CS 201
Selection Structures (2) and Repetition

Debzani Deb

Multiple-Alternative Decision Form of Nested if

- Nested if statements can become quite complex. If there are more than three alternatives and indentation is not consistent, it may be difficult for you to determine the logical structure of the if statement.
- You can code the nested if as the multiple-alternative decision described below:

```
if ( condition_1 )
   statement_1
else if ( condition_2 )
   statement_2
   .
   .
   .
else if ( condition_n )
   statement_n
else
   statement_e
```


Example

- Given a person's salary, we want to calculate the tax due by adding the base tax to the product of the percentage times the excess salary over the minimum salary for that range.

Salary Range	Base tax	Percentage of Excess
0.00 – 14,999.99	0.00	15
15,000.00 – 29,999.99	2,250.00	18
30,000.00 – 49,999.99	5,400.00	22
50,000.00 – 79,999.99	11,000,00	27
80,000.00 – 150,000.00	21,600.00	33


```
if ( salary < 0.0 )
  tax = -1.0;
else if ( salary < 15000.00 )
  tax = 0.15 * salary;
else if ( salary < 30000.00 )
  tax = (salary – 15000.00)*0.18 + 2250.00;
else if ( salary < 50000.00)
  tax = (salary – 30000.00)*0.22 + 5400.00;
else if ( salary < 80000.00)
  tax = (salary – 50000.00)*0.27 + 11000.00;
else if ( salary <= 150000.00)
  tax = (salary – 80000.00)*0.33 + 21600.00;
else
  tax = -1.0;
```


Order of Conditions in a Multiple-Alternative Decision

- When more than one condition in a multiple-alternative decision is true, only the task following the first true condition executes.
- Therefore, the order of the conditions can affect the outcome.
- The order of conditions can also have an effect on program efficiency.
- If we know that salary range 30,000 - 49,999 are much more likely than the others, it would be more efficient to test first for that salary range. For example,

```
if ((salary>30,000.00) && (salary<=49,999.00))
```


Nested if Statements with More Than One Variable

- In most of our examples, we have used nested if statements to test the value of a single variable.
- Consequently, we have been able to write each nested if statement as a multiple-alternative decision.
- If several variables are involved in the decision, we cannot always use a multiple-alternative decision.
- The next example contains a situation in which we can use a nested if statement as a "filter" to select data that satisfies several different criteria.

Example

- The Department of Defense would like a program that identifies single males between the ages of 18 and 26, inclusive.
- One way to do this is to use a nested if statement whose conditions test the next criterion only if all previous criteria tested were satisfied.
- Another way would be to combine all of the tests into a single logical expression
- In the next nested if statement, the call to `printf` executes only when all conditions are true.

Example

```
/* Print a message if all criteria are met.*/
if ( marital_status == 'S' )
   if ( gender == 'M' )
      if ( age >= 18 && age <= 26 )
         printf("All criteria are met.\n");
```

- or we could use an equivalent statement that uses a single if with a **compound condition**:

```
/* Print a message if all criteria are met.*/
if ((maritial_status == 'S') && (gender == 'M') &&
  (age >= 18 && age <= 26))
   printf("All criteria are met.\n");
```


Common if statement errors

```
if crsr_or_frgt == 'C'
  printf("Cruiser\n");
```

- This error is that there are no ( ) around the condition, and this is a syntax error.

```
if (crsr_or_frgt == 'C');
  printf("Cruiser\n");
```

- This error is that there is a semicolon after the condition. C will interpret this as there is nothing to do if the condition is true.

If Statement Style

- All if statement examples in this lecture have the true statements and false statements indented. Indentation helps the reader but conveys no meaning to the compiler.
- The word `else` is typed without indentation on a separate line.
- This formatting of the if statement makes its meaning more apparent and is used solely to improve program readability.

Tracing an if Statement

- A critical step in program design is to verify that an algorithm or C statement is correct before you spend extensive time coding or debugging it.
- Often a few extra minutes spent in verifying the correctness of an algorithm saves hours of coding and testing time.
- A **hand trace** or **desk check** is a careful, step-by-step simulation on paper of how the computer executes the algorithm or statement.
- The results of this simulation should show the effect of each step's execution using data that is relatively easy to process by hand.
- Sections 4.4 and 4.5 in the text have great step-by-step examples of using if statements to solve problems.

Switch statements

- The switch statement is a better way of writing a program when a series of if-elseif occurs.
- The switch statement selects one of several alternatives.
- The switch statement is especially useful when the selection is based on the value of a single variable or of a simple expression
  - This is called the controlling expression
- In C, the value of this expression may be of type `int` or `char`, but not of type `double`.

Figure 4.12 Example of a switch Statement with Type char Case Labels

```
switch (class) {
case 'B':
case 'b':
        printf("Battleship\n");
        break;

case 'C':
case 'c':
        printf("Cruiser\n");
        break;

case 'D':
case 'd':
        printf("Destroyer\n");
        break;

case 'F':
case 'f':
        printf("Frigate\n");
        break;

default:
        printf("Unknown ship class %c\n", class);
}
```


Explanation of Example

- This takes the value of the variable `class` and compares it to each of the cases in a top down approach.
- It stops after it finds the first case that is equal to the value of the variable `class`.
- It then starts to execute each line of the code following the matching case till it finds a `break` statement or the end of the switch statement.
- If no case is equal to the value of `class`, then the default case is executed.
  - default case is optional. So if no other case is equal to the value of the controlling expression and there is a default case, then default case is executed. If there is no default case, then the entire switch body is skipped.

Remember !!!

- The statements following a case label may be one or more C statements, so you do not need to make multiple statements into a single compound statement using braces.
- You cannot use a string such as ”Cruiser” or ”Frigate” as a case label.
  - It is important to remember that type `int` and `char` values may be used as case labels, but `strings` and type `double` values cannot be used.
- Another *very* common error is the omission of the `break` statement at the end of one alternative.
  - In such a situation, execution ”falls through” into the next alternative.
- Forgetting the closing brace of the switch statement body is also easy to do.
- In the book it says that forgetting the last closing brace will make all following statements occur in the default case, but actually the code will not compile on most compilers.

Nested if versus switch

- Advantages of if:
  - It is more general than a switch
    - It can be a range of values such as `x < 100`
  - A switch can not compare Strings or doubles
- Advantages of switch:
  - A switch is more readable
- Use the switch whenever there are ten or fewer case labels

Common Programming Errors

- Consider the statement:
  if (0 <= x <= 4)
- This is always true!
  - First it does 0 <= x, which is true or false so it evaluates to 1 for true and 0 for false
  - Then it takes that value, 0 or 1, and does 1 <= 4 or 0 <= 4
  - Both are always true
- In order to check a range use (0 <= x && x <= 4).
- Consider the statement:
  if (x = 10)
- This is always true!
  - The = symbol assigns x the value of 10, so the conditional statement evaluates to 10
  - Since 10 is nonzero this is true.
  - You must use == for comparison

More Common Errors

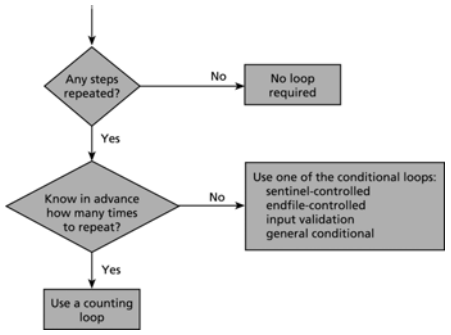
- Don’t forget to parenthesize the condition.
- Don’t forget the { and } if they are needed
- C matches each else with the closest unmatched if, so be careful so that you get the correct pairings of if and else statements.
- In switch statements, make sure the controlling expression and case labels are of the same permitted type.
- Remember to include the default case for switch statements.
- Don’t for get your { and } for the switch statement.
- **Don’t forget your break statements!!!**

Repetition and Loop

Repetition in Programs

- We have learned how to write code that chooses between multiple alternatives.
- It is also useful to be able to write code that repeats an action.
- Writing out a solution to a specific case of problem can be helpful in preparing you to define an algorithm to solve the same problem in general.
- After you solve the specific case, you need to determine whether loops will be required in the general algorithm and if so which loop structure to choose from.

Figure 5.1 Flow Diagram of Loop Choice Process



Counting Loops

- The loop shown below in pseudo code is called a counter-controlled loop (or counting loop) because its repetition is managed by a loop control variable whose value represents a count.

```
Set loop control variable to an initial value of 0
While loop control variable < final value
   ... //Do something multiple times
   Increase loop control variable by 1.
```

- We use a counter-controlled loop when we can determine prior to loop execution exactly how many loop repetitions will be needed to solve the problem.

The While Statement

- This slide shows a program fragment that computes and displays the gross pay for seven employees. The loop body is the compound statements (those between { and }).
- The **loop repetition condition** controls the while loop.

```
count_emp = 0;
while (count_emp < 7)   <-- loop repetition condition
{
   printf("Hours> ");
   scanf("%d",&hours);
   printf("Rate> ");
   scanf("%lf",&rate);
   pay = hours * rate;
   printf("Pay is $%6.2f\n", pay);
   count_emp = count_emp + 1;
}
printf("\nAll employees processed\n");
```


While Statement

- General form:

```
While (loop repetition condition)
{
   //Steps to perform.  These should eventually
   //result in condition being false
}
```

- Syntax of the while Statement:
  - Initialization. i.e. `count_emp = 0;`
  - Testing. i.e. `count_emp < 7`
  - Updating i.e. `count_emp = count_emp + 1;`
- The above steps must be followed for every while loop.
- If any of these are skipped it may produce an **infinite loop**

General While Loops

- In the above example we had `count_emp < 7`, but we may have more or less than 7 employees.
- To make our program fragment more general we should use a `printf/scanf` to get the number of employees and store it is `num_emp`.
- Now we can have `count_emp < num_emp` and our code is more general.

Computing Sum

- If we want to compute $\sum_{i=1}^{100} i$, we need to go 1+2+3+...+100
- We can use a while loop.

```
sum = 0;
currentVal = 1;
while (currentVal <= 100) {
   sum = sum + currentVal;
   currentVal = currentVal + 1;
}
printf("Sum is %d", sum);
```


Compound Assignment Operators

- Several times we have seen:
  `variable = variable <operator> expression;`
  Example: `sum = sum + currentVal;`
- where `<operator>` is a C operator
- This occurs so often, C gives us short cuts.
- Instead of writing `x = x +1` we can write:
  `x += 1.`
- W can use -=, *=, /=, and %= in the same way.

The For Statement

- A better way to construct a counting loop is to use the **for** statement.
- C provides the **for** statement as another form for implementing loops.
- As before we need to
  - **Initialize** the loop control variable
  - **Test** the loop repetition condition
  - **Update** the loop control variable.
- An important feature of the for statement in C is that it supplies a designated place for each of these three components.
- An example of the for statement is shown in the next slide.

For Example

- To compute the sum of 1 to 100:

```
int sum = 0;
int curVal;
for (curVal = 1; curVal <= 100; curVal++)
{
   sum = sum + curVal;
}
```

- Note: `curVal++` is the same as `curVal = curVal + 1` and as `curVal += 1`.

General Form of For statement

```
for (initialize; test; update)
{
  //Steps to perform each iteration
}
```

- First, the initialization expression is executed.
- Then, the loop repetition condition is tested.
- If the condition is true, the statement enclosed in { } are executed.
- After that the update expression is evaluated.
- Then the loop repetition condition is retested.
- The statement is repeated as long as the condition is true.
- For loop can be used to count up or down by any interval.

Program Style

- For clarity, it can be useful to place each expression of the for heading on a separate line.
- If all three expressions are very short, we will place them together on one line, like we did in the example.
- The body of the for loop is indented just as the if statement.

Increment and Decrement Operators

- The counting loops that we have seen have all included assignment expressions of the form
  `counter = counter + 1`
  or
  `counter++`
  or
  `counter += 1`
- This will add 1 to the variable counter. If we use a - instead of a +, it will subtract 1 from the variable counter.
- Be careful about using the ++ or -- options.

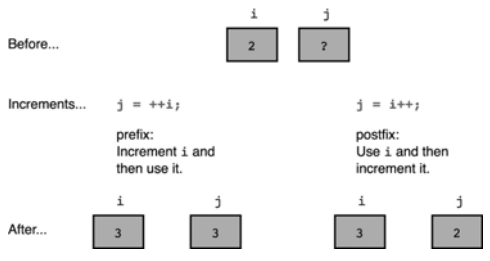
Increment and Decrement Other Than 1

- Instead of adding just 1, we can use `sum = sum + x` or `sum += x`
- Both of these will take the value of `sum` and add `x` to it and then assign the new value to `sum`.
- We can also use `temp = temp -x` or `temp -= x`
- Both of these will take the value of `temp` and subtract `x` from it and then assign the new value to `temp`.

Prefix and Postfix Increment/Decrement

- The values of the expression in which the ++ operator is used depends on the position of the operator.
- When the ++ operator is placed immediately in front of its operand (prefix increment, Ex: `++x`), the value of the expression is the variable's value after incrementing.
- When the ++ operator is placed immediately after the operand (postfix increment , Ex: `x++`), the value of the expression is the value of the variable before it is incremented.

Figure 5.6 Comparison of Prefix and Postfix Increments



More on prefix and postfix operator

- If n = 4, what will be the output of the following?

```
printf("%3d", --n);     printf("%3d", n--);
printf("%3d", n);       printf("%3d", n);
```

3 3

4 3

Conditional Loops

- In many programming situations, we will not be able to determine the exact number of loop repetitions before loop execution begins.
- Below is an example where we do not know how many times our program will repeat.

Example

- We need a program that prompts the user for a `value` and multiplies it by the value of the variable `temp`. It then stores the result in `temp`. It keeps doing this until the user enters a 0.
- The outline of the program would be as follows:

```
assign temp the value of 1
prompt the user for a value
while value does not equal 0
  assign temp the value of temp times value
  prompt the user for a value
output the value of temp
```


Program Fragment

```
temp = 1;
printf("Enter a value, 0 will stop the program> ");
scanf("%d",&value);
```
Initialization

```
while(value != 0) {
   temp = temp * value;
   printf("Enter a value, 0 will stop the program>");
   scanf("%d",&value);
}
printf("The product is %d", temp);
```
Testing

Update

- It is very common for loops to have identical initialization and update steps while performing input operations where the number of input values is not known in advance.

Sentinel Controlled Loops

- Many programs with loops input one or more additional data items each time the loop body is repeated.
- Often we don't know how many data items the loop should process when it begins execution.
- We must find some way to signal the program to stop reading and processing new data.
- One way to do this is to instruct the user to enter a unique data value, called a **sentinel** value, after the last data item.
- The loop repetition condition tests each data item and causes loop exit when the sentinel value is read.
- This is what we did in the previous example: use the value 0 to stop the loop.

Figure 5.10 Sentinel-Controlled while Loop

```
1.  /* Compute the sum of a list of exam scores. */
2.
3.  #include <stdio.h>
4.
5.  #define SENTINEL -99
6.
7.  int
8.  main(void)
9.  {
10.     int sum = 0,   /* output - sum of scores input so far */
11.         score;     /* input - current score */
12.
13.     /* Accumulate sum of all scores. */
14.     printf("Enter first score (or %d to quit)> ", SENTINEL);
15.     scanf("%d", &score);       /* Get first score. */
16.     while (score != SENTINEL) {
17.         sum += score;
18.         printf("Enter next score (%d to quit)> ", SENTINEL);
19.         scanf("%d", &score);   /* Get next score. */
20.     }
21.     printf("\nSum of exam scores is %d\n", sum);
22.
23.     return (0);
24. }
```


Sentinel Controlled for loop

- Because the for statement combines the initialization, test, and update in once place, some programmers prefer to use it to implement sentinel-controlled loops.

```
printf("Enter first score (or %d to quit)> ", sentinel);
for( scanf("%d",&score);
     score != sentinel;
     scanf("%d",&score))
{
   sum += score;
   printf("Enter next score (%d to quit)> ", sentinel);
}
```