenever you feel the need. They can be extremely useful as support for a more ambitious project.

Note that this implies that we use a 1264 page book to cover a course that goes through 812 pages. The difference is supporting material: special topics for interested students, reference material, etc.

A practical note: The slides are a bit crowded with information because some students rely (too) heavily on information there and tend to miss stuff in the book that is not on the slides. Also, it helps an instructor who has not had sufficient time to prepare (not a totally unknown phenomenon). The code is in bold 20-point font. That's sufficient for most 200-person auditoriums. In dire emergencies, I use an 18-point font. I don't really like fancy slides so there are no clever transitions or spectacular graphics. If the dark background bothers you change the design to a black-on-white design. At TAMU, we have divided

opinions on what works best in practice. If you find more and better photos to illustrate points, please send them to me; the students like nice photos as long as there is some reasonable connection to the topic at hand.

Please examine the errata (www.stroustrup.com/Programming/errata.html) before each chapter. Typos – especially in code fragments – can be disruptive to learning.

By all means add slides to suit local needs. We do that also, I have just purged all TAMU-specific slides to save you from having to do that. Feel free to add your name to the title slide, but please don't remove mine or the support site reference unless you really have radically changed a lecture. I post PowerPoint to make modification easy.

Thanks

To Lawrence "Pete" Petersen, Walter Daugherity, and Ronnie Ward for constructive comments on the slides and this guide.

Chapter 0: Notes to the reader

Obviously, "Chapter 0" is an odd number ("0 is odd" absurdity intended). Some students will read it and many won't. Few who have never programmed before will understand it. It's main purpose is to give you – the professor, teacher, teaching assistant, etc. – a general idea of what I am trying to achieve and by which means.

So, why "Chapter 0"? Because numbering in C++ – as on a ruler or a tape measure – starts at 0, not 1. To give people a clue that something unusual is going on. To indicate that this is not an ordinary chapter that's part of a series of lectures and homework. To some extent it conveys "meta information."

We don't recommend that students read Chapter 0, though many will. We don't suggest a lecture on the topics in Chapter 0. Instead, the ideas are for you to inject into lectures and discussions as needed. It does answer some of the "Why are we doing this?" and "Must we really do that?" questions that the students ask over the course. Sometimes the answer is "Now, go read Chapter 0."

However, please *you*, the instructor, read Chapter 0 and take it seriously. Few things could be as damaging as a consistent attempt to improve the course by adding more material in the early stages. Until Chapter 12, the students hover at the edge of overload and keep moving only because they constantly receive new stimulation. After each chapter they can *do* a bit more than before, even if they don't quite understand it. Please don't broaden the course by trying to make the students understand everything at this stage. The aim is to get them to the point where they can add such knowledge to a body of skills that allows them to actually use what they learn.

If you are a teaching assistant, please be sure that you have read each chapter before you try to explain things to students. This is *not* your father's C++. Also, go to the lectures or

at least carefully read the slides – it is hard to help students if you don't know what information they have already been given. The information on the slides for a chapter is at best a subset of the chapter text.

There is IMO little as deadening and damaging to a student as an attempt to teach them everything about C++'s basic types and their interactions or every details of how to do iteration, selection, and recursion early on. At that stage that's just dead knowledge. You could be a great programmer for years without being able to explain the (absurd) rules for conversion from an unsigned short to an int.

Chapters 1 and 2

To get started with programming, a student needs two things

1. A view of what good and important can be done using software. A sense of purpose. Some idealism.
2. A concrete example that they can write, run, and play with.

Both are essential. Our suggested lecture 1 is about half of each.

Chapter 1 aims to convey a sense of excitement, noble purpose, and professionalism. This is not about how to create the most blood-splattered videogames. If you want to save the world (as a technical person as opposed to a politician) you'll have to use computers to do so. The concepts, tools, and techniques we learn/teach here is what makes the world work. No, we are not fundamentally against video games; what we are against is the view that video games are the central application of computers and the attitudes that collates with people with that view. By all means, *occasionally* point out that most videogames are written in C++ using many of the techniques we teach.

Many students have a hard time imagining a computer that's not a PC with a keyboard and a screen. It is most useful to enlighten them on this point. More than half of the world's good computer-related uses (and jobs) are here. It doesn't hurt to remind the students that there is a screaming need for lots of good systems builders in industry (i.e. US, European, and other industry) implying good careers and good pay. If the students themselves don't worry, their parents and non-programming friends do. Many people (including mainstream journalists, career counselors, and parents) remember the 2000 dot-com bust and think that all technical work has been out-sourced or off-shored. It has not been and US businesses are desperate for good technical people. I consider it fair game to point out that of all US college graduates, EE and CS majors get the highest average salaries – more than pure scientists, business majors, etc.

Note the variety of "platforms" featured (the photos help). Many students have an overly firm idea of what a computer is. A professional software developer will over a career encounter radically different target platforms, from computers the size of a nail to monsters that fill rooms or effectively span continents.

Chapter 2 is all about getting the students to know enough about their programming tools to get "Hello, world!" running. There are essentially no intellectual contents in the

program itself. Unless you make the explanation complicated (don't!), the student's reaction is "Duh? What's the big deal?" There isn't any big deal yet, but there won't be one unless the students get used to their tools and get into the habit of reading the text and doing the exercises.

Always look at the chapter "Postscripts." They tend to say something significant about the chapter's subjects or where they lead.

Our lecture #1 that covers both of these chapters gives a brief overview of the course to give the students an idea of where they are going (in addition to the general talk about the importance of software) and emphasize structure of code and the fundamental aim of correctness/professionalism.

The explanation of the photos and the explanations that go with them are in most cases in the book. The planes are a Boeing 787 (only the non-flight parts of the software, such as the entertainment system is C++) and an F35 ("Joint Strike Fighter") hovering (in Ft. Worth Texas). The wind turbines are of course programmed in C++ and the truck is one used for seismic measurements for oil reservoir mapping. The collection of PCs happens to be the controls for the Keck observatory on top of Mauna Kea on Hawai'i.

Since we have been dealing with freshmen, getting "Hello, world!" to work has been plagued with many practical problems: Some students do not have accounts on the departmental servers, some do not have C++ compilers and don't know how to get them, some teaching assistants just came back from vacation or summer internships yesterday, etc. Please do everything you can to have administrative and logistical problems solved before the first lecture. If at all possible, have a meeting with TAs the week before to plan how to minimize problems and how to deal with the problems that will occur anyway (if you have 150 students, there *will* be unanticipated problems). Be available or (better) have TAs available between the first and the second lecture and make clear that you expect that everybody have "Hello, World!" running before lecture #2. If that is not the case, some students will play catch-up for weeks – or give up.

Try to be sure that TAs actually read each chapter before trying to help students. Having TAs help based on C++ knowledge from a more conventional course or from a java background is not a good idea.

In this lecture we also cover what we consider cheating and what is not cheating. Many of our freshmen (1st-year students) are quite confused about this. Is studying together cheating? Is giving a friend a copy of a completed exercise cheating? Is using a (program) library cheating? Our answers are 'no', 'yes', and 'only if you forget to quote your source'. If you have a need to expand on ethics, this may be a place to do it.

I notice that I had an – all too common – blind spot when I mentioned the uses of software skills: education. If your aim is to become a teacher (of any level up to and including university), especially a science teacher (of any level up to and including university), or someone concerned with dissemination of technical skills (trainers,

documentation writers, and more), grounding in software can be useful – or even essential. Obviously, software can play a major role in inspiring young people to take science and technology seriously: just think of robot competitions and visualization of biological systems.

Also, I tend to say "industry" when I mean "developing software for real-world use." That can of course also be done in universities, research labs, government organizations, non-profits, etc.

Somewhere during the first three lectures, make the point that every example given will run on every major platform; mention PC/Windows, Apple, Linux, and Unix. All the code in the book is available from the support website. We consider portability very important; most students don't and won't get the point. However, many are devoted to their current computer and "portability" will assure them that the course is relevant for that. We have allowed our students to use "any system with a modern C++ compiler" even though we provide direct support only for one platform supported by the university. Most of our students use Windows, but we have never had a course without a dozen or so Macs and Linux boxes.

We are using a "home brew" header **std_lib_facilities.h**. This is not ideal, but

(1) the alternative is to teach the students about several headers early on (<**string**>, <**vector**>, <**iostream**>, and <**algorithms**>) and what is needed to use them:
   a. **namespace**s
   b. the distinction between user-defined and built-in types
   c. which facility lives in which standard header
   d. **using** directives, **using** declarations, or explicit qualification.

(2) The file is all standard C++ and all will eventually be explained

On balance, we decided to simplify code for the first chapters using **std_lib_facilities.h**. Chapter 8 will explain the use of header files and namespaces. Later chapters will present **string**, **vector**, **iostream**, and **algorithms** in some detail.

We are very keen not to present students with "magic", but saying that "**#include "std_lib_facilities.h"** gives you access to the facilities of the standard library (and we'll show how a bit later) is actually far less "magic" than: "the compiler produces executable code."

You should have access to the/a support site (e.g. **www.stroustrup.com/Programming**) for the book and can download **std_lib_facilities.h** from there. If not, I attached a version to the end of this document that you can copy into a file and use.

Different systems/IDEs/communities have different conventions about where header files are located relative to .cpp files. This can be very confusing for real novices (actually, it often frustrates me also). The book is written assuming that .h and .cpp files are in the same directory/folder. That is, it uses

#include "std_lib_facilities.h"

Depending on convention, you may have to advise students to use

```
#include "../std_lib_facilities.h"      // one level up
```

or

```
#include "../../std_lib_facilities.h"   // two levels up
```

Or whatever fits local conventions. Maybe you can make a strong assumption about what is right in your environment, but I can't.

Depending on where you got **std_lib_facilities.h** from, how you stored it, and your compiler, you may get a "newline missing at the end of file" warning. IMO it is a pretty silly warning: just add a newline at the end to shut it up.

I use **.cpp** as the source suffix and **.h** as the header suffix. Again, different communities have different convention: **.cc** is not uncommon for source files; also, **.cxx** and **.C** work in many places. Also, some people (and tools) prefer **.hh** or **.H** over **.h**. Naming of files is a convention, rather than part of the language.

Our mix of students has been 60% have programmed before and 40% have never seen a line of code. One reason to move fast the first couple of weeks is to level the playing field by quickly getting to material that is new to all. Invariably, some of the students who have programmed before try to show off, which can be very intimidating to the complete novices. You need to reassure both groups:

1. You can master this material if you have not programmed before.
2. If you have programmed before you'll soon get to something that is well beyond what is usually taught in a US high school teaches: "welcome to university."

Chapter 2 Review

1. C++ is a programming language designed for a wide selection of programming tasks; see **http://www.research.att.com/~bs/applications.html** for examples.
2. In C++, strings are delimited by double quotes (").
3. The **\n** is a "special character" indicating a newline.
4. The name **cout** refers to a standard output stream. Whatever characters are "put into **cout**" will appear on the screen.
5. Anything written after the token // on a line is a comment. Comments are ignored by the compiler and written for the benefit of programmers who read the code.
6. The first line of the program is typically a comment that tells the human reader what the program is supposed to do. This kind of comment reminds us (the programmers) what we should tell the computer precisely, completely, and formally.
7. An **"#include** directive" instructs the computer to include (make available) facilities from a file.
8. The importance of **std_lib_facilities.h** (written specifically for this book) is that it makes the C++ standard library facilities available.
9. Every C++ program must have a function called **main** to tell it where to start executing.
10. A part of a C++ program that specifies an action and isn't a pre-processing directive is called a statement.

11. A C++ compiler translates source code (code that you write) from the human readable form to "machine code" that can be understood and executed by the computer.
12. The compiler looks at your source code to see if your program is grammatically correct, if every word has a defined meaning, and if there is anything else obviously wrong that can be detected without trying to actually execute the program. The compiler will not compile your program until these "compile-time errors" have been corrected.
13. The compiler is possibly the best friend you have when you program.
14. The program that links compiled program parts, often developed by different people, together to form an executable program, is (unsurprisingly) called a linker.
15. Object code and executables are not portable among systems. For example, when you compile for a Windows machine you get object code for Windows that will not run on a Linux machine.
16. Errors found by the compiler are called compile-time errors, errors found by the linker are called link-time errors, and errors not found until the program is run are called run-time errors and logic errors.
17. Generally, compile-time errors are easier to understand and fix than link-time errors and link-time errors are often easier to find and fix than run-time errors or logic errors.
18. An "Integrated Development Environment" or IDE provides facilities to help you write, debug, compile, link, and run your code.
19. An IDE usually includes an editor with helpful features like color coding to help distinguish between comments, keywords and other parts of your program code.
20. You might think you understand everything you read and everything your instructor told you in class, but repetition and practice are necessary to develop programming skills.

Chapter 3: Objects, types, and values

It is important here to make it clear that there is nothing fundamentally hard about taking a value and storing it away. However, the concept is new and surprising to non-programmers, especially non-programmers with a good Math background. The analogy with a variable being a box into which you can put a value works well. Similar, we use the analogy of the type being the shape of a box determining what can be put into the box.

It is crucial that the students do the drill and some exercises. Students who have programmed before tend to be too cavalier with that and can easily miss that something very new (relative to their high-school experience) is being put into place.

The use of **if**-statement before its real introduction in Chapter 4 is deliberate: Be sure to emphasize that it is just a new notation for something they have know since kindergarten. Similar, emphasize that the arithmetic is just middle-school stuff. The point that many programming language facilities are simply a "fancy" notation for things the students

already know is an important and recurring theme. It is well worth repeating to the students when you happen to encounter a concrete example along the way.

What *is* new here is assignment, initialization, and type.

Note that to some (especially people with a good match background) the idea of (the computer) executing on statement followed by another is surprising. Pick a slide or two and go through line for line "playing computer" to get that idea across. It doesn't take much time and avoids problems later. Slides 20-21 ("assignment and increment") and 27 ("another simple computation") (especially if you are in danger of finishing early) are good for that. Chapter 4 will reinforce.

Why bother telling students about type-safety problems at this early stage? The aim is to get them into a way of thinking that involves the machine and real-world constraints: Our language and techniques are ways of mapping solutions to hardware. The hardware is real, imposes restrictions, and no language is able to completely abstract from that. This is a course in programming (software development), not math or philosophy.

Chapter 3 Review

1. Real programs tend to produce a result based on some input we give it, rather than just doing the same thing each time we execute it.
2. An object is a region of memory with a type that specifies what kind of information can be placed in it.
3. A named object is called a variable and has a specific type (such as **int** or **string**) that determines what can be put into the object. For example, character strings are put into **string** variables and integers are put into **int** variables.
4. A statement that introduces a new name into a program and sets aside memory for a variable is called a definition.
5. **cin** is used for input. The name **cin** refers to the standard input stream (pronounced "see-in" for "character input") defined in the standard library.
6. The compiler remembers the type of each variable and makes sure that you use it according to its type, as specified in its definition.
7. The line

   cin >> name >> age;

- reads two values which can then be assigned to a string name and an int age. Note that we can read several values in a single input statement, just as we can write several values in a single output statement.

8. Several operators are shown in the table in Section 2.4.
9. A **string** keeps track of the number of characters it holds.
10. A program – or a part of a program – is type safe when objects are used only according to the rules for their type.
11. The ideal and the language rule is complete type safety. Unfortunately, a C++ compiler cannot guarantee that, but we can avoid type safety violations through a combination of good coding practice and runtime checks.
12. Always initialize your variables!

Chapter 4: Computation

Reemphasize that the students already know the control structures (if, while, for, switch) from real life ("if the light is green, cross") – all we are teaching is a different and more systematic notation.

The appeal to professionalism (emphasis on correctness) is important (and tends to go down well).

Don't go into more detail that is necessary. Please resist the temptation to be complete. Please resist the temptation to show off your expertise. That can be hard to do because inevitably some students will try to show off their expertise though "questions" about features and techniques not covered in class (or in the book) until much later. Please take such questions off line to avoid diverting the discussion and confusing and intimidating the students who has not programmed before. For the first 10 chapters or so, we have to remember to address two different audiences: people who have taken a CS or programming course before and people who have not. After that (if we have done a good job), those two groups become essentially indistinguishable. My first university Math professor said "you have seen most of this before, but we'll do it right and move fast, don't miss the boat!" in his first lecture. If you have a more homogenous class than we have, you should have an easier task, but the approach works for both groups: please don't skip stuff for "experienced programmers" and don't slow down for "genuine novices."

So far, we are just assembling a set of basic tools and skills – but we don't want to obsess about them: The aim is to build up enough of a portfolio of tools and skills so that the students can feel they can do something. Everything will be repeated as we go along in the course. Most important issues will be touched upon many times. Every language feature introduced here will eventually be used dozens of times in books examples and on slides (if you find an important feature that is not used repeatedly, please tell me).

The idea that learning is about "building up a portfolio of useful knowledge and skills" is new to many students and not always appreciated. Many basically think that classes are all about getting good grades. It is a good idea to mention the portfolio argument occasionally.

Why do we introduce **++x**? Incrementing is a fundamental idea (arguably more fundamental than addition) and **++x** directly expresses it. **x=x+1** is an indirect way of saying "increment". Also, **++x** is ubiquitous in real code. We do not want to teach anyone to write **for (int i=0; i<max; i=i+1)** …

Here is the first time we seriously go beyond the experience of students who have studied CS AP – they tend not to find the stream-of-text examples familiar. We had had complaints: "I have studied C++ for two years and you are teaching things I haven't seen before!" This was said with undisguised outrage; "Welcome to university" was my

answer. Students who have never seen a line of code two weeks before *do* understand those examples – when explained. They are excellent for students to "hand execute." We recommend hand execution as a way of getting a feel for control flow and variables. It is important to shake the students who have studied programming before or of their stupor ("I know all of this; I don't need to do the drills; I don't need to be awake during the lectures – I know all the profs can tell us for months!"). This group of students can become the best or the worst depending on whether they take note that this is not high-school programming and get excited.

Some students dismiss teaching that is not "average US high-school spoon feeding" as "bad teaching" and consider exercises and tests that are not just regurgitating examples they have seen repeatedly in the lectures "unfair." This attitude has to be defeated. It does not belong in a university. To counter, encourage idealism and a "can do" attitude needed for real-world problems.

Note that we "smuggled in" a bit of grammar. Please don't make a big deal out of it.

Chapter 4 Review

1. Computation is the essence of computer programming. (Section 4.1)
2. The most important and interesting categories of input and output are those "to and from other programs" or "to and from other parts of a program." (Section 4.1)
3. A program is a collection of cooperating parts and how they share and exchange data between them to accomplish desired tasks. (Section 4.1)
4. Program components share data stored in main memory, on persistent storage devises, or transmitted over networks. (Section 4.1)
5. The first rule for the use of parentheses is: "If in doubt, parenthesize." (Section 4.3)
6. Ugly code slows down reading and comprehension. It is also harder to verify correctness and to identify errors. (Section 4.3)
7. Always try to choose meaningful names. (Section 4.3)
8. A statement is an instruction to the computer. It is a step in the program sequence. (Section 4.5)
9. A block statement (compound statement) consists of a number of statements enclosed within two braces. (Section 4.5)
10. Block statements are used to control scope and when you want several statements to be treated as one. (Section 4.5)
11. An empty statement is a statement with nothing in it, e.g., a **';'** by itself or an empty block **{}**. (Section 4.5)
12. An if-else statement chooses between two alternatives. If the condition is true the first statement is executed, otherwise the second statement is. (Section 4.6)
13. You should always test your programs for bad input because users will eventually enter bad input and your program should behave sensibly when they do. (Section 4.6)
14. A switch-statement tests the value of the index against a set of constants. (Section 4.7.1)

15. If there is no break in the statements associated with the matched constant, the program will continue to run statements until a break or the end of the switch is encountered. (Section 4.7.1)
16. A default case in a switch statement is optional and may be placed at any point in the sequence of cases, but it is usually placed at the bottom. (Section 4.7.1)
17. Iteration in programming is a process in which a set of operations is repeated which produces a result that is closer to the desired outcome after each cycle. (Section 4.8)
18. A for-statement is functionally similar to a while statement, but management of the control variable is in the top line where it is easier to see and understand. (Section 4.8.3)
19. Never modify the control variable inside the body of a for-statement. (Section 4.8.3)
20. Use a for-loop when you want a fixed number of iterations and whenever else you reasonably can. Make this your standard loop construct. (Section 4.8.3)
21. Use a while loop when you want your program to iterate until some condition occurs, regardless of the number of iterations it takes to reach that point, and when it is not clear how that can be expressed easily in a for structure. (Section 4.8.3)
22. The job of a programmer is to express computations correctly, simply, and efficiently. (Section 4.2)
23. For large programs, applying abstraction and divide and conquer is not just an option, it is an essential requirement. (Section 4.2)
24. A function is like a small program. It is a named sequence of statements that takes input, performs some process, and produces some output. (Section 4.9)
25. To implement (call) a function, name the function and provide the required input (arguments). (Section 4.9)
26. The distinction between declarations and definitions becomes essential in larger programs where we use declarations to keep most of the code out of sight to allow us to concentrate on a single part of a program at a time. (Section 4.9.1)
27. For many things that we want to do with computers we will need a collection of data to work on. Such data is often read from input and stored in a vector. (Section 4.10)
28. The standard library sort() takes two arguments: the beginning of the sequence of elements that is to be sorted and the end of that sequence. (Section 4.10.2)
29. Each Programming language feature exists to express a fundamental idea and we can combine them in an almost infinite number of ways to write useful programs. (Section 4.11).

Chapter 5: Errors

Here is where we try to get the philosophical point of correctness across as well as impart some practical design and debugging skills. No student will understand it all or master it all, but we set a standard (a set of ideals) and give the students a place to return to when they get stuck.

Please don't try to teach students "all about exceptions." That's not possible. Here we just need a simple and standard way to write an error message and get out.

Note that the lists of errors and kinds of can be used as check lists.

Pre- and post-conditions are important and useful concepts. You could almost see the whole chapter leading up to those. However, we need to approach them through examples and alternatives, or the students don't see what problem they address.

Chapter 5 Review

1. Errors found by the compiler are called *compile-time errors* and generally consist of syntax errors and type errors. (Section 5.1)
2. Errors found by the linker are called *link-time errors*. (Section 5.1)
3. Errors found at run time are called *run-time errors* and generally consist of errors detected by the computer, errors detected by a library (e.g., the standard library), and errors detected by user code. (Section 5.1)
4. Errors found by the programmer looking for the causes of erroneous results are called *logic errors*. (Section 5.1)
5. Your program
    a. should produce the desired results for all legal inputs
    b. should give reasonable error messages for illegal inputs
    c. need not worry about misbehaving hardware
    d. need not worry about misbehaving system software
    e. is allowed to terminate after finding an error (Section 5.1)
6. Avoiding, finding, and correcting errors take 90% or more of the effort when developing serious software. (Section 5.1)
7. Here are three approaches to producing acceptable software:
    a. Organize software to minimize errors
    b. Eliminate most of the errors we made through debugging and testing
    c. Make sure the remaining errors are not serious (Section 5.1)
8. Sources of errors in programs include poor specification, incomplete programs, unexpected arguments, unexpected input, unexpected state, and code that don't do what it is supposed to. (Section 5.2)
9. Don't get overconfident: "my program compiled" doesn't mean that it will run and even if it does run, it typically will give wrong results at first until you find the flaws in your logic. (Section 5.3.3)
    a. let the caller of the function deal with bad arguments or
    b. let the called function deal with bad arguments. (Section 5.5)
10. C++ provides a mechanism, called exception handling, to help deal with errors. The fundamental idea is to separate the detection of an error (which should be done in a called function) from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored. (Section 5.6)

11. Exceptions allow us to combine the best of the various approaches to error handling. Nothing makes error-handling easy, but exceptions make it easier. (Section 5.6)
12. Logic errors are usually the most difficult type of error to find and eliminate because the computer does what you asked it to and your job is to figure out why that wasn't really what you meant. (Section 5.7)
13. Start thinking about debugging before you write the first line of code. Once you have a lot of code written it's too late to try to simplify debugging. (Section 5.8.1)
14. Decide how to report errors: a good default answer to this question is: "Use **error()** and catch **exception&** in **main().**" (Section 5.8.1)

Chapter 6: Writing a program

In this chapter and the next, we give the students a taste of what program development is. We see it as a branch of problem solving, a search for a solution, and an incremental activity. Most students will be surprises by something or other. Few students will understand all. Our aim is for students to get to the point where they see program development as an activity that (repeatedly) modifies old code to make it more useful, more correct, and more maintainable. Sometimes we don't understand all about the code we need to improve – that's normal and expected.

Note that there are two models of program development that we want to actively discourage:

(1) design the complete program, write all the code, *then* test it
(2) just start coding; add features and reorganize as needed; ship when it looks good

Please don't have too firm an idea about what students will understand, what they will not understand, and how they will understand things. Some "grok" grammars immediately; some are intrigued and spend a long time on them; others don't get it, but still get to understand the code based on the grammars. Some (few, but we think a significant few in terms of potential) read the code and *then* understand the grammar. We want to challenge the brightest while leaving the less ambitious with enough useful knowledge for them to benefit and proceed.

Read the parsing diagrams from the bottom up – from the input to the top rule.

Some students (often some of the students who has programmed before) try hard to reject our message that the structure of code is important and insist on "surging ahead" writing lots of code without distraction from "formality" or "theory." Try to convince them that that approach doesn't scale. You'll probably fail at first, but insist that such students thoroughly test any poorly structured program in ways similar to what we do in Chapters 6 and 7 and eventually most will get the point.

Note that we slip in the notion of a user-defined type – just because we need one. The idea of defining and using our own types is one that needs repeating – with many, many concrete examples. We come back to this fundamental idea in most chapters after this.

Note that we just mention and use recursion. Please don't make a big deal of recursion (or iteration); that just convinces students that there is something really difficult and scary about recursion (or iteration). We go into slightly more technical detail in Chapter 8, but for now, recursion is just a name for a neat way of getting code to work nicely.

Please don't skip the "false starts" and "wrong solutions" to save time. If you just show the final version of the program, very few students will

(1) get it (understand the program)
(2) get an idea of how to write code through gradual refinement
(3) get the idea that making mistakes and fixing them is an acceptable (and efficient) way of progressing. Many have the absurd idea that all they do has to be perfect at the first try – that's a fatal idea in the context of programming.

If a student catches a problem early, just say "thank you, you're right, we'll get to that." Do not get diverted into presenting a final solution early.

Note the errata for page 202-203. Entering 4+5+6+7 exposes the logical error in the program in a way that doesn't help us discover its cause (a premature exit with the error message never seen because the window disappears immediately). Using 4+5 6+7 exposes the problem in a way that allows us to spot the error in a way that allows us to devise a remedy. We don't invent Token_stream until we discover the need for putting back a token. Until we understand that need, making a stream will be seen only as "complication and overhead". First demonstrate the need for a stream so that Token_stream is seen as the solution to a real problem. (and then point out that "a stream with a putback" is a very general solution applicable to many input problems).

For some students, it is very important to demonstrate that programming progresses though a series of stages and that an error in an early stage is not a failure. To try to make every minor extension to a program work at the first try is both futile and inefficient. The key here is "feedback." We use our tools and techniques – e.g. the compiler and our debugging – to get feedback on our first attempts so as to make faster progress.

Walk through the code, explaining it line for line. These two chapters is also an exercise in code reading and code review. You should get to the point where you can say "and this is the same as before it just does …" quite a lot.

Note that the book has more "false starts"/"dead ends"/"errors" than the slides. Sitting through the lecture is not a substitute for reading – and vice versa.

Chapter 6 Review

1. Understanding the problem you would like your program to solve is key to a good program – after all, a program that solves the wrong problem is of little use, however elegant it may be. (Section 6.1)
2. Analysis – write a description of what should be done – this is called a set of requirements or a specification. (Section 6.2)

3. Design – an overall structure for the system including which parts the implementation should have and how they should communicate with each other. (Section 6.2)
4. Break the problem you want to solve into manageable parts, even the smallest program for solving a real problem is large enough to be subdivided. (Section 6.2.2)
5. Use pseudo-code in the early stages of design when we are not yet certain exactly what our notation means. (Section 6.3)
6. It is most important to avoid "feature creep" early in a project. (Section 6.3)
7. To read a grammar, start with the top rule and search through the rules to find a match for the tokens as they are read. (Section 6.3)
8. Useful rule: division binds tighter than addition. (Section 6.4)
9. Token is an example of a C++ user defined type. A user defined type can have member functions as well as data members. (Section 6.8)
10. A C++ user defined type often consists of two parts: the public interface and the (private) implementation details. This separated what users of the type require access to from the details required to implement the type, which we'd rather not have the user mess with.
11. Note that again and again we avoid doing complicated work and instead find simpler solutions – often relying on library facilities. This is the essence of programming.

Chapter 7: Completing a program

This chapter spends a lot of time of the structure and looks of code; that is, factors that affect comprehension and maintainability. It is a good idea to remind the students that often they are the maintainers (maybe of their own code, a few months after they first wrote it). A useful program is never finished: it will be ported, corrected, extended, etc.

When we develop software we play many different roles:

(1) designer
(2) implementer
(3) bug finder (debugging)
(4) tester (systematic search for errors)
(5) maintainer
(6) user

Maintenance, testing, etc. is *not* "somebody else's problem."

Good program structure minimizes errors, makes it easier to find bugs ("bugs lurk in messy code"), and ease changes. We impose structure to save time and effort.

Chapter 7 Review

1. When your program first starts running, you are probably about half-way finished. (Section 7.1).

2. For a large program or a program that could do harm if it misbehaves, when the program first starts running, you will be nowhere near half-way finished. (Section 7.1).
3. Once the program "basically works" the real fun begins! (Section 7.1).
4. We can't think of everything all the time, so when we stop to reflect, we find that we have forgotten something. (Section 7.1).
5. The first thing to do once you have a program that basically works is to try to break it. This is known as testing. (Section 7.3).
6. Add features incrementally (Section 7.4-5).
7. After making a number of changes (improvements) to your program, review the code to see if you can make it clearer, shorter, and improve the comments. (Section 7.6).
8. We are not finished with a program until we it is suitable for someone else to take over maintenance of the code. (Section 7.6).
9. Use symbolic constants. Don't scatter "magic constants" all over the code (Section 7.6.2).
10. Functions should reflect the structure of the program and the names of the functions should identify the logically separate parts of the code. (Section 7.6.2)
11. Look through the program for ugly code that can be hard to read; it provides hiding places for bugs. (Section 7.6.3).
12. Comments should be used to express things that cannot be expressed directly in code, such as intent (Section 7.6.4).
13. When we clean up code we might accidentally introduce errors. Always re-test the program after clean-up. (Section 7.6.4).
14. Dealing with errors is always tricky. It requires experimentation and testing because it is extremely hard to imagine what errors can occur. (Section 7.7).

Chapter 8: Function technicalities

For this chapter, most of our students breathe a sigh of relief. Finally, here are some simple geeky facts with little philosophy and no taint of problem solving!

Keep encouraging the students to define and use small functions in exercises. There often are a lot of "backsliding" to the "one large function" or "just a couple of functions with a lot of global data" styles of programming. The idea that a function is a logical unit of programming and that designing and naming functions are valuable exercises has to be learned (and reinforced by instructors). As part of the same problem, some students see global variables and "natural and simple" and resist passing arguments (because passing arguments is "complicated and inefficient").

"Inefficiency" of what you are suggesting is a surprisingly common student defense for messy code. This is especially common among "true geeks" who has programmed in high school (often being better at it than their teacher, thus getting an inflated view of their own abilities and knowledge) and hang out in discussion forums. The value of program structure and logical simplicity are hard to get across to such students. "Complicated means advanced" and "low-level means efficient/fast" are dangerous misconceptions that are hard to fight. I have found students who are addicted to the efficiency argument yet

compile with the default settings of their compiler. Pointing out that the default is "debugging" and runs up to 25 times slower than if the very same code had been run through the optimizer surprises many. Pointing out they can speed up their code by 25 times without re-coding sometimes (not always) make them think. Many have no clue what efficiency means and will happily spend hours hand-optimizing a piece of code that basically waits for a human (e.g. using unsafe low-level I/O rather than iostreams). Pointing out the there are several dimension to the notion of "efficiency" (e.g. efficient use of the student's time) and that correctness is essential in many applications ("if you don't need the correct result I can make it as fast as you like") sometimes help. Try not to be dragged into efficiency discussions in full class. It distracts, confuses, and sometimes the students will think you lost the argument because they did not understand your answer: keep the answers *really* simple or you make the problem worse for many students.

Note: Some students simply don't understand scope (on first try). Some think that arguments must be variables named the same as the parameters in the called function. This is a place where small group instruction based on looking at the students' examples is essential. You cannot (unless you have taught this several times) predict the students' misconceptions with any accuracy.

Chapter 8 Review

1. What matters is how ideas can be expressed in code, not the individual language features. (Section 8.1)
2. C++ belongs to a group of languages that also includes C, Java, and C#, so quite a few language technicalities are shared between these languages.(Section 8.1)
3. A declaration is a statement that introduces a name into a scope, specifying a type and optionally, an initializer. (Section 8.2)
4. Before a name can be used in a C++ program, it must be declared. (Section 8.2)
5. A declaration defines how something can be used; it defines the interface for a function, variable, or class. (Section 8.2)
6. In a header file the compiler reads the declarations it needs to understand our code. (Section 8.2)
7. A declaration that (also) fully specifies the entity declared is called a definition. (Section 8.2)
8. A definition of a variable sets aside (allocates) memory for that variable; consequently, you cannot define a variable twice. (Section 8.2)
9. The **extern** keyword states that a declaration is not a definition. (Section 8.2)
10. The declaration/definition distinction allows us to separate a program into many parts that can be compiled separately. (Section 8.2)
11. Constants have the same declaration syntax as variables; but they have **const** as part of their type and require an initializer. (Section 8.2.1)
12. A function declaration with a body (a function body) is a function definition. (Section 8.2.1)

13. The key to managing declarations of facilities defined "elsewhere" in C++ is the header file. (Section 8.3)
14. Our **std_lib_facilities.h** header file contains declarations for the standard library facilities we use, such as **cout**, **vector**, and **sqrt()**, together with a couple of simple utility functions such as **error()**, that are not part of the standard library. (Section 8.3)
15. A scope is a region of program text. When a variable or function is declared in some scope, it is valid (in scope) from the point of declaration to the end of that scope. (Section 8.4)
16. The main purpose of a scope is to keep names declared within it local to that scope, so they will not interfere with names declared elsewhere. (Section 8.4)
17. Whenever you can, avoid complicated nesting of scopes. Keep it simple. (Section 8.4)

Chapter 9: Class technicalities

This is another "technical" (not problem solving) chapter. It is therefore perceived as simple by many geeks. However, it does slip in many design issues through the gradual refinement of the Date example.

Do emphasize the notions of interface and invariant (9.4.3). Yes, we are teaching design to novices and few will get it all, but the lessons will be repeated and reinforced in the following chapters, so please persist. Examples makes all the difference.

No, we don't expect everyone to "get" all of this. Keep encouraging the students to define and use small classes in their exercises and projects. There are many more examples in the following chapters. There will be a lot of backsliding into styles based on global variables, data structures with lots of public data, etc. Just keep showing how simpler and less error-prone code gets with the right "little classes." Don't despise little/simple classes.

Emphasize the use of "small simple functions" to simplify code and improve readability (by naming logically separate parts of a program). Avoid simple getters and setters – if a **get()** and a **set()** for each data member is the best you can come up with you should use a **struct**.

Point out that the sequence of examples gradually brings the code nearer to our everyday notion of a date and allows the compiler to catch more and more "silly errors."

From now on these notes get sparser. My guess is that by now you need them less. A chapter plus a lecture "speaks for themselves."

Chapter 9 Review

1. A built-in type is one for which the compiler knows how to represent objects of that type, and also knows which operations can be done on it. (Section 9.1)
2. Types that are not built-in are called user-defined types (UDTs). C++ provides two kinds of UDTs, classes and enumerators. (Section 9.1)
3. Standard library types are as much a part of the language as the built-in types, but we still consider them UDTs because they are built from the same primitives and with the same techniques as the types we build ourselves. (Section 9.1)
4. If you think of some part of your program as a separate entity, it is likely that you should define a class to represent that part in your program. (Section 9.1)
5. In C++ (as in most modern languages) a class is the key building block for large programs – and is very useful for small ones as well. (Section 9.1)
6. The Interface is the part of a class declaration that users may access directly. It is identified by the term public. (Section 9.3)
7. The implementation is the part of a class declaration that users may access only indirectly, through the interface. (Section 9.3)
8. A **struct** is a **class** in which members are public by default (Section 9.3).
9. An invariant is a rule that describes a valid value. (Section 9.4.3)
10. When we define a member outside of its class, we need to say which class it is a member of, using the **class_name :: member_name** notation. (Section 9.4.4)
11. Don't put member function bodies in the class declaration unless you know that you need the performance boost from inlining tiny functions. Large functions, 5 lines of code or more, don't benefit from inlining. (Section 9.4.4)
12. An **enum** (enumeration type) is a user-defined type, specifying its set of values as symbolic constants. (Section 9.5)
13. You can define just about any operator provided by C++ for your own types, but only existing operators, such as +, -, *, /, %, [], (), ^, !, &, <, <=, >,and >=. (Section 9.6)
14. An overloaded operator must have at least one user-defined type as an operand (Section 9.6).

Chapter 10: I/O streams

Many students have only the vaguest idea of "data;" (that is lots of data items; not just a dozen variables). Files and large in-memory data structures are "magical." That's the view we have to eventually demolish. This chapter and the next primarily show how to read and write files. It's not difficult (much easier than dealing with user I/O (also covered)), but conceptually unfamiliar for many students.

Basically, the student has to know:

1. How to read a stream of values (typically a file or some format we don't control)
2. A single value (of some type) from input
3. Produce nicely formatted output ("nice" often means "matching some convention")

It doesn't matter what programming you do, you'll end up doing those three kinds of I/O – plus a bit of graphics/GUI (Chapters 12-16). This is not just for desktop processing: Even the simplest gadget typically has to have its parameters set and often deals with one or more data streams.

Chapter 10 Review

1. This chapter focuses on the basic model: how to read and write individual values, how to open, read, and write whole files.
2. The way the variety of devices is dealt with in most modern computers is to separate the detailed handling of I/O devices in device drivers and then access the device drivers through an I/O library that makes I/O devices appear as similar as possibly to the problem. (Section 10.1).
3. When dealing with file I/O, the programmers job is to set up I/O streams to appropriate data sources and destinations and then to read and write to and from those streams. (Section 10.1).
4. I/O streams can represent files, network connections, recording devices, display devices, keyboards, and interaction through graphical user interfaces. (Section 10.1).
5. The C++ standard library provides the type istream to deal with streams of input and the type ostream to deal with streams of output. (Section 10.2).
6. Most of the time, we'll assume that these "bytes on disk" are in fact characters in our usual character set. That is not always so, but we can get an awful long way with that assumption (section 10.3).
7. To read a file, we must: To read a file, we must know its name, open it (for reading), read in the characters, and close it. (Section 10.3).
8. To write a file, we must name it, open it (for writing) or create a new file of that name, write out our objects, close it. (Section 10.3).
9. Opening the file implicitly as part of the creation of an **ostream** or and **istream** and relying on the scope of the stream to take care of closing the file is the ideal. (Section 10.4).
10. When dealing with input we must expect errors and deal with them. The possibilities for input errors are limitless! However, an **istream** reduces all to four possible cases, called "the stream state":
    good() // the operations succeeded
    eof() // we hit end of input ("end of file")
    fail() // something unexpected happened
    bad() // something unexpected and serious happened (Section 10.6).
11. To get a robust read we have to deal with three problems
    a. the user typing an out-of-range value
    b. getting no value (end of file)
    c. the user typing something of the wrong type (Section 10.7)

Chapter 11: Customizing I/O

Most of "customization" is tedious details. Please encourage the attitude "let's have a quick look at what can be done so that we know what to look for in the

book/manual/online_documentation later if we need it". I tend to strongly emphasize that most of the complexity comes from pre-computer conventions, e.g. using ( ) to mean negative/debit/loss and using comma for a decimal point (no we won't show you how, look that up if you need it – an obvious example to add if you are in, say, Germany). This course/book does not cover locales.

Take the opportunity to discourage the "read a whole line and then see what it contains" way of handling text input. For simple input that's simply more work for the student. Chapter 23 – if you get that far – has the more complete treatment of text processing, incl. regular expressions. Feel free to point that out: "If you even have to do more serious text processing or text manipulation, read Chapter 23 first; that could save you a lot of time. For now, we don't need any really sophisticated stuff."

Chapter 11 Review

1. 1In this chapter, we concentrate on how to adapt the general **iostream**s framework presented in Chapter 10 to specific needs and tastes. A number of ways are presented in which we can tailor input and output to our needs (Section 11.1)
2. Our programs exist to serve humans, and humans have strong preferences. Thus, as programmers we must strive for a balance between program complexity and accommodation of users' personal tastes. (Section 11.1)
3. Output streams, **ostream**s, provide a variety of ways for formatting the output of built-in types. For user-defined types, it is up to the programmer to define suitable << operations. (Section 11.2)
4. The notations << **hex** and << **oct** do not output values. Instead << **hex** informs the stream that any further integer values should be displayed in hexadecimal and << **oct** informs the stream that any further integer values should be displayed in octal.
5. Decimal numbers have no prefix, octal numbers have the prefix **0** and hexadecimal values have the prefix **0x** (or **0X**). This is the notation for integer literals in C++ source code.
6. Some integer output manipulators are **oct**, **hex**, **dec**, **showbase**, **noshowbase**, and **setw**.
7. Formatting of floating point values is handled using **iostream** manipulators in a manner very similar to that of decimal values.
8. Some floating point manipulators are **fixed**, **scientific**, **general** (not standard), **setprecision**, and **setw**.
9. the **general** format chooses between **scientific** and **fixed** formats to present the user with the most accurate representation of a floating-point value within the precision of the **general** format, which defaults at 6 total digits.
10. Use the default (general format with precision 6) unless there is a reason not to. The usual reason not to is "because we need greater accuracy of the output".
11. Unless you set a field immediately before an output operation, the notion of "field" is not used.
12. The properties of a stream determine what operations we can perform after opening the file, and their meaning. The simplest example of this is that if we open an **istream**

for a file, we can read from the file, whereas if we open a file with an **ostream**, we can write to it.

13. The exact effect of opening a file may depend on the operating systems and if an operating system cannot honor a request to open a file in a certain way, the result will be a stream that is not in the **good()** state
14. A binary I/O is messy, somewhat complicated, and error-prone, but occasionally we must use binary I/O simply because that's the format someone chose for the files we need to read or write.
15. A typical example is an image or a sound file, for which there is no reasonable character representation: a photograph or a piece of music is basically just a bag of bits.
16. The character I/O provided by default by the **iostream** library is portable, human readable, and reasonably supported by the type system. Use it when you have a choice and don't mess with binary I/O unless you really have to.
17. An **istream** that reads from a **string** is called an **istringstream** and an **ostream** that stores characters written to it in a **string** is called an **ostringstream**.

Chapter 12: Graphs

This chapter is basically one long demo. We don't try to teach the students anything deep. We just want to show them some pictures and give them the idea that doing so is easy. After the I/O streams chapters they badly need a break and some encouragement. Also, many students consider graphics real (interesting) and I/O of numbers and text "boring and irrelevant" whatever we say.

In this lecture, I move fast from code to output (screen dumps) and have no problem finishing on time despite the high number of slides.

I usually ask for a volunteer to explain the last code example (containing yet unseen facilities) to prove that once you get the general idea of a good library (in this case graphics), you can read the code without preparation. With the slightest of help from the lecturer that "experiment" always works.

Please save essentially all implementation discussion and "advanced stuff" until the next two chapters. Please do resist the temptation to show off, but feel free to wax lyrically about graphical applications and how they consist of "code like this".

The use of FLTK (or equivalent) is a necessity. I keep wanting to re-do my graphics interface library using two more graphics/GUI libraries to give a splendid example of portability, but I have never found the time. I don't want the students to think that there is anything special about FLTK. I do emphasize that the code is portable and we always have students in the class using Windows, Linux, and Mac so this is a good demo of portability. "A window on your machine looks what a window looks like there; our code just asks for whatever kind of window your machine offers."

Some students need a lot of help setting up a Microsoft Visual Studio project with GUI. Also, students who want to run the code on their own machines may need help

downloading and installing FLTK. This has not been a big problem for our TAs, but with a large number of students (we have had 180 in one class), the logistics have to be worked out a bit in advance – don't wait until the night before☺.

Yes, that "next button" is a dirty trick that makes a GUI interface look just like "ordinary non-GUI programming." That's the point: Get the students comfortable with graphics before tackling events, widgets, and control inversion – that's what Chapter 16 is about.

I start with a relatively complex example (axes and a curve) because the course was first taught to freshmen electrical engineers, some of whom considered graphical shapes a bit too frivolous. It is good sometimes to emphasize that programming is used to noble, serious, profitable, etc. ends and not just for "fun and games."

If you feel like it, you can run the code examples from Chapters 12-16 "live." The "next button" provides for "animation" for demos. Personally, I find that distracting and prefer the slides, but others have better luck with demos than I.

Note that we have yet to introduce pointers (Ch 17), so in several places we "skate on thin ice" explaining inheritance. Just "skate along" blithely – it can be done.

Chapter 13: Graph classes

Here we start digging into the implementation of the graphics interface classes and see a few (deliberately very few) examples of FLTK use. Basically this chapter is still focused on showing examples of graphics use; we are just digging one level deeper into the code to get better examples. Only in the next chapter (Ch14) do we get to the heart of the implementation: The implementation of shape and the explanation of the basic object-oriented techniques we rely on. Please dodge those details for now: just show what can be done; that will provide the motivation for the student "getting" the techniques and concepts. The point of Lines and Text is that they provide grouping of shapes.

The color matrix example introduces **new** and unnamed objects. **Vector_ref** can be found in Appendix E, but please resists the temptation to introduce pointers to "really explain **new**." If someone asks (quite likely), just say "yes, **new** and **Vector_ref** do use pointers; we'll get to that in Chapter 17. For now, we just use them to avoid having to give a name to every object we need."

The explanation of data hiding and the reasons for making implementation variables **private** is one of the themes of the next chapter (Ch 14).

The talk covers only about half of the examples in the book, but it covers essentially all of the concepts. The book give more examples to give more familiarity with the ideas, more practice in reading code, and more useful classes for exercises and projects.

Chapter 14: Graph class design

This chapter takes on two key topics: what makes a collection of facilities a library and how do you build class hierarchies. Obviously, these topics are related. This chapter is basically about ideas, pretty fundamental and powerful ideas, but approached through examples. If you – up until the final technical slides – take the line "were looking for a good way to present a collection of shapes and the techniques for that happens to be very widely useful" you won't go far wrong. You can go wrong by spending too much time explaining "fancy worlds, like polymorphism" and trying to approach the topics comprehensibly and/or from a theoretical standpoint. At most, I do an aside "every useful and popular idea has many names; "polymorphism" is Greek for "many shapes" so it seems very appropriate that it is the name for techniques that allows us to define and use graphical shapes – and many, many other examples – well; look it up if you like."

Chapter 15: Graphing functions

This chapter is a bit messy and full of details. The aim is to give some programming practice and examples after the conceptual high point in Chapter 14 and before the mind-bending control inversion of Chapter 15. Basically, we go through a series of code examples to reinforce what has been shown before and to encourage some students that what they are learning is real ("real" to many engineering functions means "numbers and functions"; "real" to many non-engineering functions means "graphics"; and many relate to data).

The speed with which you go though the lecture can vary dramatically depending on the degree of detail (repetition) you want to go into. If the students retained all you taught so far, you'd end early (even if you used the "extra" slides showing the implementation of **Axis** provided for that emergency). More likely, you could go on forever. Do get the simulation of **exp()** done. We often prefer to actually run the code live and repeatedly. The point that the good-looking and apparently (and arguably) correct code misbehaves because of numerical instability comes as a shock to many. It is worth repeating.

The use of a global variable to control the number of terms used by **expN()** is a hack. Alternatively, we could have **expN()** hold and increment a local static variable (another hack). I don't see a good solution short of using a function object with an increment or set number of terms function – and I think function objects are too advanced and/or distracting at this point. In general, **Function** isn't a glowing example of design, it is just necessary to get graphing done, but gets bogged down in scaling and placement issues; **Axis** is better. If you like, feel free to make the point that not all code can get ideal in the first pass through. Much real-world code could be better, but isn't. There is a wide variety of reasons for that, not least "we ran out of time." Take that as a challenge of time management and programming technique, not as an excuse for sloppy coding styles.

The "simulation" technique provides the basic tools for projects where things move on the screen (usually, such projects additionally need a delay mechanism (e.g. **sleep**(); look it up; for timing, see 26.6.1) and a random number generator (see 24.7))

Chapter 16: Graphical user interfaces

Sometimes I decorate a talk with a few photos that seem relevant or warrant a detour. The GUI talk simply screams for a few examples – e.g. I have used an iPod and a airplane cockpit display – but since such examples only work if the instructor is knowledgeable and enthusiastic about a particular example and since fashions change rapidly I have left the published slides "plain and boring."

Note that by now, only about half of the examples in the book are covered by the lecture. The assumption is that students by can and will carefully read the chapter. That's not a good assumption for all students unless you have some mechanism for enforcing it: teaching assistants, labs, homework assignments, etc.

Chapter 17: Vectors: memory management

Now the "graphical holiday" is over. We are back to data structures and algorithms.

The three "vector chapters" do several things:

- introduce arrays, pointers, and free store
- show the implementation of std::vector
- introduces the definition of templates
- refines the notion of exception handling

Generally, we prefer to do one thing at a time and not doing anything "in the abstract;" all is tied down to concrete examples. That doesn't mean that general concepts, rules, and techniques are left out, but that we try to stick to the "concrete before abstract" principle.

Basically, we are building from the hardware up until we get to **std::vector** (at the end of Chapter 19). Be sure to emphasize (and repeat over the next chapters) the points that

- "close to the hardware" is a very uncomfortable and unproductive place to be/program
- The techniques and language facilities we use to build vector from low-level primitives are general and widely useful.

Note that you can have memory leaks in essentially all languages; even garbage collected languages, such as Java and C# (e.g. just stick a reference to an object into a hash table and forget about it – the garbage collector will consider it live forever).

When warning against the inconvenience and error-prone aspects of working close to the hardware there is a danger of demonizing such work. Much useful work requires knowledge of these techniques and language features – it is part of the essential bridge from higher-level code to hardware; it must exist somewhere for every system. If not in the language used, then in some other language (usually C or C++). It is the essential and common base of systems programming. We go into more details – and repeat a little bit – in Chapter 18 and also in Chapter 25, where the focus is embedded systems programming.

Note that the technique of acquiring resources in a constructor and releasing resources in the destructor is key to most modern C++ techniques (and the key to exception safety; see Chapter 19). Don't breeze past the destructor.

Chapter 18: Vectors: arrays

To be honest, I find much of this chapter tedious, but it's essential or the students will fail to appreciate pointers and arrays and will be babes in arms when it comes to C-style code. Unless they do exercises with pointers and arrays, they'll forget it all within a couple of weeks.

The explanation of copy construction and copy assignment is essential.

Chapter 19: Vectors: exceptions and templates

The reserve(), resize(), and push_back() example doesn't just show the relationship between memory and vector (which could easily be considered "magic"); it also foreshadows the general sequence concept of the STL (Ch20).

The key concepts behind effective exception handling: the standard guarantees and the RAII are only very briefly presented in the talk. There is not time to go into details. The book has a bit more information. However, the role of the destructor in all of this should not be missed (easily done if you come from a Java background).

The admission of having "cheated" by providing a range-checked vector even though the standard doesn't guarantee checking (and using a dastardly macro at that!) used to be just embarrassing. Lately, things have been more interesting. Some implementations ship libraries that check by default (e.g. the latest Microsoft C++), so now we typically have different students in a class using different library implementations without knowing it. This leads into a brief discussion of compatibility, engineering principles, tradeoffs, and (too often) performance. This is reflected in the book and in one slide.

Chapter 20: STL: containers, iterators, and algorithms

Getting the STL iterator model across is important. Don't rush, don't try to be clever, and don't try to show "more realistic" or "more advanced" examples early. For many, "getting" the point that an STL algorithm is generic with respect to both container type and element type is mindboggling and can lay the seeds for much future interest and discovery (e.g. actually, we don't need a container at all, any stream of elements will do).

If I had the time, I'd spend three lectures for this chapter. We start out with a motivating "lifting" example followed by the basics of the STL model – basically focusing on the utility of generalizing as a motivation for generic programming and the STL. This leads to a repetition of the presentation of a simple algorithm demonstrating the use of iterators (accumulate() followed by find()). This seems to be both effective and necessary.

Just a reminder: without actually using the STL algorithms and containers, many students won't really get the ideas. I have heard of students who claimed

(1) to have been present at every lecture
(2) to have been awake
(3) never to have heard of "iterators" (a year later).

I have heard the same story for "pointers". Skipping drills is a recipe for certain disaster; not doing more than one or two exercises per chapter is a recipe for likely disaster. Retention is a problem and writing code is the best way to address it.

Chapter 21: STL: maps and algorithms

This is a classical "walk through the code" presentation. Some students hate it: "boring!" Maybe so, but this ought to be (and for some is) an exciting chapter ("You can do that!" - yes; "but isn't that slow?" – no it isn't). The purpose is the get the light bulb to go on about genericity/parametization/flexibility/re-use and far too often that doesn't happen unless the student see examples – they don't get it from simply articulating principles and not everybody gets it from the drills and exercises (presumably because they didn't do those or did them with help and without real understanding).

Chapter 22: Ideals and history

Apologies for not covering many other interesting programming languages and techniques, but we have only so much space and the students only so much patience. We tend to present this chapter just before the final exam. This implies that many students treat it as a distraction from learning "stuff that'll be on the exam." If you want such students to pay attention, guarantee them that there will be questions on the exam that can be answered only if you have read this chapter.

It is not possible to cover this entire chapter in a single one-hour talk. We don't try; we cherry pick. Of course, I have an unfair advantage here, having known many of the people personally.

The red arrow on the Murray Hill Photo marks the 5th floor corridor where the people pictured worked in the early 1980s.

The languages and people are chosen partly to show C++ in its historical context, partly to show a gradual increase in the languages' ability to directly model general ideas. If I had more time, I'd increase the emphasis on the point that there can be no single best language for everybody and for everything. That's one reason that the Fortran/Cobol/Lisp specialization has modern equivalents, e.g. C++/Java/PHP. When I give the talk, I emphasize the net of connections between the individuals involved and the relatively few organizations involved.

Ideals are not ideals in the abstract; they are ideals that individuals have chosen as their ideals.

Chapter 23: Text manipulation

This is the first chapter that's focused on an application domain, rather than on a programming language features or a programming technique. Please make a big deal of that, because it is. Emphasize how the features and techniques we have learned by now come together to solve real-world problems: strings, iostreams, and maps, in particular.

Text processing is one application domain among many, but an important one because strings are an almost universal medium for human-to-human, human-to-machine, machine-to-human, and machine-to-machine communication.

This is where some students finally see the point of maps.

I find photos important here. Pointing out that at its most basic level the human genome projects was string manipulation and searching of strings (strings of A, C, G, and Ts adenine, cytosine, guanine, and thymine) really gets to some students.

Emphasize the universality of regular expressions: "This is string pattern matching; not (just) C++!" You do pattern matching like this in essentially all languages: C, C#, Javascript, Java, C++, Ruby, Python. Etc.

The text on the title slide is in runic and is the law of Skaane (part of Denmark, now part of Sweden) and has no real connection with the talk beyond being an example of text. Similarly, the table on the overview page is just an example of a table (the talk gets to tables eventually) and the container ship (the world's largest) on the applications paper is just an illustration of quite a few application areas.

Chapter 24: Numerics

Try using the Matrix library without getting into implementations details. Many/most libraries are used as black boxes (implementation unseen) and the implementation uses a combination of object-oriented, generic, and optimization techniques that makes it quite hard to read (beyond any real novice).

If you don't like Math, this chapter is no fun. If you do like Math, this chapter is just a teaser. The key point is that you can and should use the language to provide a library that models the fundaments concepts of an application domain (here, matrices for linear algebra) rather than barging ahead using the built-in language facilities directly.

Chapter 25: Embedded systems programming

Over half of all computers are embedded and a very high proportion of programming jobs are in the embedded systems industry. Yet, embedded systems seem to have no mindshare in the popular view of programming and software – the dominance of the PC application seems complete, however counterfactual and detrimental.

I take the opportunity offered by embedded systems to revisit the low-level programming language facilities and techniques. I also repeat discussions related to mapping code to

hardware. This reflects my experiences with older students and grad-students who all too often has lost (if they ever had it) any realistic idea of the mapping on higher-level code to hardware. Thus, this chapter revisits themes from Chapters 17-19 and (from a very different angle) Chapter 24.

There is a tendency to downplay the low-level facilities and the C heritage of C++. Sometimes it is even demonized – please don't do that. The mapping of higher-level constructs to hardware is important and a topic where I hear many complaints from industry about the lack of proper preparation of students (mostly students who lack a C or C++ background). Students should not consider that mapping as magic; nor should they get stuck close to the hardware with a fear of abstraction (masquerading as "concern for efficiency" or "avoiding complexity").

I chose the TEA algorithm partly because some students find encryption fascinating, partly because the style of code is so different from what the students have seen before, and partly because of its direct manipulation of bits in words (common to encryption, compression, and graphics).

The photo on page 3 is of a blood analyzer as used by emergency personnel. The plane on page 4 is a United Arab Emirates F-16 and the engine an earlier model MAN marine diesel engine as explained in Chapter 1. Engineering students respond well to the inherent idealism underlying the early slides: We build the world, (all) people rely on this kind of software, we must make it reliable – there are no excuses.

Obviously, the first part of the talk/chapter is a bit "preachy" but then the second half is bit fiddling.

Chapter 26: Testing

Despite having no real theory, this may be the most "theoretical" chapter of the book. The students don't know large systems and there is no time to introduce testing frameworks, bug databases, static analysis tools, fault injection tools, etc. All we can do is to breeze though the major themes with somewhat artificial examples. However, I consider testing so important and a so almost universally ignored topic that I couldn't leave it out. The idea of systematic searching for errors – as opposed to the ad hoc debugging is an important one, and I think one that we can get across. In my mind, designing code to help such systematic testing is the other side of the coin. My hope is to improve design and to prepare the students for a more thorough and realistic treatment of testing.

Chapter 27: The C Programming Language

I have never seen a program that could be better written in C than in C++ for any reason except the absence of a good C++ compiler on a system. I have made this – and equivalent – statements often over the last couple of decades and never had it seriously challenged. I don't consider emotional rants and insults without technical data a challenge. There are reasons (e.g., lack of compilers or lack of trained programmers) to prefer C over C++, but "performance" or "better language features" are not among them.

Nor do I think that "complexity" is a significant reason – the systems we build these days are far more complex that C or C++ and complexity must go somewhere – if not on the language (with formally defined semantics) then in user code.

So, C is a useful language which shares much with C++. The evolution of C and C++ has been intertwined since before the birth of C++. My contributions to C include:

- Function prototypes
- // comments
- **bool**
- **const**
- Initializers in for statements (C99)
- **inline** (C99)

Unfortunately, most features imported from C++ and "C with Classes" into C were incorporated in incompatible forms.

This chapter/lecture describes C from the perspective of a C++ programmer. That is, it describes what is missing and how one can program without the missing features. It also gives examples of colloquial C.

The word counting example illustrates the differences between high- and low-level programming. I used it because it would not be right if all code in this presentation was written by me. I consider **strcmp()** and the intrusive list example better examples of the use of C (and both are of course also C++).

"Emergency" copy of std_lib_facilities.h

In non emergencies, gain access to the/a book support site (e.g., **www.stroustrup.com/programming** and download support materials from there.

```
/*
	simple "Programming: Principles and Practice using C++"
	course header to be used for the first few weeks.
	It provides the most common standard headers (in the global
	namespace) and minimal exception/error support.

	Students: please don't try to understand the details of headers
	just yet. All will be explained. This header is primarily used
	so that you don't have to understand every concept all at once.
*/

#ifndef H112
#define H112 200608L

#include<iostream>
#include<fstream>
#include<sstream>
```

```
#include<cmath>
#include<cstdlib>
#include<string>
#include<cstring>
#include<vector>
#include<algorithm>
#include<stdexcept>
using namespace std;


template<class T> string to_string(const T& t)
{
	ostringstream os;
	os << t;
	return os.str();
}

struct Range_error : out_of_range { // enhanced vector range error
reporting
	int index;
	Range_error(int i) :out_of_range("Range error: "+to_string(i)),
index(i) { }
};


// trivially range-checked vector (no iterator checking):
template< class T> struct Vector : public std::vector<T> {
	typedef typename std::vector<T>::size_type size_type;

	Vector() { }
	explicit Vector(size_type n) :std::vector<T>(n) {}
	Vector(size_type n, const T& v) :std::vector<T>(n,v) {}

	T& operator[](unsigned int i) // rather than return at(i);
	{
		if (i<0||this->size()<=i) throw Range_error(i);
		return std::vector<T>::operator[](i);
	}
	const T& operator[](unsigned int i) const
	{
		if (i<0||this->size()<=i) throw Range_error(i);
		return std::vector<T>::operator[](i);
	}
};

// disgusting macro hack to get a range checked vector:
#define vector Vector

// trivially range-checked string (no iterator checking):
struct String : std::string {

	String() { }
	String(const char* p) :std::string(p) {}
	String(const string& s) :std::string(s) {}
	String(int sz, char val) :std::string(sz,val) {}
	template<class Iter> String(Iter p1, Iter p2) :
std::string(p1,p2) { }
```

```
      char& operator[](unsigned int i) // rather than return at(i);
      {
            if (i<0||size()<=i) throw Range_error(i);
            return std::string::operator[](i);
      }

      const char& operator[](unsigned int i) const
      {
            if (i<0||size()<=i) throw Range_error(i);
            return std::string::operator[](i);
      }
};



struct Exit : runtime_error {
      Exit(): runtime_error("Exit") {}
};

// error() simply disguises throws:
inline void error(const string& s)
{
      throw runtime_error(s);
}

inline void error(const string& s, const string& s2)
{
      error(s+s2);
}

inline void error(const string& s, int i)
{
      ostringstream os;
      os << s <<": " << i;
      error(os.str());
}

#if _MSC_VER<1500
      // disgusting macro hack to get a range checked string:
      #define string String
      // MS C++ 9.0 have a built-in assert for string range check
      // and uses "std::string" in several places so that macro
      // substitution fails
#endif

template<class T> char* as_bytes(T& i)    // needed for binary I/O
{
      void* addr = &i;  // get the address of the first byte
                        // of memory used to store the object
      return static_cast<char*>(addr); // treat that memory as bytes
}


inline void keep_window_open()
{
      cin.clear();
```

```
	cout << "Please enter a character to exit\n";
	char ch;
	cin >> ch;
	return;
}

inline void keep_window_open(string s)
{
	if (s=="") return;
	cin.clear();
	cin.ignore(120,'\n');
	for (;;) {
		cout << "Please enter " << s << " to exit\n";
		string ss;
		while (cin >> ss && ss!=s)
			cout << "Please enter " << s << " to exit\n";
		return;
	}
}

// make std::min() and std::max() accessible:
#undef min
#undef max

#include<iomanip>
inline ios_base& general(ios_base& b)	// to augment fixed and
scientific
{
	b.setf(ios_base::fmtflags(0),ios_base::floatfield);
	return b;
}

// run-time checked narrowing cast (type conversion):
template<class R, class A> R narrow_cast(const A& a)
{
	R r = a;
	if (A(r)!=a) error(string("info loss"));
	return r;
}


inline int randint(int max) { return rand()%max; }

inline int randint(int min, int max) { return randint(max-min)+min; }

#endif
```
