Logic

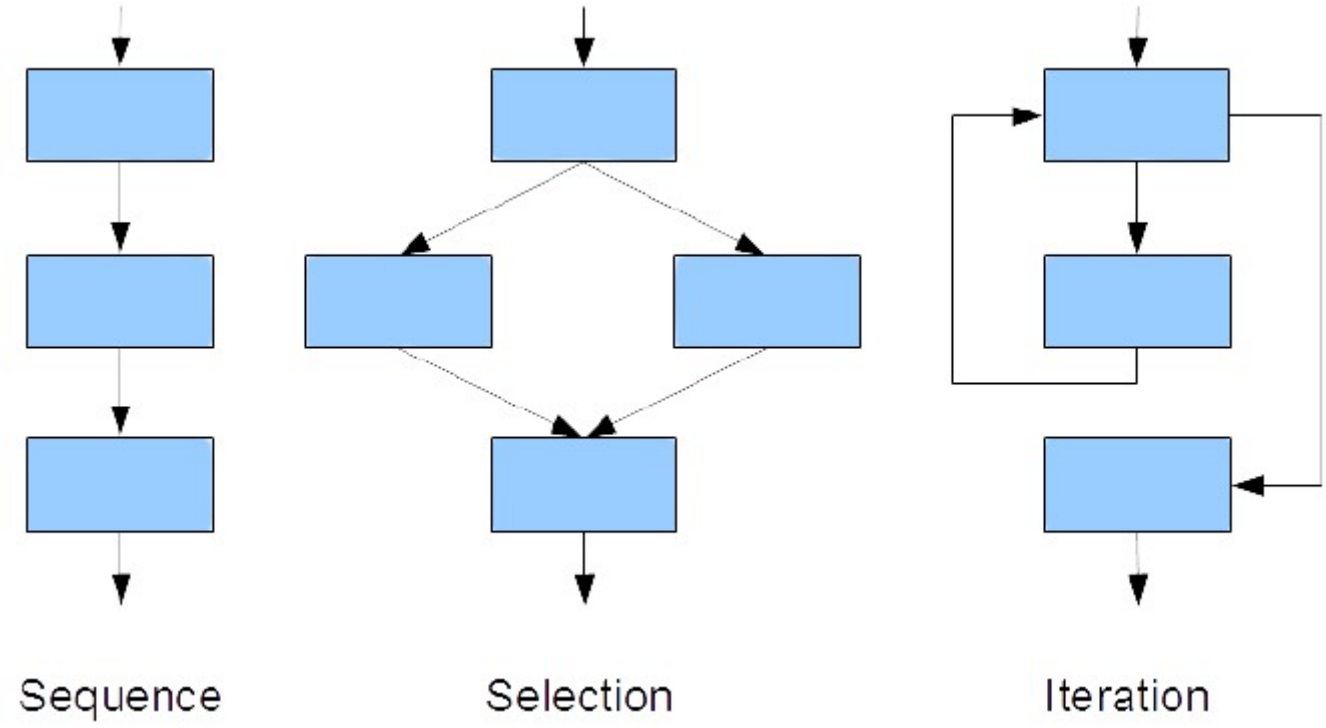
Learning Outcomes

After reading this section, you will be able to:

- Design procedures using selection and iteration constructs to solve a programming task

Introduction

A complete programming language includes facilities to implement sequential constructs, in which one statement follows another and the statements are executed in order, and two other constructs, which represent modifications of sequential constructs. Selection constructs represent different paths through the set of instructions. Iteration constructs represent repetition of the same set of instructions until a specified condition has been met. The three classes of constructs required to complete a programming language are illustrated in the figure below.



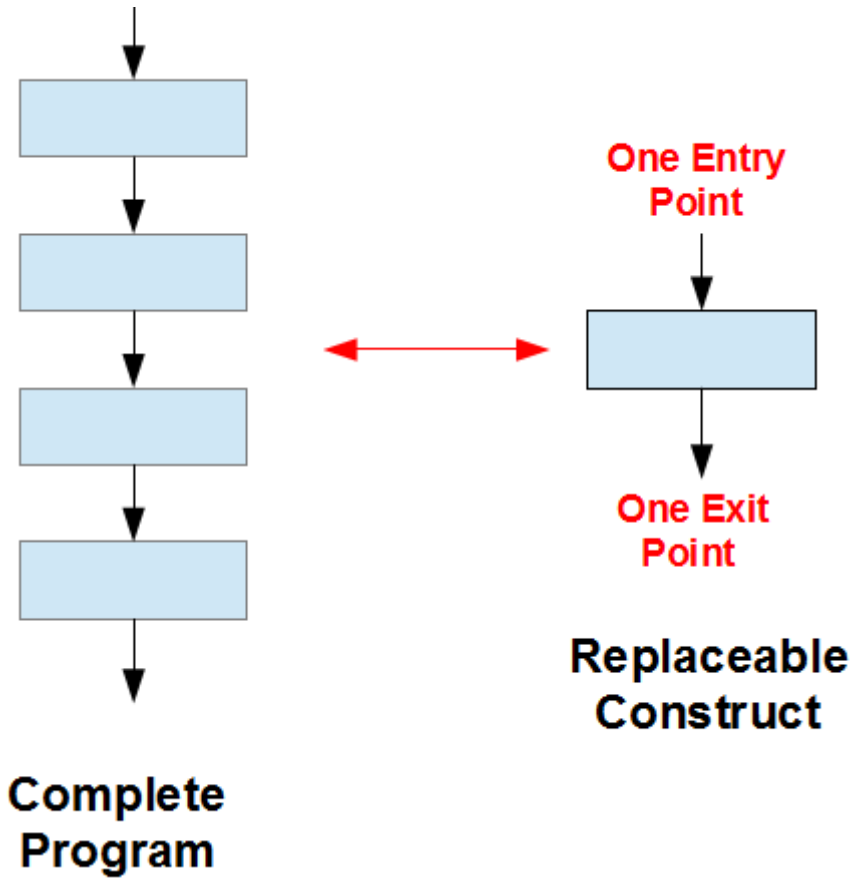
Since programmers who maintain application software are typically not those who develop that software originally and the maintenance programmers may change throughout the lifetime of a software application, it is critical that the software is not only readable but also easy to upgrade and

maintain. The principles of structured programming, which were developed in the 1960s, provide important coding guidelines that respect this objective.

This chapter introduces the selection and iteration constructs supported by the C language and describes how to implement structured programming principles in coding iterations.

Structured Programming

A structured program consists of sets of simple constructs, each of which has one entry point and one exit point. Any programmer may replace one construct with an upgraded construct without affecting the other constructs in the program or introducing errors ("bugs").



The simplest example of a structured construct is a ***sequence***. A sequence is either a simple statement or a code block. A ***code block*** is a set of statements enclosed in a pair of curly braces to be executed sequentially.

Example Simple Statement

```
// single statement
printf("I like pizza\n");
```

Example Code Block

```
// code block (upgrade)
{
    printf("I like pizza\n");
    printf("I want more pizza\n");
}
```

Unlike a single statement, a C code block does not require a terminating semi-colon of its own (after the closing brace).

Preliminary Design

During the design stage of a programming solution, it is helpful to outline the steps involved. Well-established techniques include:

- pseudo-coding
- flow charting

Clear and concise pseudo-code or flow charts improve chances are that our coding will also be clear and concise.

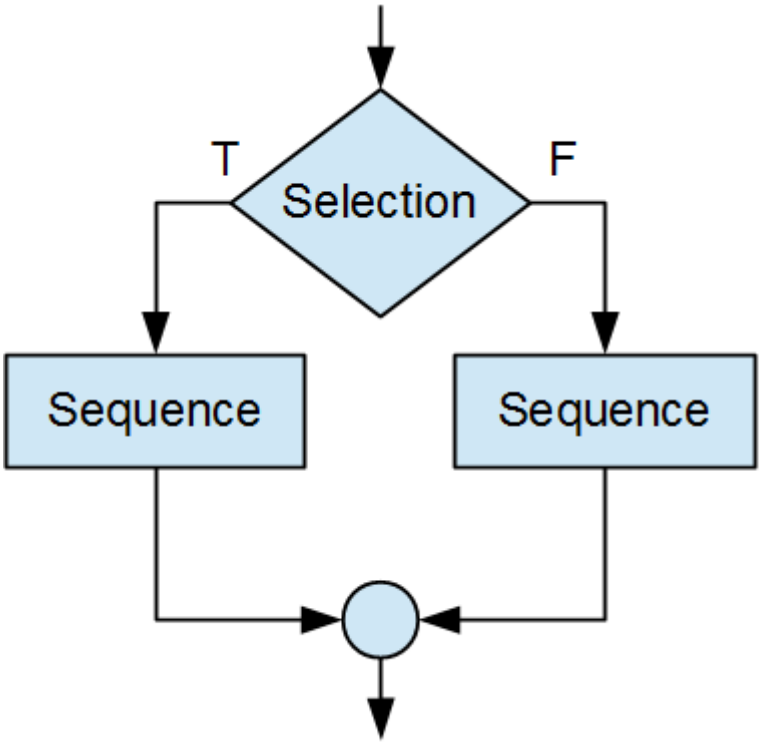
Pseudo-Code

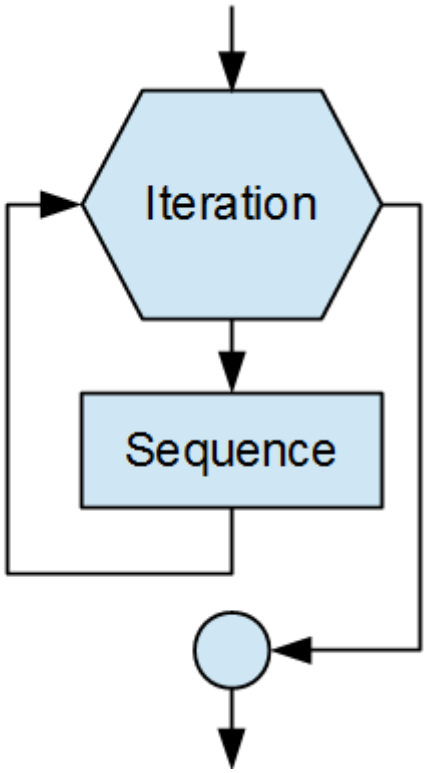
Pseudo-code is a set of shorthand notes in a human (non-programming) language that itemizes the key steps in the sequence of instructions that produce a programming solution. For example, the pseudo code for calculating the change in a vending machine might look something like

1. Declare variables for quarters and nickels
2. Calculate the number of quarters in the change
3. Calculate the remainder to be returned in nickels
4. Output the change in quarters and nickels

Flow Charts

A ***flow chart*** is a set of conventional symbols connected by arrows that illustrate the flow of control through a programming solution. Popular sets of symbols for sequences, selections and iterations are shown below:





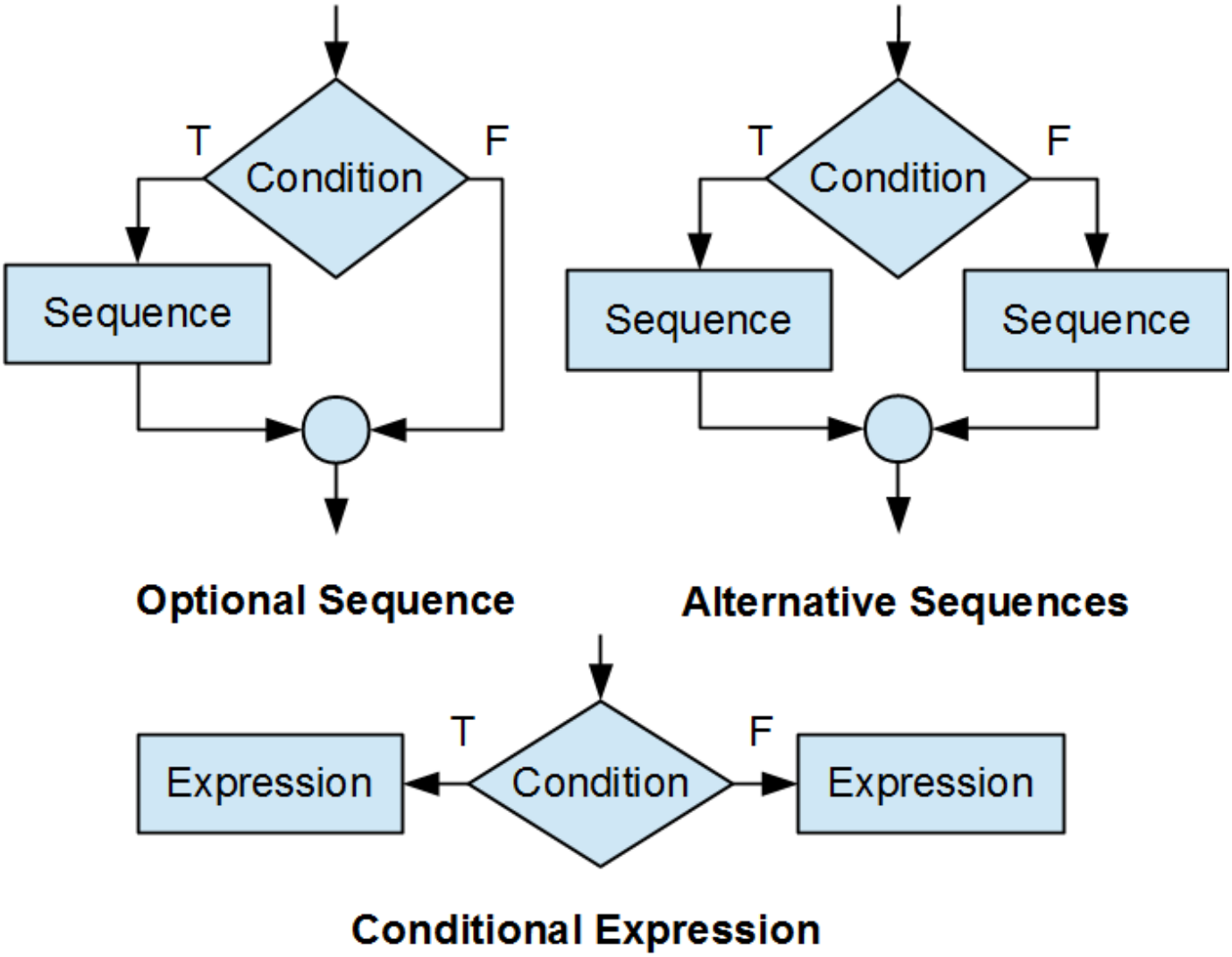
Usage of these sets with the C language is illustrated below.

Selection Constructs

The C language supports three selection constructs:

- optional path
- alternative paths
- conditional expression

The flow charts for these three constructs are shown below:



Optional Path

The simplest selection construct executes a sequence only if a certain condition is satisfied; that is, only if the condition is true. This optional selection takes the form:

```
if (condition)
   sequence
```

Parentheses enclose the condition, which may be a relational expression or a logical expression. The sequence may be a single statement or a code block.

Single Statement

```
if (likePizza == 1)
    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

```c
#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;
}
```

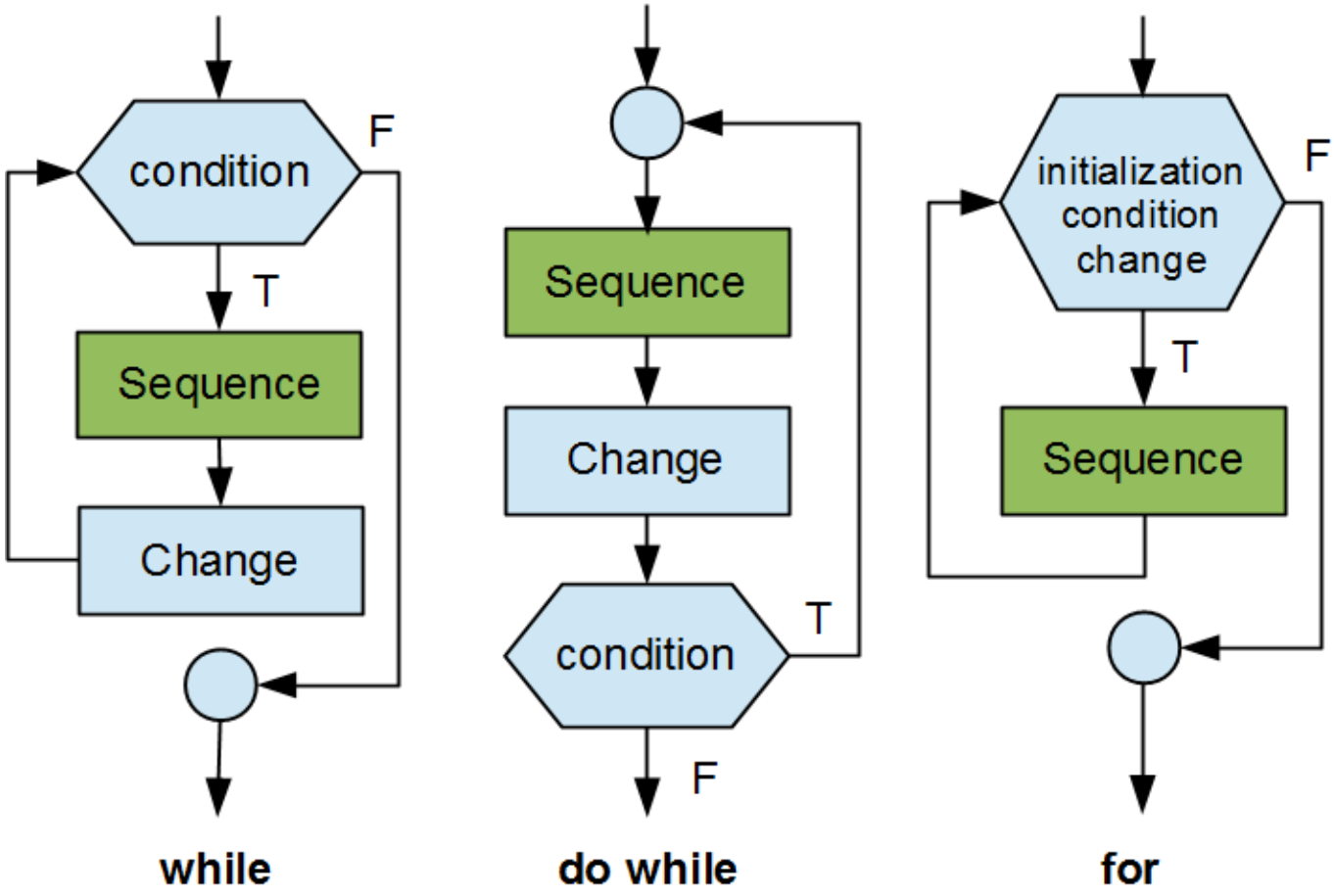
If the operands in a conditional expression are themselves expressions, the conditional expression only evaluates the operand identified by the condition.

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:

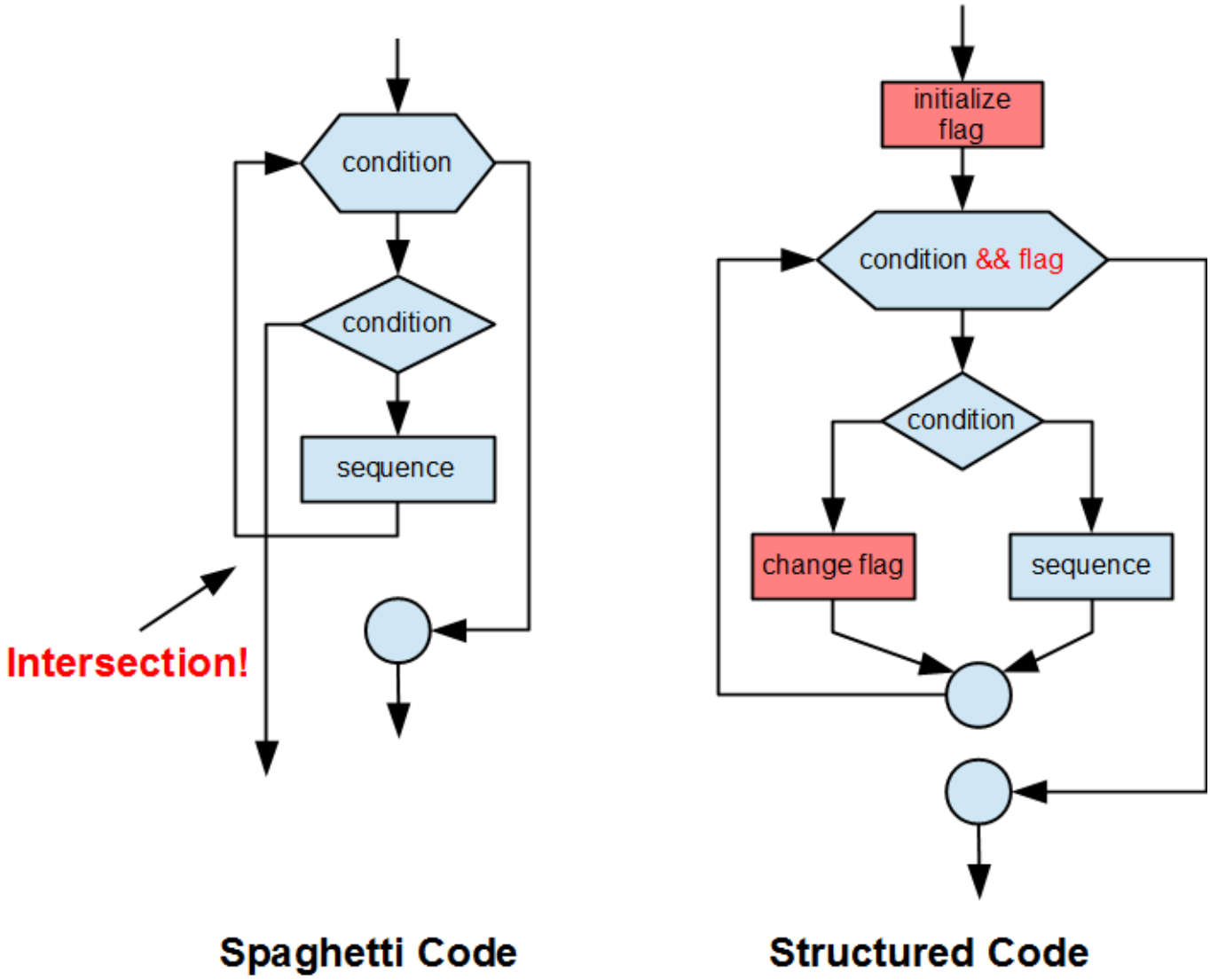
```
for (initialization; condition; change)
{
    sequence
}
```

Example

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

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
```