Pseudocode

INTRODUCTION

Flowcharts are only one of the possible formal design tools. They were one of the earliest forms used and are still very popular with many people because they give a visual image of the design. Unfortunately, the flowcharting symbols were established before the concepts of structured programming (which we will look at in greater depth in a few chapters). As a result, flowcharts aren't able to handle some of the concepts that we now assume will be present in a design. Pseudocode is another formal design tool, but it was developed along with structured programming. Pseudocode works very well with the rules of structured design and programming. But it isn't visual-it's narrative. Because pseudocode is more easily used for "structured design," it will be our primary design tool in this text.

INTRODUCTION TO PSEUDOCODE

Unlike flowcharts which have very clear and agreed upon rules, pseudocode is more individualistic. We will use the following rules:

RULES FOR PSEUDOCODE

Write only one statement per line
Capitalize initial keyword
Indent to show hierarchy
End multiline structures
Keep statements language independent

Let's look at each of these rules separately.

Write only one statement per line
By the time we get to our formal design, pseudocode in this case, we have already done our lists of major tasks, subtasks, and so on. Now our job is to convert that rough list into a formal design that can be used as the basis for coding. We have broken each task down into the smallest level of subtask or sub-subtask necessary to be sure that we have defined what needed to be done. We now convert each of those detail elements into one line in our pseudocode. Each pseudocode line will represent one "action" in our design or, as we will see later, will mark the boundaries of an action.

Let's try a variation of our payroll design from the last chapter. We'll assume our input includes name, the hours worked, and the hourly rate and that we are to write out the gross pay as well as the gross pay minus a deduction of $15 for dues and uniforms. Our list of tasks might look something like:

```
Read name, hourly rate, hours worked
Calculate gross pay and net pay
Write name, gross pay, net pay
```

Our pseudocode will just fill in the details a bit:

```
READ name, hourly rate, hours worked
Gross pay = hourly rate times hours worked
Net pay = Gross pay minus 15
WRITE name, Gross pay, Net pay
```

Notice that each statement is a single action and is written on a separate line.

Capitalize initial keywords

Look again at the pseudocode that we just completed. READ and WRITE are written in all capitals. They are "initial keywords" because they begin the statement and they are command words that give special meaning to the operation. In the statement, Net pay = Gross pay minus 15, we don't put any word in all capitals because we don't have a keyword. Net pay begins the statement, but it is not a command word. In pseudocode, we will use a limited set of keywords, primarily READ, WRITE, IF WHILE, and UNTIL. You will find it easy to recognize these keywords.

Indent to show hierarchy

In the pseudocode above, we don't have any indentation; all statements are lined up with an even left margin. As long as we have only sequence structures, we won't have indentation because we don't have any questions of hierarchy. Each statement is independent of the others. But if we have a loop or a selection, we must indicate which statements fall within the loop or which statements fall within the selection. Indentation is one of the ways we use to show the "boundaries" of these structures.

Assume that in our previous pseudocode, there is an additional deduction for workers with over 40 hours. Our pseudocode changes to this:

```
READ name, hourly rate, hours worked
Gross pay = hourly rate times hours worked
Net pay = Gross pay minus 15
IF hours worked is greater than 40
        Net pay = Net pay minus 10
ENDIF
WRITE name, Gross pay, Net pay
```

We have included a selection by using the keyword IF followed by the "condition" to be tested. IF the condition is true, we will subtract an additional $10 from Net pay. The calculation statement is indented because it is dependent on the IF statement. We have used indentation to show which statements are within the IF. (We will come back to Ifs later in this chapter to show that all the IF forms are like in pseudocode.)

End multi-line structures

In our IF structure above, we also included the line ENDIF. We're being a bit redundant here since we have already said we use the indentation to indicate what statements go within the IF structure, but the advantages of increased clarity are worth the repetition. Including the ENDIF (or ENDwhatever) makes sure the reader recognizes that we've finished with that structure and are now going on to a new statement. It also helps make sure that you know where the loop or selection is completed. The ENDIF should be aligned (same left margin) with the IF it is ending. When we get to nested Ifs (very soon), it will be particularly important to keep our IFs and ENDIFs all present and accounted for.

Keep statements language independent

This "rule" doesn't mean that you can do your pseudocode in French, Russian, Chinese, or whatever language you may know. It refers to programming languages. You should not let your pseudocode turn into the code that you will eventually write in. In this book, since we will not take our designs into code, we will be free of the temptation to gravitate toward code-like pseudocode. But what if you are taking a class in Pascal, COBOL, C, or some other language, then can your design become language *dependent?* It may be tempting to let your design grow more an more like the code you will eventually write. Resist the temptation! When you are doing your design, all your "little grey cells" should be focused on the design and your logic. When you code the program, your attention should be on writing correct syntax (because you know the logic is perfect). If you let yourself slip into designing and writing code in one step, you have two problems. First, you have only given half your attention to developing the best design you can. Second, when (not if) you find that you have a logic error, you need to make not just a simple change to your design, but to redo your code.

Some programming languages have special capabilities (commands) that are not available in other languages. Generally, you should design your program so that it is free of those special requirements. That way you can easily transport your design from one language to another. However, if you KNOW that you will be writing in a specific language and will NEVER want to convert the program to another language, you might as well design the program so that it takes advantage of the capabilities.

Advantages and Disadvantages

Pseudocode isn't perfect. We need to be aware of potential pitfalls so that we don't get caught later.

Disadvantages:

- It isn't visual: we don't get a picture of the design.

- There is no universal standard on the style or format, so one company's pseudocode might look very different from another's.

Advantages:

- It can be done easily on a word processor, and therefore
- It is easily modified.
- It implements structured design elements well.

We have already looked at flowcharting, but let's summarize the advantages and disadvantages here for comparison.

Disadvantages:

- It is difficult to modify: a simple change may mean redoing the entire flowchart.
- To do it on a computer, you need a special flowchart package.
- Structured design elements are not implemented.

Advantages:

- It is standardized: all agree on acceptable symbols.
- It is visual (but the advantage has limits as we shall see).

As we know, there is no standardization of the form to use in pseudocode. We will add to our "rules" by developing a format for our design structures. Your instructor may add additional rules for your pseudocode. If you are working as a program designer, your company will certainly have style standards that you must follow.

HOW WE STORE AND ACCESS DATA

Let's go back to our brief pseudocode segment and review what each statement indicates about storing and accessing our data. When we say READ name, hours worked, hourly rate, we are saying that we want the computer to go out to some external input device (a keyboard, a disk, etc.) and retrieve three values. These values will be stored in the computer's memory. Each memory location has an address, and we could use that memory address as the way we refer to the value when giving instructions to the computer. We could say, READ a character value and store it at address 56934. Then, read a numeric value, store it at address 61253, read a second numeric value, and store it at address 67952. Then we could tell the computer to multiply the values at addresses 61253 and 67952 and store the result of that multiplication at address 71234. But as you can see, it would get very cumbersome very soon (and also lead to some major problems in more complex programs). We could also refer to a friend's house by the exact address and say as we walk out the door "I'll be at 5198 Bentana Way, Reston, Virginia 22090, USA, if anyone needs me." Usually, though, we use a more familiar name for the address and we say, "I'll be over at Sarah's if anyone needs me."

The computer refers to the memory locations by address, but our programming languages allow us to identify a name for the memory locations. The value that is stored in memory is called a variable (because the value can vary), and the name we use to refer to it is the variable name. So when we say READ hours worked, we are really saying, "READ a value and store it in a memory address that I will refer to as hours worked." We have to be consistent with our variable names so that there is no confusion when it is time to convert the design to program code. Don't use a variable name of HoursWorked one place, then shorten it to HrsWrkd another place, and simply Hours a third place. Most programming languages have reules about how variable names can be formed. In our pseudocode, we will not limit the number of characters (as some languages do), but we will not allow the name to include a space. Some languages see a capital letter as the same character as a lowercase letter, so the variable name HOURS would refer to the same variable as hours. Other languages distinguish between capitals and lowercase. Consistency is a good habit to develop when working with computers, so (beginning with our next design) we will use lowercase and capitals consistently. We will begin variable names with a lowercase letter, but capitalize the first letter of additional "words" in the same name, as in hoursWorked.

The next line, Gross pay = hours worked times hourly rate, is a different kind of statement: an assignment. This statement is saying, "Multiply the hours worked by the hourly rate and put the result of the multiplication in a memory address that I will call Gross pay." This statement uses three memory addresses. The statement uses the variables hours worked and hourly rate but does not change their values. The result of the multiplication is "assigned" to the variable Gross pay. It is best to read this statement as "Gross pay is assigned the result of hours worked times hourly rate." We don't have a handy symbol on the keyboard to represent assigned other than the equal sign-so we use the equal sign but think of it as "assigned," not as "equal." **Assignments always go from the right side of the = sign to the left side**. Some designers prefer to use the keyword SET for assignment, in which case the statement would be SET Gross pay to hours worked times hourly rate.

The pseudocode ends with the statement WRITE Name, Gross pay, Net pay, which says "Get the values stored under the variables name, Gross pay, and Net pay and send them to the output device."

CALCULATION SYMBOLS

We used text to describe the operations in the calculations, but you are probably used to suing mathematical symbols for the operations. We all recognize the + sign as addition, but you may expect to see ÷ for division and an x for multiplication. Since the ÷ symbol isn't on the keyboard and the x could be easily confused with the letter x used as a variable, we usually use other symbols. These symbols are standard across programming languages and you should get used to seeing (and using) them.

ARITHMETIC SYMBOLS FOR CALCULATIONS

/	Division	+	Addition
*	Multiplication	-	Subtraction
^ or **	Exponentiation		

DESIGN STRUCTURES IN PSEUDOCODE

The **SEQUENCE** structure we have already seen. Since a sequence is really just an ordered list of statements, we implement it in pseudocode by giving an ordered list of pseudocode statements.

The **SELECTION** structure we have seen in its abbreviation from. A complete (two-sided in flowchart terms) IF would look like:

```
IF hoursWorked is greater than 40
        regularPay = hourlyRate times 40
        overtimeHours = hoursWorked minus 40
        overtimePay = overtimeHours * hourlyRate * 2
        grossPay = regularPay plus overtimePay
ELSE
        grossPay = hoursWorked times hourlyRate
ENDIF
```

We have three lines that line up without indentation: the IF which begins the structures, introduces the condition, and also marks the beginning of the "true" section; the ELSE which marks the end of the "true" section and the beginning of the "false" section; and finally the ENDIF which marks the end of the "false" section and of the selection structure as a whole. The indented lines between the IF and the ELSE form the "true" section-the

statements to be performed if the condition is true. The indented lines between the ELSE and the ENDIF form the "false" section-the statements to be performed if the condition is false. We will never do *both* the true and false sections since the condition could not be both true and false at the same time.

We (or the computer) perform the selection or IF structure by first testing the IF condition. If the condition is true, we continue with the statements that follow until we see the ELSE. As soon as we get to the ELSE, we know that we have completed the true section, so we skip the following lines until we get to the ENDIF. We continue "processing" with the line that follows the ENDIF.

If the condition is false, we skip all the statements until we see the ELSE, and then we begin to perform the statements, performing all statements between the SLSE and the ENDIF. When we see the ENDIF, we continue with the line that follows the ENDIF.

In flowcharting, we branched to the right or left after testing the condition and continued along that branch until we hit the connector circle indicating the selection structure was over. In pseudocode, we do the same sort of thing, but this time we "branch" to the top half or the bottom half of the structure.

In flowcharting, we had one-sided selection structures when we only had statements to perform if the condition was true. We can have the same kind of operation in pseudocode; for a selection with only a true "side," we just leave off the ELSE portion.

```
IF hoursWorked is greater than 40
        netPay = netPay minus 10
ENDIF
```

If the condition is false, we scan down looking for the ELSE, but see the ENDIF first, so we know that there is nothing to be done if the condition is false. We continue with the statement that follows the ENDIF.

Usually, we would not have a one-sided selection when the only statements would be in the false section. Most languages require that we have something in the true section. The usual solution is to rewrite the condition so that values that previously would have given a false result will now give a true result. It sounds worse than it is. Assume that we want to write num3 if it is *not* greater than num4. We could write this selection so that we write num3 when the condition num3 greater than num4 is false, as follows.

```
IF num3 greater than num4

ELSE
        WRITE num3
ENDIF
```

But then we have this awkward gap in the true section with nothing to put there. A better way would be to rewrite the condition. What's the reverse of greater than? Less than or

equal to! To say the condition num3 greater than num4 is false means than num3 must be less than num4 or equal to num4. So we can rewrite our IF structure as:

```
IF num3 less than or equal to num4
        WRITE num3
ENDIF
```

Our new structure does the same thing and is much easier to understand.

Just as we used symbols to show arithmetic operations, we frequently use symbols to show relational operations in selection statements (in loop conditions too). Usually when we're writing the condition for a selection statement, we're testing the relation between two variables or values: Is one value less than the other? Is one value greater than or equal to the other? In the previous examples, we used words to describe the relationship, but usually we use the following symbols to indicate these relations.

RELATIONAL CONDITION SYMBOLS

<	Less than	< =	Less than or equal to
>	Greater than	> =	Greater than or equal to
=	Equal to	< >	Not equal to

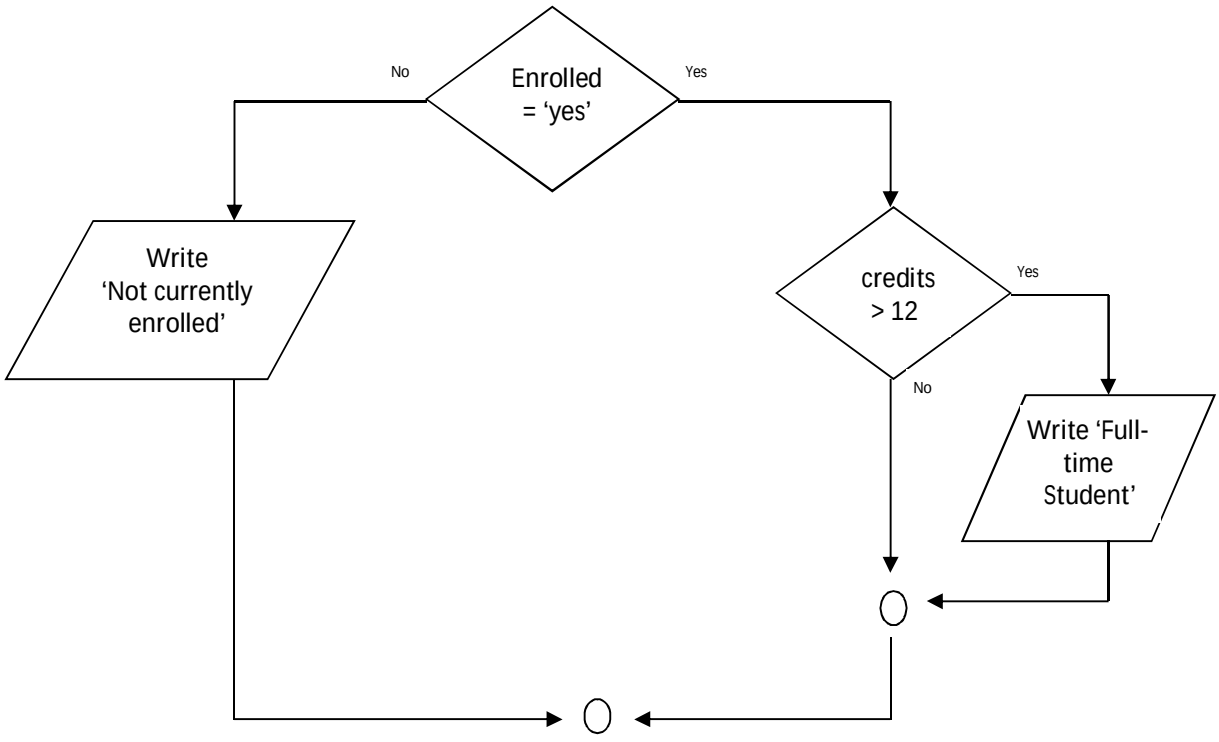
Whenever we compare two values there are only three possible relationships: the first equals the second, the first is less than the second, or the first is greater than the second. Sometimes it is helpful to be able to describe the relationship by what it is not. For example, if the first number is equal to the second, then it is not less than the second and it is not greater than the second. We can summarize these relationships in a table of opposites:

<	is the opposite of	> =
>	is the opposite of	< =
=	is the opposite of	< >

We know that we can put any statement within a selection structure. That means that we can include another selection structure-called a NESTED IF because the second IF is "nested" within the first or outer IF. Nested Ifs are not particularly difficult as long as we pay attention to where the IFs end. A nested IF must be completely contained within either the true section or the false section of the outer IF. Always look for the ENDIF when you are working with nested IFs. For example:

```
IF enrolled = 'yes'
        IF credits > 12
                WRITE 'Full-time student'
        ENDIF
ELSE
        WRITE 'Not currently enrolled'
ENDIF
```

With this IF we have a second IF completely contained in the true section of the outer IF. If the student is currently enrolled for classes, we go into the true section of the IF. In the true section, we encounter a second IF testing the number of credits the student has registered for. IF the student has registered for more than 12 credits, we write the message that the student is a 'Full-time student.' IF the student has enrolled but has 12 credits or fewer, what do we do? Nothing! There is no false (ELSE) section to the inner IF. If the credits condition is false, we usually fall to the statement after the inner ENDIF, but what follows the inner ENDIF here is the ELSE for the outer IF. Since we know we have completed bot the inner IF and the true portion of the outer IF, we fall through to the statement following the complete IF structure. When do we write 'Not currently enrolled'? When the first, outer IF is false. What would this look like in a flowchart?



Try another one.

```
IF month = 'February'
        IF day = 12
                WRITE 'Lincolns Birthday'
        ELSE
                IF day = 22
                        WRITE 'Washingtons Birthday'
                ENDIF
        ENDIF
ELSE
        IF month = 'July'
                WRITE 'Summer at last'
        ENDIF
ENDIF
```

This one's a bit more complicated. Our outer IF is the test for month = 'February.' Inside that IF we have a nested IF in the true section (day = 12) and another nested IF in the false section (month = 'July'). To make matters even worse, we have a third nested IF (day = 22) inside the false section of the first nested IF. Let's try it with some data.

Assume we have a date of March 8. Here's what happens:

month = "February'? False (go to false section)
month = 'July'? False (there is no false section-fall out of IF)

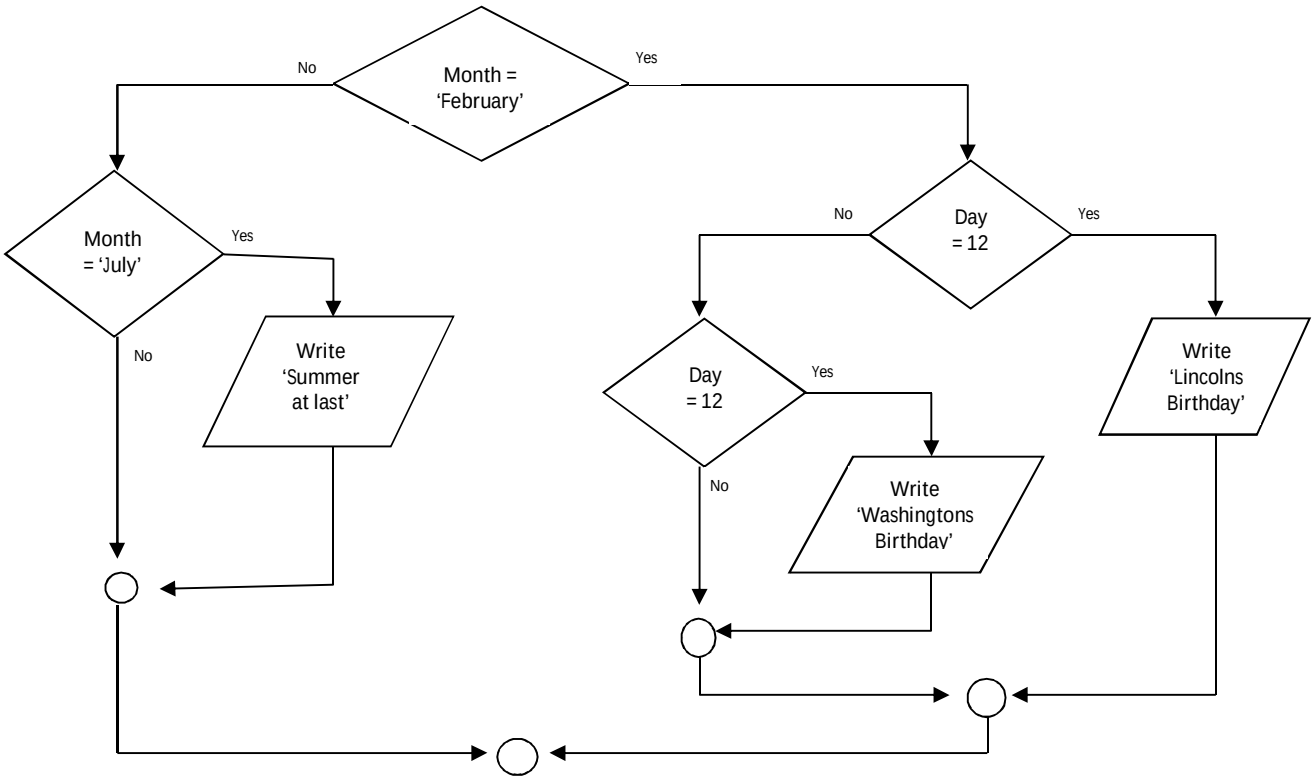
Assume we have a date of March 12. What happen?

month = 'February'? False
month = 'July'? False (Notice we never get to the day = 12 test because that is part of the true section for Month = February)

Assume we have a date of February 22. What happens?

month = 'February'? True (Go into the true section)
day = 12? False (Go into the false section of the nested IF)
day = 22? True (Write the Washingtons Birthday message)

The flowchart for this nested IF is below.



The **LOOP** structure shows pseudocode's greatest advantages over flowcharting. Since a flowchart WHILE loop begins with a decision symbol, it is easy to misread the loop as a selection structure. Unless you carefully follow the flow lines, you cannot distinguish a flowchart loop from a selection. In pseudocode, we use the keyword WHILE to introduce a while loop (seems logical) and the ENDWHILE to mark the end of the loop. The WHILE statement also includes the condition that is used to test whether the loop should be entered. All statements that are within the loop will be indented.

```
count = 0
WHILE count < 10
        READ name
        WRITE name
        ADD 1 to count
ENDWHILE
WRITE 'The End'
```

We have three statements within the WHILE loop. Our loop is controlled by the value of the count variable. As long as count has value that is less that 10, we will go into the loop. When we test the loop condition and count is greater than or equal to 10, we will not enter the loop. Where do we go?-to the statement that follows the ENDWHILE.

The first time we see the WHILE statement, count is 0, which is certainly less than 10, so we enter the loop. We read a name, write a name, and add 1 to count (count is now 1).

Our next statement is ENDWHILE which tells us to return to the WHILE statement to recheck the condition. (ENDWHILE always returns us to the WHILE.) The WHILE statement tests the loop condition (again). If the condition is true, we repeat the loop. If it is false, we exit the loop and continue processing at the first statement after the ENDWHILE.

It is very common (but certainly not required) to put the steps in a loop in a separate module. Our design would then look like the one below.

```
Mainline
count = 0
WHILE count < 10
        DO Process
ENDWHILE
WRITE 'The End'

Process
READ name
WRITE name
ADD 1 to count
```

Notice that we've added a "name" to the first module. Mainline is a common name because the first module acts as a command center, controlling module for the entire program. We can look at the mainline module and get an overview of the program. (We do something in a loop while count goes from 0 to 10 and then we write The End.) The details are pushed down into the lower level module Process. We pass control to the module by using the keyword DO, meaning "Go do the module called Process and when you finish that module, return to the statement that follows."

We will continue to keep our designs so that we have a Mainline module that "controls" the program but doesn't do any details work. Flowcharts have terminal symbols at each end of a module. Pseudocode usually starts each module with a name but does nothing to indicate the end of a module. We can recognize that one module has ended because we're at the beginning (name) of the next module or because we're at the end of the design.

Let's convert this loop into a UNTIL loop. Everything stays pretty much the same, but remember that our UNTIL loop tests the condition at the END of the loop, and we exit the loop when the condition is true. So we need to change the way we phrase the condition.

```
Mainline
Count = 0
REPEAT
        READ name
        WRITE name
        ADD 1 to count
UNTIL count > = 10
WRITE 'The End'
```

The WHILE loop “boundaries” were given by the WHILE and ENDWHILE keywords and the loop controlling condition was included after the WHILE keyword. With and UNTIL loop we are using the keyword REPEAT to mark the beginning of the loop body and UNTIL to mark the end of the loop body. The loop controlling condition follows the UNTIL to emphasize that this is a trailing decision loop structure.