

IT202
C Programming

Study Guide
Version 2.0

**© 2006 by Global Business Unit – Higher Education
Informatics Holdings Ltd
A Member of Informatics Group
Informatics Campus
10-12 Science Centre Road
Singapore 609080**

IT202 C Programming

Study Guide

Version 2.0
Revised in June 2006

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted by any form or means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Every precaution has been taken by the publisher and author(s) in the preparation of this book. The publisher offers no warranties or representations, not does it accept any liabilities with respect to the use of any information or examples contained herein.

All brand names and company names mentioned in this book are protected by their respective trademarks and are hereby acknowledged.

The developer is wholly responsible for the contents, errors and omission.

CHAPTER 1: INTRODUCTION TO C PROGRAMMING
--

Objectives	1-1
1.1 Values in C	1-2
1.1.1 Numeric Values	1-2
1.1.2 Character Values	1-2
1.2 Variables in C	1-2
1.3 Data Types and Declaration	1-3
1.3.1 Declaration	1-3
1.3.2 Data Types	1-4
1.4 Arithmetic Expressions	1-5
1.4.1 Integer Arithmetic	1-6
1.5 Assignment	1-7
1.6 Sample Programs	1-8
1.7 Programming Exercise	1-10
1.8 Revision Exercise	1-13

CHAPTER 2: BUILDING SIMPLE C PROGRAM

Objectives	2-1
2.1 Output	2-2
2.1.1 Conversion Codes	2-2
2.1.2 Size Modifiers	2-3
2.2 Input	2-3
2.2.1 Conversion Codes	2-4
2.2.2 Prompts	2-4
2.2.3 Flushing the Input Stream	2-5
2.3 Sample Programs	2-6
2.4 Programming Exercise	2-9
2.5 Revision Exercise	2-12

CHAPTER 3: THE ITERATION STRUCTURE

Objectives	3-1
3.1 Structured Programming	3-2
3.2 Conditions	3-2
3.2.1 Relational and Equality Operators	3-2
3.2.2 Logical Operators	3-3
3.3 The <i>if</i> and <i>else</i> Statement	3-5
3.4 The <i>switch</i> Statement	3-6
3.5 Sample Programs	3-7
3.6 Programming Exercise	3-18
3.7 Revision Exercise	3-23

CHAPTER 4: ITERATION

Objectives	4-1
4.1 The Iteration Structure	4-2
4.1.1 Pretest Loops	4-2
4.1.2 Posttest Loops	4-3
4.2 Accumulating and Counting	4-3
4.3 Counter-Controlled Loops	4-4
4.4 Nested Loops	4-5
4.5 Sample Programs	4-6
4.6 Programming Exercise	4-15
4.7 Revision Exercise	4-18

CHAPTER 5: FUNCTIONS

Objectives	5-1
5.1 Why Function are Used in C	5-2
5.2 The Structure of Functions	5-3
5.2.1 The Function Definition	5-3
5.2.2 Calling the Function	5-4
5.2.3 Function Prototype	5-4
5.2.4 Local Variables	5-5
5.2.5 Functions that Return a Value	5-6
5.2.6 Using Arguments to Pass Data to a Function	5-7
5.2.7 Passing Multiple Arguments	5-7
5.2.8 External Variables	5-8
5.3 Sample Programs	5-10
5.4 Programming Exercise	5-16
5.5 Revision Exercise	5-19

CHAPTER 6: ARRAYS

Objectives	6-1
6.1 Array Declaration	6-2
6.2 The Variable Defined Array	6-3
6.3 Sample Programs	6-4
6.4 Programming Exercise	6-10
6.5 Revision Exercise	6-13

CHAPTER 7: STRING

Objectives	7-1
7.1 String Variables	7-2
7.2 String Input	7-3
7.3 String Output	7-4
7.4 String Manipulation	7-5
7.5 Character Classification	7-6
7.6 Sample Programs	7-7
7.7 Programming Exercise	7-15
7.8 Revision Exercise	7-18

CHAPTER 8: FILES

Objectives	8-1
8.1 Opening Files	8-2
8.1.1 Files Modes	8-2
8.1.2 Binary versus Text Files	8-3
8.2 Closing a File	8-3
8.3 Character Access to Files	8-3
8.4 Finding The End of A File	8-4
8.5 Sample Programs	8-5

CHAPTER 9: Record-Based Data

Objectives	9-1
9.1 Structures	9-2
9.1.1 Definition and Declaration	9-2
9.1.2 The <i>sizeof</i> a Structure	9-4
9.2 Accessing Structures	9-4
9.3 Arrays of Structure	9-5
9.4 Files and Structure	9-5
9.5 Sample Programs	9-7

CHAPTER 10: PROJECT GUIDELINES 10-1

APPENDIX A: LESSON PLAN A-1

APPENDIX B: REVISION EXERCISE ANSWERS B-1

APPENDIX C: PROGRAMMING EXERCISE SOLUTIONS C-1

Chapter 1 – Introduction To C Programming

Objectives

Students would learn how to form a simple C program and expose to C language step by step. Additionally, students would know fundamentals of C language.

- Understand what is variable in C
- Know what is data type, and the data types available in C language
- Know how to declare different variable with appropriate data type
- Know and learn how arithmetic operators are used in C program
- Know how to interpret C arithmetic operation in C
- Learn how to display messages and values on screen

1.1 Values in C

There are two types of values in C known Integral and Real numbers. Due to different storage space is required for different values, students have to know the type of values in order to declare some memory spaces to store them.

1.1.1 Numeric Values

There are two categories of numeric values: real numbers, those allow decimal points; and integral, or whole numbers. In C, decimal numbers in either category can be expressed exactly as we do in normal math.

1.1.2 Character Values

Character value consists of a letter or other character surrounded by single quotes. A character is a one-byte space in memory in which the character constants, such as 'a' or 'X', can be stored. The type name for character is *char*. Some characters in ASCII are not on the keyboard and therefore would be difficult to show in character notation. In C, we represent these characters with special characters, each of which consists of a backslash (\) followed by a printable character. Even though two characters are shown within the quotes but they are stored as a single character. The following is a list of special characters in C.

\0	Null (absence of character)
\a	Audible alarm (alarm)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Apostrophe (single quote)
\"	Quote (double quote)
\?	Question mark
\\	Backlash

1.2 Variables in C

Variables represent some segment of memory location in computer. Different values placed in the storage depend on the value type. Variable should be declared prior to storage of any values. When declaring variables in C, rules must be followed, such as:

- Use alpha characters such as A through Z or a through z, numeric characters such as 0 through 9.
- Spacing in a variable name is not allowed. If the variable name contains more than one word, there should be underscore in place of the space. For example, *gross pay* should be ***GrossPay***, ***gross_pay*** or ***Gross_Pay***.
- Variable name must begin with an alpha character or an underscore, not a number. For example, ***Farley*** and ***_Age*** are valid variable names, but ***2bad*** and ***2Bad*** are not valid variable names.
- C is case-sensitive; upper- and lowercase characters are not treated the same. For example, ***Total***, ***total***, and ***TOTAL*** would be considered as three different variable names. For readability, many C programmers capitalized the first character of each word in the variable and limit the use of the underscore.
- Reserve words are not allowed. The C compiler looks for certain key words, words with special meanings, when it compiles the program. They are treated as reserved words – set aside for use by the compiler. A list of ANSI C's 32 reserved words is shown in **Table 1-1**:

Table 1-1 C's Reserved Words

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

1.3 Data Types and Declarations

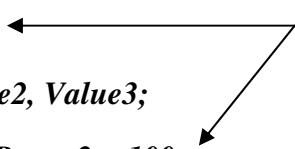
Data type is some reserve words used to specify the values whether they are integral, real, character, etc. Different storage space would be allocated depends on the Data Type. Declaration is a process of reserving storage area in order to store different values temporary.

1.3.1 Declaration

Whenever we use a value or a variable, we will declare it and its data type. Variables must be explicitly declared before they are used. Each declaration statement in C language must end with a semicolon. A declaration performs a number of functions such as (i) it directs C as to how to store the value, (ii) it allocates memory to the value or variable, and the type of declaration tells the computer how much memory required, and (iii) it initializes a variable by assigning a meaningful value to that space in memory. There is various way of declaring variables, and the following are some examples of variable declaration.

Example:

```
int GrossPay;  
int NetPay = 0; ← declaration & initialization  
int Value1, Value2, Value3;  
int Range1 = 1, Range2 = 100;
```



1.3.2 Data Types

Integral Data Type

Integral data types allow only integral, or whole, numbers with no decimal points. Integral data type can be *singed* or *unsigned*. The signed integral data type means that the value may be either positive or negative. Generally, we do not need to specify whether a variable is *singed* because by default it is so. The *unsigned* data type means the number may only be positive.

Additionally, integral data type may be either *short* or *long*, whereby it tells the computer the size of the integral number. In most of the cases, *short* is 16 bits and *long* is 32 bits. There are different ranges of value allowed for short and long integral numbers; **Figure**

1-1 shows all the ranges. The following are some examples of integral variable declaration.

Example:

```
int CurrentPage, LastPage;  
short Age;  
unsigned Value = 100;  
unsigned long LargeNumbers;  
signed short SmallValues;
```

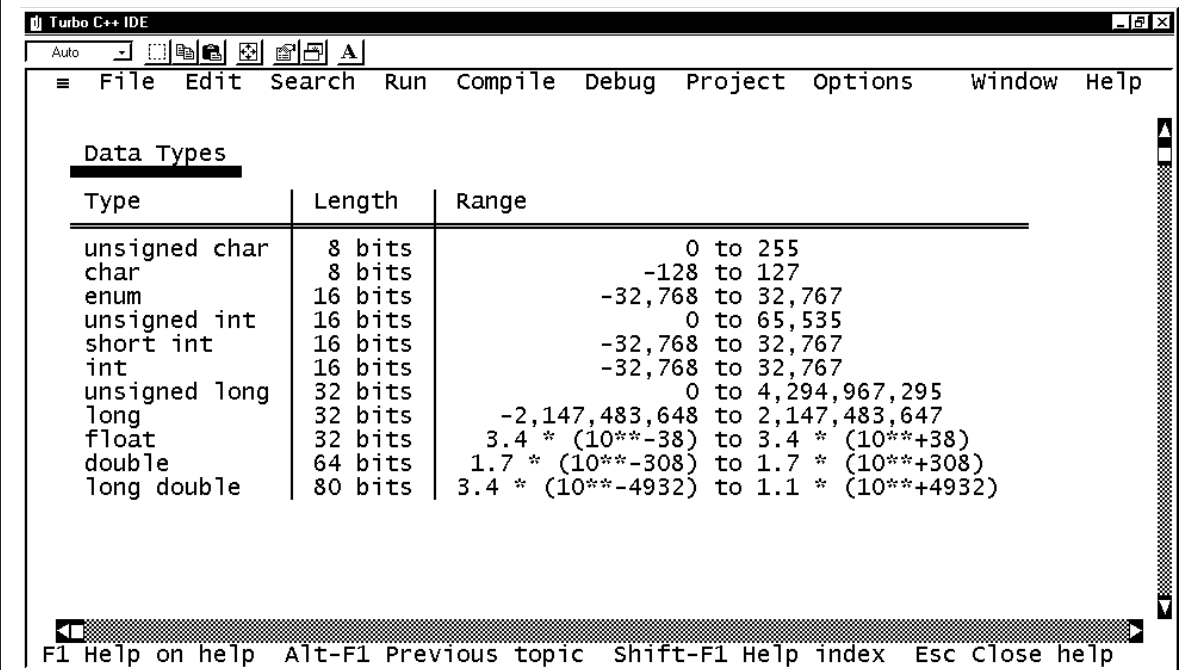
Floating-Point Data Type

Real numbers, those with decimal points, are stored as floating-point data types. For floating-point values, the computer stores the mantissa and the exponent. The memory space allocated to the number is fixed, depending on the data type we declare, and is divided between mantissa and exponent.

There are three different floating-point data types: *float*, *double*, and *long double*. These data types differ from each other in terms of the size. Generally, a *float* is 32 bits, a *double* is 64 bits, and a *long double* is 80 bits. Unlike integral data types, floating-point data types always allow positive or negative values, and so cannot have *unsigned* or *signed*. When declaring a *long double*, to be sure to state *long double*, not just *long*, because that would default to a *long int*. There are different ranges of value allowed for floating-point data types as shown in **Figure 1-1**. The following are some examples of floating-point variable declaration.

Example:

```
float WageRange = 12.78;  
double Area, Volume;  
long double Humongous;
```

Figure 1-1 Data Types and Their Range


Type	Length	Range
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	$3.4 * (10^{**-38})$ to $3.4 * (10^{**+38})$
double	64 bits	$1.7 * (10^{**-308})$ to $1.7 * (10^{**+308})$
long double	80 bits	$3.4 * (10^{**-4932})$ to $1.1 * (10^{**+4932})$

F1 Help on help Alt-F1 Previous topic Shift-F1 Help index Esc Close help

1.4 Arithmetic Expressions

In C, an expression is anything that can be simplified into a single value, such as formulas. An arithmetic expression consists of values and/or variables connected by arithmetic operators, which tell the computer how to combine the values. Many expressions, such as $12 + 9 / 3$, have more than one operator. There are strict rules about which operation is to be done first no matter where it occurs in the expression. The rules involve precedence, a hierarchy or ranked order of operations that dictates the types of operations that are to be performed before other types; and associativity rules, which dictates order if two operations have the same level of precedence. **Table 1-2** shows the arithmetic operators in precedence, highest first, and their associativity.

Table 1-2 Arithmetic Operators in Precedence

Level	Type	Associativity	Operator	Symbol	Example
1	Unary	Right to left	Negate	-	-4
			Plus	+	+4
2	Multiplicative	Left to right	Multiply	*	3 * 3
			Divide	/	10 / 5
			Remainder	%	7 % 5
3	Additive	Left to right	Add	+	10 + 9
			Subtract	-	10 - 9
4	Assignment	Right to left	Equals	=	A = 10

1.4.1 Integer Arithmetic

C performs its arithmetic to the data types on which it is currently operating. If the data types in a particular operation are integer then the result will also be integer. When dividing an integer by an integer the result will be an integer – a truncated version of what a floating-point division would yield. The result of the expression $3 / 2$ is 1, not 1.5. Since both 3 and 2 are integers, an integer calculation will be done, producing an integer result. There is not a round-off but a truncation; the result is not 2, but 1.

The remainder (or modulus) operator - % symbol, is valid only with positive integers. The result of is the remainder after dividing the value before the operator by the value after the operator. The result of the expression $5 \% 3$ is 2 because 5 divided by 3 is 1 with a remainder of 2. The following are more examples of remainder operation in C.

Example:

$$13 \% 3 = 1$$

$$1 \% 5 = 1$$

$$1 \% 3 = 1$$

$$8 \% 3 = 2$$

$$14 \% 1002 = 14$$

1.5 Assignment

As mentioned earlier, variables identify spaces in main memory. These spaces are variable because they can contain various and changeable values. Putting a value into one of these spaces is known as assignment. We usually refer to “assigning a value to a variable,” but technically, we are writing a value into the memory space identified by the variable.

Sometime, we must also make assignments as part of our program. For example, we wish to store the results of calculation in variable, but all of them follow this fundamental rule:

A variable may have only ONE value at a time

We may assign many different values to a variable, but each time we do, we write to the memory space reserved for that variable. The standard assignment operator is the equal sign (=). The following are some valid assignment statements:

Example:

$x = 17;$

$x = (y + 4) / 10;$

$Total = GrossPay + OvertimeAmount;$

1.6 Sample Programs

Program 1-1

```
#include <stdio.h>
void main(void)
{
    int a;
    a = 10;

    printf("The value in a is %i\n", a);
}
```

Output

The value in a is 10.

Program 1-2

```
#include <stdio.h>
void main(void)
{
    printf("Here is the character %c\n", 'X');
    printf("and the numbers %i and %g\n", 100, 235.654);
    printf("Now we print %i as a number\n", 'X');
}
```

Output

Here is the character X

And the numbers 100 and 235.654

Now we will print 88 as a number

Program 1-3

```
#include <stdio.h>  
void main(void)  
{  
    int a, b;  
    a =30;  
    b = 10;  
  
    printf("The value in a is:      %i\n", a);  
    printf("The value in b is:      %i\n", b);  
    printf("The addition result is:   %i\n", a+b);  
    printf("The subtraction result is: %i\n", a-b);  
}
```

Output

The value in a is:	30
The value in b is:	10
The addition result is:	40
The subtraction result is:	20

1.7 Programming Exercise

1. Write a program to give some information about yourself. It should produce something like the following:

NAME: your name

MAJOR:

OTHER COMPUTER COURSE TAKEN:

OCCUPATION:

2. Write a program to find the average of the four values 4, 42, 16.7, and .0045.

Variables:

v1, v2, v3, v4

Average

Output

The four numbers are:

4 42 16.7 0.0045

The average is:

15.676126

3. Write a program that produces a bill and coin breakdown for an amount of money. Initialize the amount in the *float* variable known as *dollars* and use the *cents* variable to keep track of the amount not yet converted to bills and coins. You will have to make use of integer arithmetic and the remainder in this program.

Variables

dollars (float)

cents (int)

Output

The coin breakdown for 7.73 dollars is:

Dollar bills: 7

Fifty cents: 1

Twenty cents: 1

Ten cents: 0

Five cents: 0

One cents: 3

4. There are 12 inches in a foot. Write a program in which you initialize the integer variable *inches* to a value, say 46, assign the number of feet to *feet*, and print the result.

Variables

Inches, feet

Output

56 inches is 3.83333 feet

1.8 Revision Exercise

1. Of the following, which are invalid variable names? Why are they invalid?
 - i. Gnash
 - ii. union
 - iii. 9Times
 - iv. too_many
2. Explain why are the following declarations are invalid?
 - i. Unsigned float zip = 46;
 - ii. Double Dip;
 - iii. Short stuff, pants = 12.6;
 - iv. Long double disaster
3. In the C you use, in what data types of the following values be stored? In what data types could they actually fit?
 - i. 91
 - ii. 45
 - iii. 48652
 - iv. 16.25
4. What are the values and data types of the following expression?
 - i. $7 / 4$
 - ii. $9 / 2. + 25 / 3$
 - iii. $6 + 4.8 / 2 * 3$
 - iv. $25 \% 5 + 12.5 * 2 / 5$
5. If $x = 5$ and $y = 2$ and both are integers, what are the values of the following expressions?
 - i. $x \% y + 14.6 / y$
 - ii. $y = 1. * x / 2 + 3.5$
 - iii. $y = y * x$
 - iv. $y = 16.2 * x / 3$

6. Briefly describe the following

- i. Which characters can be used in C variable names? What can the name start with?
- ii. Are `books` and `BOOKS` the same variable?
- iii. What does the data type say about a value or variable?
- iv. What does a declaration do in a program?
- v. Which data types are integral? Which are floating-point?
- vi. What characters do we use at the end of numbers to declare values as unsigned? Long? Unsigned long? Float? Long double?
- vii. How do we put a quote mark in string value?
- viii. How does C interpret two string values separated only by whitespace?

7. State whether the following statements are TRUE or FALSE.

- i. All single statements in C can extend over several lines in the program.
- ii. Variable names in C are not case-sensitive.
- iii. *Name* and *name* would refer to the same variable.
- iv. Variable names should never be identical to any of C's reserved words.
- v. The initialization of a variable in a declaration is optional.
- vi. If a value has no decimal point, it is integral.
- vii. An *unsigned int* cannot be zero.
- viii. There are no string variables in C.
- ix. The associativity rule applies when two operations are on the same precedence level.
- x. `+` has higher precedence than `-`.

Chapter 2 – Building Simple C Program

Objectives

In chapter two students would learn more functions such as **printf()** and **scanf()**. The chapter enables students to expose to two basic functions of C for message displaying and data entry.

- Understand how to prompt messages on the screen for data entry purposes
- Learn how to capture data by incorporating the **scanf()** function
- Understand why and when to include appropriate Conversion Codes in C program
- Understand and appreciate the purpose of Size Modifiers in C program
- Create simple data entry program to capture input temporary

2.1 Output

Output in C means display messages and values on screen. The ***printf***() function is used to produce output. When use the ***printf***() function to display something, appropriate Conversion Code should be included depends on the type of values to be displayed. The Size Modifiers enables a programmer to format the output for variable.

2.1.1 Conversion Codes

Conversion codes in C are used to reserve space in the output for some other values to print – the value of a variable or expression, and to show how those values should be converted to characters and printed. All conversion codes begin with a % symbol and with a type specifier. A list of conversion codes for ***printf***() is found in **Table 2-1**. The following example illustrates how conversion code is used.

Example:

```
void main(void)
{
    int x = 100;
    printf("The value in variable x is: %i", x);
}
```

Explanation:

All the characters in the format are printable except the conversion code ***%i***, which reserves space for an integer value to print. The format requires a value to fill in that space, so C takes the value of ***x***, and puts it there.

Table 2-1: Data Type Specifiers for printf()

Specifier	Data Type	Sample	Output
c	char	"%c", 'A'	A
d	int	"%d", 123	123
i	int	"%i", 345	345
f	float or double	"%f", 123.45	123.450000
g	float or double	"%g", 123.45 "%g", 123.00	123.45 123

2.1.2 Size Modifiers

Size modifier normally appears before the type specifier. We may set the minimum width of a print field by putting a number before the type specifier, if one exists. The format of the size modifier with width and precision is as below:

`%[width][.precision][size] type`

Example:

<u>Statement</u>	<u>Output</u>
<code>“%4i”, 123</code>	<code>*123</code>
<code>“%04i”, 123</code>	<code>0123</code>
<code>“%8.2f”, 12.34</code>	<code>***12.34</code>
<code>“%.2f”, 1223.24</code>	<code>1223.24</code>
<code>“%1.2f”, 1234.23</code>	<code>1234.23</code>
<code>“%5.1f”, 12.3678</code>	<code>*12.4</code>

*Note: * denote space*

The **width** parameter is the minimum width. If the value to be printed has more characters than the minimum, the entire value will be printed. By default, fields with specified widths will be right-justified, that is, line up on the right side. They will be padded with leading spaces. In addition to the width, the **precision** parameter comes after the width if there is one, before the size modifier if there is one, and always starts with a decimal point. The precision parameter for floating point data type specifies the number of digits after the decimal point; the value will be rounded to that number of digits. The width is the minimum width of the entire field, inclusive of signs (negative or positive), decimal points, and digits after the decimal point.

2.2 Input

The input in C enables user to enter values for e.g. data entry. The function enables programmer to prompt for data entry during program execution. Appropriate Conversion Codes should be specified in order to display the required value. Consider the following example:

Example:

scanf("control string", location, location, ..., location)

or

scanf("%f %i", &a, &b);

Explanation:

The *scanf*() function takes input characters and then divide them into sets of characters to convert to appropriate data types for the locations. The ampersands (&) in front of the variable *a*, and *b* is the locations in main memory – the memory addresses of *a*, and *b*.

2.2.1 Conversion Codes

When the *scanf*() function executes, it matches the character in the input stream with the characters in the control string. A single conversion code in the control string will match input characters following the steps below:

1. Leading white space characters (spaces, tabs, and newlines) are skipped (except for type specifier c). A previous *scanf*() will leave a newline in the input stream.
2. Subsequent characters will be taken for conversion and assignment up to the first character that is inappropriate for the data type.

For example, if the input characters are ***46.8\n (the * indicates a space), a *%i* conversion code would skip the three spaces and store the value 46 at the memory location of the variable, leaving the characters .8\n for the next match. The conversion stopped there because a decimal point is not an appropriate character for an integer conversion.

2.2.2 Prompts

The *printf*() before each *scanf*() is known as prompt. The *printf*() exist to display something on the screen to tell the person at the keyboard what to type in.

2.2.3 Flushing the Input Stream

If the person at the keyboard is not careful to give responses that fit the control string in the *scanf()*, there is a chance that extra characters will be left in the input stream waiting to be converted by the next *scanf()*. Consider the following example:

Sample program run:

Enter an integer: **1.23**

Enter a float: Integer: 1. Float: 0.23000

In the sample program run, 1.23 was typed in as the integer. The *scanf()* converted to the first inappropriate character, the decimal point, and assigned the value 1 to the next variable (of the *float* type). At that point, the input stream contained .23. The next *scanf()* found characters in the stream, and they were appropriate for a *float*, so without waiting for any further input, it converted them and assigned the value 0.23 to the next variable (of the *float* type). To clean up some of the problems we may encounter from characters left in the input stream, we can empty the stream with the following statement:

while (getchar() != '\n');

OR

fflush(stdin);

2.3 Sample Programs

Program 2-1

```
#include <stdio.h>
#define FEET_METER 3.2808

void main(void)
{
    char letter = 'A';
    int meters = 2;
    float feet = 25.8;

    printf("The character %c can also be interpreted as the number %i\n",
        letter, letter);
    printf("At %f feet per meter, %i meters is %g feet.\n",
        FEET_METER, meters, meters * FEET_METER);
    printf("%f feet is %g meters.\n", feet, feet / FEET_METER);
}
```

Output

The character A can also be interpreted as the number 65.

At 3.280800 feet per meter, 2 meters is 6.5616 feet.

25.799999 feet is 7.86394 meters.

Program 2-2

```
#include <stdio.h>

void main(void)
{
    long eastville = 322536, westport = 643, gotham = 6445821;

    printf("Eastville had %6li people.\n", eastville);
    printf("Westport has %6li people.\n", westport);
    printf("Gotham has %6li people.\n", gotham);
}
```

Output

Eastville has 322536 people.

Westport has 643 people.

Gotham has 6445821 people.

Program 2-3

```
#include <stdio.h>

void main(void)
{
    float x = 1.2345;
    int y = 1;

    printf("Number positions: 123456\n");
    printf("The value of x is %6.2f\n", x);
    printf("The value of x is %6.2f\n", x);
    printf("The value of y is %6.2i\n", y);
}
```

Output

```
Number positions: 123456
The value of x is      1.23
The value of x is      1.2
The value of y is      01
```

Program 2-4

```
#include <stdio.h>

void main(void)
{
    float food, drink, tip, total, tax, bill;

    printf("Food total: ");
    scanf("%f", &food);
    printf("Beverages: ");
    scanf("%f", &drink);
    printf("Tip: ");
    scanf("%f", &tip);
    total = food + drink + tip;
    tax = total * 0.06;
    printf("Tax: %6.2f\n", tax);
    bill = total + tax;
    printf("Please pay: %6.2f\n", bill);
}
```

Output

Food total: **34.82**

Beverages: **16.75**

Tip: **6**

Tax: 3.45

Please pay: 61.02

Program 2-5

```
#include <stdio.h>
```

```
void main(void)
```

```
{ int i;
```

```
  long l;
```

```
  float f;
```

```
  double d;
```

```
  printf("Enter values for an int and a long: ");
```

```
  scanf("%i %li", &i, &l);
```

```
  printf("Your int is %i and long is %li.\n\n", i, l);
```

```
  printf("Now, enter values for a float and a double: ");
```

```
  scanf("%f %lf", &f, &d);
```

```
  printf("Your float is %f and double is %f.\n\n", f, d);
```

```
  printf("Enter more values for a float and a double: ");
```

```
  scanf("%f %f", &f, &d);
```

```
  printf("Your float is %g and double is %g.\n", f, d);
```

```
}
```

Output

Enter values for an int and a long: **524 79735**

Your int is 524 and long is 79735

Now, enter values for a float and a double: **12.345 12.34567890123**

Your float is 12.345000 and double is 12.345679

Enter more values for a float and a double: **98.765 98.7654321**

Your float is 98.765 and double is 12.3457

2.4 Programming Exercise

1. Write a program that accepts two numbers from the keyboard and prints the following information.

Variables

first

second

Output

First number? **7**

Second number? **2**

The second number goes into first 3 times
with a remainder of 1.

The quotient is 3.5.

2. Write a program to print out a customer bill for Ajax Auto Repair. The parts and labor charges are input and a 6 percent sales tax is charged on parts but not on labor. Your program should display the following information.

Variables

Parts

Labor

SalesTax

Total

Output

PARTS? **104.50**

LABOR? **182.15**

AJAX AUTO REPAIR

SERVICE INVOICE

PARTS	\$104.50
LABOR	182.15
SALES TAX	6.27
TOTAL	\$292.92

3. Write a program that will accept keyboard input of various coins and return the total value.

Variables

input (value from keyboard)

total (to accumulate the value of the inputs)

Output

Fifty cents? **3**

Twenty cents? **2**

Ten cents? **1**

Five cents? **2**

One cents? **3**

Your total is: \$ 1.93.

4. Write a program that accepts a number of seconds from the keyboard and converts it into days, hours, minutes, and seconds. Use integer arithmetic and the remainder operator.

Variables

seconds

Output

How many seconds? **106478**

Days: 1

Hours: 5

Minutes: 34

Seconds: 38

5. Ajax would like a program to compute an employee's paycheck. The employee's gross pay is the hours worked times the hourly pay. Income tax withholding, FICA tax, payroll savings plan, retirement, and health insurance are subtracted from the gross pay. From time to time the various rates for these deductions change, so the values should be put in #define directives.

Constants

FIT_RATE	15% of gross pay
FICA_RATE	6.2% of gross pay
SAVINGS_RATE	3% of gross pay
RETIREMENT_RATE	8.5% of gross pay
HEALTH_INS	\$3.75 of employee

Variables

Hours	
HourlyPay	
GrossPay	
FIT	Federal income tax withholding
FICA	Social security tax withholding
Savings	Payroll savings
Retirement	
NetPay	Gross pay less deductions

Output

HOURS? **40**

HOURLY PAY? **7.50**

GROSS PAY: \$300.00

FEDERAL INCOME TAX: \$45.00

FICA: \$18.60

PAROLL SAVINGS: \$ 9.00

RETIREMENT: \$25.50

HEALTH INSURANCE: \$ 3.75

NET PAY: \$198.15

2.5 Revision Exercise

1. Write the *printf*() conversion codes to provide the proper output from the values given. The vertical bar (|) shows the whole field including spaces (* denotes a space). If vertical bars are not given, no width should be specified.

a. 46	46*
b. 8.046	**8.05
c. 73.28	73.3*
d. 86425	*086425
e. 214	214.00
f. 'C'	67
g. 67	C
h. 35.6	***35.600

2. Show the output of these values given the *printf*() conversion codes (use * to denote space).

a. 'A'	%3i
b. 'A'	%c
c. 4.26	%6.1f
d. 425	%g
e. 381.67	%3.1f
f. 16	%4.3i
g. 1.2345	%.2f
h. 52	%5i

3. Briefly describe the following:
 - a. What is the purpose of a conversion code in either *printf*() or *scanf*()?
 - b. What character begins a conversion code and what character ends it?
 - c. What is a size modifier in a conversion code, and what are the possible ones for *printf*() and *scanf*()?
 - d. What is a width parameter in a conversion *printf*() code?
 - e. What is the precision parameter in a *scanf*()?

- f. What is a prompt, and how do we display one?
 - g. How can we empty the input stream?
4. State whether the following statements are TRUE or FALSE.
- i. All conversion codes begin with an `'*'`.
 - ii. When inputting integer data into an *int* variable using `%i`, blanks are not skipped.
 - iii. When inputting char data into a *char* variable using `%c`, blanks are not skipped.
 - iv. `printf()` and `scanf()` have control strings.
 - v. *It* is not possible to print a floating-point number without printing a decimal point.
5. Fill in the blanks with most appropriate word.
- i. The directive _____ establishes a character replacement for a C program.
 - ii. The `%c` conversion code is best used for printing _____ data.
 - iii. The notation _____ means the address of the variable *a*.

Chapter 3 – The Selection Structure

Objectives

The Selection Structure in C enables decisions to be incorporated. Students would learn and appreciate the purpose of including Selection Structure in C program. Additionally, students would expose to various Selection Structure programs in C.

- Understand the concept and purpose of Selection Structure in C
- Know how to incorporate decision making in C program
- Learn *if*, *switch* reserve words in C and how to include them in program
- Appreciate the importance and know the application of Selection Structure in C program

3.1 Structured Programming

The technique of structured programming simplifies the programming process by using only three types of programming patterns, called control structures, to build programs. By using them in various combinations you can write any program. The three structures are:

Sequence. A sequence structure is one operation after another. This is the structures been demonstrated in the previous chapter's program.

Selection. A selection structure is a choice between sets of operations. There will be some sample programs on Selection structure in this chapter.

Iteration. An iteration structure is a repetition of a set of operations, which would be discussed on next chapter.

3.2 Conditions

A condition enables the computer to perform certain tasks base on the outcome of evaluation. The statements execution would not be in any sequence whereby some statements might not be executed. Execution is normally base on the outcome of conditions – true or false. A condition contains one or more comparisons that relate one to another. Comparisons can be achieved by incorporating Relational operators and Logical operators in a condition. A condition in C typically consists of one or more comparisons that relate one value to another.

Example:

$$x + 10 > 20$$

Explanation:

The above example condition compares the value of the expression $x + 10$ with the value of 20. An expression is anything that reduces to a single value. The comparison operator, $>$ tells the computer how the comparison should be made.

3.2.1 Relational and Equality Operators

The comparison operator comes from one of two categories: relational operators and equality operators. The relational operators have higher precedence than equality operators. **Table 3-1** shows a list of relational and equality operators. The equal operator ($==$) is not the same as the assignment operator ($=$).

Table 3-1: Relational, Equality, and Logical Operators

Operator	Symbol	Example
Unary – Right-to-left associativity		
Logical NOT	!	!(a > b)
Relational – Left-to-right associativity		
Greater	>	a + b > c – 3
Less	<	x < y
Greater or equal	>=	Total >= 1000
Less or equal	<=	Cost <= 20
Equality – Left-to-right associativity		
Equal	==	value == y
Not equal	!=	NewAmount != OldAmount
Logical – Left-to-right associativity		
AND	&&	Value1 && Value2
OR		Value1 Value2

3.2.2 Logical Operators

Conditions can consist of one or more comparison. Multiple conditions can be tied together with two or more logical operators – AND (&&) and OR (||). The AND (&&) operator is higher in precedence than OR (||) operator. Both of them however, are lower than comparison operators, but higher than assignment operator. Order of evaluation can be adjusted any way we want by using parameters.

AND Operator

If the comparison on both sides is true, then the whole condition is true. If even one comparison is false, then the whole condition is false.

<u>C1</u>	&&	<u>C2</u>	<u>Result</u>
True		True	True
True		False	False
False		True	True
False		False	False

Example:

Conditions	Result
$2 + 7 \neq 38 / 2$	False
$16 \% 3 < 22 == 6 / 3 \geq 2$	True
$9 > 7 + 3 \ \&\& \ 7 \% 3 == 1$	True

OR Operator

By using the OR (||) operator, if either or both of the comparisons are true, then the whole condition is true. Both comparisons would have to be false for the whole condition to be false.

<u>C1</u>		<u>C2</u>	<u>Result</u>
True		True	True
True		False	True
False		True	True
False		False	False

Example:

Conditions	Result
$6 > 2 \ \ \ 25 / 5 == 4$	True
$7 \leq 9 \ \ \ 6 > 5 \ \&\& \ 7 == 2$	True
$(7 \leq 9 \ \ \ 6 > 5) \ \&\& \ 7 == 2$	False

NOT Operator

The NOT operator acts on only one expression. The logical NOT operator makes what was true false, and what was false true.

3.3 The *if* and *else* Statement

A selection structure in C is represented by *if* statement. Referring to **Figure 3-1** and **Figure 3-1**. The selection structure shown in **Figure 3-1** has one task/statement to be performed. In contrast the structure shown in **Figure 3-2** contain few tasks/statements to be performed base on the outcome of the condition evaluation. In a selection structure, if a condition produce a true outcome, the immediate statement(s) would be performed. However, if a condition produce a false outcome, the statement(s) after the *else* clause would be performed.

Figure 3-1: Selection structure with one task/statement

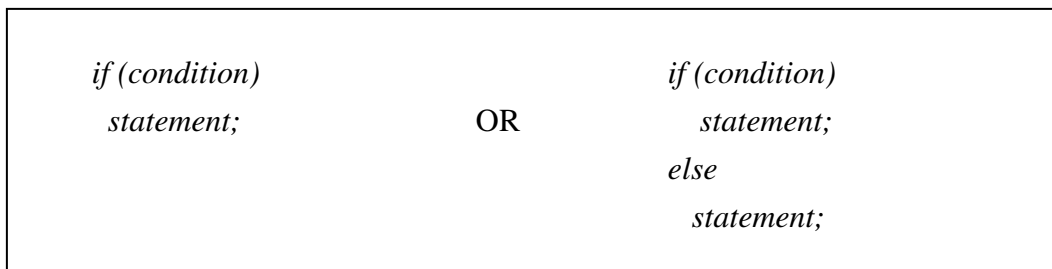
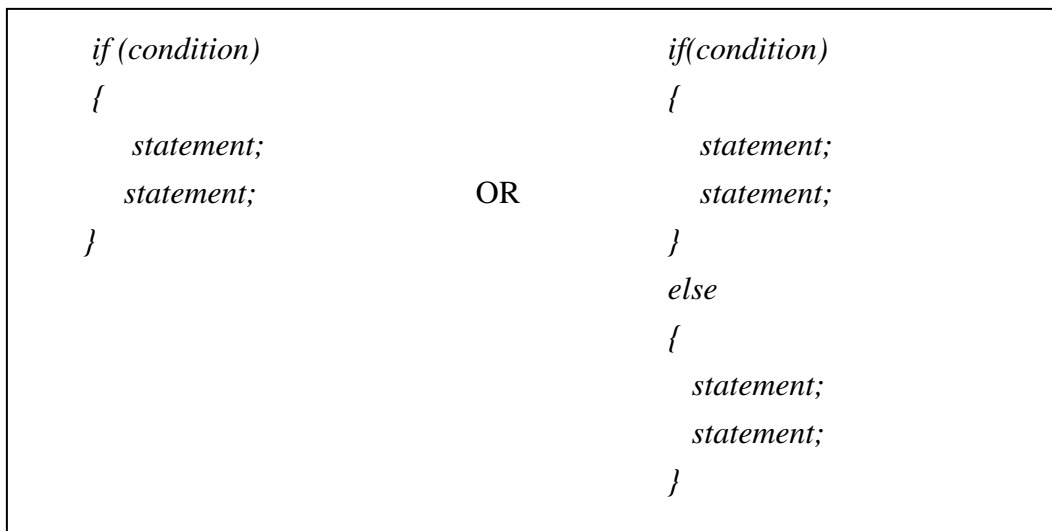


Figure 3-2: Selection structures with more than one tasks/statements



3.4 The *switch* Statement

Apart from *if* statement, a selection structure in *switch* statement is an alternative method. The *switch* statement simplifies the selection structure especially when there are multiple branches for statement execution. However, the *switch* statement has some limitations such as it only allow integral expressions or any other integral value and the *case* statement only evaluates integral value such as *int* or *char*. **Figure 3-3** shows a basic structure for the *switch* statement.

Figure 3-3: Switch structure

```
switch (integral_expression)
{
    case integral_value:
        statement;
        statement;
        .....
        break;
    case as many as are necessary;
    default:
        statement;
        statement;
}
```

The *integral_expression* following the *switch* key word must evaluate to some integral data type, *char* or *int*; floating-point results are not allowed. The value of the expression becomes a case value to be matched to the possible *case* identifiers within the statement block following the *switch*. For example, if the *integral_expression* evaluated to 6, the *switch* would look for *case 6*..

3.5 Sample Programs

Program 3-1

```
#include <stdio.h>  
#define SECRET 10  
  
void main(void)  
{ int guess;  
  
    printf("What's your guess? ");  
    scanf("%i", &guess);  
    if (guess == SECRET)  
        printf("You guessed that ");  
        printf("the secret number was %i.\n", SECRET);  
}
```

Output

What's your guess? **10**

You guessed that the secret number was 10.

What's your guess? **42**

The secret number was 10.

Program 3-2

```
#include <stdio.h>

#define SECRET 0.1
#define ACCEPT 0.01
void main(void)
{ float guess;

    printf("What's your guess? ");
    scanf("%f", &guess);
    if (guess >= SECRET - ACCEPT && guess <= SECRET + ACCEPT)
        printf("You guessed that ");
        printf("the secret number was %g.\n", SECRET);
}
```

Output

What's your guess? **0.1**

You guessed that the secret number was 0.1.

What's your guess? **0.15**

The secret number was 0.1.

Program 3-3

```
#include <stdio.h>

void main(void)
{ float weight, price = .2;

    printf("Enter weight of apple: ");
    scanf("%f", &weight);
    printf("Eve's ");
    if (weight > 10)
    { printf("Premium ");
      price = price + .1;
    }
    else
    { printf("Juicy ");
    }
    printf("Apple. $%4.2f.\n", price);
}
```

Output

Enter weight of apple: **9.2**

Eve's Juicy apple. \$0.20.

Enter weight of apple: **10.1**

Eve's Premium apple. \$0.30.

Program 3-4

```
#include <stdio.h>

void main(void)
{ float weight, price = .2;

    printf("Enter weight of apple: ");
    scanf("%f", &weight);
    printf("Eve's ");
    if (weight > 10)
    { printf("Premium ");
      price = price + .1;
    }
    else
    { if (weight > 8)
      { printf("Juicy ");
      }
      else
      { if (weight > 6)
        { printf("Snack ");
          price = price - .05;
        }
        else
        { printf("Cooking ");
          price = price - .1;
        }
      }
    }
    printf("Apple. $%4.2f.\n", price);
}
```

Output

Enter weight of apple: **12**
Eve's Premium apple. \$0.30.

Enter weight of apple: **7.5**
Eve's Snack apple. \$0.15.

Program 3-5

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

void main(void)
{
    float Amt=0, TaxRate;
    char Code;

    gotoxy(5,10);
    printf("Enter your country code [S/M/I]: ");
    scanf(" %c", &Code);
    gotoxy(5,11);
    cprintf("Enter your purchase amount: ");
    scanf("%f", &Amt);
    if (toupper(Code) == 'S')
    {
        gotoxy(38,10);
        printf("Singapore (8% tax rate)\n");
        TaxRate = 0.08;
    }
    else
        if(toupper(Code) == 'M')
        {
            gotoxy(38,10);
            printf("Malaysia (6% tax rate)\n");
            TaxRate = 0.06;
        }
    else
        if(toupper(Code) == 'I')
        {
            gotoxy(38,10);
            printf("Indonesia (3% tax rate)\n");
        }
    else
    {
        gotoxy(38,10);
        printf("Invalid code\n");
    }
    gotoxy(5,14);
    printf("Tax amount: %.2f", Amt * TaxRate);
    gotoxy(5,15);
    printf("Amount payable: %.2f", Amt + Amt * TaxRate);
}
```

Output

Enter your country code [S/M/I]: **S** Singapore (8% tax rate)

Enter your purchase amount: **100**

Tax amount: 8.00

Amount Payable: 108.00

Enter your country code [S/M/I]: **M** Malaysia (6% tax rate)

Enter your purchase amount: **100**

Tax amount: 6.00

Amount Payable: 106.00

Enter your country code [S/M/I]: **I** Indonesia (3% tax rate)

Enter your purchase amount: **100**

Tax amount: 3.00

Amount Payable: 103.00

Program 3-6

```
#include <stdio.h>

void main(void)
{
    int Age;

    printf("Enter your age (1-99 only): ");
    scanf("%i", &Age);
    if(Age > 1)
        if(Age < 99)
            printf("Valid age!\n");
        else
            printf("Invalid age!\n");
}
```

Output

Enter your age (1-99 only): **125**

Invalid age!

Enter your age (1-99 only): **-1**

Invalid age!

Enter your age (1-99 only): **18**

Valid age!

Program 3-7

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    char Gender;

    printf("Enter your gender [F/M]: ");
    scanf(" %c", &Gender);
    if(toupper(Gender) != 'M' && toupper(Gender) != 'F')
        printf("Invalid entry!\n");
    else
        if(toupper(Gender) == 'F')
            printf("\nFemale");
        else
            printf("\nMale");
}
```

Output

Enter your gender [F/M]: **F**

Female

Enter your gender [F/M]: **M**

Male

Enter your gender [F/M]: **O**

Invalid entry!

Program 3-8

```
#include <stdio.h>

void main(void)
{
    char Gender;

    printf("Enter your gender [F/M]: ");
    scanf(" %c", &Gender);
    switch(Gender)
    {
        case 'm':
        case 'M':
            printf("male!\n");
            break;
        case 'f':
        case 'F':
            printf("Female!\n");
            break;
        default:
            printf("Inavlid entry!\n");
    }
}
```

Output

Enter your gender [F/M]: **F**

Female

Enter your gender [F/M]: **M**

Male

Enter your gender [F/M]: **O**

Invalid entry!

Program 3-9

```
#include <stdio.h>

void main(void)
{
    int Choice;
    float Value1, Value2;

    printf("Enter value 1: ");
    scanf("%f", &Value1);
    printf("Enter value 2: ");
    scanf("%f", &Value2);
    printf("\n1. Multiply");
    printf("\n2. Divide");
    printf("\n3. Subtract");
    printf("\n4. Add");
    printf("Enter your choice [1-4]: ");
    scanf("%i", &Choice);
    switch(Choice)
    {
        case 1:
            printf("\nThe result is: %g", Value1*Value2);
            break;
        case 2:
            printf("\nThe result is: %g", Value1/Value2);
            break;
        case 3:
            printf("\nThe result is: %g", Value1-Value2);
            break;
        case 4:
            printf("\nThe result is: %g", Value1+Value2);
            break;
    }
}
```

Output

Enter value 1: **40**

Enter value 2: **30**

1. Multiply
2. Divide
3. Subtract
4. Add

Enter your choice [1-4]: **1**

The result is 1200

Enter value 1: **40**

Enter value 2: **30**

1. Multiply
2. Divide
3. Subtract
4. Add

Enter your choice [1-4]: **3**

The result is 10

Enter value 1: **40**

Enter value 2: **30**

1. Multiply
2. Divide
3. Subtract
4. Add

Enter your choice [1-4]: **4**

The result is 70

Program 3-10

```
#include <stdio.h>

void main(void)
{ float price;
  char grade;

  printf("Enter grade of apple: ");
  scanf(" %c", &grade);
  printf("Eve's ");
  switch (grade)
  { case 'P':
    case 'p':
      price = .3;
      printf("Premium ");
      break;
    case 'J':
    case 'j':
      price = .2;
      printf("Juicy ");
      break;
    case 'S':
    case 's':
      price = .15;
      printf("Snack ");
      break;
    default:
      price = .1;
      printf("Cooking ");
  }
  printf("Apple. $%4.2f.\n", price);
}
```

Output

Enter grade of apple: **P**

Premium Apple \$0.30.

Enter grade of apple: **s**

Snack Apple \$0.30.

3.6 Programming Exercise

1. Write a program to compare two numbers with executions similar to the following.

Variables

Number1, number2

Outputs

Enter two numbers **48 52.33**

52.33 is greater than 48.

Outputs

Enter two numbers **88 88**

They are equal.

2. Adams County (country code A) has a 7 percent sales tax rate; the rest of the state a 6 percent rate. Write a program to print out the amount owed on purchase including sales tax, given the amount of the purchase and the county.

Variables

Purchase

County

TaxRate

Outputs

AMOUNT OF PURCHASE? **100**

COUNTY? **A**

TOTAL BILL: 107.00

AMOUNT OF PURCHASE? **100**

COUNTY? **D**

TOTAL BILL: 106.00

3. Write a program to assign grade points according to a letter score. An A is 4 grade points; B is 3; C is 2; D is 1; and F is 0. Use the *else if* construct.

Variables

grade

grade_points

Output

Letter grade: **B**

Grade point: 3

Letter grade: **D**

Grade point: 1

4. The Ace Courier Service charges \$10 for the first pound or fraction thereof and \$6 per pound for anything over one pound. Write a program that figures the charges.

Variables

weight

Output

WEIGHT: **.7**

CHARGE: 10

WEIGHT: **2.5**

CHARGE: 19

WEIGHT: **4.2**

CHARGE: 29.2

5. Social security (FICA) tax is currently 7.65 percent of earnings up to \$50.00 for the year. Write a program that accepts earnings for the current week and previous earnings up to the current week, and returns the amount of FICA tax to be withheld.

Variables

CurrentEarnings

PrevEarnings

Output

This week's pay? **700**

Previous pay? **12600**

FICA to withhold: \$53.55

This week's pay? **1850**

Previous pay? **50200**

FICA to withhold: \$15.30

6. Write a program that prompts the user to enter two positive integers, and then tests whether the larger integer is exactly divisible by the smaller one. In the process, check the input values are both valid (greater than zero), and establish which of them is the larger.

Variables

value1

value2

Output

Enter two integer values: **20 30**

30 is larger than 20

30 is not divisible by 20

Enter two integer values: **-1 30**

Invalid value(s) entered!

Enter two integer values: **40 20**

40 is larger than 20

40 is divisible by 20

7. The ABC Insurance Company determines auto insurance rates based on a driver's age, the number of tickets in the last three years, and the value of a car. The base rate is 5 percent of the value of the car. Drivers under 25 years of age pay 15 percent over the base, and drivers from 25 through 29 pay 10 percent over. A driver with one ticket pays 10 percent over the rates already figured. Two tickets draws a 25 percent extra charges; three tickets adds 50 percent; and drivers with more than three tickets are refused. Write a program to show a driver's insurance premium.

Variables

Declare any appropriate variables

Outputs

DRIVER'S AGE? **35**

NUMBER OF TICKETS? **1**

VALUE OF CAR? **10000**

PREMIUM: \$550

DRIVER'S AGE? **29**

NUMBER OF TICKETS? **2**

VALUE OF CAR? **15000**

PREMIUM: \$1031.25

DRIVER'S AGE? **19**

NUMBER OF TICKETS? **3**

VALUE OF CAR? **850**

PREMIUM: \$73.3125

3.7 Revision Exercise

1. In mathematics, we might state a range from x as $1 < x < 5$. Show how we should write the following range expression in C.

- i. $1 < x < 5$
- ii. $16 \geq x \geq -7$
- iii. $22.6 \geq x \geq 12.03$
- iv. $36 < x \leq 115$

2. If $a = 1$, $b = 2$, and $c = 3$, are the following conditions true or false?

- i. $a \geq c - b$
- ii. $b / 2 == a \mid \mid c < 3$
- iii. $b < c - 2 \mid \mid a * 3 \geq c \ \&\& \ b > a$
- iv. $5 \mid \mid !b \ \&\& \ c$

3. With traffic lights, R (for red) means “Stop”, Y means “Caution”, and G means “Go”. Any other color letter means “Weird”. Given the statement below, write the program segment that prints what the color letter means. Use else if construct.

```
printf("Color letter:");  
scanf("%c", &color);
```

4. What is the difference in these two program segments? What value of x is produced if its initial value is 2?

```
if(x > 0)  
    x = x + 2;  
else if (x > 2)  
    x = x + 4;
```

```
if(x > 0)  
    x = x + 2;  
if (x > 2)  
    x = x + 4;
```

5. Briefly describe the following.

- i. How do we put more than one statement inside an if branch?
 - ii. Must a selection structure have an else branch?
 - iii. What is the different between = and ==?
 - iv. Are braces required around the true and false branches?
 - v. What is the difference between nested ifs using the else if construct and sequential ifs?
 - vi. What are the three logical operators and what is their precedence?
6. Using the following C code to answer questions (i) to (iii).

```
int x = -1;  
if (x * 30 / 10 > 5 - 6 * x)  
    x = x * 10;  
else  
    x = x + 10;
```

- i. What would be contained in *x* after executing the above segment of codes?
 - ii. If “*int x = -1*” is replaced with “*int x = -10*”, what would be contained in *x* after executing the above segment of codes?
 - iii. If “*x = x * 10*” is replace with “*x = ++(++x) * 10*”, and “*x = x + 10*” is replaced with “*x = ++(++x) * 10*”, what would be contained in *x* after the execution.
7. Fill in the blanks with most appropriate word(s).
- i. Enclosing something within something else is called _____.
 - ii. A(n) _____ must evaluate to either true or false.
 - iii. Multiple comparisons are tied together with _____.
 - iv. The tree control structures are _____, _____ and _____.
 - v. A program that uses human language rather than a computer language is said to be written in _____.
 - vi. One operation after another is called the _____ structure.

- vii. The _____ structure determines which of several alternatives will be executed.
- viii. A condition typically consists of one or more comparisons that compare one value to another. A comparison has the form _____.
- ix. The logical AND operator is the _____ symbol in C.
- x. The logical NOT operator is the _____ symbol in C.

Chapter 4 – Iteration

Objectives

The Iteration Structure in C enables some operations to be repeated until certain conditions are fulfilled or no longer valid. It saves the programmer's time in repeating similar codes/statements/instructions in a program. Additionally, the Iteration Structure simplifies and organizes lengthy programs. At the end of this lesson, students would have achieved the following objectives.

- Understand the concept and purpose of Iteration Structure in C
- Know how to incorporate repetition in C programs
- Understand the concept of pretest and posttest loops in C
- Learn and appreciate the *while, do...while* reserve words in C and how to include them in C program
- Appreciate the importance of Iteration Structure in C program

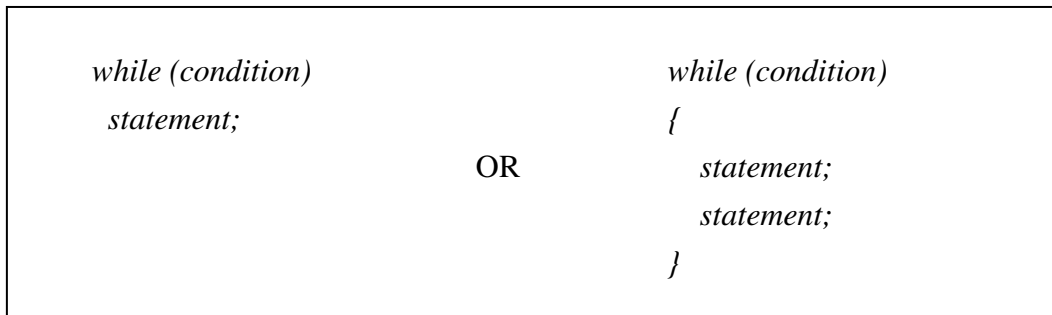
4.1 The Iteration Structure

The Iteration Structure is also known as loop or repetition. In C, a loop enables certain operations/tasks/instructions to be repeated. Those operations/tasks/instructions are repeated base on condition(s) outcome – true or false. There are two types of loops in C known as pretest and posttest loops. A pretest test loop as shown in **Figure 4-1** evaluates the outcome of condition(s) prior to tasks/operations. In contrast a posttest loop as shown in **Figure 4-2** would enable tasks/operations to be performed at least once prior to the condition(s) evaluation.

4.1.1 Pretest Loops

A pretest loop begins with the keyword *while*. The general form of this configuration is shown in **Figure 4-1**. Condition is enclosed in parenthesis; each statement in the block is indented one level and ends with a semicolon; and there is no semicolon after the block's closing brace. The C compiler requires the punctuation. The indentation and line endings are for program readability.

Figure 4-1: Pretest Loop



4.1.2 Posttest Loops

The body in posttest loop would be executed once no matter what. The posttest loops begins with a **do** and has general form as shown in **Figure 4-2**. There is semicolon at the end of condition.

Figure 4-2: Posttest Loop

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

4.2 Accumulating and Counting

Accumulating adds or multiplies few values together. Accumulating is quite common in loops especially in finding out total value for a range of values entered. Counting keeps track of number of times certain operations or instruction executed. A counter variable is normally declared to count by adding 1 each time the operation is performed.

Accumulating

Total = Total + value;

OR

Total += value;

Total = Total * value;

OR

Total *= value;

Counting

++times; or

times++; or

times = times + 1; or

times += 1;

4.3 Counter-Controlled Loops

A counter-controlled loop allows some operations to be executed for a fix number of times continuously without any interruption. In a counter-controlled loop there are five essential elements such as (i) initialization, (ii) test/condition, (iii) body, (iv) counter and (v) the end part. A typical counter-controlled loop's structure is shown in **Figure 4-3**.

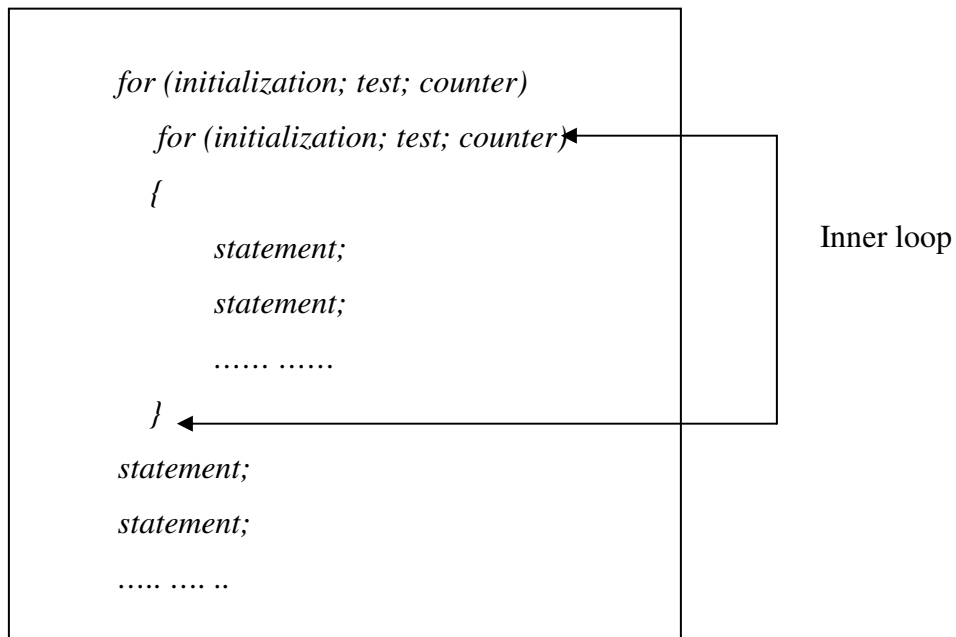
Figure 4-3: Counter-controlled loop

```
for (initialization; test; counter)
{
    statement;
    statement;
    .....
}
```

4.4 Nested Loops

A loop within another is known as nested loop. When there is any nested loops the inner one would always be executed and repeated prior to the outer one. **Figure 4-4** is the structure of a nested loop.

Figure 4-4: Nested Loop



4.5 Sample Program

Program 4-1

```
#include <stdio.h>

void main(void)
{ float price;
  short quantity;
  char answer;

  printf("Do you wish to enter a purchase (Y/N)? ");
  scanf(" %c", &answer);
  while (answer == 'Y' || answer == 'y')
  { printf("Enter 'price quantity': ");
    scanf("%f %hi", &price, &quantity);
    printf("The total for this item is $%6.2f.\n", price * quantity);
    printf("Another (Y/N)? ");
    scanf(" %c", &answer);
  }
  printf("Thank you for your patronage.\n");
}
```

Output

Do you wish to enter a purchase (Y/N)? y

Enter 'price quantity': **1.98 6**

The total for this item is \$11.88

Another (Y/N)? **Y**

Enter 'price quantity': **4.29 15**

The total for this item is \$ 64.35

Another (Y/N)? **n**

Thanks you for your patronage.

Program 4-2

```
#include <stdio.h>

void main(void)
{ float price;
  short quantity;
  char answer;

  do
  { printf("Enter 'price quantity': ");
    scanf("%f %hi", &price, &quantity);
    printf("The total for this item is $%6.2f.\n", price * quantity);
    printf("Another (Y/N)? ");
    scanf(" %c", &answer);
  }while (answer == 'Y' || answer == 'y');
  printf("Thank you for your patronage.\n");
}
```

Output

Enter 'price quantity': **2.45 12**

The total for this item is \$29.40

Another (Y/N)? **Y**

Enter 'price quantity': **.99 4**

The total for this item is \$ 3.96

Another (Y/N)? **n**

Thanks you for your patronage.

Program 4-3

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
    int Choice;
    float Value1, Value2;
    do{
        clrscr( );
        printf("Enter value 1: ");
        scanf("%f", &Value1);
        printf("Enter value 2: ");
        scanf("%f", &Value2);
        printf("\n1. Multiply");
        printf("\n2. Divide");
        printf("\n3. Subtract");
        printf("\n4. Add");
        printf("\n5. Quit");
        printf("\nEnter your choice [1-5]: ");
        scanf("%i", &Choice);
        switch(Choice)
        {
            case 1:
                printf("\nThe result is: %g", Value1*Value2);
                break;
            case 2:
                printf("\nThe result is: %g", Value1/Value2);
                break;
            case 3:
                printf("\nThe result is: %g", Value1-Value2);
                break;
            case 4:
                printf("\nThe result is: %g", Value1+Value2);
                break;
            case 5:
                printf("\nThank you");
                break;
            default:
                printf("\nInvalid choice!");
        }
    }while (Choice != 5);
}
```

Output

Enter value 1: **40**

Enter value 2: **30**

1. Multiply
2. Divide
3. Subtract
4. Add
5. Quit

Enter your choice [1-4]: **1**

The result is 1200

Enter value 1: **40**

Enter value 2: **30**

1. Multiply
2. Divide
3. Subtract
4. Add
5. Quit

Enter your choice [1-4]: **3**

The result is 10

Enter value 1: **40**

Enter value 2: **30**

1. Multiply
2. Divide
3. Subtract
4. Add
5. Quit

Enter your choice [1-4]: **5**

Thank you.

Program 4-4

```
#include <stdio.h>
#include <conio.h>

#define Code 999

void main(void)
{
    int Password, Valid, Try=1;

    do{
        clrscr( );
        gotoxy(8,10);
        printf("Enter password: ");
        scanf("%i", &Password);
        if(Password != 999)
        {
            printf("Invalid!");
            Valid = 0;
            Try += 1;
        }
        else
            Valid = 1;
    } while (!Valid && Try <= 3);
    if(Try > 3)
        printf("\n\n\tMaximum 3 try only!");
    else
        printf("\n\n\tAccess approved.");
}
```

Output

Enter password: 111

Invalid!

Enter password: 222

Invalid!

Enter password: 333

Invalid!

Maximum 3 try only!

Program 4-5

```
#include <stdio.h>

void main(void)
{ float price;
  float total = 0;
  short quantity;

  printf("Enter 0 0 to quit.\n");
  printf("Enter 'price quantity': ");
  scanf("%f %hi", &price, &quantity);
  while (price != 0)
  { printf("The total for this item is $%6.2f.\n", price * quantity);
    total += price * quantity;
    printf("Enter 'price quantity': ");
    scanf("%f %hi", &price, &quantity);
  }
  printf("Your total is $%6.2f.\n", total);
}
```

Output

Enter 0 0 to quit.

Enter 'price quantity': **6.35 8**

The total for this item is \$50.80.

Enter 'price quantity': **2.50 10**

The total for this item is \$25.00.

Enter 'price quantity': **0 0**

Your total is \$75.80.

Program 4-6

```
#include <stdio.h>

void main(void)
{ float price;
  float total = 0;
  short quantity, items = 0;

  printf("Enter 0 0 to quit.\n");
  printf("Enter 'price quantity': ");
  scanf("%f %hi", &price, &quantity);
  while (price != 0)
  { printf("The total for this item is $%6.2f.\n", price * quantity);
    total += price * quantity;
    items += 1;
    printf("Enter 'price quantity': ");
    scanf("%f %hi", &price, &quantity);
  }
  printf("Your total is $%6.2f for %hi different items.\n", total, items);
}
```

Output

```
Enter 0 0 to quit.
Enter 'price quantity': 22.95 3
The total for this item is $68.85.
Enter 'price quantity': 7.29 8
The total for this item is $58.32.
Enter 'price quantity': 15 4
The total for this item is $60.00.
Enter 'price quantity': 0 0
Your total is $187.17 for 3 different items.
```

Program 4-7

```
#include <stdio.h>

void main(void)
{ int count;

    for (count = 1; count <= 3; count += 1)
        printf("%i\n", count);
        printf("Finished, but why is the count %i?\n", count);
}
```

Output

```
1
2
3
Finished, but why is the count 4?
```

Program 4-8

```
#include <stdio.h>

void main(void)
{ int n, count, factorial;

    printf("Enter a positive integer: ");
    scanf("%i", &n);
    printf("Integer Factorial\n", n);
    for ( ; n >= 1; --n)
    { printf("  %3i ", n);
      factorial = 1;
      for (count = 1; count <= n; ++count)
        factorial *= count;
      printf("%i\n", factorial);
    }
}
```

Output

Enter a positive integer: **5**

Integer	Factorial
5	120
4	24
3	6
2	2
1	1

4.7 Programming Exercise

1. Write a program to assign a letter grade given a numeric score: 90 or above is an A; 80, B; 70, C; 60, D; and below 60, F. The program should continue to accept values until a negative number is input. The program should print how many of each letter grade were assigned after the input is completed. Use else if construct in your program.

Variables

score	(score input)
a_s, b_s, c_s, d_s, f_s	(counters for letter grades)

Output

SCORE? 92

THE GRADE IS A

:

SCORE? -1

2	A's
2	B's
1	C's
0	D's
1	F's

2. Write a program to create a multiplication table for all combinations of two numbers from 1 to 8.

Variables

Multiplier

Multiplicand

Output

	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	2	4	6	8	10	12	14	16
3	3	6	9	12	15	18	21	24
4	4	8	12	16	20	24	28	32
5	5	10	15	20	25	30	35	40
6	6	12	18	24	30	36	42	48
7	7	14	21	28	35	42	49	56
8	8	16	24	32	40	48	56	64

3. Write a program that iterates through the odd numbers less than 30, and outputs the square of each number.

Variable

x

Output

1 9 25 49 81 121 169 225 289 361 441 529 625 729 841

4. Create a program that loops 30 times, but only outputs numbers that are not divisible by 3 or 5. Decide on the most appropriate form of loop, and use an *if* statement inside it.

Variable

x

Output

3 5 9 10 12 15 18 21 24 25 27 30

5. Write C codes to accept two integer values and display the multiplication result on the screen. Your program should iterate until user enter two negative values.

Variable

x, y

Output

Enter two integer values: **10 2**

The multiplication result is 20

Enter two integer values: **10 -2**

The multiplication result is -20

Enter two integer values: **-10 -2**

End of run.

6. Write a C program that produces the following patterns using 2 *for* loops.

Variables

x, y

Output

```
1  1
2  2  1
3  3  2  1
4  4  3  2  1
5  5  4  3  2  1
6  6  5  4  3  2  1
7  7  6  5  4  3  2  1
```

7. Write a program to produce the following output. Use nested for loops.

Variables

x, y

Output

```
1
1  2
1  2  3
1  2  3  4
1  2  3  4
1  2  3
1  2
1
```

4.8 Revision Exercise

1. Rewrite these *while* loops using the *for* statements.

<i>x = 14;</i>	<i>y = 65;</i>
<i>While (x >= 3)</i>	<i>while (y <= 85)</i>
<i>{</i>	<i>{</i>
<i>printf("%i\n", x);</i>	<i>printf("%i\n", y);</i>
<i>x -= 5;</i>	<i>y += 5;</i>
<i>}</i>	<i>}</i>

2. Rewrite these *for* statements using *while* loops.

```
for (x = 250; x >= 100; x -= 50)  
    printf("%i\n", x);  
  
for (y = 1226; y <= 1426; y += 2)  
    printf("%i\n", y);
```

3. What will be the output be from the following program segment?

```
for (a = 1; a <= 5; ++a)  
{  
    printf("%i", a);  
    for (b = a; b >= 1; --b)  
        printf("%i", b);  
    printf("\n");  
}  
printf("%i %i\n", a, b);
```

4. Which of the following code segment is functionally equivalent to the statement below?

*y += a * c++;*

- a. *y += a * ++c;*
- b. *y = (y + a) * c; c++;*

- c. $y = y + a * c; c++;$
- d. $c++; y = y + a * c;$
- e. none of the above

5. Use the following code to answer questions (i) to (x).

```
int t = 0; b = 0; s = 0; z = 0;
scanf("%i", &t);
{
    s += t;
    ++z;
    scanf("%i", &t);
}
b = s / z;
```

- i. Which of the variable is called an accumulator variable?
- ii. Which of the variable is called a counter variable?
- iii. Which of the variable is tested against a sentinel?
- iv. Is it necessary to initialize *t* to zero in the code?
- v. -1 is known as what?
- vi. Is it necessary to initialize *z* to zero in the code?
- vii. Is it necessary to initialize *b* to zero in the code?
- viii. Is it necessary to initialize *s* to zero in the code?
- ix. All **for** loop can be rewritten as a **while** loop.
- x. All **for** loop counter variables must be incremented/decremented with either ++ or --.

6. How does accumulation work?

7. Why is initialization of an accumulator variable important?

8. How does counting differ from accumulation?

9. What five elements are necessary in a counter-controlled loop?

10. Rewrite the following **for** statements using **while** loop.

```
for (x = 20; x >= 100; x -= 50)
    printf("%d\n", x);
```

11. State the output produce by the following C codes.

```
int x = 1;
for (x=11, x >= 1; x -= 3)
    printf("%4i", x * x);
```

12. Given the following C program, indicate the output produce.

```
void main( void)

{ int i = 0, x = 0;

  for (i = 1; i < 20; i *= 2)
  {
    x++;
    printf("%4i ", x);
  }
  printf("\nx = %i", x);
}
```

13. Rewrite the following *while* loops using *for* statement.

```
x = 14;
while (x >= 3)
{
    printf("%d", x);
    x -= 5;
}
```

```
x = 65;
while (y <= 85)
{
    printf("%d", y);
    x += 5; }
```

14. Answer questions (i) through (iii) by using the following C codes.

```
#include <stdio.h>  
main()  
{  
    int a, b;  
    for (a = 1; a < 3; ++a)  
        for (b = 1; b <= 3; b++)  
            printf("%4d", a+b);  
}
```

- i. Indicate what would be the final value stored in variable *a*?
- ii. Indicate what would be the final value stored in variable *b*?
- iii. Indicate the output produce by the program.

Chapter 5 – Function

Objectives

Functions in C enable program segments to be grouped as modules. The introduction of function simplifies and organizes C program's structure. There are some reasons and advantages of using functions in C. Functions would enable (i) modularity to be implemented easily, (ii) increase program's readability by grouping of some similar tasks, (iii) improve debugability where errors are easily identified and isolated, (iv) allows repeatability where some segment of codes could be repeated without repeating retyping, and (v) enables reusability where new programs are written by using as much code as possible from old programs. At the end of this chapter, students would have learned what is function and how to include functions in program as follow.

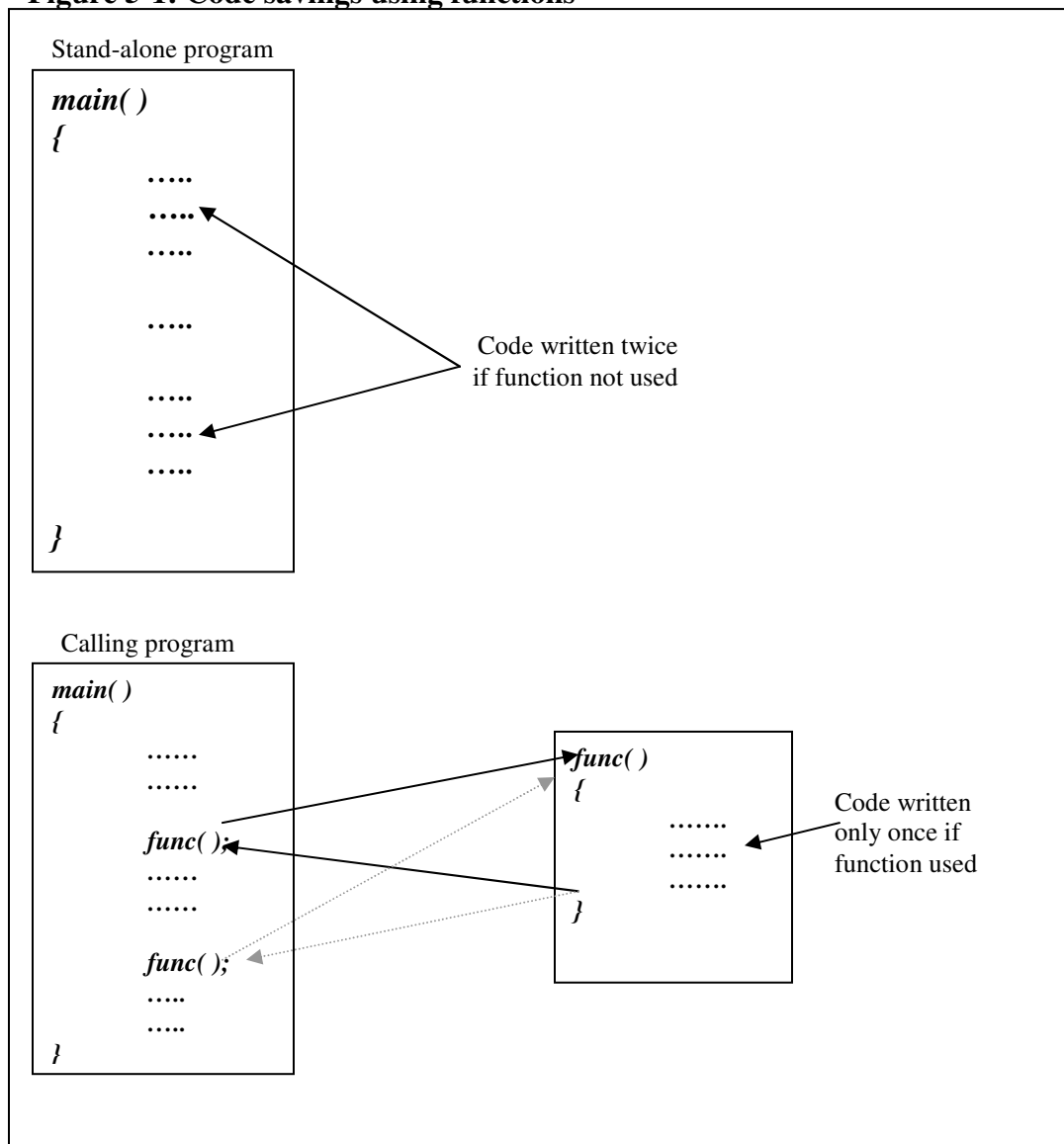
- Understand the concept of functions in C
- Learn and include function prototype and function definition in C program
- Know what is global/external and local variables, when to include them in C program
- Appreciate the purpose of parameter passing and how it works in C program
- Learn and understand how to incorporate functions in program for validation check

5.1 Why Functions are used in C?

Avoiding unnecessary repetition of code

Suppose you have a section of code in your program that calculates the square root of a number, you don't want to have to write the same instructions all over again. Instead, in effect, you want to jump to the section of code that calculates square roots and then jump back again to the normal program flow when you are done. In this case, a single section of code can be used many times in the same program. The saving of code is illustrated in **Figure 5-1**.

Figure 5-1: Code savings using functions



Program organization

By using the idea of subroutine, make it easier to organize programs and keep track of what they were doing. If the operation of the program could be divided into separate activities and each activity placed in a separate subroutine, each subroutine could be written and checked out more or less independently. Separating the code into modular functions also made programs easier to design and understand.

Independence

By making subroutines as independent from the main program and from one another possible. For instance, subroutines were invented that had their own “private” – that is, variables that couldn’t need to worry about accidentally using the same variable names in different subroutines; the variables in each subroutine were protected from inadvertent tampering by other subroutines. Thus it was easier to write large and complex programs.

5.2 The Structure of Functions

There are three program elements involved in using a function: the function definition, the call to the function, and the function prototype.

5.2.1 The Function Definition

The function itself is referred to as function definition. The definition starts with a line that includes the function name, among other elements.

Example:

```
void func(void ) ← Function definition, note that  
                    there is NO semicolon at the end.  
{  
    statement;  
    statement;  
    ....  
    ....  
}
```

Explanation:

Take note that the declarator doesn't end with a semicolon. It is not a program statement; it tells the compiler that a function is being defined. The function definition continues with the body of the function: the program statements that do the work. The body of the function is enclosed in braces. In the above example, the *func* is a user-defined function name.

5.2.2 Calling the Function

As with the C library functions we have seen, such as *printf()* and *getche()*, our user-written function *func()* is called from *main()* simply by using its name, including the parentheses following the name. The parentheses let the compiler know that you are referring to a function and not to a variable or something else. Calling a function like this is a C statement, so it ends with a semicolon.

Example:

```
void main( )  
{  
    .....  
    func( );  
    .....  
    ....  
}
```

Function call, note that
there is a semicolon at
the end.

5.2.3 Function Prototype (Declaration)

The function prototype (declaration) is a third function related element. The function prototype looks very much like the declarator line at the start of the function definition, except that it ends with a semicolon. A function is declared in a similar way at the beginning of a program before it is called. The function declaration tells the compiler the name of the function; the data type the function returns and data types of the function's arguments. In our example, the function returns nothing and takes no arguments; hence there are two voids.

Example:

```
void func(void );
```

Function prototype, note that there is semicolon at the end.

The key thing to remember about the prototype is that the data type of the return value must agree with that of the declarator in the function definition, and the number of arguments and their data types must agree with those in the function definition.

A function declares a function

A function call executes a function

A function definition is the function itself

5.2.4 Local Variables

We could declare variables in our function, these variables are known as local variables, whereby they are only known within the function, and other functions are not allowed to access them. A local variable will be visible to the function it is defined in, but not to others. A local variable used in this way is known as automatic variable, because it is automatically created when a function is called and destroyed when the function returns. The length of time a variable lasts is called lifetime.

Example:

```
void func(void)
```

```
{
```

```
    int x, y;
```

```
    .....
```

```
    .....
```

```
}
```

Local variables which are visible within *func* only

5.2.5 Functions that Return a Value

A function that uses no arguments but returns a value is a slightly more complicated kind of function: one that returns a value. When the function is called, it gets a certain piece of information and returns it to you. The function *getche()* operates in just this way, when it is called – without giving it any information – and it returns the value of the first character typed on the keyboard.

Example:

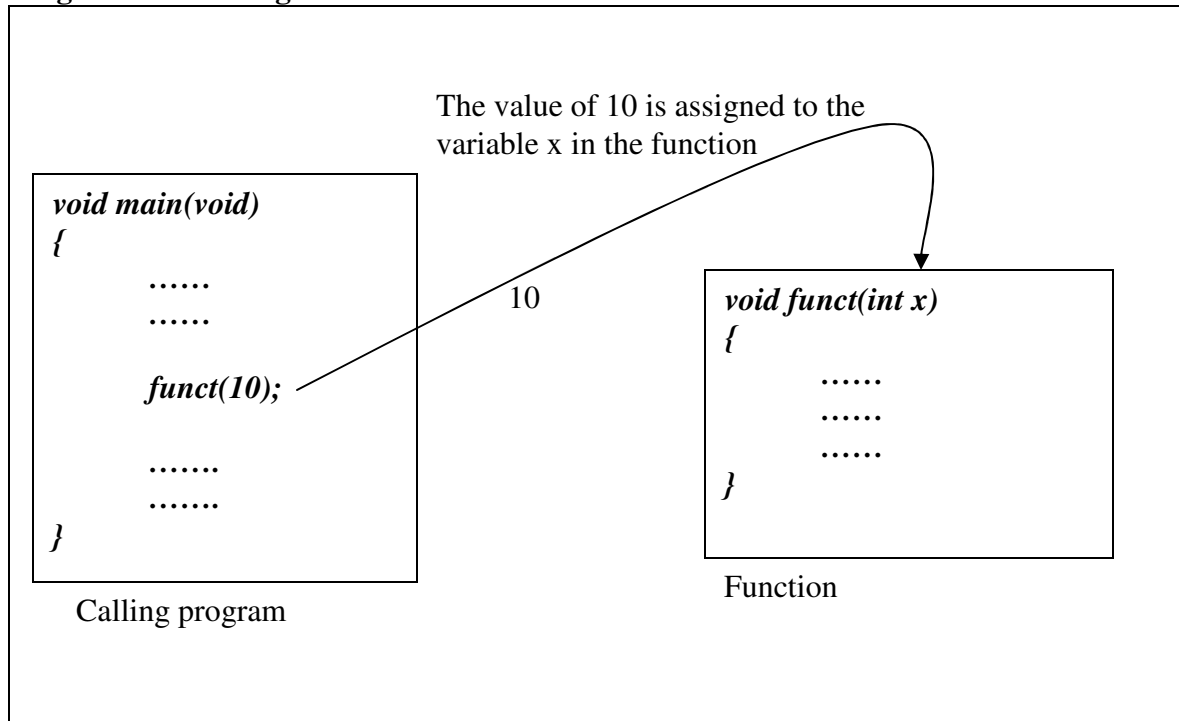
```
int funct(void);  
void main(void)  
{   int y;  
    ....  
    ....  
    y = funct( );  
    ....  
}  
  
int funct(void)  
{   int x;  
    ...  
    ...  
    return (x);  
}
```

The return statement has two purposes. First, executing it immediately transfers control from the function back to the calling program. Second, whatever is inside the parentheses following *return* is returned as a value to the calling program. The *return* statement need not be at the end of the function; as soon as it's encountered, control will return to the calling program. The *return* statement can also be used without a value following it, and no value would be returned to the calling program. The *return* statement can only be used to return a single value.

5.2.6 Using Arguments to Pass Data to a Function

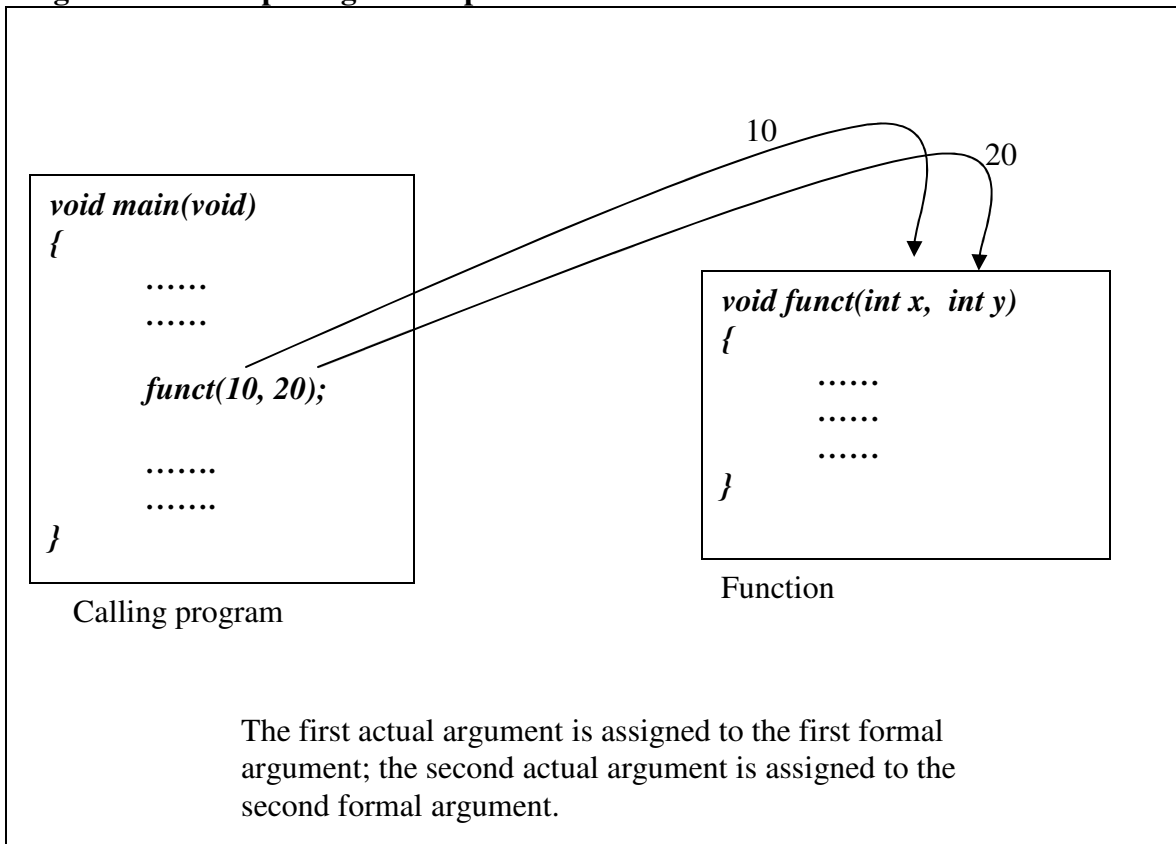
The mechanism used to convey information to a function is the argument. In the function definition, a variable name could be placed in the parentheses. This ensures that the value included between parentheses in the main program is assigned to the variable between parentheses in the function definition. This is shown schematically in **Figure 5-2**.

Figure 5-2: Passing a value to a function



5.2.7 Passing Multiple Arguments

We can pass as many arguments as we like to function. The process of passing two arguments is similar to one. The value of the first actual argument in the calling program is assigned to the first formal argument in the function, and the value of the second actual argument is assigned to the second formal argument, as shown in **Figure 5-3**.

Figure 5-3: Multiple arguments passed to function

5.2.8 External Variables

Sometimes, it is desirable to use a variable known to all the functions in a program, rather than just one. This is true when many different functions must read or modify a variable, making it clumsy or impractical to communicate the value of the variable from function to function using arguments and return values. In this case, we use an external variable, or sometimes also known as global variable. The use of external variables should be limited, due to some reasons such as, first, external variables are not protected from accidental alteration by functions that have no business modifying them. Second, external variables use memory less efficiently than local variables.

Example:

```
#include <stdio.h>
```

```
void func1(void);
```

```
void func2(void);
```

```
int x, y;
```

External variables that could be
accessed in func1 and func2.

```
void main(void)
```

```
{
```

```
    .....
```

```
    .....
```

```
}
```

5.3 Sample Program

Program 5-1

```
#include <stdio.h>

void info(void);

void main(void)
{ double gross, tax;
  int dependents;

  printf("Dependents (including employee)? ");
  scanf("%i", &dependents);
  info( );
  printf("Write the check!\n");
}

void info(void)
{ int emp_id, emp_num, dept;

  printf("Employee ID? ");
  scanf("%i", &emp_id);
  emp_num = emp_id / 10;
  printf("Employee #%i in the ", emp_num);
  dept = emp_id % 10;
  switch (dept)
  { case 1:
    printf("Information Systems");
    break;
    case 2:
    printf("Accounting");
    break;
    default:
    printf("Manufacturing or other");
  }
  printf(" department.\n");
}
```


Program 5-2

```
#include <stdio.h>
```

```
void info(void);
double gross_pay(void);
void main(void)
{ double gross, tax;
  int dependents;

  printf("Dependents (including employee)? ");
  scanf("%i", &dependents);
  info();
  gross = gross_pay();
  printf("Write the check!\n");
}
```

```
double gross_pay(void)
{ double hours, rate, gross;

  printf("Hours? ");
  scanf("%lf", &hours);
  printf("Rate? ");
  scanf("%lf", &rate);
  gross = hours * rate;
  printf("Gross pay: %7.2f\n", gross);
  return gross;
}
```

```
void info(void)
{ int emp_id, emp_num, dept;

  printf("Employee ID? ");
  scanf("%i", &emp_id);
  emp_num = emp_id / 10;
  printf("Employee #%i in the ", emp_num);
  dept = emp_id % 10;
  switch (dept)
  { case 1:
    printf("Information Systems");
    break;
    case 2:
    printf("Accounting");
    break;
    default:
    printf("Manufacturing or other");
  }
  printf(" department.\n");
}
```

```
}
```

Program 5-3

```
#include <stdio.h>
```

```
void Sub(void); //function prototype/declaration
```

```
void main(void)
```

```
{
```

```
    printf("\nThis message was displayed in the Main function");
```

```
    Sub( );
```

```
    printf("\nBack to Main function again");
```

```
}
```

```
void Sub(void) //function definition
```

```
{
```

```
    printf("\n Hi this message is in the Sub function");
```

```
}
```

Program 5-4

```
#include <stdio.h>
#include <conio.h>

float Value1, Value2; // global variables (for all the functions to access)

void Display(void); // function prototype/declaration
void Enter(void);
float Calculate(float, float);

void main(void)
{
    float Result; // local variable (access within main( ) only)

    Display( );
    Enter( );
    Result = Calculate(Value1, Value2);
    printf("\n\tThe addition result is: %g", Result);
}

void Display(void) // function definition
{
    gotoxy(8,10);
    printf("Enter 1st value: ");
    gotoxy(8,11);
    printf("Enter 2nd value: ");
}

void Enter(void) // function definition
{
    gotoxy(32,10);
    scanf("%f", &Value1);
    gotoxy(32,11);
    scanf("%f", &Value2);
}

float Calculate(float x, float y) // definition
{
    return x + y;
}
```

Program 5-5

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

char Gender, Mode; // global variables (for all the functions to access)

int Validate(char, char, char);
void enterGender(void);
void enterMode(void);

void main(void)
{
    enterGender( );
    enterMode( );
    printf("\n\tBye!");
}

void enterGender(void)
{
    int Okay;

    do{
        gotoxy(10,10);
        printf("Enter gender [F/M]: ");
        gotoxy(35,10);
        scanf(" %c", &Gender);
        Okay = Validate(Gender, 'F', 'M');
        if (!Okay)
        {
            gotoxy(10, 15);
            printf("Invalid!");
            getch( );
            gotoxy(10, 15);
            clrhol( ); // clear end of line at col 10, row 15
        }
    }while (!Okay);
}

void enterMode(void)
{
    int Okay;

    do{
        gotoxy(10,11);
```

```
printf("Enter payment mode [C/N]: ");
gotoxy(35,11);
scanf(" %c", &Mode);
Okay = Validate(Mode, 'C', 'N');
if (!Okay)
{
        gotoxy(10, 15);
        printf("Invalid!");
        getch();
        gotoxy(10, 15);
        clrhol();    // clear end of line at column 10, row 15
    }
}while (!Okay);
}

int Validate(char Field, char x, char y)
{
    if (toupper(Field) != toupper(x) && toupper(Field) != toupper(y))
        return 0;
    else
        return 1;
}
```

5.4 Programming Exercise

1. Create a function whose job is to print a page heading. It should print the next page number passed to it. Use the following driver – Main() function segment – to test your function.

Driver

```
void main(void)
{
    int p;
    for (p = 1; p <= 5; ++p)
        page(p);
}
```

Function and variable

```
page( )
    Page_no    (int)
```

Output

```
Major document    Page 1
Major document    Page 2
```

2. Write a program that accepts any number from the keyboard and tells you whether it is a nonnegative integer. The number should be sent to the function *int_test()*, which returns either the integer value, or -1 if the number is negative, or zero if it is nonnegative but not an integer. Inputs should continue until zero is input.

Functions and variables

```
main( )
    input          (from keyboard)
    integer        (return from the function)

int_test( )
    value          (from main( ) function)
    result         (value to return)
```

Output

```
Your number: 48
The number is 48,
Your number: -14.3
The number is negative
Your number: 12.562
The number is not an integer
Your number: 0
```

3. Write a program to make change in coins. The *main()* function should accept input of the purchase and amount tendered, and the *change()* function should print the number of dollars, fifty cents, twenty cents, ten cents and one cent.

Functions and variables

```
main( )
    purchase
    tendered
change(amount)
    cents --→ convert from the float amount to the int cents.
```

Output

Purchase: **2.50**

Amount tendered: **4**

Dollars: 1

Fifty cents: 5

Twenty cents: 0

Five cents: 0

One cents: 0

4. Write a simple calculator so that you can input an expression with two values separated by an operator and the computer will print out the proper result. Your calculator should include the ^operator for exponentiation. All calculations should be done in an appropriate function (the function will be small) and the results printed in *main()*.

Functions and Variables

main()

op ---→ operator

x, y ---→ Values for calculation

add()

subtract()

multiply()

divide()

exponentiation()

a, b ----→ Local variables for each function return

Output

Enter expression (q to quit): **50 + 45.20**

95.20

5.5 Revision Exercise

1. What is a function definition?
2. What does a function call do?
3. Describe the actions of the **return** statement.
4. What does the return type or declaration **void** mean?
5. Differentiate between the terms “lifetime” and “visibility”.
6. Differentiate between the terms “local” and “global”.
7. Differentiate between the terms “internal” and “external”.
8. State the output produce by the following C program.

```
#include <stdio.h>
int funct (int);

void main(void)
{
    int x;
    for (x = 1; x < 10; ++x)
        printf("%4i", funct(x));
}
int funct(int y)
{
    return y * y;
}
```

9. The following program accepts two numeric values and displays the multiplication result on the screen. *Calculate()* is a user-defined function that performs multiplication and returns the result to main program body. Answer questions (i) through (v) by using the following C program

```
#include <stdio.h>

void main(void)
{
    int v1, v2, v3;
    printf("Enter value 1 and value 2 ");
    scanf("%i %i", &v1, &v2);
    v3 = Calculate(v1, v2);
}
```

```

    printf("Result = %i ", v3);
}

```

- i. There are how many argument(s) in *Calculate*()?
- ii. What is the appropriate data type for the argument(s) in *Calculate*()?
- iii. Write out the function declaration for *Calculate*().
- iv. Write a complete function definition for the *Calculate*() function.
- v. Suppose the values entered for *v1* and *v2* are 10 and 20 respectively, the multiplication result would be 20. State the variable that holds this result in the main program body.

10. Answer questions (i) to (v) by using the following C program segment

```

#include <stdio.h>
int funct (int);

void main(void)
{
    int a, x;
    for (x = 1; x < 10; ++x)
    {
        a = funct(x);
        printf("%4i", a);
    }
}

int funct(int y)
{
    return y * y;
}

```

- i. There is one argument found in the user-defined function – *funct*. (TURE/ FALSE)
- ii. The first value stored in variable a is 1. (TURE/ FALSE)
- iii. The user-defined function – *funct* will iterate for 10 times. (TURE/ FALSE)
- iv. The statement *++x* could also be written as *x = x + 2*. (TURE/ FALSE)
- v. The statement *%4i* would display 4 zeros in front of each number displayed on the screen. (TURE/ FALSE)

11. Use the following C code to answer questions (i) through (x).

```
double F(float x, unsigned n)
{
    int i = n;
    double t = 1;
    for (; i >= 1; --i)
        t *= x;
    return t;
}
```

- i. The above function has ____ argument(s).
- ii. If *n* is 2 and *x* is 3.0, what would get returned by the above function?
- iii. How many times does the loop in the above code executed?
- iv. The number that *F* returns is of type _____.
- v. If *n* is 1, then the function always return *x* (TRUE/FALSE).
- vi. If *x* is -2.0 and *n* is 2, *F* would return _____.
- vii. The variable(s) ____ is/are local to function *F*.
- viii. *F* computes the factorial of *x*(TRUE/FALSE).
- ix. *F* computes the same thing as the *pow()* function (TRUE/FALSE).
- x. If *n* is -1 and *x* is 10, what would be returned by *t*?

12. Choose the best answer for each of the situation described below function definitions.

- i. A function called sample generates and returns an integer quantity.
 - a. void sample(void)
 - b. char sample(int quantity)
 - c. int sample(void)

- ii. A function called `root` accepts two integer arguments and returns a floating-point result.
 - a. `float root(int a, b)`
 - b. `float root(int a, int b)`
 - c. `void root(void)`

- iii. A function called `convert` accepts a character and returns another character.
 - a. `void convert(char a)`
 - b. `int convert(char)`
 - c. `char convert(char a)`

- iv. A function called `process` accepts an integer and two floating-point quantities (in that order), and returns a double-precision quantity.
 - a. `double process(float a, float b, int c)`
 - b. `double process(float a, b, int c)`
 - c. `double process(int a, float a, float c)`

- v. A function called `value` accepts two double-precision quantities and a short-integer quantity (in that order). The input quantities are processed to yield a double-precision value which is displayed as a final result.
 - a. `void value(double a, double b, short i)`
 - b. `double value(double a, b, short i)`
 - c. `printf((double(double a, b, short i))`

Chapter 6 – Arrays

Objectives

Array is a consecutive storage location in computer's memory that stores similar type of data. The array implementation and application is important if a list of elements/items or records were to be manipulated. At the end of this chapter, students would have learned what is array and how to include array in program as follow.

- Understand the concept of arrays in C
- Learn how to declare single dimensional array of different types and store data into them
- How to manipulate the values stored in arrays
- Appreciate the purpose of array and array application such as Searching

6.1 Array Declaration

Array is a consecutive storage location in computer's memory that stores similar type of data. Array is something like a table or list that hold a group of items/elements of the same type. A name is normally assigned to an array and the required slots would be defined in the array's declaration. The following example shows how an array with 10 slots that stores only integer values are declared. Additionally, array could be initialized to some values prior to operations or manipulation as below.

Array Declaration

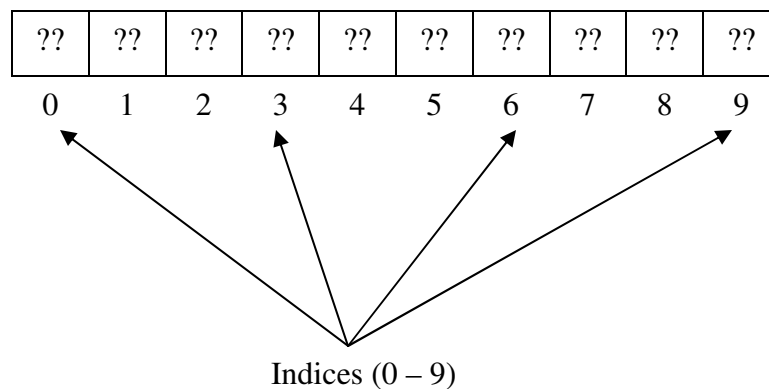
```
int SalesArray[10];
```

Array Initialization

```
int SalesArray[10] = {0};
```

Once an array is declared, individual slot could be used for storage and these slots are referenced via index. The first slot starts at zero for example if an array contain ten slots the last slot's index would be nine. The following **Figure 6-1** represents the organization of array and the indices.

Figure 6-1: Array declared with 10 slots without any initialization



Each slot in the array could be referred by specifying the array name and the index. For example, the statement ***SalesArray[2]*** would refer to third slot's stored value (if any).

6.2 The Variably Defined Array

In some situations, it is important for us to access different elements, essentially different variables, by changing the index. The advantage of using variably defined variable in the array index is, it would reduce/simplifies the array operations. Consider an array with hundred elements, and if the value of each element were to be displayed/printed, individual access would require the same /statement code to be repeated hundred times in a program. This would not happen if the index were declared variably. Consider the following example of codes for displaying all the elements in the array namely *SalesArray*.

Without variably defined variables

```
printf("%i",SalesArray[0]);  
printf("%i",SalesArray[1]);  
.....  
.....  
  
printf("%i",SalesArray[98]);  
printf("%i",SalesArray[99]);
```

With variably defined variables

```
for (idx = 0; idx <= 99; ++idx)  
    printf("%i",SalesArray[idx]);
```

6.3 Sample Program

Program 6-1

```
#include <stdio.h>

#define DAYS_IN_WEEK 6

void main(void)
{ short sales[DAYS_IN_WEEK] = {3806, 28, 4522, 1183, 47, 12};
  short day;

  for (day = 1; day <= DAYS_IN_WEEK; ++day)
    printf("Sales for day %hi =%5hi\n", day, sales[day - 1]);
}
```

Output

```
Sales for day 1 = 3806
Sales for day 2 =  28
Sales for day 3 = 4522
Sales for day 4 = 1183
Sales for day 5 =  47
Sales for day 6 =  12
```


Program 6-2

```

#include <stdio.h>

#define FLEET 5

int truck_in(void);

void main(void)
{ float weights[FLEET] = {0};
  int truck;
  int tot_trucks = 0;
  float tot_weight = 0;
  truck = truck_in();
  while (truck) { printf("Weight? ");
    scanf("%f", &weights[truck - 1]);
    truck = truck_in();
  }
  printf("\nShipping Report\n");
  printf("   Truck Weight\n");
  for (truck = 1; truck <= FLEET; ++truck)
    if (weights[truck - 1])
      { printf("   %5i %6.1f\n", truck, weights[truck - 1]);
        ++tot_trucks;
        tot_weight += weights[truck - 1];
      }
  printf("   ----- \n");
  printf("Total %5i %6.1f\n", tot_trucks, tot_weight);
}

int truck_in(void)
{ int truck;

  do
  { printf("Truck (1 to %i, 0 to quit)? ", FLEET);
    scanf("%i", &truck);
  } while (truck < 0 || truck > FLEET);
  return truck;
}

```

Output

Truck (1 to 5, 0 to quit)? **7**

Truck (1 to 5, 0 to quit)? **4**

Weight? **1825.8**

Truck (1 to 5, 0 to quit)? **72**

Weight? **883.5**

Truck (1 to 5, 0 to quit)? **1**

Weight? **829.4**

Truck (1 to 5, 0 to quit)? **0**

Shipping Report

	Truck	Weight
	1	829.4
	2	883.5
	3	1825.8
	-----	-----
Total	4	3538.7

Program 6-3

```
#include <stdio.h>
#define SIZE 5

void main(void)
{
    int List[5]={0}, idx;

    for (idx = 0; idx < SIZE; ++idx)
    {
        printf("\nEnter an integer value %i: ", idx);
        scanf("%i", &List[idx]);
    }
    printf("\nList");

    for (idx = 0; idx < SIZE; ++idx)
    {
        printf("\n%i", List[idx]);
    }
}
```

Output

Enter an integer value 1: **67**

Enter an integer value 2: **58**

Enter an integer value 3: **67**

Enter an integer value 4: **48**

Enter an integer value 5: **98**

List

67

58

67

48

98

Program 6-4

```
#include <stdio.h>
#define SIZE 5

void main(void)
{
    int List[5]={0}, idx, Target, Found;

    for (idx = 0; idx < SIZE; ++idx)
    {
        printf("\nEnter an integer value %i: ", idx);
        scanf("%i", &List[idx]);
    }

    printf("Enter an integer value to search: ");
    scanf("%i", &Target);

    idx = 0;
    do{
        if (Target == List[idx])
        {
            printf("\nTarget %i found at index %i", Target, idx);
            Found = 1;
        }
        else
            Found = 0;
        ++idx;
    } while (!Found && idx < SIZE);
    if (!Found)
        printf("\nNot found");
}
```

Output

Enter an integer value 1: **67**
Enter an integer value 2: **58**
Enter an integer value 3: **67**
Enter an integer value 4: **48**
Enter an integer value 5: **98**
Enter an integer value to search: **48**
Target 48 found at index 3.

Program 6-5

```

#include <stdio.h>
#define SIZE 5

void main(void)
{
    int List[5]={0}, idx ;

    for (idx = 0; idx < SIZE; ++idx)
    {
        printf("\nEnter an integer value %i: ", idx);
        scanf("%i", &List[idx]);
    }

    printf("\n\nData\tData * 2");
    printf("\n----\t-----");
    for (idx = 0; idx < SIZE; ++idx)
        printf("\n%i\t%i", List[idx], List[idx] * 2);
}

```

Output

Enter an integer value 1: **67**
 Enter an integer value 2: **58**
 Enter an integer value 3: **97**
 Enter an integer value 4: **48**
 Enter an integer value 5: **98**

Data	Data * 2
-----	-----
67	134
58	116
97	194
48	96
98	196

6.4 Programming Exercise

1. Write a program to input a set of five numbers and print them out in the reverse order of input. Put both the input and printing in loops.

Variables

a[]

x

Output

Input five numbers:

0? **235**

1? **256.23**

2? **458**

3? **265.12**

4? **5**

Here they are: 5 265.12 458 256.23 235

2. Write a program that accepts input of five values from the keyboard and prints out the values and their difference from the mean (average) value.

Variables

Value[]

Mean

Count

Output

Enter 5 values separated by space.

46.2 12.6 32.654 6 25.44

Number	Value	Difference
1	46.20	21.62
2	12.60	-11.98
3	32.65	8.08
4	6.00	-18.58
5	25.44	0.86

3. Write a program to accept five integral values into an array namely *List*, at the end of data entry, the program will prompt user to enter a value to search from the array. Your program should display appropriate message notifying the user whether the search element is found in the array.

Variables

List[]

Target, counter

OutputEnter value[0]: **10**Enter value[1]: **20**

Enter value[2]: **35**

Enter value[3]: **12**

Enter value[4]: **8**

Enter a value to search from the array: **8**

8 was found at index 4

Enter a value to search from the array: **100**

100 was not found in the array

6.5 Revision Exercise

1. How is an indexed variable similar to any other variable?
2. How does an indexed variable differ from a normal variable?
3. What is an array?
4. Why do we use array?
5. What is the first index value for any indexed variable in C?
6. In the array declaration *float stuff[10]*; how many elements are allocated and what is the index of the last allocated variable?
7. Given the declaration *float stuff[5] = {1.1, 2.2, 3.3}*; what will be the value of *stuff[1]*? Of *stuff[3]*? Of *stuff[5]*?
8. Of what data type is an index for question 7?
9. Answer questions (i) through (v) regarding an array called *fractions*.
 - i. Define a symbolic constant SIZE to be replaced with the replacement text 10.
 - ii. Declare an array fractions with SIZE elements of type float and initialize the elements to 0.
 - iii. Name the fourth element from the beginning of the array.
 - iv. Name the last element from the beginning of the array.
 - v. Name the first element from the beginning of the array.
10. Given the following array's declaration statements, indicate whether valid or invalid.
 - i. `char a[10] = "Alice";`
 - ii. `char a[] = "Alice";`
 - iii. `char a[10] = 'A', 'l', 'i', 'c', 'e';`
 - iv. `char a[10] = {0};`
 - v. `char a[5] = "Alice";`

11. State whether the following statements are TRUE or FALSE.

- i. C checks for subscripts that are out of range of an array's declared size.
- ii. Negative subscripts are allowed in C.
- iii. The end of an array can easily be tracked through the use of a sentinel value.
- iv. Unlike other variables, arrays must be initialized when they are declared.
- v. When using subscripted variables, the subscript can be any non-negative integral values.
- vi. Individual elements of an array are stored contiguously.

12. Fill in the blanks for the following statements.

- i. We refer to a set of subscripted variables as a(n) _____.
- ii. Given the declaration *float y[100]*; the highest value subscript for *y* would be _____.
- iii. Given the declaration *float x[80]*; the lowest value subscript for *x* would be _____.
- iv. A(n) _____ arranges a list of values in order.
- v. A(n) _____ operation begins with a list of values in order (sorted), and adds one more to the list.

13. Choose the best answer for the following statements.

- i. The process of placing the elements of an array into either ascending or descending order is called.
 - a. Sorting
 - b. arranging
 - c. placing

- ii. When referring to an array, the position number contained within parenthesis is called a _____.
 - a. number
 - b. value
 - c. subscript
- iii. The elements of an array are related by the fact that they _____.
 - a. are integers
 - b. contain values
 - c. have same name and type

Chapter 7 – String

Objectives

String in C is a list of alphanumeric characters declared as a character array. In chapter 7, students will learn how to incorporate string variables in C programs. Additionally, students would learn how to manipulate string variables such as character conversion, format string output, and validation check of string characters. At the end of this lesson, students would be able to achieve the following objectives.

- Understand the concept of string variables in C programs
- Learn how to incorporate string variables in C programs
- Know how to manipulate string variables such as character conversion, formatting, and validation check.
- Know and appreciate when should string variables be included and the purpose of using string variables

7.1 String Variables

A string variable in C is simply a list of characters declared as array. When declaring string variables, sufficient storage space should be allocated. The string variables' storage space normally requires a NULL ('\0') character at the end of it. C will automatically assign a NULL character at the end of a string variable. The following examples show how string variables are declared and being initialized.

Example – Declaration:

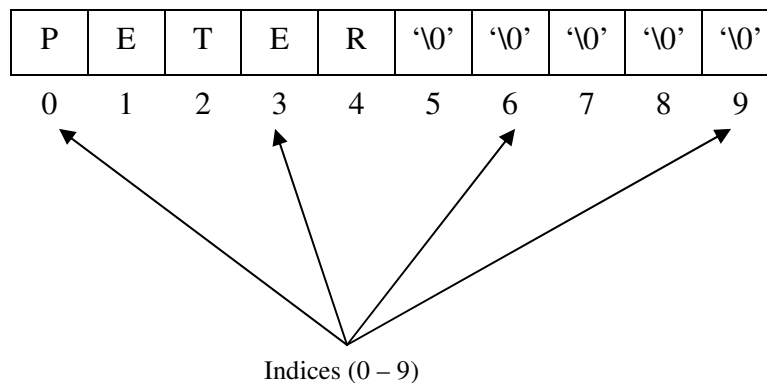
```
char Name[10];
```

Example – Initialization:

```
char Name[10] = { 'P', 'E', 'T', 'E', 'R'};  
char Name[10] = "Peter";  
char Name[ ] = "Peter";
```

Once a string variable is declared, characters could be stored in the allocated storage space. Each character would be accessed/referenced via index. The first character starts at zero and the last character is always a NULL character. The following **Figure 7-1** represents the organization of a string variable for the first initialization example above.

Figure 7-1: String variable initialization



7.2 String Input

A string variable could be used to store any ASCII characters. There are two ways of capturing value into string variable, the *scanf()* and *gets()* function. The main difference between the two functions is the *scanf()* function requires data specifier while the other does not.

scanf(“%s”, Name);

OR

gets(Name);

The *scanf()* function takes a set of characters from the input stream (from the current position to the next whitespace and write them beginning at the address name. It will also add a null at the end of the characters to maintain C’s concept of string.

If a character array (string variable) has 10 elements/slots for storage, we can effectively store only 9 characters. If user enter a 15-characters to the array, it overflows the space allocated and will mess up something else. We can limit the number of characters that will be converted by putting a width modifier in front of the s type code.

Example:

char product[9];

scanf(“%9s”, product);

Explanation:

The above *scanf()* example will stop conversion at the first whitespace or a maximum of 9 characters. If it reaches the maximum, characters left over remain in the stream. Whitespace is the default delimiter for *scanf()*, which means that inputting a single string with whitespace in it is not possible. If there might be two-word product names, such as *Blue Swan*, we would have to use something other than *scanf()*. In this case, we may use *gets()*.

The *gets()* function does not limit the number of characters assigned to the string. Typically, *scanf()* leaves newline in the input stream after finishes. When *gets()* executes and you are expecting the program to stop while you enter something. The *gets()* sees the newline, which it interprets as an empty string – perfectly acceptable to it – and it assigns

the empty string to the input field. It is important to be sure that the input buffer is empty before the call to *gets()*, this could be done through flushing the input stream as below:

Example:

```
char Name[30];  
:  
:  
fflush(stdin);  
gets(Name);
```

7.3 String Output

A string variable's content could be displayed by using either the *printf()* or *puts()* function. The *%s* conversion code outputs an entire string. Its argument must be an address of an array of *chars*. The content in a string variable could also be manipulated such as format the display to the required length or spaces, for example, by including the required width and precision.

```
printf ("%s", Name);  
printf ("%20s", Name);  
printf ("% -20s", Name); /* left justification */  
printf ("%20.10", Name); /* width = 20, no. of characters to print = 10
```

OR

```
puts(Name);
```

7.4 String Manipulation

There are various string functions available for manipulation purpose. Those functions are available in the *string.h* header file. The following are some commonly used string functions.

strlen() – it returns the number of characters in the string.

```
Char Name[40] = "Peter";  
printf("String size is: %i", strlen(Name));
```

strcpy() - copies a string from location to another.

```
char Destination[40];  
char Source[40] = "Peter";  
strcpy(Destination, Source);
```

strncpy() – enable maximum number of characters to be copied only.

```
char Destination[40];  
char Source[40] = "Peter";  
strncpy(Destination, Source, 20);
```

strcat() – concatenates strings, puts one string at the end of another.

```
char First[40] = "Peter";  
char Second[40] = "Pan";  
printf("%s", strcat(First, Second));
```


strcmp() – compares two strings and return a positive value if the first string is greater than the second string, zero if both strings are equal, or negative value if first string is less than second string.

```
char First[40] = "Peter";  
char Second[40] = "Pan";  
if( strcmp(First, Second) == 0)  
    printf("They are equal");
```

strncmp() – compares only up to maximum characters specified.

```
char First[40] = "Peter";  
char Second[40] = "Pan";  
if( strncmp(First, Second, 3) == 0)  
    printf("The first three characters are equal");
```

7.5 Character Classification

As mentioned earlier, string is merely collection of characters (character array). Each character stored in a string variable can be manipulated individually such as from lowercase to uppercase, etc. There are various character classification functions available in *ctype.h* header file.

isalnum() – returns nonzero if the character is alphanumeric: 0-9, A-Z, or a-z.

isalpha() – returns nonzero if the character is alphabetic: A-Z or a-z.

isctrl() – returns nonzero if the character is a control code: ASCII 1-31.

isdigit() – returns nonzero if the character is a decimal digit: 0-9.

isgraph() – returns nonzero if the character is printable, not including space.

islower() – returns nonzero if the character is lowercase: a-z.

isprint() - returns nonzero if the character is printable. Including space.

ispunct() - returns nonzero if the character is punctuation.

isspace() - returns nonzero if the character is whitespace: space, form feed (\f), newline (\n), return (\r), horizontal tab (\t), or vertical tab (\v).

isupper() - returns nonzero if the character is uppercase: A-Z.

isxdigit() - returns nonzero if the character is a hexadecimal digit: 0-9, A-F.

7.6 Sample Program

Program 7-1

```
#include <stdio.h>

#define LENGTH 10

void out_string(char string[]);

void main(void)
{ char product[LENGTH];
  float price;

  printf("Enter price and product: ");
  scanf("%f %s", &price, product);
  printf("The price of a ");
  out_string(product);
  printf(" is %.2f.\n", price);
}

void out_string(char string[])
{ int index = 0;

  while (string[index] != '\0')
  { printf("%c", string[index]);
    ++index;
  }
}
```

Output

Enter price and product: **12.98 widget**

The price of a widget is 12.98

Enter price and product: **12.98, widget**

The price of a , is 12.98

Program 7-2

```
#include <stdio.h>
#define LENGTH 10

void in_line(char line[], int max);
void out_string(char string[]);

void main(void)
{ char string[LENGTH];

    printf("Your input> ");
    in_line(string, LENGTH);
    while (string[0] != '\0')
    { printf("    Stored: ");
      out_string(string);
      printf("\nYour input> ");
      in_line(string, LENGTH);
    }
}

void in_line(char line[], int max)

{ int index = 0;

    line[index] = getchar();
    while (line[index] != '\n' && index < max - 1)
    { ++index;
      line[index] = getchar();
    }
    if (line[index] != '\n')
        while (getchar() != '\n');
    line[index] = '\0';
}

void out_string(char string[])
{ int index = 0;

    while (string[index] != '\0')
    { printf("%c", string[index]);
      ++index;
    }
}
```

Output

Your input> **Hi there**

Stored: Hi there

Your input> **How are you?**

Stored: How are

Program 7-3

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void main(void)
{
    char Name[40];
    int idx;

    printf("\nEnter your name [max 40]: ");
    fflush(stdin);
    gets(Name);
    Name[0] = toupper(Name[0]);
    for (idx = 1; Name[idx] != '\0'; ++idx)
    {
        if (isspace(Name[idx - 1]))
            Name[idx] = toupper(Name[idx]);
        else
            Name[idx] = tolower(Name[idx]);
    }
    printf("\nAfter conversion: %s", Name);
}
```

Output

Enter your name [max 40]: rosemary reis

After conversion: Rosemary Reis

Program 7-4

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

void main(void)
{
    char Name[40];
    int idx=0, strSize, Valid;

    printf("\nEnter your name [max 40]: ");
    fflush(stdin);
    gets(Name);
    strSize = strlen(Name);
    do{
        if (isalpha(Name[idx]) || isspace(Name[idx]))
            Valid = 1;
        else
            Valid = 0;
        ++idx;
    }while (Valid && idx < strSize);
    if(!Valid)
        printf("Name field contain invalid character!");
    else
        printf("Your name is: %s", Name);
}
```

Output

Enter your name [max 40]: rosemary 123
Name contain invalid character!

Enter your name [max 40]: rosemary reis
Your name is rosemary reis

Program 7-5

```

#include <stdio.h>
#include <string.h>
#include <ctype.h>

void main(void)
{
    char Text[20] = "C PROGRAMMING";

    printf("\n*%s*", Text);
    printf("\n*%20s*", Text);
    printf("\n*%-20s*", Text);
    printf("\n*%5.3s*", Text);
    printf("\n*%8.3s*", Text);
    printf("\n*%8.5s*", Text);
    printf("\n*%8.6s*", Text);
}

```

Output

```

* C      P R O G R A M M I N G *
*              C P R O G R A M M I N G *
* C      P R O G R A M M I N G      *
*      C      P *
*              C      P *
*              C      P R O *
*      C      P R O G *

```

Program 7-6

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{ int in;

  do
  { printf("Type a letter and <enter>: ");
    in = getchar( );
    getchar( );
  } while (!isalpha(in));
  puts("Finally, an alpha character!");
}
```

Output

Type a letter and <enter>: **5**

Type a letter and <enter>: **8**

Type a letter and <enter>: \

Type a letter and <enter>: **d**

Finally, an alpha character!

Program 7-7

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{ char string[ ] = "23 skidoo.";
  int chr = 0;

  while(string[chr] != '\0')
  { if (isalpha(string[chr]))
    printf("'%c' is alpha\n", string[chr]);
    else
    printf("'%c' is not alpha\n", string[chr]);
    ++chr;
  }
}
```

Output

```
'2' is not alpha
'3' is not alpha
' ' is not alpha
's' is alpha
'k' is alpha
'i' is alpha
'd' is alpha
'o' is alpha
'o' is alpha
'.' is not alpha
```


Program 7-8

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{ char string[] = "mYRnA H. bALthAZaR, III";
  int chr;

  printf("Before: %s\n", string);
  string[0] = toupper(string[0]);
  for (chr = 1; string[chr] != '\0'; ++chr)
    if (string[chr - 1] == ' ')
      string[chr] = toupper(string[chr]);
    else
      string[chr] = tolower(string[chr]);
  printf("After : %s\n", string);
}
```

Output

Before: mYRnA H. bALthAZar, III

After: Myrna H. Balthazar, Iii

7.7 Programming Exercise

1. Write a function to return a nonzero if the character is a vowel(a, e, i, o, u – upper or lower case), or zero if it is not a vowel. Test the function in a program that allows input of string – *gets()* and a single character output – *putchar()* of the characters in the string with the vowels highlighted as shown.

main()	isvowel()
string	character
pos (character position in string)	result

Output

Aloysius Washington

<A>l<o>ys<i><u>s W<a>sh<I>ngt<o>n

2. In the ASCII code, an uppercase letter has a value 32 less than the corresponding lowercase letter. Write a program that will allow you to input any string and print in all uppercase. The function *caps()* should make the actual changes in the string. Do not use any string function.

Functions and Variables

```
main( )

    string[ ]

caps( )
    string[ ], sub
```

Output

Your input? **How are you?**

The output: HOW ARE YOU?

Your input? **Please pay \$23.90**

The output: PLEASE PAY \$23.90

3. Write a program to compare two strings, the program should print out which string is greater or that they are equal.

Variables:

String1

String2

OutputsFIRST STRING? **ABNER**SECOND STRING? **CRUMP**

CRUMP IS GREATER THAN ABNER

FIRST STRING? **NORBERT**SECOND STRING? **Nancy**

Nancy IS GREATER THAN NORBERT

FIRST STRING? **MARY**SECOND STRING? **MARY**

MARY EQUALS MARY

4. Good people all have last name begin with the letter G through L; all the others are normal. Write a program that differentiates the good people from normal people.

Variable

Name

OutputsName? **Annie**

Annie is a normal person

Name? **Gary**

Gary is a good person

Name? **Ben**

Ben is a normal person

Name? **Lawrence**

Lawrence is a normal person

7.8 Revision Exercise

1. How can we establish our own delimiters using *scanf()*?
2. What two functions can we use to display strings?
3. What does the precision parameter do in *printf()*?
4. What function returns the number of characters in a string?
5. What two functions will copy characters from one location to another? How do they differ?
6. What two functions concatenate one string to another? How do they differ?
7. What two functions compare two strings? How do they differ? How are the comparisons made?
8. Fill in the blanks with most appropriate word(s).
 - i. A string value or literal is enclosed in _____.
 - ii. Each character in memory takes up _____ byte(s).
 - iii. A string is really just a (n) _____.
 - iv. The conversion code “%[^\#@]s” would tell scanf to _____.
9. State whether the following statements are TRUE or FALSE.
 - i. There is no difference between how C interprets ‘x’ and “x”.
 - ii. There is no difference between how C interprets ‘x’ and ‘X’.
 - iii. There is no difference between how C interprets ‘x’ and “X”.
 - iv. A null will be stored at the end of x, given the declaration – *char x[3] = {‘1’, ‘2’};*
 - v. The automatic declaration of x – *char x[10]* will automatically put zeros throughout x.
 - vi. A string variable control string can be sent to scanf as its first argument.

- vii. C issues an error if you assign characters beyond the memory locations allocated for an individual string.
- viii. The *strlen()* function returns the number of characters in its argument, and does not count the null.

Chapter 8 – File

Objectives

So far all the data captured is temporary held in computer's memory, and once power is off, the data lost too. In this chapter, students would learn how to keep and store data into a data file. There is various file manipulation techniques would be covered in this chapter. At the end of the chapter students would learn the following.

- Understand the concept of file in C
- Learn how to declare file variables and various types of file
- Know how to write data and retrieve data from a text file
- Learn how to delete data from a text file
- Create simple application program for file handling such as adding, searching, viewing and deleting of data

8.1 Opening Files

Opening a file in C means telling C that the type of file variable should be used to stored data. Prior to opening a file, a file variable must be declared. A file could be declared as *FILE* type and open using the *fopen()* function.

Declaring a File Variable

```
FILE *StudentFile;
```

Opening File

```
FILE *StudentFile;  
StudentFile = fopen("c:\\Data\\Student.Dat", "a+");
```

8.1.1 File Modes

In the Declaring File Variable example, the *StudentFile* is the file pointer, and it is a user-defined name. In the second example- Opening File, the *fopen()* function takes two arguments – the data file name and the file modes. There are three types of file modes available in C.

“r” – Opens an existing file with file position indicator at the beginning of the file.

“w” – Creates a new, empty file with the file position indicator at the beginning. If there is any data exist in the file, it would be erased.

“a” – Opens an existing file or, if the file does not exist, creates a new one. The file position indicator would be placed at the end.

All the above file modes could be issued in conjunction with a plus “+” symbol to indicate the file is for both writing and reading operations.

8.1.2 Binary versus Text Files

Test file try to match the internal storage of data in C to the typical formatting of text for the operating system being used. The possible adjustments are in the line endings and end-of-file indications.

Binary files are most assume nothing about the operating system and don't make any special adjustments or translations. What is in the program or main memory will be written to the file verbatim. The binary file mode can be declared by including a b in the mode string either the second or last position. For example, "rb+" or "rb", or "wb", and so forth.

8.2 Closing a File

The act of closing a file writes the contents of the current buffer, if it has changed, into the appropriate place in secondary memory storage, making changes permanent, and deallocations the memory space for both the file buffer and the file description. The *fclose()* function closes a file.

Closing a File

```
FILE *StudentFile;  
.....  
.....  
fclose(StudentFile);
```

8.3 Character Access to Files

Character access to files also known as formatted access. The *fprintf()* function is normally used in character access, it transfers the data to a file. The *fprintf()* function have the same meaning as *printf()*. The difference is that instead of the characters appearing on the screen, they will be written to the file at the location of the file position indicator. The file must be opened in some mode that will allow writing to it – "w", "wb", "a", "ab", or anything with a plus sign in it such as "rb+". To retrieve a file's content, the *scanf()* function is used.

Writing to File - text file

```
fprintf(StudentFile, "%s %i\n", Name, Age);
```

Both the *scanf()* and *fscanf()* have the same arguments, parameters, and return values, but *fscanf()* has an additional argument – the pointer to the file description. Both *scanf()* and *fscanf()* have an additional conversion code that is of limited use in the keyword input of *scanf()*. When writing to a text file, some delimiters were included, the slashes, to separate values in the file. We can use those delimiters to tell a *scanf()* string conversion when to stop reading characters. Instead of using %s for a string conversion, we will use the %[] code. Inside the brackets we will put a caret (^) followed by any characters we want to be recognized as delimiters – those that end a string conversion. For example, %[^\, -] will end a string conversion at either a comma or dash, whichever comes first. Whitespace, the usual default delimiter, becomes just another character when use %[].

Retrieving from File (text file)

```
fscanf(StudentFile, "%s %i%c", Name, &Age);
```

Example:

```
fprintf(StudentFile, "%s/ %i\n", Name, Age);  
...      ...  
...      ...  
fscanf(StudentFile, "%[^/]%c %i%c", Name, &Age);
```

8.4 Finding The End of A File

The *feof()* function checks to see if the end-of-file indicator is set. In almost all cases, this indicator is set by an attempt to read or beyond the end of the file.

Finding End of The File

```
if (feof(StudentFile))  
    printf("End of the file");
```

8.5 Sample Program

Program 8-1 (Adding)

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    char Name[40];
    int ID=0;
    char Gender;
    float Amt;
    FILE *sfile;

    if ((sfile = fopen("c:\\stud.dat", "a+")) == NULL)
        ID = 1;
    else
    {
        do{
            fscanf(sfile, "%i %[^/]%c %c %f%c", &ID, Name, &Gender, &Amt);
        }while (!feof(sfile));
        ID += 1;
    }

    printf("Student ID: %04i", ID);
    printf("\nName:    ");
    fflush(stdin);
    gets(Name);
    printf("Gender:  ");
    scanf(" %c", &Gender);
    printf("Amount:  ");
    scanf("%f", &Amt);
    printf("\nRecord saved!");
    fprintf(sfile, "%i %s/ %c %.2f\n", ID, Name, Gender, Amt);
    fclose(sfile);
}
```

Program 8-2 (Disaply)

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    char Name[40];
    int ID;
    char Gender;
    float Amt;
    FILE *sfile;

    if ((sfile = fopen("c:\\stud.dat", "a+")) == NULL)
        printf("File Empty!");
    else
    {
        while (!feof(sfile))
        {
            fscanf(sfile, "%i %[^/]%*c %c %f%*c", &ID, Name, &Gender, &Amt);
            if (feof(sfile))
                break;
            printf("%04i %s %c %.2f\n", ID, Name, Gender, Amt);
        }
    }
    printf("\nEnd of file, press any key to exit.");
    fclose(sfile);
}
```

Program 8-3 (Searching)

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void main(void)
{
    char Name[40], Target[40];
    int ID, Found = 0;
    char Gender;
    float Amt;
    FILE *sfile;

    if ((sfile = fopen("c:\\stud.dat", "r")) == NULL)
        printf("File Empty!");
    else
    {
        printf("Enter Name to search: ");
        fflush(stdin);
        gets(Target);
        while (!feof(sfile) && Found == 0 )
        {
            fscanf(sfile, "%i %[^/]%*c %c %f%*c", &ID, Name, &Gender, &Amt);
            if (strcmp(Target, Name) == 0)
                Found = 1;
        }
        if (Found)
        {
            printf("ID:   %04i\n", ID);
            printf("Name: %s\n", Name);
            if (toupper(Gender) == 'F')
                printf("Gender: Female\n");
            else
                printf("Gender: Male\n");
            printf("Amount: %.2f\n", Amt);
        }
    }
    fclose(sfile);
}
```

Program 8-4 (Deleting)

```

#include <stdio.h>
#include <ctype.h>
#include <string.h>

void main(void)
{
    char Name[40], Target[40];
    int ID, Found = 0;
    char Gender;
    float Amt;
    FILE *sfile, *temp;

    temp = fopen("c:\\temp.dat", "w");
    if ((sfile = fopen("c:\\stud.dat", "r")) == NULL)
        printf("File Empty!");
    else
    {
        printf("Enter Name to delete: ");
        fflush(stdin);
        gets(Target);
        while (!feof(sfile))
        {
            fscanf(sfile, "%i %c[^/]%c %c %c%f%c", &ID, Name, &Gender, &Amt);
            if (feof(sfile))
                break;
            if (strcmp(Target, Name) != 0)
                fprintf(temp, "%i %s/ %c %c%f\n", ID, Name, Gender, Amt);
            else
            {
                Found = 1;
                printf("ID: %04i\n", ID);
                printf("Name: %s\n", Name);
                if (toupper(Gender) == 'F')
                    printf("Gender: Female\n");
                else
                    printf("Gender: Male\n");
                printf("Amount: %.2f\n", Amt);
            }
        }
        if (!Found)
            printf("Record not found!\n");
        printf("\nRecord deleted.");
        fclose(sfile);
        fclose(temp);
        remove("c:\\stud.dat");
        rename("c:\\temp.dat", "c:\\stud.dat");
    }
}

```

Chapter 9 – Record-Based Data

Objectives

When data is manipulated in records that group individual but related values, it is used to have a mechanism that holds each record together. In this chapter we shall look at a mechanism. At its conclusion, you should understand:

- The characteristics of C's method for accessing records
- How these records are defined, declared, initialized, and accessed
- How to use records in conjunction with files

9.1 Structures

A structure is a complex data type made up of other data types. For example, we could combine the three strings (*char* arrays) for name, address, and phone number into one structure. We may access the entire structure – move it into secondary storage, for example; or we could access an individual member of the structure – change the address, for example.

9.1.1 Definitions and Declarations

Structures are data types that we make up ourselves. Their components are the existing data types. Working with a structure requires a step that is already done for us with the standard data types – providing a structure definition, where we tell C the makeup of the structure. The general format of the definition is as below:

```
struct tag
{
    member definitions
}
```

The tag is the name of new data type. It is equivalent of *float* or *int* and is used in much the same way – we will declare variables of that data type to be used in the program. In the member definitions we will define each of the individual variables that make up the structure.

Example:

```
struct emp_record
{
    char name[40];
    int age;
    float basicPay;
};
```

The above example with tag of *emp_record* contains definitions of a 30-element *char* array referred to as *name*, an *int* referred to as *age* and a *float* referred to as *basicPay*. Take note that there is a semicolon after the closing brace. Any valid data type can be a member of a structure. The structure definition only tells C the makeup of the structure. It

does not allocate memory. To use the new data type, we will have to declare variables of that type. The following is an example of the declaration:

Example:

```
struct emp_record staff;
```

The above example allocates memory space for the variable *staff* of data type *emp_record*. The key word struct must be included whenever we refer to a structure data type. Structure tags are defined externally, outside of a function, are visible globally, in fact, the structure can be declared anywhere in the program. It is often a good idea to define structure externally, where they are visible to all functions. C also allows us to both define and declare structures in the same statement. The overall general form is as below:

```
struct tag  
{  
    member definitions  
} names;
```

Example:

```
struct emp_record  
{  
    char name[40];  
    int age;  
    float basicPay;  
}full_time, part_time;
```

9.1.2 Initializations

We may also explicitly initialize structures at the time of their declarations. The important criterion for initializations is that the values are listed in the exact same order as the members in the structure. The entire definition, declaration, and initialization could be contained in one statement. Please take note that no initialization should be made in a

definition as a definition of structure allocates no memory space and sets up no variables. The following is an example of a structure initialization:

Example:

```
struct emp_record full_time = {"Richard Brandon", 35, 3500.00};
```

9.1.3 The sizeof a Structure

The result of a sizeof operation is the number of bytes in the expression or data-type name following sizeof. If data-type names are used, they must be enclosed in parentheses. Defining a structure defines a data type, and we can declare variables of that data type. The *sizeof* operator will give us the number of bytes in the structure data type or variable.

9.2 Accessing Structures

Structures are most often accessed – read or assigned – by accessing individual members. Since we may have many variables of the same structure type and each of these variables will have the same member names, we must tie the member name to the specific structure variable with a member operator, a dot (.). Consider the following example:

Example:

```
struct emp_record
{
    char name[40];
    int age;
    float basicPay;
}full_time, part_time;
    :
    :
    printf("Employee's name: %s", full_time.name);
    printf("Employee's age: %i", full_time.age);
    :
    :
```

9.3 Arrays of Structures

We may declare an array of structure base on the number of records that we would like to store. The following example shows how we could declare an array of structures:

Example:

```
struct emp_record staff[4];
```

To access the basicPay of the second staff in the array we would refer to *staff[1].basicPay*. To access the third character of the name member of the second staff we would refer to *staff[1].name[2]*. The array may be initialized by putting a block of values after the declaration, the following is an example of the initialization:

Example:

```
struct emp_record staff[4] =  
{{    "Barzoom", 20, 2300},  
 {    "LaRue", 30, 2300},  
 {    "Annie", 20, 2000},  
 {    "Peter", 25, 2800}};
```

9.4 Files and Structures

A structure can be viewed as a fixed-length object stored at an address in memory. This kind of description fits perfectly with the operation of *fread()* and *fwrite()*. We can easily move structures to and from files by passing either function the address of the structure, the number of bytes in the structure (which we can get using *sizeof*), and the number structures. The following example shows how *fread()* and *fwrite()* functions are used with structures:

Example – *fwrite()*:

```
struct emp_record staff;  
FILE *emp;  
:  
:
```

```
fwrite(&emp, sizeof staff, 1, emp);  
:  
:
```

Example – *fread()*:

```
struct emp_record staff;  
FILE *emp;  
:  
:  
while(fread(&emp, sizeof staff, 1, emp) == 1)  
:  
:
```

9.5 Sample Program

Program 9-1

```
#include <stdio.h>

#define NAME_CHRS 30

struct employee_rec
{ char name[NAME_CHRS];
  int dependents;
  float pay_rate;
};

void main(void)
{ struct employee_rec full_time = {"Beulah Barzoom", 4, 12.63};

  printf("Employee's name: %s.\n", full_time.name);
  printf("Dependents: %i.\n", full_time.dependents);
  printf("Change pay rate from %.2f to > ", full_time.pay_rate);
  scanf("%f", &full_time.pay_rate);
  printf("Confirm new pay rate: %.2f.\n", full_time.pay_rate);
}
```

Output

Employee's name: Beulah Barzoom.

Dependents: 4.

Change pay rate from 12.63 to > **14.25**

Confirm new pay rate: 14.25

Program 9-2

```

#include <stdio.h>

#define TAX_RATE .16                /* General tax rate */
#define DEP_REDUCTION .02          /* Reduction for each dependent */
#define NAME_CHRS 30

struct employee_rec
{ char name[NAME_CHRS];
  int dependents;
  float pay_rate;
};

void main(void)
{ struct employee_rec employee;
  float hours, gross, tax;
  double net;

  printf("Employee name: ");
  gets(employee.name);
  while (employee.name[0] != '\0')    /* Stop at empty string */
  { printf("Hours, pay rate, other dependents: ");
    scanf(" %f %f %i",
          &hours, &employee.pay_rate, &employee.dependents);
    gross = hours * employee.pay_rate;
    tax = (TAX_RATE - DEP_REDUCTION * employee.dependents) * gross;
    net = gross - tax;
    printf(" Pay to: %s  $$$%.2f**\n", employee.name, net);
    while (getchar( ) != '\n');      /* Flush stream */
    printf("Employee name: ");
    gets(employee.name);
  }
}

```

Output

Employee name: **Maynard Freebisch**

Hours, pay rate, other dependents: **36 10.83 2**

Pay to: Maynard Freebisch *****350.89****

Employee name: **Gilda Garfinkle**

Hours, pay rate, other dependent: **44 15.25 4**

Pay to: Gilda Garfinkle *****630.74****

Employee name:

Program 9-3

```
#include <stdio.h>
#include <stdlib.h>
#define CHRS 30

struct employee_rec
{ char name[CHRS];
  int dependents;
  float pay_rate;
};

void main(void)
{ struct employee_rec employee;
  FILE *employ;

  if ((employ = fopen("EMPLOYEE.DAT", "wb")) == NULL)    /* Open file */
  { printf("Cannot open file.\n");
    exit(EXIT_FAILURE);
  }

  printf("Employee name: "); /****** Input and write to file */
  gets(employee.name);
  while (employee.name[0] != '\0')          /* Empty string */
  { printf("Pay rate, other dependents: ");
    scanf(" %f%i", &employee.pay_rate, &employee.dependents);
    fwrite(&employee, sizeof employee, 1, employ);
    while (getchar() != '\n');
    printf("Employee name: ");
    gets(employee.name);
  }
  fclose(employ);
}
```


Output

Employee name: **Maynard Freebisch**

Pay rate, other dependents: **10.83 2**

Employee name: **Gilda Garfinkle**

Pay rate, other dependents: **15.25 4**

Employee name: **Beulah Barzoom**

Pay rate, other dependents: **12.63 4**

Employee name: **LeRoy LaRue**

Pay rate, other dependents: **9.50 -1**

Employee name:

Program 9-4

```
#include <stdio.h>
#include <stdlib.h>

#define CHRS 30

struct employee_rec
{ char name[CHRS];
  int dependents;
  float pay_rate;
};

void main(void)
{ struct employee_rec employee;
  float hours, gross, tax;
  double net;
  FILE *employ;

  if ((employ = fopen("EMPLOYEE.DAT", "rb")) == NULL)    /* Open file */
  { printf("Cannot open file.\n");
    exit(EXIT_FAILURE);
  }
  while (fread(&employee, sizeof employee, 1, employ) == 1)    /* Pay */
  { printf("Hours for %s: ", employee.name);
    scanf("%f", &hours);
    gross = hours * employee.pay_rate;
    tax = (.16 - .02 * (employee.dependents + 1)) * gross;
    net = gross - tax;
    printf(" Pay to: %s $**%.2f**\n", employee.name, net);
  }
  fclose(employ);
}
```

Output

Hours for Maynard Freebisch: **36**

Pay to: Maynard Freebisch *****350.89****

Hours for Gilda Garfinkle: **44**

Pay to: Gilda Garfinkle *****630.74****

Hours for Beulah Barzoom: **39**

Pay to: Beulah Barzoom *****463.02****

Hours for LeRoy LaRue: **3**

Pay to: LeRoy LaRue *****23.94****

Student Information

Project Proposal

- One copy of project proposal consists of the cover page and the proposal details (refer to appendix D) should be submitted on lesson 4.
- Marks will be awarded to project proposal approved by project supervisor.
- No project development should be started without obtaining the approval of project proposal from respective supervisor/lecturer.

Documentation Format (Project Documentation and Project Proposal)

- All documents (proposal and final project documentation) should be **type written using word processor**.
- The cover page (refer to appendix E) should contain student's particulars.
- The final project documentation should be bound properly with diskette attach in the back cover of the project.
- Any hand written material found in final project documentation will result in marks deduction.
- Standard font style of Times New Roman
- Font size of 12 (except Program Listing)
- Line spacing of 1.5 (except Program Listing)
- Start each chapter/part on new page.
- The content page should show all the relevant chapters/pages clearly.
- **MUST** have continuous and type written page numbers.

Dateline and Submission

- The dateline for project proposal submission is on **lesson 4** and final documentation on **lesson 14**.
- Late submissions will cause **10 marks to be deducted per day**.
- Items to be submitted on lesson 14:
 - ✧ One copy of the original final project documentation (no photocopy project documentation is acceptable)
 - ✧ The approved project proposal
 - ✧ One copy of the source program, and EXE version on diskette
- The submitted project will not be returned to students. Students are required to keep a copy for own references.

Backup

- Students should always backup their program and documentation.
- Students should always make use of Virus Scan software to scan their diskette to prevent diskette from infecting by viruses.

Integrity and Plagiarism

- Any project is found copied from either classmates or outside sources will not be marked.
- Lecturers have the right not to accept any project is found lacking of substance or copy from some other sources.

Sample Project

There won't be any sample project to be given to students. Students are required to develop their project according to the stated project specification. There is no specific format to the project. The most important criterion is to produce all the required parts in their project.

Project Presentation

There will be twenty percent on Project Presentation, and each student **MUST** pass the Project Presentation/Viva (with minimum score of ten marks) in order to pass this unit. Failures to do so will result in a fail grade for this module.

Basic Project Requirements

All the project documentation and program should clearly indicate the file implementation process. Candidates may either choose to implement text file or binary file in the program. Should there be any absence of file implementation will result in a “D” grade regardless of the quality of the project documentation.

All codes in program should be written using standard C language, should there be any other language found in the program such as C++, or Object Oriented, etc will result in a “D” grade to be awarded.

Candidates must include basic file manipulation process such as Add, Delete, Search and Delete of records in the project's program. Failures to do so will result in a “D” to be awarded.

Project Guidelines

Students are required to document their projects based on the Suggested Table of Content. On Lesson 14, students are required to submit only ONE original copy on documentation and a soft copy of program. Students are reminded that the submitted project will not be returned and students are expected to photocopy one own reference.

Suggested Table of Contents

1. Current System
2. New System
3. Specification
 - 3.1 Input Specification
 - 3.2 Screen Design
 - 3.3 CRT Forms
 - 3.4 Processing & Validations
4. Program Listing
5. Testing
 - 5.1 Test Log
 - 5.2 Test Cases
 - 5.3 Test Log
6. Implementation
 - 6.1 User Manual
7. Conclusion
 - 7.1 Strengths
 - 7.2 Weaknesses
 - 7.3 Enhancements

1. Current System

Provide some background information about the types of problem currently faced in a company. A detail description on all the problems encountered by the user. Why there is a need to implement a new system, and who initiated this project. Students may approach this by identifying problematic areas in the company and why they need to change the manual or automated system to a new system.

2. Proposed New System

Outline the proposed new system. Students may approach this by looking into functions available in the proposed system. Detail description on features available in the new system. Students are reminded to substantiate that the proposed new system is able to solve existing problems faced by the users.

3. Program Specification

3.1 Input Specification

In this section, students have to show all of the **input variables** that are used in program. Information such as Item Name, Description, Data Type and Size should be described in detail.

3.2 Screen Design

The Screen Design section is a diagrammatic representation of all the screens in the program. Links among different screens are clearly shown. For example, the Main Menu links to individual screens such as Data Entry, Edit etc.

3.3 CRT Form

The CRT is used to draw individual screen as stated in the Screen Design section. The CRT form will be distributed on the first lesson, students are required to make some copies depends on the number of screens in Screen Design section. The objective of using CRT form is allow the required co-ordinate (such column and row number) to be viewed during coding. When draw the CRT Form, try to use pencil so as to facilitate changes. A copy of standard CRT Form is available in Appendix C.

Standard should be adopted when preparing CRT form, the following is the required standard:

- X - alphanumeric field
- 9 - numeric field

For example:

CRT Form

<u>Add New Student</u>
Name: xxxxxxxxxxxxxxxx
Age: 99
Date of Birth: 99/99/99
Gender: x (F-female / M-male)
Address: xxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Save the record (Y/N)? x

3.4 Processing & Validation

In this section, it shows all the data processing activities involved in the program. This can be details like how data are validated, the algorithm used for certain data processing tasks, for e.g. checking data entry for gender type, update the quantity field of the master file record, print a subtotal at the end of the report, calculate the average of a set of numeric values. These are the program requirements that your program should reflect. The following are some possible checking or validations that you may include in the Processing Requirement section.

Suggested Types of Validation

1. Format check - the acceptable entry in terms of characters, digits etc. normally applicable to alphanumeric or alphabetic field.
2. Emptiness - where the entry for a particular field is compulsory, normally applicable to all the fields.
3. Uniqueness - where no duplication/similar entry is allowed, normally applicable to key field such NRIC, Identification NO. etc.
4. Size check - where a particular field entry should not exit the stated size, normally applicable to alphanumeric and alphabetic fields.
5. Range check - where the entry of a particular field should be within specific range, normally applicable to numeric fields.

Suggested Calculation

1. Formulas used to derive certain results

Suggested Update Processes

1. Updating of files - how related file(s) is updated.

For example:

Name

1. The acceptable entry for name is purely letters of alphabet, otherwise error message "Invalid! Accept letters of alphabet from A to z only." will be displayed.
2. The maximum size allowed for name field is 20, otherwise error message "Invalid! The maximum number of characters allowed is 20 only" will be displayed.
3. The name field should have data entry, otherwise error message "Invalid! You must enter the name" will be displayed.

Payment_amount

1. The acceptable payment amount range from 350.00 to 5,000.00, otherwise error message "Invalid! Accept payment amount range from \$350.00 to \$5,000.00 only" will be displayed.
2. The Payment_amount will be updated to a field known as YTD in the Student Master file by the following algorithm:

Locate the related record in the Student Master file
YTD = YTD + Payment_amount

4. Program Listing

A copy of an updated error-free program listing should be included in this section. Students are reminded that good habits like giving comments, using meaningful identifier names, good program layout with proper indentation. These attributes will increase the ease of maintenance at later stage. Try to avoid very long module. A long module should be broken down into smaller modules with each performing a single subtask. All these features will help to improve the readability of the program.

5. Program Testing

5.1 Test Plan

A proper Test Plan should be produced to test the program. Students may adopt Top-down or Bottom-up testing strategy. Testing begins at the menu level while Bottom-up testing focuses on individual program and follow by the menu level. An outline of the total number of test cases included must be outlined here, for example:

<u>Test case</u>	<u>Objectives</u>
1	To check for the correct activation of first option in the main menu

5.2 Test Cases and Results

All test cases outlined in the test plan previously are carried out with the actual running of the program. This section of the testing documentation should include further elaboration of each test case. Each test case should have the following components:

Test Case	:	1
Objective	:	Explain the purpose of this test case
Test Data	:	Show actual test data used in this test case
Expected Test Result	:	Explain the expected output you expect the program to produce if the test data are used
Actual Test Result	:	This should be produced by your program in actual testing environment. You should capture and print out the screen to support your testing activities. Attach this copy below the test case as an Actual Test result. DO NOT add anything to the print out.
Conclusion	:	Write a short conclusion as to whether any programming error has been discovered.

How to capture the actual test result:

1. Press ALT+PRINTSCREEN keys simultaneously to capture the screen
2. Switch to MS Word immediately and PASTE the screen.
3. If output screen in DOS mode, press ALT+ENTER to switch the screen to window size before capture the screen.

6. User Manual

Students are required to document user instructions showing the user's step by step usage of different functions of the program. There must be enough coverage on each function outlined.

The students should include some **printed copy of screen capture** when guiding the users with step by step instructions. This gives the users a better idea of how the screen would look like and the ways to respond to such screens. The user manual must be detail enough for a non-computer literate to operate and no computer jargon should be included.

7. Conclusion

7.1 Program weaknesses

Briefly explain the weaknesses of the program. You may comment on the performance of the program.

7.2 Program Strengths

Include some comments on the strengths of the program. Specify in what ways it benefits users.

7.3 Program Enhancements

Include some suggestions that would help to enhance the performance of the program in the future.

Recommended reference text

Title	Author	Publisher
The Art of Programming - Computer Science with C	Steven C. Lawlor	West Publishing Company
C How to program	H.M. Deitel / P.J. Deitel	Prentice Hall
Programming with C	Byron Gottfried	Mc Graw Hill
C Programming Using Turbo C++	Robert Lafore	SAMS Publishing

Assessments

75% (70-100)	A	Outstanding good performance, showing insight, creative thinking or flair in excess of the normal expectations for good sound works. Well organised, clear and accurate documentation. Does not have to be perfect but flaws and omissions should be minor in relation to the quality of work taken as a whole.
65% (60-69)	B	Very good performance, showing some insight, creative thinking or flair in excess of the normal expectations for a sound performance. Close to a “A” performance but falling short in some significant respect. When the work is considered as a whole some shortcomings cannot be overlooked.
55% (50-59)	C	A sound performance. Competent and appropriate response to the task. Significant shortcomings and not serious enough to gravely weaken the whole work.
45% (40-49)	D	A fair attempt. Some acceptable elements achieved, but the work may lack organisations and lack coherence as a whole, or have other important weaknesses.
35% (35-39)	(Fail)	A borderline failure. The minimum standard expected.
25% (15-34)	(Fail)	Not up to the minimum standard, but some credit earned.
<15% (0-15)	(Fail)	Unacceptable work or not submitted.

Appendix D

Project Title:

Software Requirement:

Hardware Requirement:

Abstract:

Project status: Reject/Pending/Refinement/Accept

Date of Approval:

Approved by:

Signature:

Appendix E**Cover Page**

Student Name:

Lecturer/Supervisor Name:

Student Number:

Student Contact Number:

Class Code:

Center Code: 0101

Subject Name: C Programming

Country: Singapore

Subject Code: IT202

Unit Code: IT202 - C Programming

Release date: September 2002

LESSON NO.	TOPICS COVERAGE	ACTIVITIES
1. Theory	Introduction to C Forming a C program, variables, data types and declarations, arithmetic expressions, assignment and simple output.	Discuss Key terms and concepts.
2. Practical	Building a C program Developing a style, directives, streams, output and input.	Practical
3. Theory	Theory on Selection structure & Iteration structure Conditions, if statement, switch statement, loops in C, counter controlled, nested loops.	Practical
4. Practical	Practical exercises on Selection structure & Iteration structure	Practical
5. Practical	Function External variables, recursion and existing Functions.	Practical
6. Practical	Arrays Indexed variable names, array declaration, array functions and applications.	Practical
7. Practical	Strings String Values, variables, assignments, input & output, string manipulation, Character classification and conversion.	Practical
8. Practical	Files input and output File identifiers, buffered input and output, opening files, closing a file, character access to files.	Practical
9	Online Test	Practical
10. Practical	Record-based Structure Structures in C, Records, Array of records	Practical
11. Theory	Progress test	
12. Practical	Project development work	Practical
13. Practical	Project development work	Practical
14. Theory	Project submission	Practical

Reference book

The Lesson Plan and Study Guide was developed base the following reference book:

Title: The Art of C Programming

Author: Steven C. Lawlor

Publisher: ITP (1996)

Revision Exercise

Chapter 1

1.

- i. Gnash is a valid variable name
- ii. Union is an invalid variable name because it is a reserve word
- iii. 9Times is an invalid variable name because cannot begin with a digit
- iv. too_many is a valid variable name

2.

- i. not a valid declaration because there is no such variable type as unsigned float
- ii. not a valid declaration because there is no such variable type as Double (it's capitalization)
- iii. valid declaration
- iv. not a valid declaration because semicolon missing

3.

- i. 91 - int
- ii. 45 - int
- iii. 48652 – long int
- iv. 16.25 – float/double

4.

- i. 1 (int)
- ii. 12.5 (float/double)
- iii. 13.2 (float/double)
- iv. 5.0 (float/double)

5.

- i. 8.3
- ii. 6
- iii. 10
- iv. 27

6.

- i. A-Z, upper- or lowercase, 0-9, and underscore. A name can start with anything but a number.
- ii. No, C is case-sensitive
- iii. Whether it is represented by straight binary or exponential notation, and how many bits it occupies

- iv. Directs C how to store values in the variable, allocates space in memory, and, possibly, initializes the variable
- v. Char and int (with the possible modifiers signed, unsigned, short, and long) are integral; float, double, and long double are floating point.
- vi. U, L, UL, F and L again for long double, but the decimal point differentiates it from a long int.
- vii. We must use the special character \'
- viii. As one continuous string value

7.

- i. true
- ii. false
- iii. false
- iv. true
- v. true
- vi. true
- vii. false
- viii. There are no string variables in C.
- ix. true
- x. false

Chapter 2

1.

- i. %-3i
- ii. %6.1f
- iii. %5.1f
- iv. ^%06li
- v. %.2f
- vi. %i/%d
- vii. %c
- viii. %9.3E

2.

- i. |*65|
- ii. |A|
- iii. |***4.3|
- iv. |425|
- v. |381.7|
- vi. |*016|
- vii. |1.23|
- viii. |***52|

3.

- i. The first argument to the *printf*() function is the control string that defines the format of the output line. Subsequent arguments provides values to fill the spaces left in the format.
- ii. Conversion code begin with a %a and end with a type specifier.
- iii. Size modifiers further define the data type of the type specifier. The modifiers *h* and *l* preceding *i* specify *short* or *long ints*, and *L* preceding *f* specifies long double. With *scanf*(), *lf* specifies a double.
- iv. The width parameter specifies a minimum width for the field for *printf*().
- v. Precision for *printf*() states the number of decimal places for a floating-point field (or significant digits for a %g field) and the minimum number of digits with leading zero fill for integral fields.
- vi. Prompts are reminders displayed on the screen before inputs. They are displayed using *printf*().
- vii. *while (getchar() != '\n');* or *fflush(stdin)* are used to flush the input stream of unexpected or unwanted characters.

4.

- i. false
- ii. false
- iii. true
- iv. true
- v. false

5.

- i. *define*
- ii. *char*
- iii. *&a*

Chapter 3

1.

- i. $(1 < x \ \&\& \ x < 5)$
- ii. $(16 \geq x \ \&\& \ x \geq -7)$
- iii. $(22.6 \geq x \ \&\& \ x \geq 12.03)$
- iv. $(36 < x \ \&\& \ x \leq y \ \&\& \ y < 115)$

2.

- i. true
- ii. true
- iii. true
- iv. true

3.

```
if(color == 'r')
    printf("stop");
else if(color == 'y')
    printf("caution");
else if (color == 'g')
    printf("go");
else
    printf("weird");
```

4.

The two segments don't do the same thing. In the first code statement, the second *if* can never be true since it is part of the *else* which only gets executed if $x \leq 0$. In the second code segment the second *if* is always true when the first one is true, so it is redundant. In both segments the segment *if* ($x > 2$) could be taken out (*int* the second case `{ }` would have to be added however, to keep the functionality the same). If x 's initial value were 2, x would get 4 in the first segment and in the second segment x would get 8.

5.

- i. By enclosing the statements within braces
- ii. No, if the false branch is to take no action
- iii. The `=` operator makes an assignment; the `==` operator makes a comparison.
- iv. Only if there is more than one statement in the branch.
- v. The entire set of nested *ifs* is one structure; if it is set up using *else if*, only one branch will execute. Sequential *ifs* are separated structures, and many may execute.
- vi. Logical operators, in order of precedence, are *not* (`!`), *and* (`&&`), and *or* (`||`).

6.

- i. 9
- ii. 0
- iii. 10

7.

- i. nested
- ii. condition
- iii. logical operators
- iv. sequence, selection & iteration
- v. Pseudocode
- vi. sequence
- vii. selection
- viii. expression comparison_operator expression
- ix. `&&`
- x. `!`

Chapter 4

1.

```
for (x = 14; x >= 3; x -= 5)
    printf("%d\n", x);
```

```
for (y = 65; x <= 85; x += 5)
    printf("%d\n", y);
```

2.

```
x = 250.0;
while (x >= 100)
{
    printf("%f\n", x);
    x -= 50;
}
```

```
y = 1226;
while (y <= 1426)
{
    printf("%f\n", yx);
    x -= 2;
}
```

3.

```
1 1
2 2 1
3 3 2 1
4 4 3 2 1
5 5 4 3 2 1
6 0
```

4.

a. $y = y + a * c; c++;$

5.

- i. both s and z
- ii. z
- iii. t
- iv. not necessary
- v. sentinel value
- vi. yes
- vii. yes
- viii. yes
- ix. true
- x. false

6.

An accumulation process adds (or multiplies, or whatever) the value of a variable to some other values and stores the result in the original variable.

7.

If the accumulator variable contains garbage before the accumulation, it will contain garbage after.

8.

Counting is accumulation, but with difference that it adds the same value each time instead of different values.

9.

Any counter-controlled loop must contain an initialization, a test, a body, a counter and end.

10.

```
x = 20;  
while (x >= 100)  
{  
    printf("%d\n", x);  
    x -= 50;  
}
```

11.

1 2 3 4 5

 $x = 5$ **12.**

```
for (x = 14; x >= 3; x -= 5)
```

```
    printf("%d", x);
```

```
for (y = 65; y <= 85; x += 5)
```

```
    printf("%d", y);
```

13.

i. 3

ii. 4

iii. 2 3 4 3 4 5

Chapter 5

1.

The function definition starts with the declaration of its return types, its name, and initialized local variables, and it contains all of the code that makes the function operate.

2.

The function call passes values to a function and sets the function in operation.

3.

The return statement ceases execution of the function, passes the value of the expression (if any) back to the calling point to substitute for the call, and continues program execution at the calling point.

4.

A void return type means that no value is being returned by a function. A void declaration means that no variables are being declared to be initialized by passed values. In other words, the function call can have no arguments.

5.

The lifetime of a variable is the part of the program in which memory is allocated for that variable. The visibility of a variable (or functions) is the part of the program in which the variable can be accessed (or function called) by name.

6.

Global lifetime or visibility is from declaration throughout the entire program. Local is within only a certain section, such as within one function.

7.

Internal is within a function; external is outside of any function. Internal declarations are local; external declarations are global.

8.

1 4 9 16 25 36 49 64 81

9.

- i. two
- ii. *int*
- iii. *int Calculate(int, int);* or *int Calculate(int x, int y);*
- iv. *int Calculate(int x, int y)*
 {
 *return (x * y);*
 }
- v. v3

10.

- i. true
- ii. true
- iii. false
- iv. false
- v. false

11.

- i. two
- ii. 0
- iii. n times
- iv. double
- v. FALSE
- vi. 0
- vii. n, t, i
- viii. FALSE
- ix. TRUE
- x. 0

12.

- i. c
- ii. b
- iii. c
- iv. c
- v. a

Chapter 6**1.**

An indexed variable, like any other variable, has its own name, its own value, and its own place in memory.

2.

Changing the index, changes the element or variable referred to.

3.

An array is a set of indexed elements of the same data type. In C all the elements will be contiguous in memory.

4.

One reason for using arrays is to create a large number of variables. Another is so that we can change the variable name in our program by changing the value of the index.

5.

Indexed variable sets in C always start with the index zero.

6.

Ten variables are allocated with; the last index being 9, that is, *stuff[9]*.

7.

stuff[1] will be 2.2. Since only *stuff[0]*, *stuff[1]*, and *stuff[2]* have initialization values, *stuff[3]* will be zero, *stuff[5]*, while we can actually access it, is beyond the end of the array, so its value is garbage.

8.

An index will be an integral data type. If a floating-point value is given, it is truncated to an integer.

9.

- i. `#define SIZE 10`
- ii. `float fractions[SIZE] = {0};`
- iii. `fractions[3];`
- iv. `fractions[9];`
- v. `fractions[0];`

10.

- i. valid
- ii. valid
- iii. invalid
- iv. valid
- v. invalid

11.

- i. FALSE
- ii. FALSE
- iii. TRUE
- iv. FALSE
- v. TURE
- vi. TRUE

12.

- i. array
- ii. 99
- iii. 0
- iv. sort
- v. insert

13.

- i. a
- ii. c
- iii. c

Chapter 7**1.**

By using the brackets and the caret indicating the characters that are not acceptable in the conversion.

2.

printf(), using the *%s* conversion code, or *puts*().

3.

It states the maximum number of characters that will print in a field.

4.

strlen()

5.

strcpy() copies a string from one address to another, including the terminating null. *strncpy()* does the same, but will not exceed a maximum number of bytes, even if the null is not copied.

6.

strcat() copies a string from one address to the end of a string at another address. *strncat()* does the same, but will not copy more than a maximum number of bytes.

7.

strcmp() and *strncmp()* compare bytes a position at a time from each address given. The comparison is made according to the numeric value of the ASCII codes. The comparison stops at the first difference or, in the case of *strncmp()*, at a maximum number of bytes.

8.

- i. double quotes
- ii. 1
- iii. array of character
- iv. input up to a # or @ is reached

9.

- i. FALSE
- ii. FALSE
- iii. FASLE
- iv. TRUE
- v. FALSE
- vi. TRUE
- vii. FALSE
- viii. TRUE

Programming Exercise

Chapter 1

1.

```
#include <stdio.h>
void main(void)
{
    printf("NAME: Steven\n");
    printf("MAJOR: Diploma in Informtation Technology\n");
    printf("OTHER COMPUTER COURSE TAKEN: Web Publishing\n");
    printf("OCCUPATION: Account Manager\n");
}
```

2.

```
#include <stdio.h>
void main(void)
{   int v1, v2;
    float v3, v4;

    v1=4;
    v2=42;
    v3=16.7;
    v4=.0045;

    printf("The four numbers are: \n");
    printf("%d %d %g %g", v1, v2, v3, v4);
}
```

3.

```
#include <stdio.h>
void main(void)
{   float dollars = 7.73;
    int cents = 773;

    printf("The coin breakdown for %10.2f dollars is: \n");
    printf("Dollar bills : %d\n", cents / 100);
    cents = cents - cents / 100 * 100;
    printf("Fifty cents : %d\n", cents / 50);
    cents = cents - cents / 50 * 50;
    printf("Twenty cents : %d\n", cents / 20);
    cents = cents - cents / 20 * 20;
    printf("Ten cents : %d\n", cents / 10);
    cents = cents - cents / 10 * 10;
    printf("Five cents : %d\n", cents / 5);
    cents = cents - cents / 5 * 5;
    printf("Cents : %d\n", cents / 1);
}
```

4.

```
#include <stdio.h>
void main(void)
{   float feet;
    int inches = 46;

    feet = inches / 12.0;
    printf("%d inches is %.5ff feet.\n");
}
```

Chapter 2

1.

```
#include <stdio.h>
void main(void)
{   int first, second;

    printf("First number?");
    scanf("%d", &first);
    printf("Second number?");
    scanf("%d", &second);
    printf("The second goes into the first %d times\n", first / second);
    printf("with a remainder of %d.\n", first % second);
    printf("The quotient is %3.1f.\n", ((float)first / second);
}
```

2.

```
#include <stdio.h>
void main(void)
{   int input;
    float total = 0.0;

    printf("PARTS?");
    scanf("%f", &Parts);
    printf("LABOR?");
    scanf("%d", &Labor);
    SalesTax = Parts * 0.06;
    Total = Parts + Labor + SalesTax;
    printf("\n AJAX AUTO REPAIR\n SALES INVOICE\n");
    printf("PARTS      $%7.2f\n", Parts);
    printf("LABOR      $%7.2f\n", Labor);
    printf("SALES TAX $%7.2f\n", SalesTax);
    printf("TOTAL      $%7.2f\n", Total);
}
```


3.

```
#include <stdio.h>
void main(void)
{   int input;
    float total = 0.0;

    printf("Fifty cents?");
    scanf("%d", &input);
    total = total + input * 0.5;
    printf("Twenty cents?");
    scanf("%d", &input);
    total = total + input * 0.25;
    printf("Ten cents?");
    scanf("%d", &input);
    total = total + input * 0.1;
    printf("Five cents?");
    scanf("%d", &input);
    total = total + input * 0.05;
    printf("One cents?");
    scanf("%d", &input);
    total = total + input * 0.01;
    printf("Your total is %7.2f", total);
}
```

4.

```
#include <stdio.h>
void main(void)
{   long int seconds;

    printf("\nhow many second?");
    scanf("%li", &seconds);

    printf("Days: %2li", seconds / 86400);
    seconds = seconds - seconds / 86400 * 86400;
    printf("\nHours: %2li", seconds / 3600);
    seconds = seconds - seconds / 3600 * 3600;
    printf("Minutes: %2li", seconds / 60);
    seconds = seconds - seconds / 60 * 60;
    printf("\nHours: %2li", seconds);
}
```

5.

```
#include <stdio.h>
#define FIT_RATE 0.15
#define FICA_RATE 0.062
#define SAVINGS_RATE 0.03
#define RETIREMENT_RATE 0.085
#define HEALTH_INS 3.75

void main(void)
{   float Hours, HourlyPay, GrossPay, FIT, FICA, Savings, Retirement, NetPay;
```

```

printf("HOURS?");
scanf("%f", &Hours);
printf("HOURLY PAY?");
scanf("%f", &HourlyPay);

GrossPay = Hours * HourlyPay;
FIT = FIT_RATE * GrossPay;
Savings = SAVINGS_RATE * GrossPay;
Retirement = RETIREMENT_RATE * GrossPay;
NetPay = GrossPay - FIT - FICA - Savings - Retirement - HEALTH_INS;

printf("\nGROSS PAY          $7.2f\n", GrossPay);
printf("\nFEDERAL INCOME TAX $7.2f\n", FIT);
printf("\nFICA                $7.2f\n", FICA);
printf("\nPAYROLL SAVINGS      $7.2f\n", Savings);
printf("\nRETIREMENT           $7.2f\n", Retirement);
printf("\nHEALTH INSURANCE     $7.2f\n", HEALTH_INS);
printf("\n\nNET PAY            $7.2f\n", NetPay);
}

```

Chapter 3

1.

```

#include <stdio.h>
void main(void)
{ float number1, number2;

    printf("Enter two number: ");
    scanf("%f %f", &number1, &number2);

    if (number1 > number 2)
        printf("%f is greater than %f", number1, number2);
    else
        if(number1 < number2)
            printf("%f is greater than %f", number2, number1);
        else
            printf("They are equal");
}

```

2.

```

#include <stdio.h>
void main(void)
{ float Purchase, TaxRate;
  char County;

    printf("\nAMOUNT OF PURCHASE? ");
    scanf("%f", &Purchase);
    fflush(stdin);

```

```
printf("\nCOUNTY? ");
scanf("%c", &County);

if (County == 'A')
    TaxRate = 0.07;
else
    TaxRate = 0.06;
printf("TOTAL BILL: $%.2f\n", Purchase + Purchase * TaxRate);
}
```

3.

```
#include <stdio.h>
void main(void)
{   char grade;
    int grade_point;

    printf("\nLetter grade: ");
    scanf(" %c", &grade);

    if (grade == 'A' || grade == 'a')
        grade_point = 4;
    else if (grade == 'A' || grade == 'a')
        grade_point = 4;
    else if (grade == 'B' || grade == 'b')
        grade_point = 3;
    else if (grade == 'C' || grade == 'c')
        grade_point = 2;
    else if (grade == 'D' || grade == 'd')
        grade_point = 1;
    else if (grade == 'F' || grade == 'f')
        grade_point = 0;
    printf("Grade points %d", grade_point);
}
```

4.

```
#include <stdio.h>
void main(void)
{   float weight;

    printf("\nWEIGHT? ");
    scanf("%f", &weight);

    if (weight <= 1.0)
        printf("CHARGE: 10\n");
    else
        printf("CHARGE: %.1f\n", 10 + (weight - 1.0) * 6.0);
}
```

5.

```
#include <stdio.h>
void main(void)
{   float CurrentEarnings, PrevEarnings;

    printf("\nThis week's pay? ");
    scanf("%f", &CurrentEarnings);
    printf("\nPrevious pay? ");
    scanf("%f", &PrevEarnings);

    if (PrevEarnings + CurrentEarnings <= 50400.0)
        printf("FICA to withhold: $%04.2f\n", CurrentEarnings * 0.0765);
    else
        if (PrevEarnings < 50400.0 && PrevEarnings + CurrentEarnings > 50400.0)
            printf("FICA to withhold: $%04.2f\n", (50400.0 - PrevEarnings) * 0.0765);
        else
            printf("FICA to withhold: $0.0\n");
}
```

6.

```
#include <stdio.h>
void main(void)
{   int value1, value2;

    printf("\nEnter two different integer values: ");
    scanf("%d%d", &value1, &value2);

    if(value1 > 0 && value2 > 0)
    {
        if (value1 > value2)
            if(value1 % value2 == 0)
            {
                printf("%d is larger than %d\n", value1, value2);
                printf("%d is divisible by %d\n", value1, value2);
            }
            else
            {
                printf("%d is larger than %d\n", value1, value2);
                printf("%d is not divisible by %d\n", value1, value2);
            }
        else
            if (value2 > value1)
                if(value2 % value1 == 0)
                {
                    printf("%d is larger than %d\n", value2, value1);
                    printf("%d is divisible by %d\n", value2, value1);
                }
                else
                {

```

```

        printf("%d is larger than %d\n", value2, value1);
        printf("%d is not divisible by %d\n", value2, value1);
    }
}
else
    printf("Invalid value(s) entered!");
}

```

7.

```

#include <stdio.h>
void main(void)
{
    float car_value, premium;
    int age, tickets;

    printf("\nDRIVER'S AGE? ");
    scanf("%d", &age);
    printf("\nNUMBER OF TICKETS? ");
    scanf("%d", &tickets);
    printf("\nVALUE OF CAR? ");
    scanf("%d", &car_value);

    if (tickets > 3)
        printf("COVERAGE DENIED\n");
    else
    {
        premium = 0.05 * car_value;
        if (age < 25)
            premium = premium + 0.15 * premium;
        else if (age <= 29)
            premium = premium + 0.10 * premium;
        if(tickets == 1)
            premium = premium + 0.10 * premium;
        else if(tickets == 2)
            premium = premium + 0.25 * premium;
        else if(tickets == 3)
            premium = premium + 0.50 * premium;
        printf("PREMIUM: $5.2f\n", premium);
    }
}

```

Chapter 4**1.**

```

#include <stdio.h>
void main(void)
{
    int score, a_s=0, b_s=0, c_s=0, d_s=0, f_s=0;

    printf("\nSCORE? ");
    scanf("%d", &score);
}

```

```

while(score != 0)
{
    if (score >= 0 && score <= 100)
    {
        printf("THE GRADE IS ");
        if(score >= 90)
        {
            a_s++;
            printf("A");
        }
        else if(score >= 80)
        {
            b_s++;
            printf("B");
        }
        else if(score >= 70)
        {
            c_s++;
            printf("C");
        }
        else if(score >= 60)
        {
            d_s++;
            printf("D");
        }
        else
        {
            d_s++;
            printf("F");
        }
        printf("\nSCORE?");
        scanf("%d", &score);
    }
    printf("\n%d A's\n", a_s);
    printf("\n%d B's\n", b_s);
    printf("\n%d C's\n", c_s);
    printf("\n%d D's\n", d_s);
    printf("\n%d F's\n", f_s);
}

```

2.

```

#include <stdio.h>
void main(void)
{   int multiplier, multiplicand;

    for(multiplier = 1; multiplier <= 8; multiplier++)
    {
        for(multiplicand = 1; multiplicand <= 8; multiplicand++)
            printf("%d", multiplier * multiplicand);
        printf("\n");
    }
}

```

```
}
```

3.

```
#include <stdio.h>
```

```
void main(void)
```

```
{ int x;
```

```
    for(x = 1; x <= 30; x+=2)
```

```
        printf("%4d\n", x * x);
```

```
}
```

4.

```
#include <stdio.h>
```

```
void main(void)
```

```
{ int x;
```

```
    for(x = 1; x <= 30; x++)
```

```
        if(x % 3 == 0 || x % 5 == 0)
```

```
            printf("%4d", x);
```

```
}
```

5.

```
#include <stdio.h>
```

```
void main(void)
```

```
{ int x, y;
```

```
    do{
```

```
        printf("Enter two integer values: ");
```

```
        scanf("d%d", &x, &y);
```

```
        printf("The multiplication result is: %d", x * y);
```

```
    } while (x > 0 && y > 0);
```

```
    printf("End of run.");
```

```
}
```

6.

```
#include <stdio.h>
```

```
void main(void)
```

```
{ int x, y;
```

```
    for (x = 1; x <= 5; ++x) {
```

```
        printf("%i", x);
```

```
        for (y = x; y >= 1; --y)
```

```
            printf("%4d", y);
```

```
        printf("\n");
```

```
    }
```

```
}
```

7.

```
#include <stdio.h>

void main(void)
{   int x, y;

    for(x = 1; x <= 4; x++) {
        for(y = 1; y <= x; y++)
            printf("%4d", x);
        printf("\n");
    }
    for(x = 4; x >= 1; x--) {
        for(y = 1; y <= x; y++)
            printf("%4d", x);
        printf("\n");
    }
}
```

Chapter 5

1.

```
#include <stdio.h>
void page(int page);

void main(void)
{   int p;

    for(p = 1; p <= 5; ++p)
        page(p);
}

void page(int page)
{
    printf("Major Document\t\tPage %d\n\n", page);
}
```

2.

```
#include <stdio.h>
int int_test(float value);

void main(void)
{   float input;
    do
    {
        printf("Your number: ");
        scanf("%d", &input);
        if(input)
        {
            if(!int_test(input))
```



```

        printf("The number is not an integer\n");
    else if(int_test(input) != -1)
        printf("The number is %g\n", input);
    else
        printf("The number is negative\n");
    }
}while(input);
}

```

```

int int_test(float value)
{
    int result;

    if (value < 0.0)
        result = -1;
    else if ((float)(int)value == value)
        result = (int) value;
    else
        result = 0;
    return result;
}

```

3.

```

#include <stdio.h>
void change(int amount);

void main(void)
{
    float purchase, tendered;
    int cents;

    printf("Purchase: ");
    scanf("%f", &purchase);
    printf("Amount tendered: ");
    scanf("%f", &tendered);
    cents = ((tendered - purchase) * 100.0);
    if(((tendered - purchase) * 100.0) - cents > 0.5)
        cents++;
    change(cents);
}

void change(int amount)
{
    printf("Fifty cents: %d\n", amount / 50);
    amount -= (amount / 50 * 50);
    printf("Twenty cents: %d\n", amount / 20);
    amount -= (amount / 20 * 20);
    printf("Ten cents: %d\n", amount / 10);
    amount -= (amount / 10 * 10);
    printf("Five cents: %d\n", amount / 5);
    amount -= (amount / 5 * 5);
    printf("One cents: %d\n", amount);
}

```

4.

```

#include <stdio.h>
#include <math.h>
double add(double a, double b);
double subtract(double a, double b);
double multiply(double a, double b);
double divide(double a, double b);
double exponentiate(double a, double b);

void main(void)
{
    char op;
    double x, y;

    printf("Enter expression separated with space (q to quit): ");
    while(scanf("%lf %c %lf", &x, &op, &y))
    {
        switch(op)
        {
            case '+':
                printf(" %g\n", add(x, y));
                break;
            case '-':
                printf(" %g\n", subtract(x, y));
                break;
            case '*':
                printf(" %g\n", multiply(x, y));
                break;
            case '/':
                printf(" %g\n", divide(x, y));
                break;
            case '^':
                printf(" %g\n", exponentiate(x, y));
                break;
            default:
                printf(" Invalid operator\n");
        }
        printf("Enter expression separated with space (q to quit): ");
    }
}

double add(double a, double b)
{
    return a + b;
}

double subtract(double a, double b)
{
    return a - b;
}

```

```
double multiply(double a, double b)
{
    return a * b;
}
```

```
double divide(double a, double b)
{
    if(b == 0)
    {
        printf(" Can't divide by");
        return 0;
    }
    else
        return a / b;
}
```

```
double exponentiate(double a, double b)
{
    return pow(a, b);
}
```

Chapter 6

1.

```
#include <stdio.h>
```

```
void main(void)
{
    int a[5], x;

    printf("Input five numbers\n");
    for(x=0; x<=4; x++)
    {
        printf("%d", x);
        scanf("%d", &a[x]);
    }
    printf("Here they are: ");
    for(x=4; x>=0; x--)
        printf("%4d", x);
}
```

2.

```
#include <stdio.h>
```

```
void main(void)
{
    float value[5], mean = 0.0;
    int count;
```



```
int pos = 0;

gets(string);
while(string[pos] != '\0')
{
    if(isvowel(string[pos]))
    {
        putchar('<');
        putchar(toupper(string[pos]));
    }
    else
        putchar(string[pos]);
    ++pos;
}

int isvowel(char character)
{
    int result;

    switch(toupper(character))
    {
        case 'A':
        case 'E':
        case 'I':
        case 'O':
        case 'U':
            result = 1;
            break;
        default:
            result = 0;
    }
    return result;
}
```

2.

```
#include <stdio.h>
#include <ctype.h>
#define CHRS 80

void caps(char string[]);

void main(void)
{
    char string[CHRS];

    while(printf("Your input?"), gets(string), string[0] != '\0')
    {
        caps(string);
        printf("The output: %s\n", string);
    }
}
```

```
void caps(char string[])
{   int pos=0;
    while(string[pos] != '\0')
    {
        string[pos] = toupper(string[pos]);
        pos++;
    }
}
```

3.

```
#include <stdio.h>
```

```
#include <string.h>
```

```
void main(void)
```

```
{
    char string1[20], string2[20];

    printf("FIRST STRING:");
    fflush(stdin);
    gets(string1);
    printf("SECOND STRING:");
    fflush(stdin);
    gets(string2);

    if(strcmp(string1, string2) == 0)
        printf("%s EQUAL %s", string1, string2);
    else if(strcmp(string1, string2) > 0)
        printf("%s IS GREATER THAN %s", string1, string2);
    else
        printf("%s IS GREATER THAN %s", string2, string1);
}
```

4.

```
#include <stdio.h>
```

```
#include <ctype.h>
```

```
void main(void)
```

```
{   char initial, name[50];
    int x = 0;

    printf("Name?");
    fflush(stdin);
    gets(name);
    initial = name[0];

    while(name[x] != '\0');
    {
        if(name[x - 1] == ' ')
            initial = name[x];
        x++;
    }
}
```

```
if(toupper(initial) >= 'G' && toupper(initial) <= 'L')
    printf("%s is a good person\n", name);
else
    printf("%s is a bad person\n", name);
}
```

Informatics Virtual Campus



IVC is an interactive system designed exclusively for Informatics students worldwide! It allows students to gain online access to the wide range of resources and features available **anytime, anywhere, 24hours per day, and 7days a week!**

In order to access IVC, students need to log-in with their user ID and password.

Among the many features students get to enjoy are e-resources, message boards, and online chat and forum. Apart from that, IVC also allows students to download assignments and notes, print examination entry cards and even view assessment results.

With IVC, students will also be able to widen their circle of friends via the discussion and chat rooms by getting to know other campus mates from around the world. They can get updates on the latest campus news, exchange views, and chat about common interest with anyone and everyone, anywhere.

Among the value-added services provided through IVC are **global orientation** and **e-revision**.

Global orientation is where new students from around the world gather at the same time for briefings on the programmes they undertake as well as the services offered by Informatics.

e-Revision on the other hand is a scheduled live text chat session where students and facilitators meet online to discuss on assessed topics pre-exams. Students can also post questions and get facilitators to respond immediately. Besides that, students can obtain revision notes, and explore interactive exam techniques and test banks all from this platform.

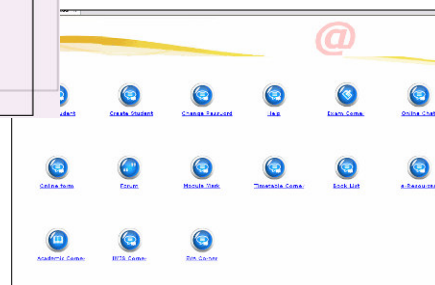
In a nutshell, IVC is there to ensure that students receive the best academic support they can get during the course of their education pursuit with Informatics. It could give students the needed boost to excel well beyond expectations.

IVC at a Glance

- Accessible 24 hours per day, 7 days a week
- Notes and assignment downloads
- Print exams entry cards
- View assessment results
- Message boards
- Chat and discussion rooms
- Global orientation
- E-Revision



Screen shot of IVC login page



Screen shot of IVC menu