C – REVISION

3.1 Introduction

Before writing a program to solve a particular program, we must **have a thorough** understanding of the problem and a carefully planned approach solution approach.

3.2 Algorithm

An algorithm is defined as a procedure for solving a problem in terms of

1. the actions to be executed
2. the order in which these actions are to be executed

The solution to any computing problem involves executing a series of actions in a specific order.

Program control refers to specifying the order in which statements are to be executed in a computer program.

3.3 Pseudocode

Psuedocode is an artificial and informal language that helps you develop algorithms. They help to "think out" a program before attempting to write it in a programming language. Psuedocode only consists of action and decision statement.

3.4 Control Structures

All C programs are built in terms of only three control structures. These control structures include:

- **sequence structure**:
  - unless directed otherwise, the computer executes C statements one after the other in the order in which they're written

Flowcharts :
A flowchart is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are useful for developing and representing algorithms, and clearly show how control structures operate
*rectangle symbol/action symbol - indicates any type of action including a calculation or an input/output operation. The flow lines in the figure indicate the order in which
the actions are performed

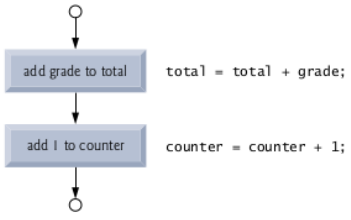


Fig. 3.1 | Flowcharting C's sequence structure.

Flowcharts for complete algorithms begin and end with rounded rectangles containing the words "begin" and "end" respectively.When drawing only a portion of an algorithm we omit the rounded rectangle symbols in favor of using small circle symbols, also called **connector symbol**

The most important flowcharting symbol is the diamond symbol, also called the decision symbol, which indicates that a decision is to be made.

- selection structure:

C provides three types of selection structures in the form of statements. These are

- The **if** selection statement either selects (performs) an action if a condition is true or skips the action if the condition is false.
- The **if … else** selection statement performs an action if a condition is true and performs a different action if the condition is false
- The **switch** selection statement performs one of many different actions, depending on the value of an expression.

- iteration structure:

C provides three types of iteration structures in the form of statements namely:

- **while**
- **do … while**
- **for**

C has only seven control statements: sequence, three types of selection and three types of iteration. C programs are formed by **combining as many** of each type of control statement as is appropriate for the algorithm the program implements.

Important:
There's only one other way control statements may be connected, this is done through a method called control-statement nesting. Thus, any C program we'll ever need to build can be constructed from only these seven different types of control statements combined in only two ways.

3.5 The if selection statement

Selection statements are used to choose among alternative courses of action. The if selection statement only checks one condition statement, if that condition is false, the action related to that condition won't be performed. I
n other words The if selection statement performs an indicated action only when the condition is true otherwise the action is skipped.

C code corresponds closely to the pseudocode (of course you'll also need to declare the necessary variables applicable to the condition) . This is one of the properties of pseudocode that makes it such a useful program-development tool.

e.g. if selection

If student's grade is greater than or equal to 60
Print "Passed"

C equivalent:

```
if ( grade >= 60 ) {
   puts( "Passed" );
} // end if
```

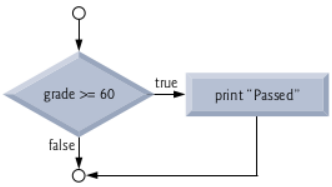


Fig. 3.2 | Flowcharting the single-selection if statement.

NOTE:

- indentation is important in any programming language. It helps emphasize the inherent structure of structured programs
- **diamond symbol –** represents decision symbol /decision to be made.
  - Decisions can be based on conditions containing relational or equality operations
  - if the expression evaluates to *zero* its treated as false, and if it evaluates to *nonzero* its treated as true

3.6 The if...else selection statement

The if…else selection statement allows you to specify that different actions are to be performed when the condition is true and when it's false.

e.g.

For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
else
    Print "Failed"
```

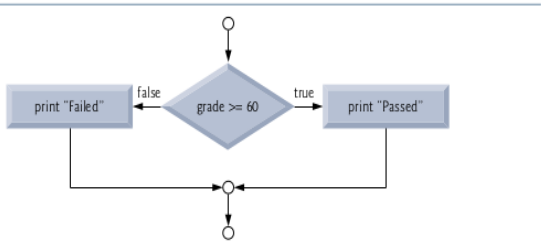


Fig. 3.3 | Flowcharting the double-selection `if...else` statement.

good programming practice:

- Indent both body statements of an if … else statement
- If there are several levels of indentation, each level should be indented the same additional amount of space.

The preceding pseudocode *If…else* statement may be written in C as

```
if ( grade >= 60 ) {
   puts( "Passed" );
} // end if
else {
   puts( "Failed" );
} // end else
```

Nested if...else statements

Nested if … else statements test for multiple cases by placing if…else statements inside if…else statements.

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

Or =>

This pseudocode may be written in C as

```
if ( grade >= 90 ) {
   puts( "A" );
} // end if
else {
   if ( grade >= 80 ) {
      puts( "B" );
   } // end if
   else {
      if ( grade >= 70 ) {
         puts( "C" );
      } // end if
      else {
         if ( grade >= 60 ) {
            puts( "D" );
         } // end if
         else {
            puts( "F" );
         } // end else
      } // end else
   } // end else
```

You may prefer to write the preceding `if` statement as

```
if ( grade >= 90 ) {
   puts( "A" );
} // end if
else if ( grade >= 80 ) {
   puts( "B" );
} // end else if
else if ( grade >= 70 ) {
   puts( "C" );
} // end else if
else if ( grade >= 60 ) {
   puts( "D" );
} // end else if
else {
   puts( "F" );
} // end else
```

- **A set of statements contained within a pair of braces is called a compound statement or a block.**

Important definitions:

- A **syntax error** is caught by the compiler
- A **logic error** has its effect at execution time
  - A fatal logic error causes a program to fail and terminate prematurely
  - A non-fatal logic error allows a program to continue executing but to produce incorrect results

3.7 The while iteration statement

An iteration statement (also called an repetition statement or loop) allows you to specify that an action is to be repeated **while some condition remains true.** Eventually, the condition will become false. At this point, the iteration terminates, and the first pseudocode statement after the iteration structure is executed.

e.g.

```
product = 3;
while ( product <= 100 ) {
   product = 3 * product;
}
```

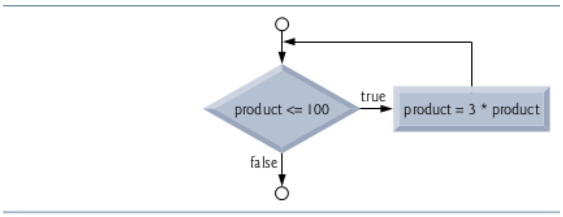


Fig. 3.4 | Flowcharting the `while` iteration statement.

Common programming error

-Not providing in the body of a while statement an action that eventually causes the condition in the while to become false. Normally, such an iteration structure will never terminate—an error called an "infinite loop."

3.8 Formulating algorithms case study 1: counter controlled iteration

We solve several variations of a class-averaging problem. (code written in c source files) [PROGRAM IN TB]

- *counter controlled iteration:*
  This technique uses a variable called a counter to specify the number of times a set of statements should execute. (**while and for loops are examples of this**)
  - The iteration terminates when the number of iterations exceed the the specified value in the count.

(end pg 114 the bottom)

3.9 Formulating Algorithms with top-down, stepwise Refinement
Case study 2: Sentinel controlled iteration

- *sentinel controlled iteration:*
  We define a **sentinel value** , some random value, such that if when we do not know how many iterations to have in our loop , once we input this sentinel value it'll result in the termination of the loop.

User will input values, once the user enters the sentinel value it will result in the termination of the condition
Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value.

Top-Down Stepwise Refinement - (READ THROUGH AGAIN TB)

This is a technique that's essential to the development of well-structured programs.
We define the :

- **Top:** The top is a single statement that conveys the program's overall function. As such, the top is, in effect, a complete representation of a program. Unfortunately, the top rarely conveys a sufficient amount of detail for writing the C program, thus we have to define the refinement
- **refinement steps**: divide the top into a series of smaller tasks and list these in the order in which they need to be performed.
  *- 1st refinement is usually the outlying steps you will follow do in your program, e.g. declaration of variable , possible operations to apply throughout your programs, and how these variable will be used (*an initialization phase that initializes the program variable)
  *- 2nd refinement is usually the initialization of the variables, checks conditions for which possible errors may occur etc. (*a processing phase that inputs data values and adjusts program variables accordingly)
  -etc... There could be many refinement steps
  *- **final refinement:*** a termination phase that calculates and prints the final results.

Good Programming practice and error prevenetion

- When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as by printing an error message) rather that allowing the fatal error to occur.
- In a sentinel-controlled loop, **explicitly remind the user** what the sentinel value is in prompts requesting data entry.

Converting between types explicitly and implicitly

Dividing two integers results in integer division in which any fractional
part of the calculation is truncated (i.e., lost). Because the calculation is performed first,
the fractional part is lost before the result is converted to a floating point value.

To produce a floating point calculation with integer values, we must create temporary values that are floating point numbers. C provides the unary cast operator to accomplish this task:

e.g. average = (float) total/counter ;

Using a cast operator in this manner is called **explicit conversion**. Cast operators are available for most data types they're formed by placing parentheses around a type name. Each cast operator is a unary operator, i.e., an operator that takes only one operand.

Formatting Floating-Point Numbers

floating point values can be expressed with a certain ***precision.***

- Precision = rounded to a certain number of digits.

When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions. The value in memory is unaltered.

e.g. %.2f

note:
- floating point numbers can not be compared for equality

(ended 3.10)

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements

{PROBLEM DISCUSSED AND SOLVED}

Software Engineering Observation 3.6
Many programmers write programs without ever using program-development tools such as pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs.

3.11 Assignment operators

C provides several assignment operators for abbreviating assignment expressions. The following table makes reference to a few of these abbreviated assignment expressions:

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 3.11 | Arithmetic assignment operators.

3.12 Increment and Decrement Operators

C defines the following **unary** operators:

- increment (**++**) : can be preincrement/postincrement
- decrement **(--)** : can be preincrement / post-increment

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 3.12 | Increment and decrement operators

Pre-incrementing (pre-decrementing) a variable causes the
variable to be incremented (decremented) by 1, then its new value is used in the expression in which it appears. Post-incrementing (post-decrementing) the variable causes the current value of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by 1.

Figure 3.14 lists the precedence and associativity of the operators introduced to this point. The operators are shown top to bottom in decreasing order of precedence.

Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - *(type)* ++ *(prefix)* -- *(prefix)*	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 3.14 | Precedence and associativity of the operators encountered so far in the text.

3.13 Secure Programming

Arithmetic Overflow:
Arithmetic overflow: value that's too large to store in an int variable.
It can cause undefined behavior, possibly leaving a system open to attack

The maximum and minimum integers that can be stored in a variable are represented by constants
- INT_MAX
- INT_MIN

They are defined in the header <limits.h>.
You can see your platform's values for these constants by opening the header <limits.h> in a text editor.

Important to check especially in industrial strength code that arithmetic calculation will not overflow or underflow. Logic operations are especially useful in this regard.

Unsigned Integers:
In general, counters that should store only non-negative values should be declared with unsigned before the integer type.

As you'll see, the end-of-file (EOF) indicator—which is introduced in the next chapter and is often used to terminate sentinel-controlled loops—is also a negative number.

Scanf_s and printf_s:
The more secure versions of **printf** and **scanf** called **printf_s** and **scanf_s.**
If given warnings that say that scanf is deprecated (it should no longer be used) ,you should consider using scanf_s instead. The same can be applied to printf aswell.

Chapter 4: C Program control

4.1 Introduction

In this chapter we discuss:

- iteration (continued)– for, do...while
- switch multiple selection
- break (exiting immediately from control statement) and continue statements (for skipping the remainder of the body of an iteration statement, then proceeding with the next iteration of the loop)
- logical operators

4.2 Iteration Essentials

Most programs involve iteration, or looping. A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.
We've defined:

1. Counter-controlled iteration- used when :
- we know in advance exactly how many times the loop will be executed
- counter variable usually incremented by 1 until the desired number of iterations are completed

2. Sentinel-controlled iteration - used when:
- The precise number of iterations isn't known in advance, and
- The loop includes statements that obtain data each time the loop is performed
- The sentinel value indicates "end of data." - Sentinels must be distinct from regular data items.

4.3 Counter-controlled Iteration

{REQUIREMENTS of counter controlled interation defined}

Notes: Good programming and Error prevention tips

1. *Control counting loops with integer values – Floating point variables may result in imprecise counter values and inaccurate termination tests.*

2. *Too many levels of nesting can make a program difficult to understand. As a rule, try to avoid using more than three levels of nesting.*
3. *Indentation and vertical spacing greatly improve pogram readability*

4.4 for Iteration Statement

The for iteration statement handles all the details of counter-controlled iteration. **The counter variable, loop continuation condition are defined and incremented in one line.**

***For statement header components*

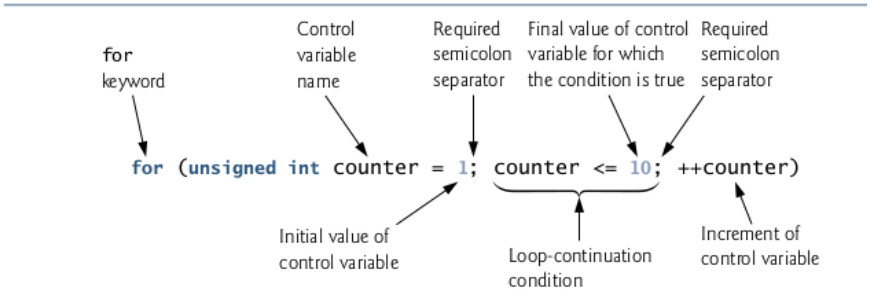


Fig. 4.3 | for statement header components.

***Control variables defined in a **for** HEADER Exist only until the loop terminates.*
Attempting to access the control variable after the for statement's closing right brace (}) is a compilation error.

***off-by-one Error*
Note: Error prevention
Using the final value in the condition of a while or for statement and using the <= relational operator can help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be counter <= 10 rather than counter < 11 or counter < 10 .

General format of a for statement

```
for (initialization; condition; increment) {
   statement
}
```

Comma seperated list of expressions

{COME BACK} – INTERESTING- THINK OF PROGRAMS TO APPLY THESE TO!!

Expressions in the for Statement's Header Are Optional

- If the condition expression is omitted, C assumes that the loop-continuation condition is true, thus creating an infinite loop.
- You may omit the initialization expression if the control variable is initialized before the for statement.
- The increment expression may be omitted if the increment is calculated by statements in the for statement's body or if no increment is needed.

Increment Expression Acts Like a Standalone Statement

{interesting}

4.5 for Statement: Notes and observations

{observations made regarding for iteration: some general knowledge}

4.6 Examples using the for statement

{Examples showing methods of varying the control variable in a for statement.}
{see TextBook}

- The header **<math.h>** (line 4) should be included whenever a math function is used.
- Formating numeric output:
  - **precision** = number of places decimal places. e.g. **%.2f = 2 decimal value float**

4.7 Switch multiple selection statement

multiple selection: an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken

C provides the switch multiple-selection statement to handle such decision making.
It consists of:

- case labels
- default case: If no match occurs for the different case labels, the default case is executed
- statements to execute for each case + break statement

Reading character input:
The **getchar()** function (from <stdio.h>) reads one character from the keyboard and the character can be stored in an integer or character variable. Characters are represented as small integer values.

Characters can be read with **scanf** by using the conversion specifier **%c.**

Note the EOF character, which is typically used in sentinel controlled iteration and contains a value of -1 normally.

- Linux EOF = *<ctrl>d*

4.8 do...while Iteration statement

The do...while is similar to the while Iteration statement. The **do…while** statement tests the loop-continuation condition after the loop body is performed. Therefore, the loop body will always execute at least once.

```
do {
    statements
} while (condition); // semicolon is required here
```

do...while statement flowchart

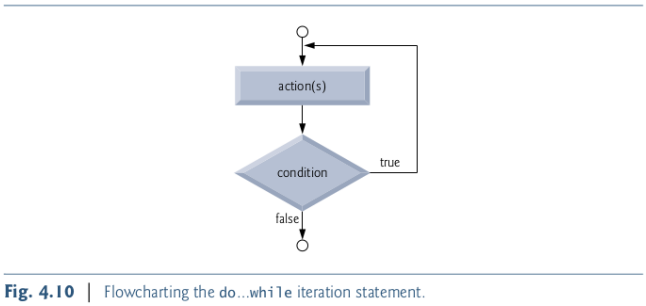


Fig. 4.10 | Flowcharting the do...while iteration statement.

4.9 Break and continue statement

Purpose: The break and continue statements are used to alter the flow of control, especially in iteration statements

break *statement*
The break statement, when executed in a while , for , do … while or switch statement, causes an immediate exit from that statement. Program execution continues with the next statement after that while , for , do … while or switch

continue *statement*
The continue statement, when executed in a while , for or do … while statement, skips the remaining statements in that control statement's body and performs the next iteration of the loop.

4.10 Logical Operators

Purpose: To test multiple simple conditions in the process of making a decision, instead of using if...else statements continually

logical operators may be used to form more complex conditions by combining simple conditions. The logical operators are **&&** (logical AND), **||** (logical OR) and **!** (logical NOT, also called logical negation).

*Logical AND (**&&**) operators*

Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution. In this case, we can use the logical operator && as follows:

```
if (gender == 1 && age >= 65) {
    ++seniorFemales;
}
```

C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1. Although C sets a true value to 1 it accepts any nonzero value as true.

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Fig. 4.13 | Truth table for the logical AND (&&) operator.

*Logical OR (**||**) operator*

Suppose we wish to ensure at some point in a program that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the || operator, as in the following program segment:

```
if (semesterAverage >= 90 || finalExam >= 90) {
    puts("Student grade is A");
}
```

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Fig. 4.14 | Truth table for the logical OR (||) operator.

*Logical Negation **(!)** operator*

The logical negation operator has only a single condition as an operand (and is therefore a unary operator). The logical negation operator is placed before a condition when we're interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if (!(grade == sentinelValue)) {
    printf("The next grade is %f\n", grade);
}
```

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

Summary of Operator Precedence and Associativity

Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - ! ++ *(prefix)* -- *(prefix)* (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 4.16 | Operator precedence and associativity.

The ***_Bool*** *data type*

The C standard includes a boolean type—represented by the keyword _Bool —which can hold only the values 0 or 1.

Assigning any nonzero value to a **_Bool** sets it to 1. The standard also includes the **<stdbool.h>** header, which defines **bool** as a shorthand for the type **_Bool** , and **true** and **false** as named representations of **1** and **0 ,** respectively.

In other words, when declaring a **bool** variable, we can assign the values/statements **true** or **false** , which is equivalent to the values **1** and **0** respectively to the declared bool variable.

{ Read the remaining information for additional information}

Chapter 5: C functions

5.1 Introduction

Chapter objectives: This chapter describes some key features of the C language that facilitate the design, implementation, operation and maintenance of large programs. The best way to develop and maintain a large program is to construct it from smaller pieces, each of which is more manageable than the original program, also known as the divide and conquer approah.

5.2 Modularizing Programs in C

Functions are used to modularize programs. Programs are typically written by combining new functions you write with pre-packaged functions available in the C standard library.

Functions are invoked by a function call, which specifies the function name and provides information (as arguments) that the function needs to perform its designated task.

We define caller and called functions.!

5.3 Math Library Functions

NOTE: Function arguments may be constants, variables, or expressions. It is also common use to make use of functions specified in the C standard library instead of creating the same functions that have been defined in them.

Examples of commonly used math functions include:

Function	Description	Example
`sqrt(x)`	square root of x	`sqrt(900.0)` is 30.0 `sqrt(9.0)` is 3.0
`cbrt(x)`	cube root of x (C99 and C11 only)	`cbrt(27.0)` is 3.0 `cbrt(-8.0)` is -2.0
`exp(x)`	exponential function e^x	`exp(1.0)` is 2.718282 `exp(2.0)` is 7.389056
`log(x)`	natural logarithm of x (base e)	`log(2.718282)` is 1.0 `log(7.389056)` is 2.0
`log10(x)`	logarithm of x (base 10)	`log10(1.0)` is 0.0 `log10(10.0)` is 1.0 `log10(100.0)` is 2.0
`fabs(x)`	absolute value of x as a floating-point number	`fabs(13.5)` is 13.5 `fabs(0.0)` is 0.0 `fabs(-13.5)` is 13.5
`ceil(x)`	rounds x to the smallest integer not less than x	`ceil(9.2)` is 10.0 `ceil(-9.8)` is -9.0
`floor(x)`	rounds x to the largest integer not greater than x	`floor(9.2)` is 9.0 `floor(-9.8)` is -10.0
`pow(x, y)`	x raised to power y (x^y)	`pow(2, 7)` is 128.0 `pow(9, .5)` is 3.0

Fig. 5.2 | Commonly used math library functions. (Part 1 of 2.)

Function	Description	Example
`fmod(x, y)`	remainder of x/y as a floating-point number	`fmod(13.657, 2.333)` is 1.992
`sin(x)`	trigonometric sine of x (x in radians)	`sin(0.0)` is 0.0
`cos(x)`	trigonometric cosine of x (x in radians)	`cos(0.0)` is 1.0
`tan(x)`	trigonometric tangent of x (x in radians)	`tan(0.0)` is 0.0

Fig. 5.2 | Commonly used math library functions. (Part 2 of 2.)

5.4 Functions

Functions allow you to modularize a program. All variables defined in function definitions are local variables—they can be accessed only in the function in which they're defined. Most functions have a list of parameters that provide the means for communicating information between functions via arguments in function calls. A function's parameters are also local variables of that function.

Motivations for functionalizing a Program includes:

1. *The divide-and-conquer approach makes program development more manageable*
2. *Another motivation is software reusability*
3. *Third motivation is to avoid repeating code in a program. Packaging code as a function allows it to be executed from other locations in a program simply by calling the function*

IMPORTANT!:

- Each function should be limited to performing a single, well-defined task, and the function name should express that task. This facilitates abstraction and promotes software reusability.
- If you cannot choose a concise name that expresses what the function does, it's possible that your function is attempting to perform too many diverse tasks. It's usually best to break such a function into smaller functions—this is sometimes called decomposition.

5.5 Function definitions

{ This section gives a few examples on how customised functions can be written **}**

5.5.1 Square function

{ see Textbook **}**

Format of a function definition:
The format of a function definition is as follows:

```
return-value-type function-name(parameter-list)
{
    statements
}
```

The following are defined in the definition:

- function-name : is any valid identifier
- **return-value-type**: is the data type of the result returned to the caller
  NOTE: The return-value-type void indicates that a function does not return a value.
- Together, the return-value-type, function-name and parameter-list are sometimes referred to as the function **header**
- ***parameter-list:*** is a comma-separated list that specifies the parameters received by the function when it's called. If a function does not receive any values, parameter-list is void . A type must be listed explicitly for each parameter.

Function Body
The statements within braces form the function body, which is also a block. Variables can be declared in any block, and blocks can be nested (but functions connot be nested).

Returning control from a function

three ways to return control from a called function to the point at which a function was invoke
If the function does not return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement
return;
If the function does return a result, the statement
return *expression* ;
returns the value of expression to the caller.

5.5.2 maximum function

{ see textbook}

5.6 Function Prototypes: A deeper look

Purpose: Function prototypes are important in that the compiler uses function prototypes to validate function calls.

Compilation errors:
A function call that does not match the function prototype is a compilation error. An error is also generated if the functiSton prototype and the function definition/header disagree.

Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.

A function prototype placed outside any function definition applies to all calls to the function appearing after the function prototype in the file. A function prototype placed in a function body applies only to calls made in that function.***[interesting]***

5.7 Function Call Stack and Frames [Re - read]

{ Important: **STACK** discussed as well as important definitions – important as coder **}**

5.8 Headers

Header: contains the function prototypes, and definitions of various data types and constants needed by those functions.

[*Examples of some C standard Libraries given. i.e. Fig 5.10 – each containing function prototypes pertaining to a specific purpose*]

You can create custom headers. Programmer-defined headers should also use the .**h filename extension**. A programmer-defined header can be included by using the **#include** preprocessor directive.

Programmer-defined headers are enclosed in quotes ("") rather than angle brackets (<>).

5.9 Parsing arguments by value or by reference

There are two ways to pass arguments—*pass-by-value* and *pass-by-reference*. When arguments are passed by value, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect an original variable's value in the caller. When an argument is passed by reference, the caller allows the called function to modify the original variable's value.

It's possible to achieve pass-by-reference by using the **address operator** and the i**ndirection operator.**

5.10 Random Number generator

Purpose:We now take a brief and, hopefully, entertaining diversion into simulation and game playing. In this and the next section, we'll develop a nicely structured game-playing program that includes multiple functions discussed previously.

C standard library function **rand** is from the **<stdlib.h>** header. The rand function generates an integer between 0 and RAND_MAX.

Used as e.g. **i = rand() ; =** *returns a value between 0 -randmax*

- **rand % x (**a.k.a scaling**):** *returns value between 0 to x-1*
- ***i** (shift)***+ rand % x:** *value returned is between i + x*

Randomizing the random number generator:

Note: Function rand actually generates **pseudorandom numbers ,** exactly the same sequence of values is printed each time the program is executed.

A program can be conditioned to produce a different sequence of random numbers for each execution. This is called **randomizing** and is accomplished with the standard library function **srand .**

- **Function srand takes an unsigned int argument and seeds function rand to produce a different sequence of random numbers for each execution of the program.**

  - i.e ***srand(seed)***

  - Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied

- To randomize without entering a seed each time, use a statement like:

  srand(time(NULL)); - Note: The function prototype for time is in **<*time.h*>** , be sure to include the header

Generalised scaling and shifting of random numbers
The values produced directly by rand are always in the range:

0 ≤ rand() ≤ RAND_MAX

We can generalize any range to produce a random number by defining the:

- *width* of the range: determined by the number used to scale **rand()** with the (%) remainder operator
- *scaling factor:* determined by the starting number of the range

This can be generalized as follows:

n = a + rand() % b

- **a** = shifting value
- **b =** scaling factor/width of range = a +b
- note! : **(a +b-1)** = final value of range

5.11 Example: A game of chance: Introducing Enum

{ game of scraps simulated- (understand the code) **}**

Enumerations
enumeration: creates a programmer-defined type
An enumeration, introduced by the keyword enum , is a set of integer constants represented by identifiers. Enumeration constants help make programs more readable and easier to maintain. Values in an enum start with 0 and are incremented by 1. e.g. type "status" created

```
enum Status { CONTINUE, WON, LOST };
```

Q: Do you see how this can be used in programming of games? Not needed but efficient

5.12 Storage Classes [Read Through Theory: IMPORTANT - understanding not thoroughly there]

Note: identifiers are used for variable names. The attributes of variables include name, type, size and value.

C provides the storage class specifiers ***auto , register , extern*** and ***static.***

An identifier's scope is where the identifier can be referenced in a program i.e. where the variable can be used in a program.

Local Variables
Local variables have automatic storage duration by default, so keyword auto is rarely used.

Static storage class
Keywords extern and static are used in the declarations of identifiers for variables and functions of static storage duration.
Ing n

5.13 Scope rules [Read through theory: important]

The ***scope of an identifier*** is the portion of the program in which the identifier can be referenced.

The four identifier scopes are:

- function scope
- file scope
- block scope: Identifiers defined inside a block have block scope. Block scope ends at the terminating right brace (**}**) of the block.
- function-prototype scope.

{{ scoping1.c }}

5.14 Recursion [Important: understanding not here!]

5.15 Example using recursion: Fibonacci series

{ Theory discusses with regards to the recursive problem, i.e. Fibonacci series, as well as problems that may arise with regards to recursive problems **}**

5.16 Recursion vs Iteration [Theory: Important extra information]

Purpose: we compare the two approaches and discuss why you might choose one approach over the other in a particular situation

5.17 Secure C Programming (CHAPTER: SUMMARY GIVEN: USEFUL)

Chapter 6: C Arrays

6.1 Introduction

This chapter introduces data structures. **Arrays** are data structures consisting of related data items of the same type.

6.2 Arrays

Note: variable/Array name = identifier

An array is a group of contiguous memory locations that all have the same type. To refer to a particular location or element in the array, we specify the array's name and the position number of the particular element in the array.

The position number in square brackets is called the element's **index** or **subscript.**

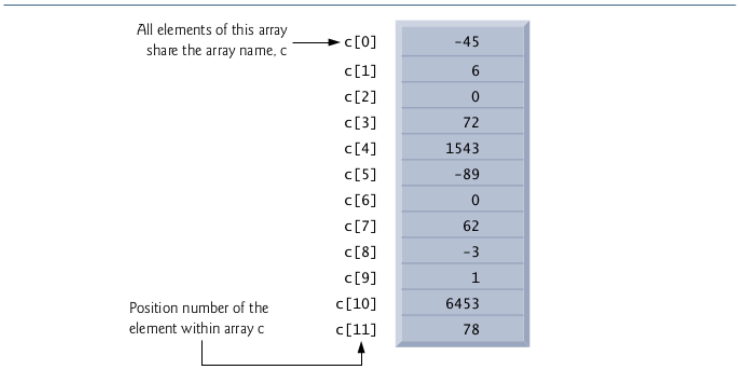


Fig. 6.1 | 12-element array.

Operators	Associativity	Type
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 6.2 | Operator precedence and associativity. (Part 2 of 2.)

6.3 Defining arrays

Arrays occupy space in memory.You specify the type of each element and the number of elements each array requires so that the computer may reserve the appropriate amount of memory.

e.g. *defining arrays* :

```
int b[100], x[27];
```

Arrays may contain other data type e.g. ***type char, double, float, etc...***

Note: Although we can define multiple arrays in 1 line, its best to define them in separately to indicate the purpose of each array.

6.4 Array Examples

Purpose: This section presents several examples that demonstrate how to define and initialize arrays, and how to perform many common array manipulations.

6.4.1 Defining an array and using a loop to initialize array element entries

Note: Like any other variables,uninitialized array elements contain garbage values.

Note: type ***size_t*** , according to the C standard represents an unsigned integral type

This type is recommended for any variable that represents an array's size or indices.

{ example done – InitializeArray.c **}**

6.4.2 Initializing an Array in a Definition with an Initializer List

The elements of an array can also be initialized when the array is defined by following the definition with *an equals sign and braces, {}* , containing a comma-separated list of array initializers.

Note: By not initializing all the elements of an array using an initializer list, the remaining uninitialized elements will be set to 0.

6.4.3 Specifying an array size with a symbolic constant and intializing array elements with calculations

The **#define preprocessor directive** is introduced in this program.

```
#define SIZE 5
```

Using symbolic constants to specify array sizes makes programs more modifiable/scalable. As programs get larger, this technique becomes more useful for writing clear, easy to read, maintainable programs.

Note: symbolic/preprocessor directive constants are not variables. Only use CAPITAL LETTERS to declare these symbolic constants.

6.4.4 Summing the elements of an array

{ Example done **}**

6.4.5 Using arrays to summarise survey results

{ Example done: Analyzing a studentpoll – Interesting implementation **}**

Note: *When manipulating arrays, be sure to inlude error checking procedures.Programs should validate the correctness of all input values to prevent erroneous information from affecting a program's calculations.*

Ended with array survery code – confirm if correct on, slides and textbook

6.4.6 Graphing Array Element values with Histograms

{ Example:done reads numbers from an array and graphs the information in the form of bar chart or histogram **}**

6.4.7 Rolling a die 60,000,000 times and summarizing the results in an array

6.5 Using Character Arrays to store and Manipulate Strings

In this chapter, we show that integers are not the only datatype that can be stored in Arrays. We look at storing strings in character Arrays . Thus far , printf , is the only pre-processing directive that allows us to print strings

6.5.1 Initializing a character Array with a String

character Array can be intilised with a string literal. The initial string contains , the initial specified characters and a *string terminating character called the null character*.

```
char string1[] = "first";
```

The escape sequence representing the null character is '\0'.

6.5.2 Initializing a character array with an initializer list of characters

Character arrays also can be initialized with individual character constants in an initializer list:

```
char string1[] = {'f', 'i', 'r', 's', 't', '\0'};
```

6.5.3 Accessing the character in a string

Because a string is really an array of characters, we can access individual characters in a string directly using array index notation.

6.5.4 Inputting into a character Array

We also can input a string directly into a character array from the keyboard using **scanf** and the conversion specifier **%s**.

For example:

```
char string2[20];
```

The statement:

```
scanf("%19s", string2);
```

reads a string from the keyboard into string2. **Function scanf() will read characters until a space, tab, newline or end-of-file indicator is encountered.**
We used the conversion specifier %19s so that scanf reads a maximum of 19 characters and does not write characters into memory beyond the end of the array string2.

6.5.5 Outputting a character array that represents a String

Strings stored in character arrays can be printed/displayed using the **%s conversion specifier** and **printf() function** .

```
printf("%s\n", string2);
```

6.5.6 Demonstrating Character Arrays

To print the individual characters of a string in a character array , the conversion specifier **%c** and **for loop** should be used.

Example: operations with character arrays, character/string input and ouput

```
1  // Fig. 6.10: fig06_10.c
2  // Treating character arrays as strings.
3  #include <stdio.h>
4  #define SIZE 20
5
6  // function main begins program execution
7  int main(void)
8  {
9     char string1[SIZE]; // reserves 20 characters
10    char string2[] = "string literal"; // reserves 15 characters
11
12    // read string from user into array string1
13    printf("%s", "Enter a string (no longer than 19 characters): ");
14    scanf("%19s", string1); // input no more than 19 characters
15
16    // output strings
17    printf("string1 is: %s\nstring2 is: %s\n"
18           "string1 with spaces between characters is:\n",
19           string1, string2);
```

Fig. 6.10 | Treating character arrays as strings. (Part 1 of 2.)

```
20
21    // output characters until null character is reached
22    for (size_t i = 0; i < SIZE && string1[i] != '\0'; ++i) {
23       printf("%c ", string1[i]);
24    }
25
26    puts("");
27 }
```

```
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Fig. 6.10 | Treating character arrays as strings. (Part 2 of 2.)

6.7 Passing Arrays to functions

To pass an array argument to a function, specify the array's name without any brackets.

For example, if array `hourlyTemperatures` has been defined as

```
int hourlyTemperatures[HOURS_IN_A_DAY];
```

the function call

```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

Recall that all arguments in C are passed by value. C automatically passes arrays to functions by reference. **The array's name evaluates to the address of the array's first element**. Because the starting address of
the array is passed, the called function knows precisely where the array is stored. Therefore, when the called function modifies array elements in its function body, it's modifying the actual elements of the array in their original memory locations.

The %p conversion specifier normally outputs addresses as hexadecimal numbers, but this is compiler dependent.

For a function to receive an array through a function call, the function's parameter list must specify that an array will be received. For example:

```
void modifyArray(int b[], size_t size)
```

Difference between passing an entire array and passing an array element

Basically, when passing an entire array , a function similar to the above is declared where the parameter list includes an array.
Passing an array element,is done by accessing each array element individually using indexing, and modifying the value on its own and not in the array.

```
void modifyArray(int b[], size_t size)
{
   // multiply each array element by 2
   for (size_t j = 0; j < size; ++j) {
      b[j] *= 2; // actually modifies original array
   }
}

// in function modifyElement, "e" is a local copy of array el
// a[3] passed from main
void modifyElement(int e)
{
   // multiply parameter by 2
   printf("Value in modifyElement is %d\n", e *= 2);
}
```

using the constant qualifier with Array parameters

To ensure the contents of an array are not modified , either by a function or other means, the array should be preceded by the qualifier **const** . Any attempts to modify the array elements will result in a compilation error.
This specifies that the array is constant and cannot be modified.

6.10 Searching Arrays

This section covers the linear search and Binary search technique. There may instances where we need to search for a specific **key** in an arrray, or whether it contains that value.

6.10.1 Searching an array with linear search

Know program code: LinearSearchArray.c
(linear search algorithm)

Traverse through the Array one element at a time and return the index , at which the key was found.

6.10.2 Search an Array with Binary search (come back)

know/understand program code: BinarySearchArray.c
(binary search function/algorithm)

For large arrays linear searching is inefficient. If the array is sorted, the high-speed binary search tech-nique can be used. . There is a tremendous increase in performance compared to a linear search of a sorted array, which requires comparing the search key to an average of half of the array elements.

Code tips: low = Lowest index in array ; high = highest index in array

L16:

6.8 Sorting Arrays

***Bubble sort**

This works by making n-1 passess through an array of size n, and sorting elements by swapping them arround.

The technique is to make several passes through the array. On each pass, successive pairs of elements (element 0 and element 1, then element 1 and element 2, etc.) are compared. If a pair is in increasing order (or if the values are identical), we leave the values as they are. If a pair is in decreasing order, their values are swapped in the array.

Fig 6.15: Sorting an arrays values in ascending order

The sorting is performed by the nested for loops (lines 21–34). If a swap is necessary, it's performed by the three assignments.

```
hold = a[i];
a[i] = a[i + 1];
a[i + 1] = hold;
```

6.9 Case Study: Computing Mean, Median, Mode using Arrays

Computers are commonly used for survey data analysis to compile and analyze the results of surveys and opinion polls. This example includes most of the common manipulations usually required in array problems, including passing arrays to functions. The program computes the mean, median and mode of the 99 values.

***Fig. 6.16** | Survey data analysis with arrays: computing the mean, median and mode of the data*
Example Code

6.11 Multi-dimensional Arrays

The main take away is that multi-dimensional arrays are a table representation of information unlike normal 1 dimensional arrays.
Arrays in C can have multiple indices. A common use of multidimensional arrays, which the C standard refers to as multidimensional arrays, is to represent tables of values consisting of information arranged in rows and columns. To identify a particular table element, we must specify two indices:

6.11.1 Illustrating double scripted array

The indices a[i][j] , uniquely identify each element in an array.

6.11.2 Initialiszing a double scripted array

A multidimensional array can be initialized when it's defined. For example, a two-dimensional array could be defined and initialized with:

```
int b[2][2] = {{1, 2}, {3, 4}};
```

If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0.

Fig. 6.21 | Initializing multidimensional arrays. (Part 1 of 2.)

Example

6.11.3 Setting the elements in one row

Basically illustrates how we can set each element in a specific row using a *for* iteration statement.

6.11.4 Totalling the Elements in a Two Dimensional Array

Basically illustrates the format of totalling the elements in a 2D array. Where the the number of rows and columns depend on the size of the 2D array.

```
total = 0;
for (row = 0; row <= 2; ++row) {
   for (column = 0; column <= 3; ++column) {
      total += a[row][column];
   }
}
```

6.11.5 Two dimensional Array Manipulations

Figure 6.22 performs several other common array manipulations on a 3-by-4 array studentGrades using for statements. Each row of the array represents a student and each column represents a grade on one of the four exams the students took during the semester.

***Fig. 6.22** | Two-dimensional array manipulations. (Part 1 of 3.)*

Example

DEBUGGING

It is a systematic process of identifying and fixing errors (bugs) in a computer program.
Errors may lead to wrong results or code that does not compile
Debug Methods:

- using Printf() statement:
  - interrupt program at specific exit(-1)
  - Print out the data in a particular variable to see if it is as expected
- use comment blocks
- using debugger

Remainder half: focus on application not theory: Note taking set aside for now

7.1 Introduction

Pointer variable stores the memory location of a value/variable. Used to simulate call by reference

7.2 Pointer variable definition and initialization

Pointer declaration:

- * used to declare pointer variables

```
int *myPtr;
```

- Initialise pointers to 0, NULL,or an address

```
int *myPtr1 = NULL;
int *myPtr2;
myPtr2 = 0x22FF7C;
```

7.3 Pointer Operators

& (address operator)

- Returns address of operand

***** indirection/dereferencing operaror

- * not limited to pointer declaration, also used as operand
- * provides access to memory/variable that its operand points to
- Access can be to read the value or to change the value

```
int x = 5; // assign
int *xPtr; // declare
xPtr = &x; // assign
printf("%d", *xPtr);
*xPtr = 7; // indirec
```

7.4 Passing arguments to functions by reference

Call by reference means the use of the pointers refers to the use of pointers indirectly.

- Call by reference with pointer arguments:
  - Pass address of argument using address operator (&)
  - Allows you to change the value at the specific location in memory
  - Arrays are not passed with &, because the array name is already an adress/reference (pointer)
- ***** (operator)

```
void Double( int *myPointer )
 {
    *myPointer = 2 * ( *myPointer );
 }
```

Example: Cube by refennce

Example: Swap numbers

7.5 Using the const qualifier with pointers

- const qualifier
- const qualifier with pointers
  -
  -
  -
  -

7.6 Bubble sort using Pass-by-reference

7.7 Sizeof Operator

returns the size of opearands in bytes. **e.g.** for arrays size of 1 elements × number of elements in array.

- if `sizeof( int )` equals 4 bytes, then

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

will print 40

- Sizeof can be used with:
  - *variable names*
  - *type names*
  - *constant values*

Example

7.8 Pointer arithmetic and pointer expressions

7.9 Relationship between Pointers and Arrays

12.3 Dynamic Memory Allocations

- malloc()
  - contained in stdlib.h
  - allocates memory during execution time, for an array of int (or whichever required datatype) of size *numberofElements*

```
newPtr = malloc( numberOfElements * sizeof( int ) );
```

- Free()
  - frees memory allocated previously
  - ***always free dynamically allocated memory to prevent memory leaks***

Example

By using dynamic memory allocations pointers can be used exactly in the same way as arrays and it no long will need to only point to the memory location of the a variable.

7.10 Arrays of Pointers

- Array of strings
  - e.g

```
const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

7.11 Case Study: Card shuffling and Dealing Simulation

Problem statement

Design and implement an algorithm that shuffles and deals a 52-card deck

C structures

10.1 Structures

```
struct coord {
   double x;
   double y;
};
```

Collection of a variety of random variables under one name.

Structures can contain variables of the same type or different types.Commonly used to define records that can be stored in files. Structures can be combined with pointers to create linked lists, stacks and qeueus and trees.

Example III

```
struct node {
   struct node *leftPtr;
   struct node *rightPtr;
};
```

10.2 Structure Definitions

Structures are essentially user defined datatypes.

10.3 Iniatializing structures

Example 1
Example 2

10.4 Accessing members of structures

- **.**(*dot operator*)
  - used to access strcture members using the name
- -> (arrow operator)
  - used to access members of a structure using a pointer to the structure

```
        c.x = 10.5;                 c.y = 12.3;
(*cPtr).x = 10.5;           (*cPtr).y = 12.3;
  cPtr->x = 10.5;             cPtr->y = 12.3;
```

10.5 Using Structures with functions

Example

10.6 Typedef

Creates synonyms (aliases) for previouly defined datatypes. *Typedef* can be used to create shorter type names.

```
/*Option 1*/
    struct coord {
        double x;
        double y;
    };

    typedef struct coord Point;
  Point pa, pb;
```

```
/*Option 2*/
    typedef struct {
        double x;
        double y;
    } Point;

    Point pa, pb;
```

- **Defines a new type name Point as synonym for type struct coord**

Example

- **This is an important property to make use of reglarly and not only for structs.**

Chapter 10
Array of strcuctures

Example

Malloc and structures

Similar to regular datatypes where malloc was used to allocate memory at run time for pointers to those datatype, *malloc* can also be used to allocate memery to pointers that are of the specifc *structs*

10.7 Example: Card Shuffling and Dealing

Problem statement

Design and implement an algorithm that shuffles and deals a 52-card deck

Interesting Problem make sure to fully undertand

Chapter 11:File Processing

11.1 Introduction

- Data Files:

Data files are used for permanent storage of large amounts of data, unlike in variables and arrays data is only stored termporarily (when the program is executing) and is deleted once the program terminates.

Hence Importance: The introduction of data files which can created , updated and processed by C programs to resolve this issue.

- File types in C:
  - Binary files (non -readable)
  - Text Files
    *data in the form of characters
    *Able to read and write to
- Opening files
  - Accessing and modifying files requires the use of pointeR of type *FILE* ,which points to the file to be accessesing (memory of the file to be accessed)

```
FILE *cfPtr;
cfPtr = fopen( "someFilename.xyz", "r" );
```

- Closing files
  - files should always be closed after use

```
fclose( cfPtr );
```

File opening modes

Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, *discard* the current contents.
a	Open or create a file for writing at the end of the file—i.e., write operations *append* data to the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for reading and writing. If the file already exists, *discard* the current contents.
a+	Open or create a file for reading and updating; all writing is done at the end of the file—i.e., write operations *append* data to the file.
rb	Open an existing file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append: open or create a file for writing at the end of the file in binary mode.
rb+	Open an existing file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append: open or create a file for update in binary mode; writing is done at the end of the file.

11.2 Files and Streams

Each file in C is viewed as a sequence of bytes. Files end with a *end-of-file* marker or files end at a specified byte

Stream created when a file is opened
A stream is a communication channel between a file and program. Meaning that external files are accessed in programs to store data retrieve stored data.

Three files with associated streams are automatically opened with program execution:

- stdin – standard input (keyboard)
- stderr – standard error (screen)
- stdout – standard output (screen)

Read/Write Functions in Standard Library
Useful functions from the standard library to know and be able to apply:

```
int fgetc( FILE *stream )
```

```
char *fgets( char *str, int count, FILE *stream );
```

```
int fscanf( FILE *stream, const char *format, ... );
```

```
int fprintf( FILE *stream, const char *format, ... )
```

11.3 Creating a File for sequential Access

```
int feof( FILE *stream );
```

- Returns true (1) if the *end-of-file* marker reached
  - useful in file processing when you want to know when to stop processing data.

To reset the file pointer to the beginning of the file, use:

```
void rewind( FILE *stream );
```

Example 1: FileProcessing1.c
Example 2: FileProcessing2.c

Chapter 11:

1.3 The Data hierarchy

- Bit – smallest data Item 0/1
- Byte- 8 bits
- Field – group of characters conveying meaning
- Record – group of related fields
- File – group of related records
- Database
- Data File: We distinguish two things in a data file
  - Record Key/ID

ID	NAME	SURNAME
253	Hashim	Amla
254	Temba	Bavuma
255	Quinton	De Kock
256	Morné	Morkel
257	...	...

  - Sequential File
    *Typically records sorted by Key

11.6 Write to/Read from Binary (OR Text)file

- Unformated I/O functions

```
fwrite
```

```
fread
```

- Writing and Reading Structures

```
fwrite( &myObj, sizeof( struct myStrct ), 1, fPtr );
```

```
fread( &myObj, sizeof( struct myStrct ), 1, fPtr );
```

- Writing and Reading Array of Structures

```
fwrite( &myObj, sizeof( struct myStrct ), numElements, fPt
fread( &myObj, sizeof( struct myStrct ), numElements, fPtr
```

- numElements – is the number of array elements

11.7 Sequential Write to binary file

Example 1: SequentialFileWrite.c
Example 2: SequentialFileRead.c

11.5 Random Acess Files

11.7.1 Writing to a random Acess File

- ***fseek()***
  - Purpose: sets file position pointer to a specific position)
  - *documentation of document on slide*

```
int fseek( FILE *stream, long int offset, int whence );
```

Example

```
struct coord xPnt = { 1.0, 5.0 };
fseek( fPtr, 4 * sizeof( struct coord ), SEEK_SET );
fwrite( &xPnt, sizeof( struct coord ), 1, fPtr );
```

Explain: sets pointer to file at the 5th position from the beginning of file ,using fseek, at which the 1 byte of data of type *struct coord* is written to the file.

11.8 Reading data from a Random Acess file

- Reading data with fseek()

Example

```
struct coord xPnt;
fseek( fPtr, 4 * sizeof( struct coord ), SEEK_SET );
fread( &xPnt, sizeof( struct coord ), 1, fPtr );
```

Explain: read specific number of bytes from a file into memory.

11.6 Creation of Binary file for random Access 1

Purpose: *example illustrates the creation of a binary file.*

Example1 : RndmAccBnryF.c

Purpose: Writing data to a random access file

Example 2: WrndmAccBnryF.c

Purpose: Reading data from random Access file

Example 3: RrndmAccBnryF.c

Chapter 12

12.1 Introduction

We distinguish between two different data types that differ with respect to memory:

- Static data types
- Dynamic data types

12.2 Self-Referencing structures

- Self-Referencing Structure

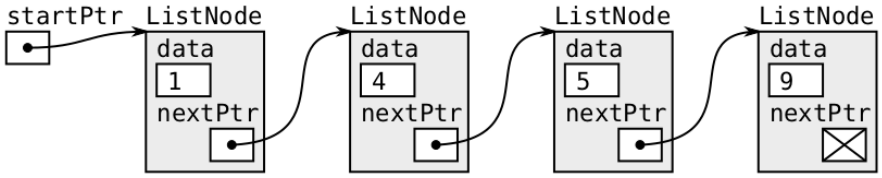
12.3 Dynamic memory Allocations

- Dynamic Structures and Dynamic memory Allocations

- Dynamic memory management is made possible by the following functions ; malloc() , sizeof() , free()

12.4 Linked Lists

- Linear collection of self referential stuctures, with multiple Nodes connected to each other via a link
- Linked lists are accessed via a pointer to the first element and from there subsequent Nodes are accessed via the link pointer Member (within the structure) of the current Node.



- Last elemement link to be set to Null which indicates the end of the list

The most important functions associated with linked lists are:

- **searching**
- **insertion**
- **deletion**

Example Code for each function

Each of these functions are discussed with the appropriate graphics in the slides

<u>Purpose:</u> Linked lists are applicable when the number of elements in a linked lists is not know in advance, and unlike arrays its size can be dynamically increased if more data becomes available unlike arrays which are static data structure

12.5 Stacks

Constrained version of liked lists.Elements may only be inserted or deleted from the top (front) of stack

```
struct stackNode {
    int data;
    struct stackNode *nextPtr;
};
typedef struct stackNode StackNode;
```

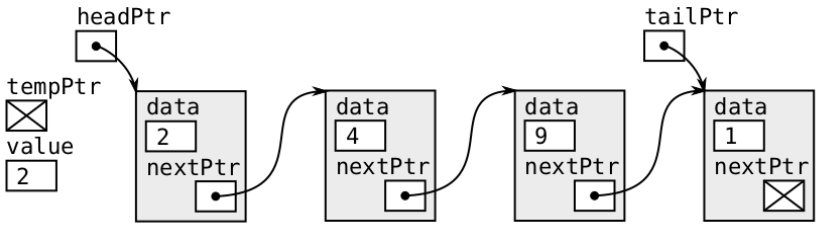
- Insertion in stacks (PUSH)
  - In a stack what happens is the stack pointer is always pointing at the top of the stack, when a new item is added to the stack, the stackPointer moves 1 position higher and then pointing to the latest point/Node/Element added
- **Deletion in stacks (POP)**

12.6 Queues

Constrained version of Linked lists. Elemenst may only be deleted from the front of a stack and inserted at the back (tail).

```
struct queueNode {
    int data;
    struct queueNode *nextPtr;
};
typedef struct queueNode QueueNode;
```

- **Insertion in queues (Enque)**
- Deletion in queues (Dequeue)



Explain

12.7 Trees

Contains 2 or more self-referential members.

- Binary tree

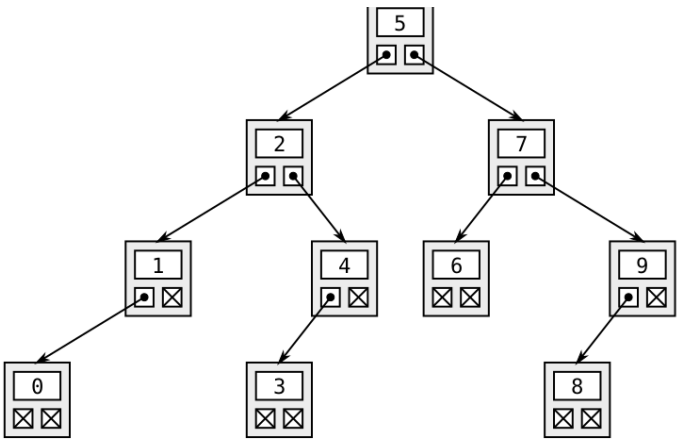
- Contains 2 self-referential members

```
struct treeNode {
  struct treeNode *leftPtr;
  int data;
  struct treeNode *rightPtr;
};
typedef struct treeNode TreeNode
typedef TreeNode *TreeNodePtr;
```

Creating and handling trees requies a combination of:

- structures
- pointers
- recursion/recursive functions

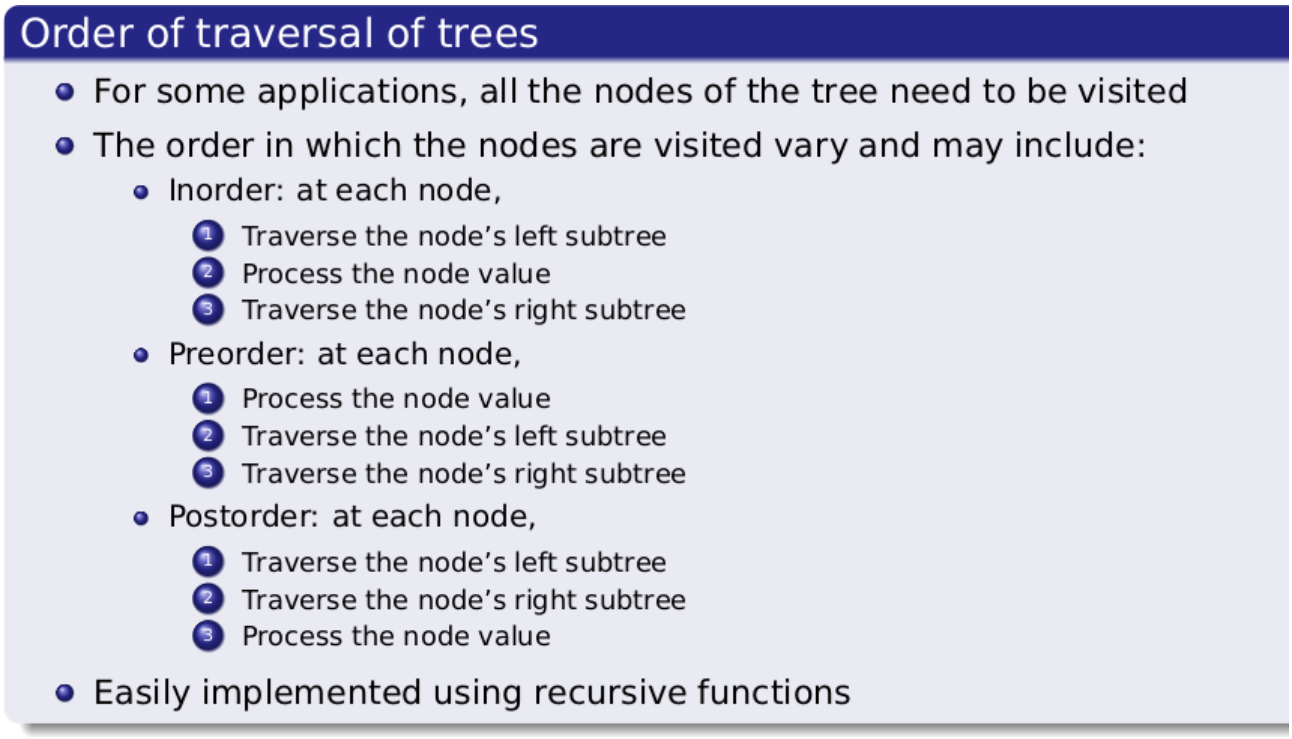
Sorted binary Trees



- Each left subtree, less than parent Node and each right tree greater or equal to the parent tree (if duplicates allowed)

Insertion in binary trees

Traversal of trees



Travering binary trees based on the type of order is illustrated on the slides to provide appropriate understanding. Clear that that traversion of trees is recursuve , but clarity on the function of the recursion such as if applied in loop to be confirmed.

Chapter 15: Object Oriented Programming

15.1-15.2
15.3-15.13 The c++ language
15.14 Introduction to object technology

Chapter 16: Classes and ...

...

Example: cppTest.cpp