```
    printf("I like pizza\n");
```

Code Block (more than a single statement)

```
if (likePizza == 1)
{
     printf("I like pizza\n");
     printf("I want more pizza\n");
}
```

The program executes the sequence only if `likePizza` is equal to `1`. Otherwise, the program bypasses the sequence altogether.

Alternative Paths

The C language supports two ways of describing alternative paths: an binary select construct and a multiple selection construct.

Binary Selection

The binary selection construct executes one of a set of alternative sequences. This construct takes the form

```
if (condition)
   sequence
else
   sequence
```

Parentheses enclose the condition, which may be a relational expression or a logical expression. The sequences may be ***single statements*** or ***code blocks***. The program executes the sequence following the if only if the condition is true. The program executes the sequence following the ***else*** only if the condition is false.

Single Statement

```
if (likePizza == 1)
    printf("I like pizza\n");
```

```
else
    printf("I hate pizza\n");
```

Code Block (more than a single statement)

```
if (likePizza == 1)
{
    printf("I like pizza\n");
}
else
{
    printf("I hate pizza\n");
    printf("I don't want pizza\n");
}
```

> **READABILITY AND MAINTAINABILITY**
>
> Although it is not required to create a code block for single-line statements, it is suggested for maximum readability and maintainability of code that you create a code block.
>
> All code examples provided in these notes will assume this suggested style including the placement of the opening and closing curly braces be on their own dedicated lines (Allman).

Multiple Selection

For three alternative paths, we append an ***if else*** construct to the ***else*** keyword.

```
if (condition)
   sequence
else if (condition)
   sequence
else
   sequence
```

If the first condition is true, the program skips the second and third sequences. If the first condition is false, the program skips the first sequence and evaluates the second condition. The program executes the second sequence only if the first condition is false and the second

condition is true. The program executes the third sequence and skips the first two only if both conditions are false.

Compound Conditions

The condition in a selection construct may be a ***compound*** condition. A compound condition takes the form of a logical expression (see the section on ***Logical Expressions*** in the chapter on Expressions).

```
if (age > 12 && age < 16)
{
    printf("Student Fare - no id required\n");
}
else if (age > 15 && age < 20)
{
    printf("Student Fare - id is required\n");
}
else if (age < 13)
{
    printf("Child ride for free!\n");
}
else if (age >= 65)
{
    printf("Senior Fare - id is required\n");
}
else
{
    printf("Adult Fare\n");
}
```

Case-by-Case

The case-by-case selection construct compares a condition - simple or compound - against a set of constant values or constant expressions. This construct takes the form:

```
switch (condition)
{
case constant:
       sequence
       break;
case constant:
```

```
        sequence
        break;
default:
        sequence
}
```

If the condition matches a constant, the program executes the sequence associated with the case for that constant. The break; statement transfers control to the closing brace of the switch construct. Braces around the statements between case labels are unnecessary.

If a break statement is missing for a particular case, control flows through to the subsequent case and the program executes the sequence under that case as well.

The program executes the sequence following default only if the condition does not match any of the case constants. The **`default`** case is optional and this keyword may be omitted.

For example, the following code snippet compares the value of choice to **`'A'`** or **`'a'`**, **`'B'`** or **`'b'`**, and **`'C'`** or **`'c'`** until successful. If unsuccessful, the code snippet executes the statements under **`default`**.

```
char choice;
double cost;

printf("Enter your selection (a, b or c) ? ");
scanf("%c", &choice);

switch (choice)
{
case 'A' :
case 'a' :
    cost = 1.50;
    break;
case 'B' :
case 'b' :
    cost = 1.10;
    break;
case 'C' :
case 'c' :
    cost = 0.75;
    break;
default:
```

```
    choice = '?';
    cost = 0.0;
}

printf("%c costs %.2lf\n", choice, cost);
```

Conditional Expression

The ***conditional expression*** selection construct is shorthand for the ***alternative path*** construct. This ternary expression combines a condition and two sub-expressions using the `? :` operators:

```
 condition ? operand : operand
```

If the condition is true, the expression evaluates to the operand between `?` and `:`. If the condition is false, the expression evaluates to the operand following `:`.

Example

```
#include <stdio.h>

int main(void)
{
    int minutes;
    char s;

    printf("How many minutes left ? ");
    scanf("%d", &minutes);

    s = minutes > 1 ? 's' : ' ';    // Conditional Expression

    printf("%d minute%c left\n", minutes, s);

    return 0;
}
```

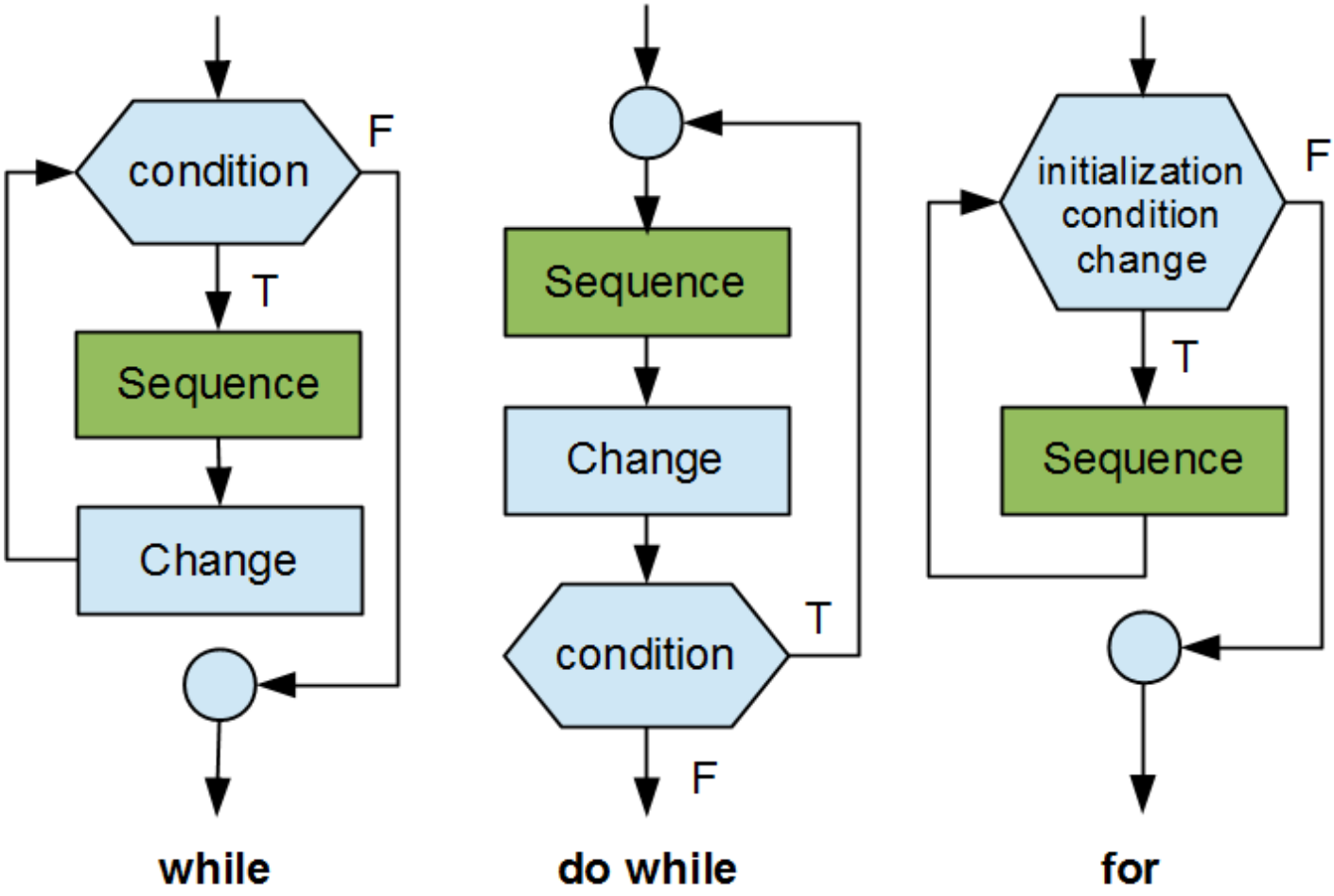
If the operands in a conditional expression are themselves expressions, the conditional expression only evaluates the operand identified by the condition.

Iteration Constructs

The C language supports three iteration constructs:

- while
- do while
- for

Three instructions control the execution of an iteration: an initialization, a test condition, a change statement. The test condition may be simple or compound. The flow charts for the three constructs are shown below.



If the change statement is missing or if the test condition is always satisfied, the iteration continues without terminating and the program can never terminate. We say that such an iteration is an infinite loop.

while

The ***while*** construct executes its sequence as long as the test condition is true. This construct takes the form:

```
while (condition)
{
    sequence
}
```

Example

```
slices = 4;
while (slices > 0)
{
    slices--;
    printf("Gulp! Slices left %d\n", slices);
}
```

The above code produces the following output:

```
Gulp! Slices left 3
Gulp! Slices left 2
Gulp! Slices left 1
Gulp! Slices left 0
```

If the condition is never true (for example, if initially slice = 0), this construct never executes the sequence.

do while

he ***do while*** construct executes its sequence ***at least once*** and continues executing it as long as the test condition is true. This construct takes the form:

```
do {
    sequence
} while (condition);
```

Example

```
slices = 4;
do {
    slices--;
    printf("Gulp! Slices left %d\n", slices);
} while (slices > 0);
```

The above code produces the following output:

```
Gulp! Slices left 3
Gulp! Slices left 2
Gulp! Slices left 1
Gulp! Slices left 0
```

If we change the initial value to slices = 12 and the test condition to slices < 5, this iteration displays once and stops because the test condition is false.

```
slices = 12;
do {
    slices--;
    printf("Gulp! Slices left %d\n", slices);
} while (slices < 5);
```

The above code produces the following output:

```
Gulp! Slices left 11
```

This code contains a ***semantic error***: if the initial value was 5, the iteration would never end!

for

The ***for*** construct groups the initialization, test condition and change together, separating them with semi-colons. This construct takes the form:

```
for (initialization; condition; change)
{
```

```
    sequence
}
```

Example

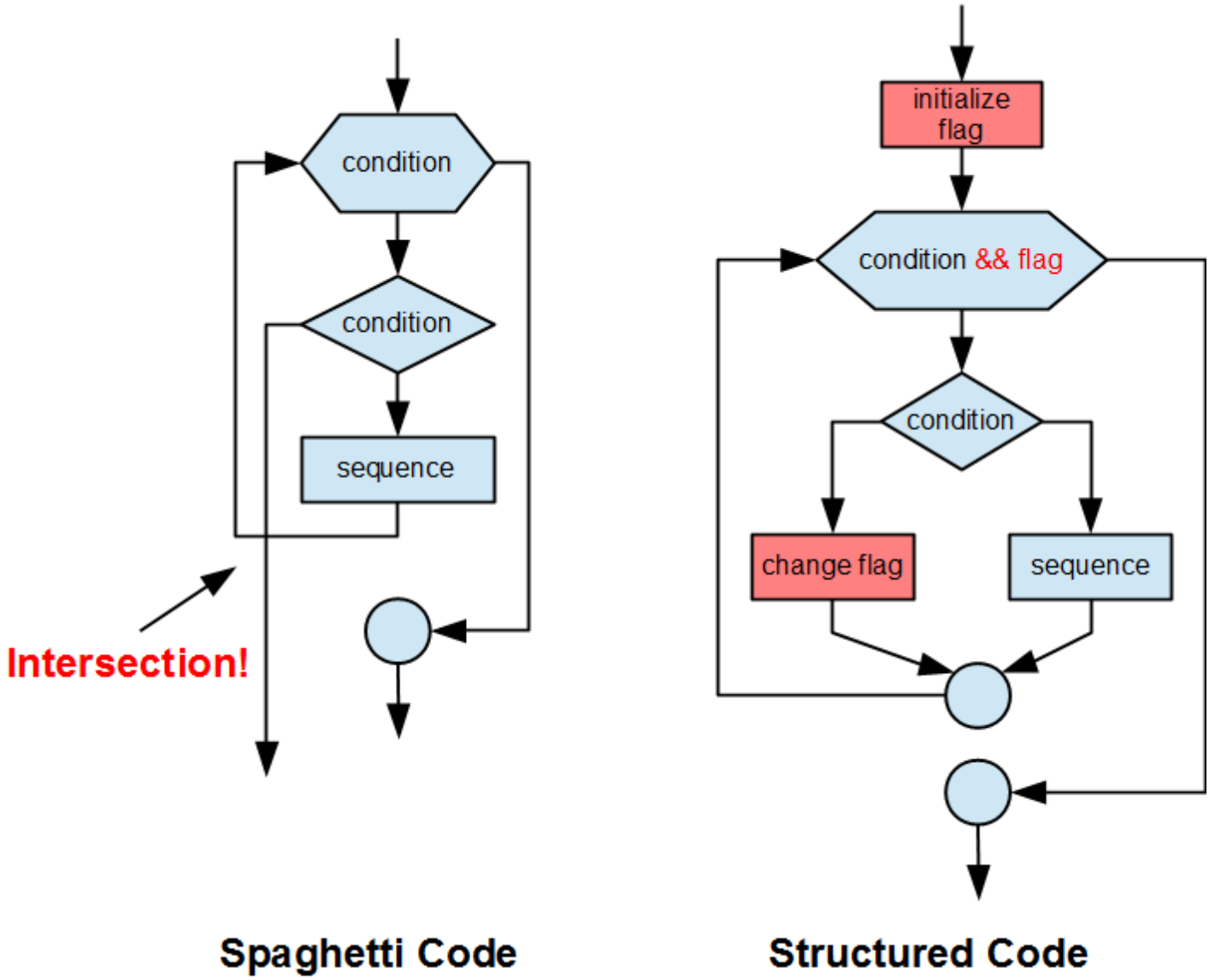
```
for (slices = 4; slices > 0; --slices)
{
    printf("Gulp! Slices left %d\n", slices - 1);
}
```

The above code produces the following output:

```
Gulp! Slices left 3
Gulp! Slices left 2
Gulp! Slices left 1
Gulp! Slices left 0
```

Flags

Flagging is a method of coding iteration constructs within the ***single-entry single-exit principle*** of structured programming. Consider the flow-chart on the left side in the figure below. This design contains a path that crosses another path.



Flags are variables that determine whether an iteration continues or stops. A flag is either true or false. Flags helps ensure that no paths cross one another. By introducing a flag, we avoid the jump and multiple exit, obtain a flow chart where no path crosses any other and hence an improved design.

Example

The following code example demonstrates a flag to terminate the iteration prematurely.

```
#include <stdio.h>

int main(void)
{
   int value;
   int done = 0;  // flag
```

```
    int total = 0; // accumulator

    for (i = 0; i < 10 && done == 0; i++)
    {
        printf("Enter integer (0 to stop) ");
        scanf("%d", &value);

        if (value == 0)
        {
            done = 1;
        }
        else
        {
            total += value;
        }
    }

    printf("Total = %d\n", total);

    return 0;
}
```

Example execution of the code above:

```
Enter integer (0 to stop) 45
Enter integer (0 to stop) 32
Enter integer (0 to stop) 3
Enter integer (0 to stop) -6
Enter integer (0 to stop) 0
Total = 74
```

The test condition is compound (logical expression) due to the evaluation of both, the iterator `i` and the flag `done`. If **`done == 1`**, the iteration stops.

> **IMPORTANT**
>
> Until you learn how to evaluate and rationalize when to break the single-entry single-exit principle, you should apply the control flag approach to eloquently manage process flow. Listed below are some cases to avoid:

- `break` (the `switch` construct should be the only construct using this, and only 1 per case)
- `continue`
- `exit`
- `goto`
- `return`
- `exit`
- iterator variable (should only be changed in a single place)

Avoid Jumps (Optional)

Designing a program with jumps or intersecting paths makes it more difficult to read. We refer to program code that contains paths that cross one another as spaghetti code. The roots of spaghetti coding lie in assembly languages (second-generation languages). Assembly languages include jump instructions. Jump instructions migrated to high-level languages as assembly language programmers started coding in higher-level languages. Spaghetti code was a serious problem in the 1960's. To improve readability, many programmers started to advocate complete avoidance of jump statements and introduced ***flags*** as the good-design alternative.

Nested Constructs

Enclosing one logic construct within another is called ***nesting***.

Nested Selections

A selection within another selection is called a ***nested selection***.

Example

```
if (grade < 50)
{
    if (sup == 1)
    {
        printf("Sup\n");
    }
    else
```

```
    {
        printf("Failed\n");
    }
}
else
{
    printf("Pass\n);
}
```

Dangling Else

An ambiguity arises in a ***nested if else*** construct that contains an optional sequence (***if***). Consider the following code snippet:

```
// Problem: Ambiguity
if (grade < 50)
    if (sup == 1)
        printf("Sup\n");
else // <-- Does this belong to 'if (grade < 50)' OR 'if (sup == 1)'?
    printf("Pass\n);
```

It is unclear as to which `if` the `else` belongs:

```
// Interpretation #1:
if (grade < 50)  // <-- The formatting suggests to this 'if'
    if (sup == 1)
        printf("Sup\n");
else
    printf("Pass\n);
```

... OR if the `else` is indented ...

```
// Interpretation #2:
if (grade < 50)
    if (sup == 1) // <-- now the formatting suggests this 'if'
        printf("Sup\n");
    else
        printf("Pass\n);
```

The C language always attaches the dangling `else` to the **innermost** `if` (regardless of how it is indented, interpretation #2 above is how it would be executed).

To guarantee the desired behaviour (interpretation #1 from above), we use code blocks (curly braces) to ensure the intended flow:

```
if (grade < 50)
{
    if (sup == 1)
    {
        printf("Sup\n");
    }
}
else
{
    printf("Pass\n);
}
```

Nested Iterations

An iteration within another iteration is called a ***nested iteration***.

The program below includes a nested iteration:

```
// Rows and Columns
// row_columns.c

#include <stdio.h>

int main(void)
{
    int i, j;

    for (i = 0; i < 5; i++)
    {
        for (j = 0; j < 5; j++)
        {
            printf("%d,%d  ", i, j);
        }

        printf("\n");
```

```
        }

        return 0;
    }
```

The output of the code above:

```
0,0  0,1  0,2  0,3  0,4
1,0  1,1  1,2  1,3  1,4
2,0  2,1  2,2  2,3  2,4
3,0  3,1  3,2  3,3  3,4
4,0  4,1  4,2  4,3  4,4
```

Style Guidelines

Learning Outcomes

After reading this section, you will be able to:

- Self-document programs using comments and descriptive identifiers

Introduction

A well-written program is a pleasure to read. The coding style is consistent and clear throughout. The programmer looking for a bug sees a well-defined structure and finds it easy to focus on the portion of the code that is suspect. The programmer looking to upgrade the code sees how and where to incorporate changes. Although several programmers may have contributed to the code throughout its lifetime, the code itself appears to have been written by one programmer.

This chapter describes the coding style used throughout these notes and recommended for an introductory course in programming. The style is referred to as the Allman coding style.

Identifiers

All identifiers in a program should be self-descriptive. The reader should not have to search through the code for their meaning. It is better to embed the meaning in the name, rather than to explain it in a comment elsewhere in the code. By all means, avoid referring the reader to a document external to the code itself.

A program with short names is easier to read than one with long names. The human eye infers the meaning of a word from just a few letters that make up that word and the context within which the word is used. Reading long identifiers tires the eyes when searching through code. We follow the sophisticated conventions of our own languages and complying with them makes our programs all the more readable. Nouns describe objects, verbs describe actions.

Notations, such as Hungarian notation, that incorporate the type into the identifier will clutter source code unnecessarily. C compilers know the type of each identifier and readers do not need

reminders in every place the identifier appears.

When selecting identifiers:

- adopt self-descriptive names, adding comments only if clarification is necessary
- prefer nouns for variable identifiers
- keep variable identifiers short - `temp`, rather than `temporary`, `id`, rather than `identification`,
- avoid cryptic identifiers - use just enough letters for the eye to infer the meaning from the context but no less (if you want to represent 'amount owed', `ao` is cryptic, while `amtOwed` is clear but concise)
- keep the identifiers of counters very short - use `i` rather than `loop_counter`, and `n` rather than `numberOfIterations`. This is context dependant and should only be applied to iterators and counters otherwise, the name becomes meaningless or cryptic.
- avoid decorating the identifier with Hungarian or similar notations (data type)
- use "camelNotation" (first letter of each word is capitalized with the exception of the first word)
- avoid underscore characters which are commonly used in system libraries to avoid conflicts

Layout

Professionals in the field of human-computer interaction confirm that layout and arrangement affects comfort and accessibility. Poorly laid out code frustrates and promotes misreading's.

Typographers, artists, and photographers know that negative space surrounding an image is as important as the image itself. Space itself can visually separate, making it easier to find something and draw attention to a certain part of a page.

Layout tools at our disposal include:

- indentation
- line length
- braces
- spaces
- comments

Indentation

Indentation helps define where a code block starts and ends, clearly showing the structure of our logic. The recommended indent in C programs is a tab of 4 or 8 characters.

Example:

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        for (k = 0; k < n; k++)
        {
            int ijk = i * j * k;
            if (ijk != 0)
            {
                printf(" %4d", ijk);
            }
            else
            {
                printf("    ");
            }
        }
        printf("\n");
    }
    printf("\n");
}
printf("That's all folks!!!\n");
```

Using tabs for indentation rather than spaces enables other programmers to adjust the indentation without difficulty in their own text editors. Using 8 characters per tab position heavily indents code to the far right and identifies code that may be a hog of compute cycles and a likely candidate for refactoring.

To minimize the effects of indentation with switch constructs, we align the subordinate case labels with the switch keyword:

```
switch(c)
{
case 'A' :
```

```
case 'a' :
        cost = 1.50;
        break;
case 'B' :
case 'b' :
        cost = 1.10;
        break;
case 'C' :
case 'c' :
        cost = 0.75;
        break;
default:
        c = '?';
        cost = 0.0;
}
```

Line Length

The practical limit on line length is 80 columns, including indentation. Many windows default to an 80-column width and break longer lines into chunks that are then difficult to read. Lines longer than 80 columns either truncate or wrap in hard copy printouts, which confuses readers.

String literals pose a special problem. We break them into a set of sub-string literals separated only by whitespace. C compilers discard the whitespace and concatenate the sub-string literals into a single string literal.

```
printf("%d substrings"
       " display as a"
       " single string"
       "\n", 3);
```

Produces the following output:

```
3 substrings display as a single string
```

Braces

The style of bracing used in these notes is that proposed by Eric Allman. We put the opening brace on its own line indented to the preceding statement and the closing brace on its own line in alignment with the opening brace.

```
if (i == 7)
{
    cost = 1.75;
    printf("Congrats!\n");
}
```

```
if (i == 7)
{
    cost = 1.75;
    printf("Congrats!\n");
}
else
{
    cost = 2.75;
    printf("Good luck next time!\n");
}
```

The opening and closing braces for a `do/while` construct is an exception:

```
do {
    printf("Guess i : ");
    scanf("%d", &i);
    if (i == 7)
    {
        cost = 1.75;
        printf("Congrats!\n");
    }
    else
    {
        cost = 2.75;
        printf("Good luck next time!\n");
    }
} while (i != 7);
```