The Order of Arguments in Conditionals
The passage talks about making code easy to understand when using symbols like `<` and `>`. It suggests a rule: put the variable that doesn't change much on the right side. This makes the code clearer, like normal writing. For example, use "while (bytes_received < bytes_expected)" instead of the other way around. It also mentions an old practice called "Yoda Notation" in languages like C and C++, where you flip the order to prevent mistakes. But nowadays, compilers usually catch these errors, so "Yoda Notation" isn't as important. Overall, it says it's important to write code that's easy to read and understand.

The Order of if/else Blocks
When writing if/else statements, it's helpful to think about the order of the conditions to make your code clearer and easier to follow. Here are a few tips: First, try to deal with the positive condition before the negative one (for instance, use `if (debug)` instead of `if (!debug)`). Second, tackle the simpler condition first to help keep your code straightforward and fit it on a single screen. Third, consider addressing the more significant or unusual condition first to highlight it, similar to how people naturally pay attention to something unique or unexpected. However, sometimes these tips might conflict with each other, and you'll have to decide which ordering makes the most sense. Organizing your if/else statements well can really improve how readable and functional your code is.

The ?: Conditional Expression (a.k.a. “Ternary Operator”)
The discussion about the ternary operator (cond ? a : b) in programming notes its ability to shorten code by summarizing complex conditions into one line, which can enhance readability and reduce repetition, as shown in an example with string appending. However, the text also warns that using the ternary operator can make code less clear and harder to manage, especially with complex calculations, like those involving exponents. The main point is that while the ternary operator can make code neater, it's best used for simple situations to help rather than hinder understanding. The text emphasizes that making code easy to understand should be more important than just making it shorter.

Avoid do/while Loops
The excerpt explains that the do/while loop in programming, although common in languages like Perl, can be tricky because the condition check comes after the code block. This setup is less straightforward than other loops like if, while, and for loops, where the condition is usually at the beginning. It might make readers double-check the code more often, which is not very efficient. Also, do/while loops can cause extra confusion or repeated code, particularly when using the continue statement. Bjarne Stroustrup, who created C++, advises using while loops instead because they show the condition first, which makes the code clearer and reduces errors. Therefore, it's generally better to use while loops rather than do/while loops.

Returning Early from a Function
The passage talks about a common misunderstanding that functions shouldn't have more than one return statement. It says that actually, using early returns can make code easier to understand and faster. It mentions that modern programming languages have ways to handle cleanup tasks neatly, like using destructors in C++, or try-finally blocks in Java and Python. Some people think it's best to have only one exit point in functions to make sure cleanup tasks are done, but the passage suggests that modern languages offer better solutions. It also says that in languages like pure C, where there aren't built-in features for cleanup, using early returns in big functions with lots of cleanup tasks can be tricky.

The Infamous goto
This excerpt talks about the `goto` statement in programming, specifically in C and the Linux kernel. Although `goto` is often unnecessary and can cause problems in many modern programming languages, it can be useful in certain situations. The text points out a good use of `goto` in making it easier to exit functions cleanly, helping to manage resource cleanup effectively in one place and reducing errors. However, it also cautions against using `goto` recklessly, as it can create messy and confusing "spaghetti code," particularly when `goto` statements overlap or jump backwards, making the code hard to follow and maintain. The main takeaway is that while `goto` can be helpful in some cases, its use should generally be limited to keep the code clear and organized.

Minimize Nesting
The main points are about the problems with code that has too many layers, which can be hard to understand. Each extra layer means keeping track of more conditions at once. The given example shows how these layers build up, often from small changes that ignore how readable the code is. To improve this, the idea of "returning early" is suggested, which means dealing with errors right away to keep the code simple. Also, using "continue" in loops can help by skipping steps that aren't needed, making the code easier to read. The goal is to write code that's straightforward and clear, even for someone new to it.

Can You Follow the Flow of Execution?
This chapter explains the importance of knowing how both basic and advanced parts of programming work to keep code easy to read and understand. It discusses how programming tools like threading, signals, exceptions, function pointers, anonymous functions, and virtual methods can make it hard to see where a program starts and ends. These tools can make code more efficient and avoid repetition, but using them too much can make the code hard to understand and fix. The chapter recommends using these tools carefully to avoid making the code too complex, comparing overly complex code to a tricky Three-Card Monte game where it's hard to follow the actual sequence of events. It suggests that developers should find a good balance between making their code complex and keeping it easy to manage and work with.