By applying ***deMorgan's law***, we can often re-write a compound condition in a more readable form.

Shorthand Assignments

The C language also supports shorthand operators that combine an arithmetic expression with an assignment expression. These operators store the result of the arithmetic expression in the left operand.

Integral Operands

C has 5 binary and 2 unary shorthand assignment operators for integral (`int` and `char`) operands.

Binary Operands

The binary operators yield the same result as shown in the longhand expressions listed alongside:

Expression	Shorthand	Longhand	Meaning
operand `+=` operand	`i += 4`	`i = i + 4`	add 4 to i and assign to i
operand `-=` operand	`i -= 4`	`i = i - 4`	subtract 4 from i and assign to i
operand `*=` operand	`i *= 4`	`i = i * 4`	multiply i by 4 and assign to i
operand `/=` operand	`i /= 4`	`i = i / 4`	divide i by 4 and assign to i
operand `%=` operand	`i %= 4`	`i = i % 4`	remainder after i / 4 and assign to i

Unary Operands

The unary operators yield the same result as shown in the longhand expressions listed alongside:

Expression	Shorthand	Longhand	Meaning
++operand	`++i`	`i = i + 1`	increment i by 1
operand++	`i++`	`i = i + 1`	increment i by 1
--operand	`--i`	`i = i - 1`	decrement i by 1
operand--	`i--`	`i = i - 1`	decrement i by 1

We call the unary operator that precedes its operand a ***prefix*** operator and the unary operator that succeeds its operand a ***postfix*** operator.

The difference between the ***prefix*** and ***postfix*** expressions is in the value of the expression itself. The ***prefix*** operator changes the value of its operand and sets the expression's value to be the changed value. The ***postfix*** operator sets the expression's value to the operand's original value and then changes the operand's value. In other words, the ***prefix*** operator changes the value before using it, while the ***postfix*** operator changes the value after using it.

```c
// Prefix and Postfix Operators
// pre_post.c
#include <stdio.h>

int main(void)
{
        int age = 19;

        printf("Prefix:  %d\n", ++age);
        printf("         %d\n", age);
        printf("Postfix: %d\n", age++);
        printf("         %d\n", ++age);

        return 0;
}
```

Floating-Point Operands

C has 4 binary and 2 unary shorthand assignment operators for floating-point (`float` and `double`) operands.

Binary Operands

The binary operators yield the same result as in the longhand expressions listed alongside:

Expression	Shorthand	Longhand	Meaning
operand `+=` operand	`x += 4.1`	`x = x + 4.1`	add 4.1 to x and assign to x
operand `-=` operand	`x -= 4.1`	`x = x - 4.1`	subtract 4.1 from x and assign to x
operand `*=` operand	`x *= 4.1`	`x = x * 4.1`	multiply x by 4.1 and assign to x
operand `/=` operand	`x /= 4.1`	`x = x / 4.1`	divide x by 4.1 and assign to x

Unary Operands

Expression	Shorthand	Longhand	Meaning
++operand	`++x`	`x = x + 1`	increment i by 1.0
operand++	`x++`	`x = x + 1`	increment i by 1.0
--operand	`--x`	`x = x - 1`	decrement i by 1.0
operand--	`x--`	`x = x - 1`	decrement i by 1.0

The prefix and postfix operators operate on floating-point operands in the same way as on integral operands.

Ambiguities

Compact use of shorthand operators can yield ambiguous results across different platforms. Consider the following longhand statements:

```
int i = 5;
int j = i++ + i; // *** AMBIGUOUS ***
```

One compiler may increment the first ***i*** before the addition, while another compiler may increment ***i*** after the addition. The C language does not address this ambiguity and only stipulates that the value must be incremented before the semi-colon. To avoid ambiguity, we re-write this code to make our intent explicit:

```
int i = 5;
i++;           // ++ before
int j = i + i; // j is 12
int i = 5;
int j = i + i; // j is 10
i++;           // ++ after
```

Casting

The C language supports conversions from one type to another. To convert the type of an operand, we precede the operand with the target type enclosed within parentheses. We call such an expression a cast. Casting expressions take one of the forms listed below:

Cast Expression	Meaning
(`long double`) operand	`long double` version of operand
(`double`) operand	`double` version of operand
(`float`) operand	`float` version of operand
(`long long`) operand	`long long` version of operand
(`long`) operand	`long` version of operand
(`int`) operand	`int` version of operand

Cast Expression	Meaning
(`short`) operand	`short` version of operand
(`char`) operand	`char` version of operand

Consider the example below. To obtain the number of hours in fractional form, we cast minutes to a `float` type and then divide it by 60. The input and output are listed on the right:

```
// From minutes to hours
// cast.c
#include <stdio.h>

int main(void)
{
        int minutes;
        float hours;

        printf("Minutes ? ");
        scanf("%d", &minutes);
        hours = (float)minutes / 60;
        printf("= %.2lf hours\n", hours);

        return 0;
}
```

Without the type cast, the output for the same input would have been 0.00 hours.

Mixed-Type Expressions

Since CPUs process integral expressions and floating-point expressions differently (using the ALU and the FPA respectively), they only handle expressions with operands of the same type. For expressions with operands of different types, we need rules for converting operands of one type to another type.

The C language using the following ranking:

`long double`	higher
`double`	...
`float`	...
`long long`	...
`long`	...
`int`	...
`short`	...
`char`	lower

There are two distinct kinds of expressions to consider with respect to type coercion:

- assignment expressions
- arithmetic and relational expressions

Assignment Expressions

Promotion

If the left operand in an assignment expression is of a higher type than the right operand, the compiler promotes the right operand to the type of the left operand. For the example below, the compiler promotes the right operand (loonies) to a `double` before completing the assignment:

```
// Promotion with Assignment Operators
// promotion.c
#include <stdio.h>

int main(void)
{
        int loonies;
        double cash;
```

```
        printf("Loonies ? ");
        scanf("%d", &loonies);
        cash = loonies; // promotion
        printf("Cash is $%.2lf\n", cash);

        return 0;
}
```

Narrowing

If the left operand in an assignment expression is of a lower type than the right operand, the compiler truncates the right operand to the type of the left operand. For the example below, the compiler truncates the type of the right operand (cash) to an `int`:

```
// Truncation with Assignment Operators
// truncation.c
#include <stdio.h>

int main(void)
{
        double cash;
        int loonies;

        printf("Cash ? ");
        scanf("%lf", &cash);
        loonies = cash; // truncation
        printf("%d loonies.\n", loonies);

        return 0;
}
```

Arithmetic and Relational Expressions

C compilers promote the operand of lower type in an arithmetic or relational expression to an operand of the higher type before evaluating the expression. The table below lists the type of the promoted operand.

Left Operand	Right Operand							
	long double	double	float	long long	long	int	short	char
long double	long double	long double	long double	long double	long double	long double	long double	long double
double	long double	double	double	double	double	double	double	double
float	long double	double	float	float	float	float	float	float
long long	long double	double	float	long long	long long	long long	long long	long long
long	long double	double	float	long long	long	long	long	long
int	long double	double	float	long long	long	int	int	int
short	long double	double	float	long long	long	int	short	short
char	long double	double	float	long long	long	int	short	char

Example

```
1034 * 10   evaluates to 10340    // an int result
1034 * 10.0 evaluates to 10340.0  // a double result
1034 * 10L  evaluates to 10340L   // a long result
1034 * 10.f evaluates to 10340.0f // a float result
```

Compound Expressions

A compound expression is an expression that contains an expression as one of its operands. C compilers evaluate compound expressions according to specific rules called rules of precedence. These rules define the order of evaluation of expressions based on the operators involved. C compilers evaluate the expression with the operator that has the highest precedence first.

The order of precedence, from highest to lowest, and the direction of evaluation are listed in the table below:

Operator	Evaluate From
`++ -- (postfix)`	left to right
`++ -- (prefix) + - & ! (all unary)`	right to left
`(type)`	right to left
`* / %`	left to right
`+ -`	left to right
`< <= > >=`	left to right

Operator	Evaluate From
`== !=`	left to right
`&&`	left to right
`||`	left to right
`= += -= *= /= %=`	right to left

To change the order of evaluation, we introduce parentheses. C compilers evaluate the expressions within parentheses (( )) before applying the rules of precedence. For example:

```
  2 + 3   * 5 evaluates to 2 + 15, which evaluates to 17
( 2 + 3 ) * 5 evaluates to 5 * 5, which  evaluates to 25
3 / (double)2 evaluates to 3 / 2.0, which evaluates to 1.5
(double)(3 / 2) evaluates to (double)1, which evaluates to 1.0
```

Information

Learning Outcomes

After reading this section, you will be able to:

- Define the units for storing information on a modern computer
- Introduce the memory model for programming a modern computer
- Introduce the addressing system for accessing the memory of a modern computer

Introduction

The information stored in a computer includes program instructions and program data. This information is stored in bits in RAM. The instructions and data take the form of groups of bits. The two most common systems for interpreting information stored in RAM are the binary and hexadecimal numbering systems.

This chapter defines these numbering systems and their units and describes the memory model for addressing different groups of bits stored in the part of RAM associated with a program.

Fundamental Units

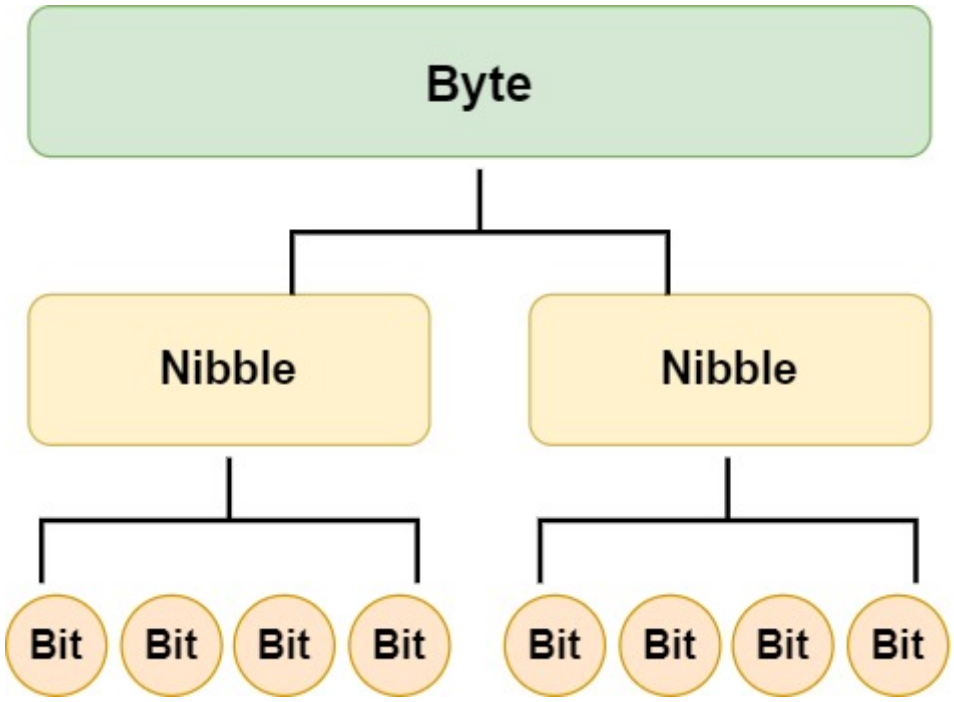
Bits

The most fundamental unit of a modern computer is the binary digit or bit. A bit is either on or off. One (1) represents on, while zero (0) represents off.

Since bits are too numerous to handle individually, modern computers transfer and handle information in larger units. As programmers, we define some of those units.

Bytes

The fundamental addressable unit of RAM is the byte. One byte consists of 2 nibbles. Each nibble consists of 4 bits.



One byte can store any one of 256 (2^8) possible values in the form of a bit string:

Bit Value	Decimal Value
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
...	...
00111000	56
...	...
11111111	255

The bit strings are on the left. The equivalent decimal values are on the right. Note that our counting system starts from 0, not from 1.

Words

We call the natural size of the execution environment a word. A word consists of an integral number of bytes and is typically the size of the CPU's general registers. Word size may vary from CPU to CPU. On a 16-bit CPU, a word consists of 2 bytes. On a Pentium 4 CPU, the general registers contain 32 bits and a word consists of 4 bytes. On an Itanium 2 CPU, the general registers contain 64 bits, but a word still consists of 4 bytes.

Hexadecimal

The decimal system is not the most convenient numbering system for organizing information. The hexadecimal system (base 16) is much more convenient.

Two hexadecimal digits holds the information stored in one byte. Each digit holds 4 bits of information. The digit symbols in the hexadecimal number system are {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. The characters A through F denote the values that correspond to the decimal values 10 through 15 respectively. We use the **0x** prefix to identify a number as hexadecimal (rather than decimal - base 10).

Bit Value	Hexadecimal Value	Decimal Value
00000000	0x00	0
00000001	0x01	1
00000010	0x02	2
00000011	0x03	3
00000100	0x04	4
...	...	
00111000	0x38	56

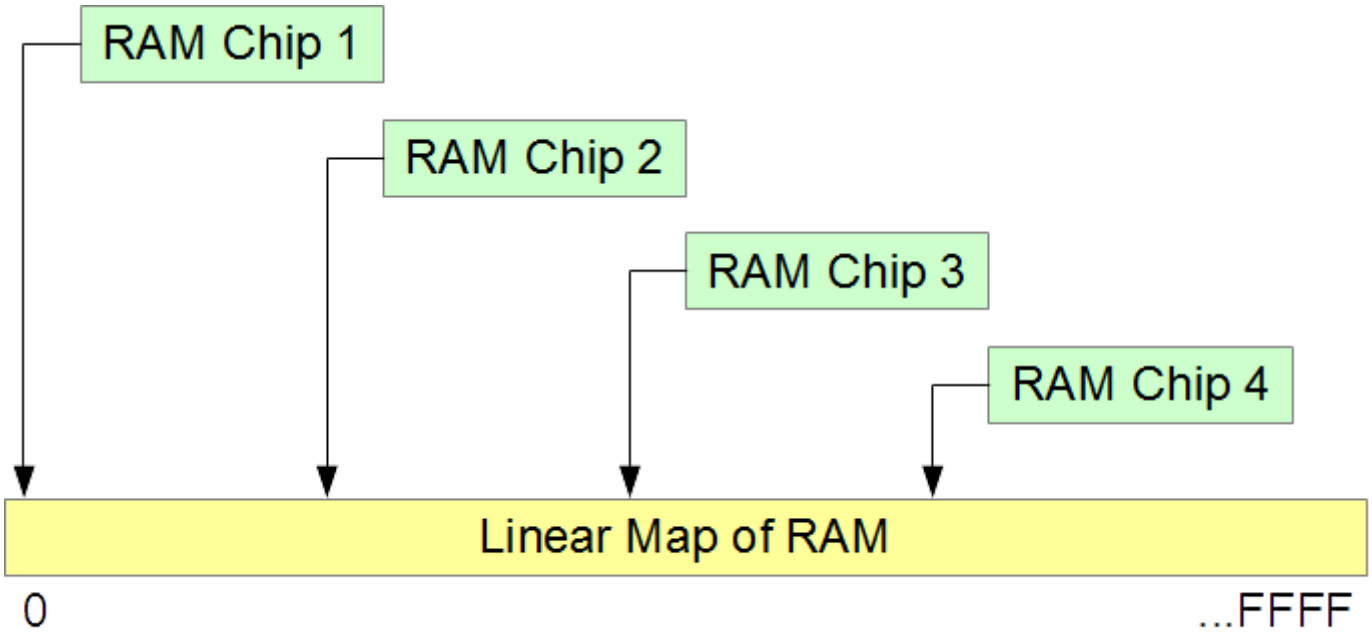
Bit Value	Hexadecimal Value	Decimal Value
...	...	
11111111	0xFF	255

For example, the hexadecimal value 0x5C is equivalent to the 8-bit value 01011100_2, which is equivalent to the decimal value 92.

To learn how to convert between hexadecimal and binary refer to the chapter entitled Data Conversions in the Appendices.

Memory Model

The memory model for organizing information stored in RAM is linear. Any byte in memory is accessible through a map that treats each actual physical memory location as a position in a continuous sequence of locations aligned next to one another.



Addresses

Each byte of RAM has a unique address. Addressing starts at zero, is sequential, and ends at the address equal to the size of RAM less 1 unit.

For example, 4 Gigabytes of RAM

- consists of 32 (= 4 * 8) Gigabits
- starts at a low address of `0x00000000`
- ends at a high address of `0xFFFFFFFF`

Size:	1 Byte		1 Byte		1 Byte		...	1 Byte	
Hex:	1 Nibble	1 Nibble	1 Nibble	1 Nibble	1 Nibble	1 Nibble	...	1 Nibble	1 Nibble
Value:							...		
Address:	0x00000000		0x00000001		0x00000002		...	0xFFFFFFFF	

> **NOTE**
>
> Each byte, and not each bit, has its own address. We say that RAM is byte-addressable.

Sets of Bytes

The abbreviations for sets of bytes are:

- Kilo or k (=1024): 1 Kilobyte = 1024 bytes ~ 10^3 bytes
- Mega or M (=1024k): 1 Megabyte = 1024 * 1024 bytes ~ 10^6 bytes
- Giga or G (=1024M): 1 Gigabyte = 1024 * 1024 * 1024 bytes ~ 10^9 bytes
- Tera or T (=1024G): 1 Terabyte = 1024 * 1024 * 1024 * 1024 bytes ~ 10^{12} bytes
- Peta or P (=1024T): 1 Petabyte = 1024 * 1024 * 1024 * 1024 * 1024 bytes ~ 10^{15} bytes
- Exa or E (=1024P): 1 Exabyte = 1024 * 1024 * 1024 * 1024 * 1024 * 1024 bytes ~ 10^{18} bytes

> **NOTE**
>
> The multiplying factor is 1024, not 1000. 1024 bytes is 2^{10} bytes, which is approximately 10^3 bytes.

Limit on Addressability (Optional)

The maximum size of the memory that the CPU can access depends on the size of its address registers. The highest accessible address is the largest address that an address register can hold:

- 32-bit address registers can address up to 4 GB (Gigabytes) (addresses can range from 0 to 2^{32}-1, that is 0 to 4,294,967,295).
- 36-bit address registers can address up to 64 GB (Gigabytes) (addresses can range from 0 to 2^{36}-1, that is 0 to 68,719,476,735).
- 64-bit address registers can address up to 16 EB (Exabytes) (addresses can range from 0 to 2^{64}-1, that is 0 to 18,446,744,073,709,551,615).

Segmentation Faults

The information stored in RAM consists of information that serves different purposes. We expect to read and write data, but not to execute it. We expect to execute program instructions but not to write them. So, certain architectures assign the data read and write permissions, while assigning instructions read and execute permissions. Such permission system helps trap errors while a program is executing. An attempt to execute data or to overwrite an instruction reports an error. Clearly, the access has been to the wrong segment. We call such errors a segmentation faults.

Logic

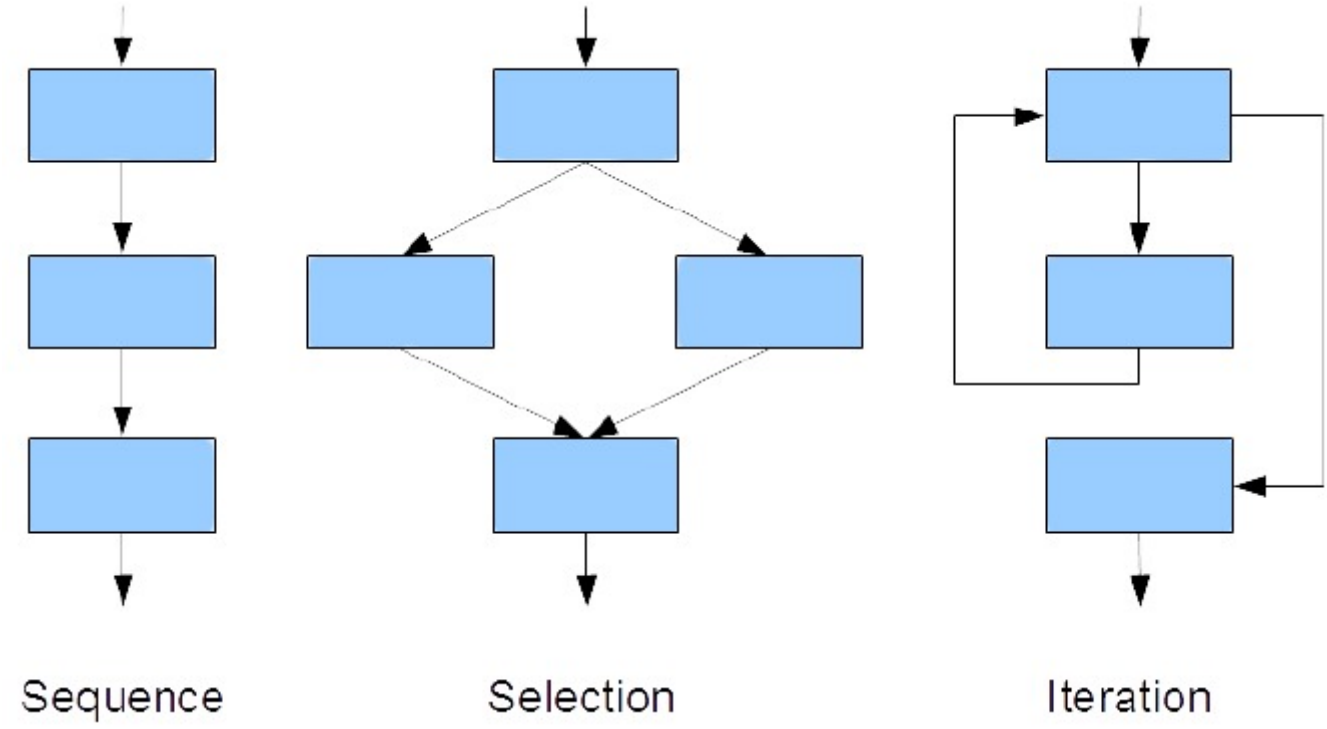
Learning Outcomes

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.

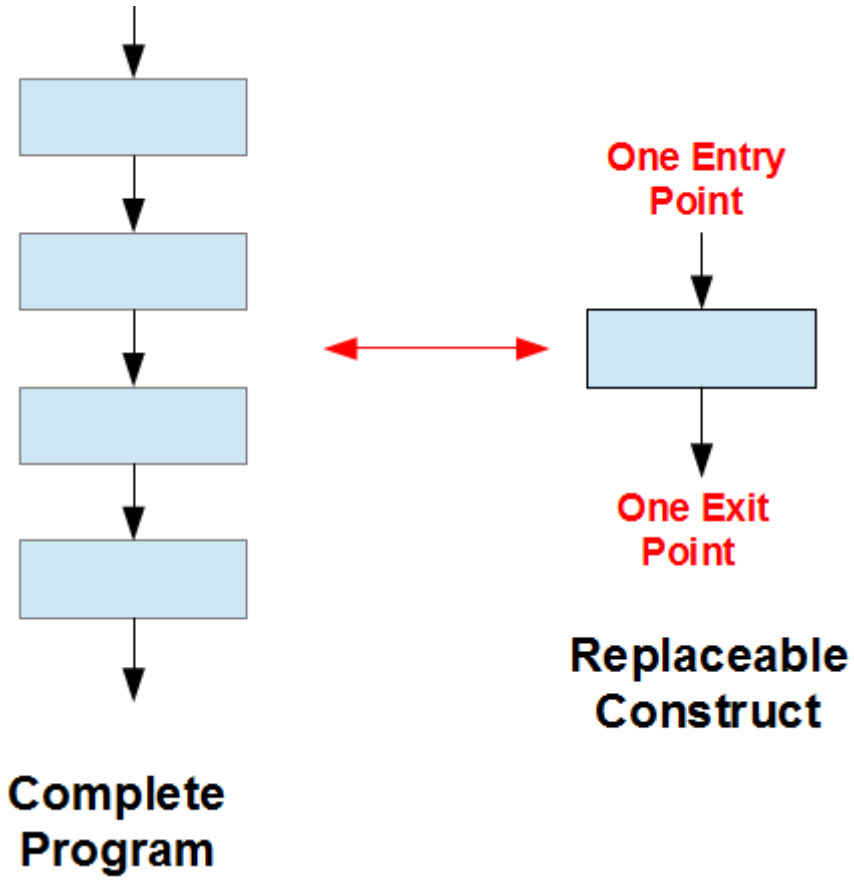


Since programmers who maintain application software are typically not those who develop that software originally and the maintenance programmers may change throughout the lifetime of a software application, it is critical that the software is not only readable but also easy to upgrade and maintain. The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

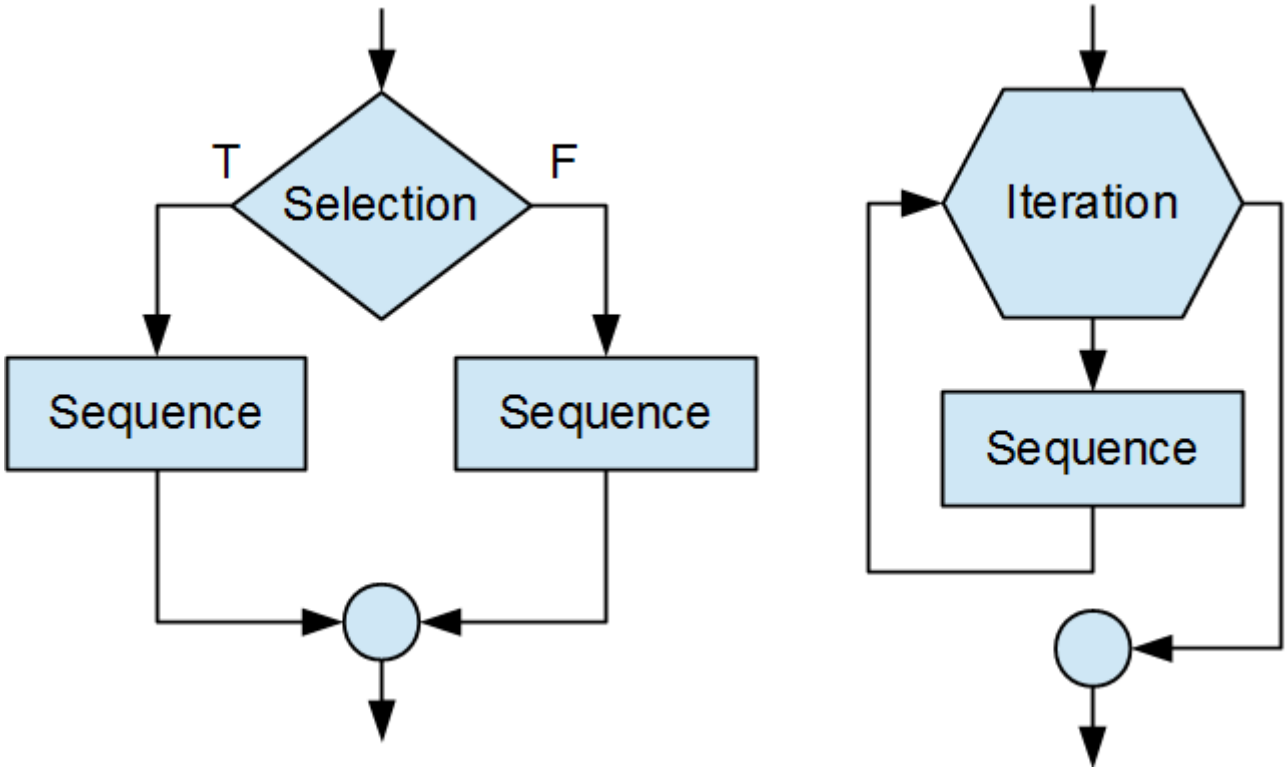
Pseudo-code is a set of shorthand notes in a human (non-programming) language that itemizes the key steps in the sequence of instructions that produce a programming solution. For example, the pseudo code for calculating the change in a vending machine might look something like

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:



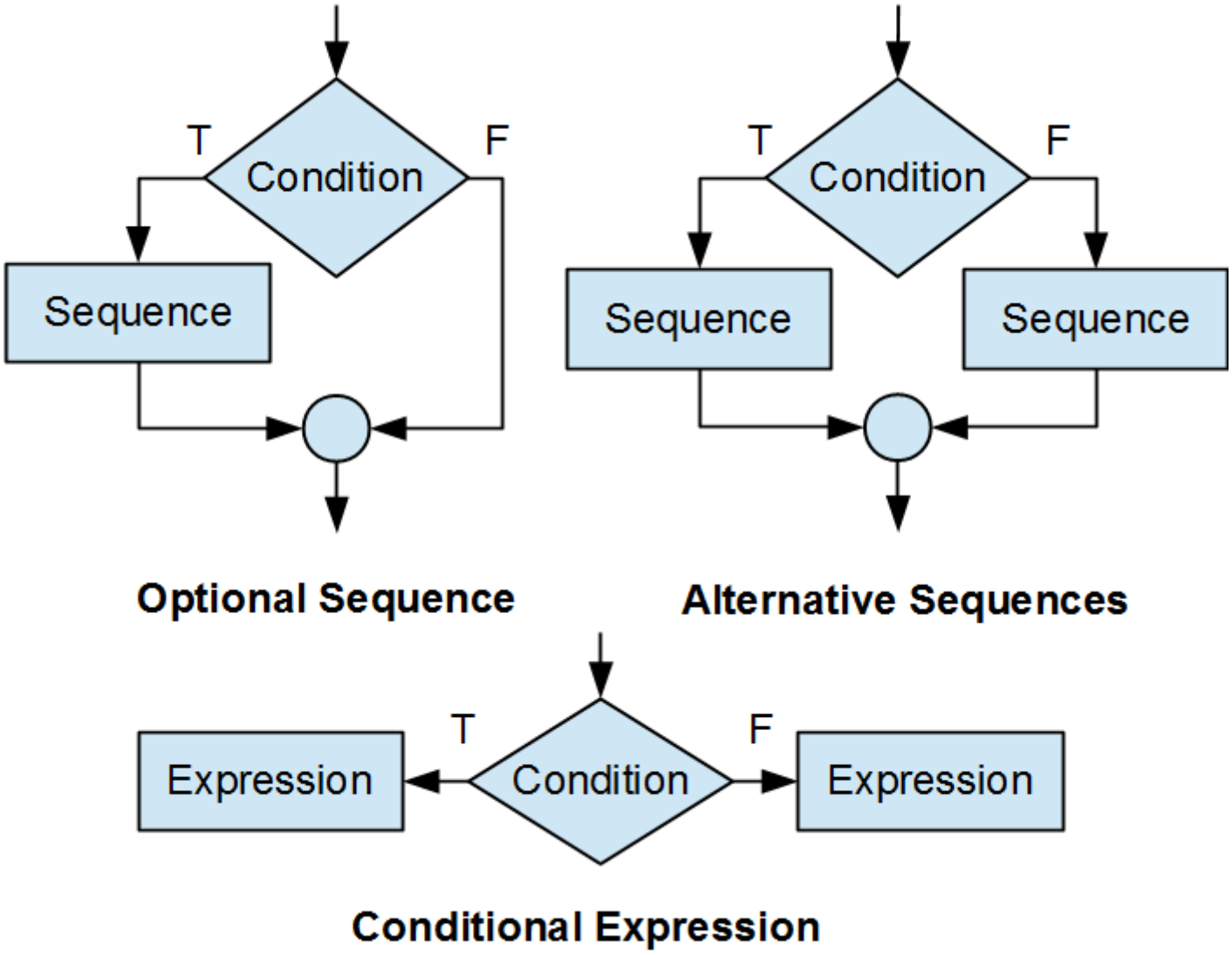
Usage of these sets with the C language is illustrated below.

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)
```