



South Valley University

Faculty of Commerce

Statistical Computer Programming

2022-2023

جامعة جنوب الوادي
South Valley University

Prepared By

Dr\ Saddam Hussein Ahmed

PhD- University of Lincoln- UK

INTRODUCTION

Computer programming were and still a major milestone in the last era of the human history. It changed the human life for better by minimizing the effort to do the most complex and timely jobs in a matter of seconds or even less.

The knowledge of computer programming is essential to everyone, specially 'students' who wants to build their skills for a better chance in the jobs market upon their graduation. Developing a program involves a series of steps. The programmer defines a problem, plans a solution, codes the program, tests the program and, finally, documents the program. Usually, the programmer defines what he knows and the objective, selects a program to use, debugs the program in stages after completion to ensure no errors are introduced and then documents the design, development and testing of the program. With the ever-changing face of computer technology, programming is an exciting and always challenging environment that few programmers ever dream of leaving. Furthermore, programming is a platform to showcase creativity, especially in problem-solving and entertainment. Programming develops new business ideas to resolve a particular problem.

This course starts with a gentle introduction to learn programming which is designing prototypes using the flow chart model. This helps the student to understand the various

programming concepts. From an applicability perspective, this course will study computer programming using one of the most popular languages 'C++'. However, the course will be driven by the commercial and statistical needs to enrich your knowledge and support your future commercial career.

We will also study the 'MATLAB' programming platform and the SPSS software. Both are statistical software's that offers wide range of ready available tools to easily solve the various statistical problems that would further polish your thinking and skills.

Dr\ Saddam Hussein Ahmed

Associate Professor

PhD- University of Lincoln- UK



Table of Contents

Part 1

Chapter 1 Algorithm and Flow Chart	5
------------------------------------	---

Part 2

Chapter 1 C++ Programming Basics	25
Chapter 2 Using Microsoft visual studio.	59
Chapter 3 Loops and Decisions in C++.	77



Part 3

Chapter 1 MATLAB programming: An introduction	142
Chapter 2 Plotting in MATLAB	142
Chapter 3 Vectors and matrices in MATLAB	150
Chapter 4 Statistical MATLAB Applications	170



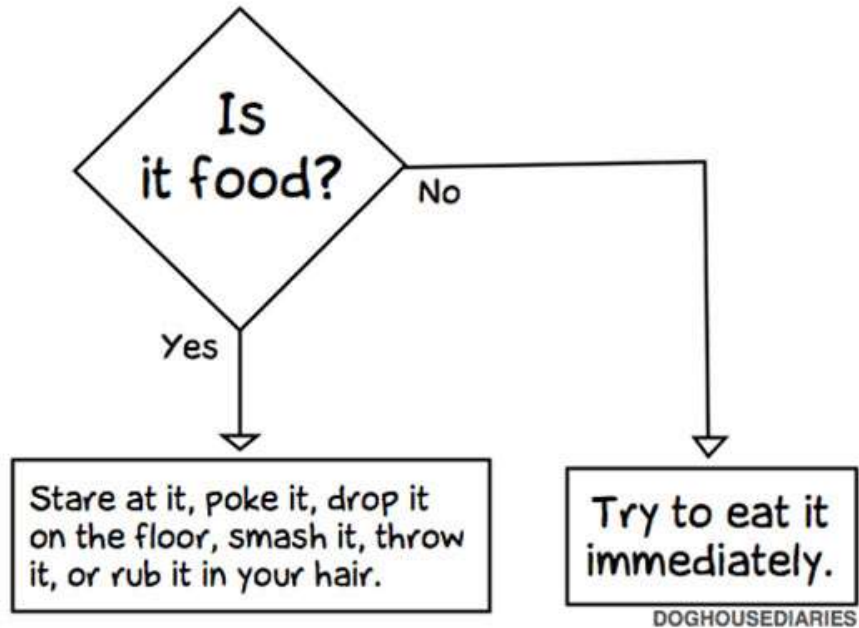
Part 4

Chapter 1 An introduction to SPSS	189
Previous Exams	147
References	130
Communication with Lecturer	251



Part 1

How Babies Make Decisions



جامعة جنوب الوادي

Chapter 1 Algorithm and Flow Chart

جامعة جنوب الوادي
South Valley University

An algorithm is a set of instructions, sometimes called a procedure or a function that is used to perform a certain task. This can be a simple process, such as adding two numbers together, or a complex function, such as adding effects to an image. For example, in order to sharpen a digital photo, the algorithm would need to process each pixel in the image and determine which ones to change and how much to change them in order to make the image look sharper. Most computer programmers spend a large percentage of their time creating algorithms. (The rest of their time is spent debugging the algorithms that do not work properly.) The goal is to create efficient algorithms that do not waste more computer resources (such as RAM and CPU time) than necessary. This can be difficult, because an algorithm that performs well on one set of data may perform poorly on other data. As you might guess, poorly written algorithms can cause programs to run slowly and even crash. Therefore, software updates are often introduced, touting “improved stability and performance”. While this sounds impressive, it also means that the algorithms in the previous versions of the software were not written as well as they could have been.

An algorithm generally takes some input, carries out a number of effective steps in a finite Given a list of numbers, you can easily order them from largest to smallest with the simple instruction “Sort these numbers.” A computer, however, needs more detail to sort numbers. It must be told to search for the smallest number, how to find the smallest number, how to compare numbers together, etc. The operation “Sort these numbers” is ambiguous to a computer because the computer has no basic operations for sorting. Basic operations used for writing algorithms are known as primitive operations or primitives. When an algorithm is written in computer primitives, then the algorithm is unambiguous and the computer can execute it.

Algorithms should be composed of a finite number of operations and they should complete their execution in a finite amount of time. Suppose we wanted to write an algorithm to print all the integers greater than 1. Our steps might look something like this:

1. Print the number
2. Print the number 5
3. Print the number 7

While our algorithm seems to be pretty clear, we have two problems. First, the algorithm must have an infinite number of steps because there are an infinite number of integers greater than one. Second, the algorithm will run forever trying to count to infinity. These problems violate our definition that an algorithm must halt in a finite amount of time. Every algorithm must reach some operation that tells it to stop.

Advantages and Disadvantages of Algorithms

Advantages The use of algorithms provides a number of advantages. One of these advantages is in the development of the procedure itself, which involves identification of the processes, major decision points, and variables necessary to solve the problem. Developing an algorithm allows and even forces examination of the solution process in a rational manner. Identification of the processes and decision points reduces the task into a series of smaller steps of more manageable size. Problems that would be difficult or impossible to solve wholesale can be approached as a series of small, solvable subproblems. The required specification aids in the identification and reduction of subconscious biases. By using an algorithm, decision-making becomes a more rational process. In addition to making the process more rational, use of an algorithm will make the process more efficient and more consistent. Efficiency is an inherent result of the analysis

and specification process. Consistency comes from both the use of the same specified process and increased skill in applying the process. An algorithm serves as a mnemonic device and helps ensure that variables or parts of the problem are not ignored. Presenting the solution process as an algorithm allows more precise communication. **Finally**, separation of the procedure steps facilitates division of labor and development of expertise.

A final benefit of the use of an algorithm comes from the improvement it makes possible. If the problem solver does not know what was done, he or she will not know what was done wrong. As time goes by and results are compared with goals, the existence of a specified solution process allows identification of weaknesses and errors in the process. Reduction of a task to a specified set of steps or algorithm is an important part of analysis, control, and evaluation. Disadvantages One disadvantage of algorithms is that they always terminate, which means there are some computational procedures—occasionally even useful ones—which are not algorithms. Furthermore, all computational procedures, whether they terminate or not, can only give computable results, so you cannot, for example, design a program which determines a busy beaver number more quickly than could be done by actually running the associated types of turing machines.

Disadvantages One disadvantage of algorithms is that they always terminate, which means there are some computational procedures—occasionally even useful ones—which are not algorithms. Furthermore, all computational procedures, whether they terminate or not, can only give computable results, so you cannot, for example, design a program which determines a busy beaver number more quickly than could be done by actually running the associated types of turing machines.

FLOW CHART

A flow chart, or flow diagram, is a graphical representation of a process or system that details the sequencing of steps required to create output. A typical flow chart uses a set of basic symbols to represent various functions, and shows the sequence and interconnection of functions with lines and arrows. Flow charts can be used to document virtually any type of business system, from the movement of materials through machinery in a manufacturing operation to the flow of applicant information through the hiring process in a human resources department. 86 Computer Basics with Office Automation.

Each flow chart is concerned with one particular process or system. It begins with the input of data or materials into the system and traces all the procedures needed to convert the input into its final output form. Specialized flow chart symbols show the processes that take place, the actions that are performed in each step, and the relationship between various steps. Flow charts can include different levels of detail as needed, from a high-level overview of an entire system to a detailed diagram of one component process within a larger system. In any case, the flow chart shows the overall structure of the process or system, traces the flow of information and work through it, and highlights key processing and decision points.

Flow charts are an important tool for the improvement of processes. By providing a graphical representation, they help project teams to identify the different elements of a process and understand the interrelationships among the various steps. Flow charts may also be used to gather information and data about a process as an aid to decision-making or performance evaluation. For example, the owner of a small advertising agency who hopes to reduce the time involved in creating a print ad might be able to use a flow chart of the process to identify and eliminate

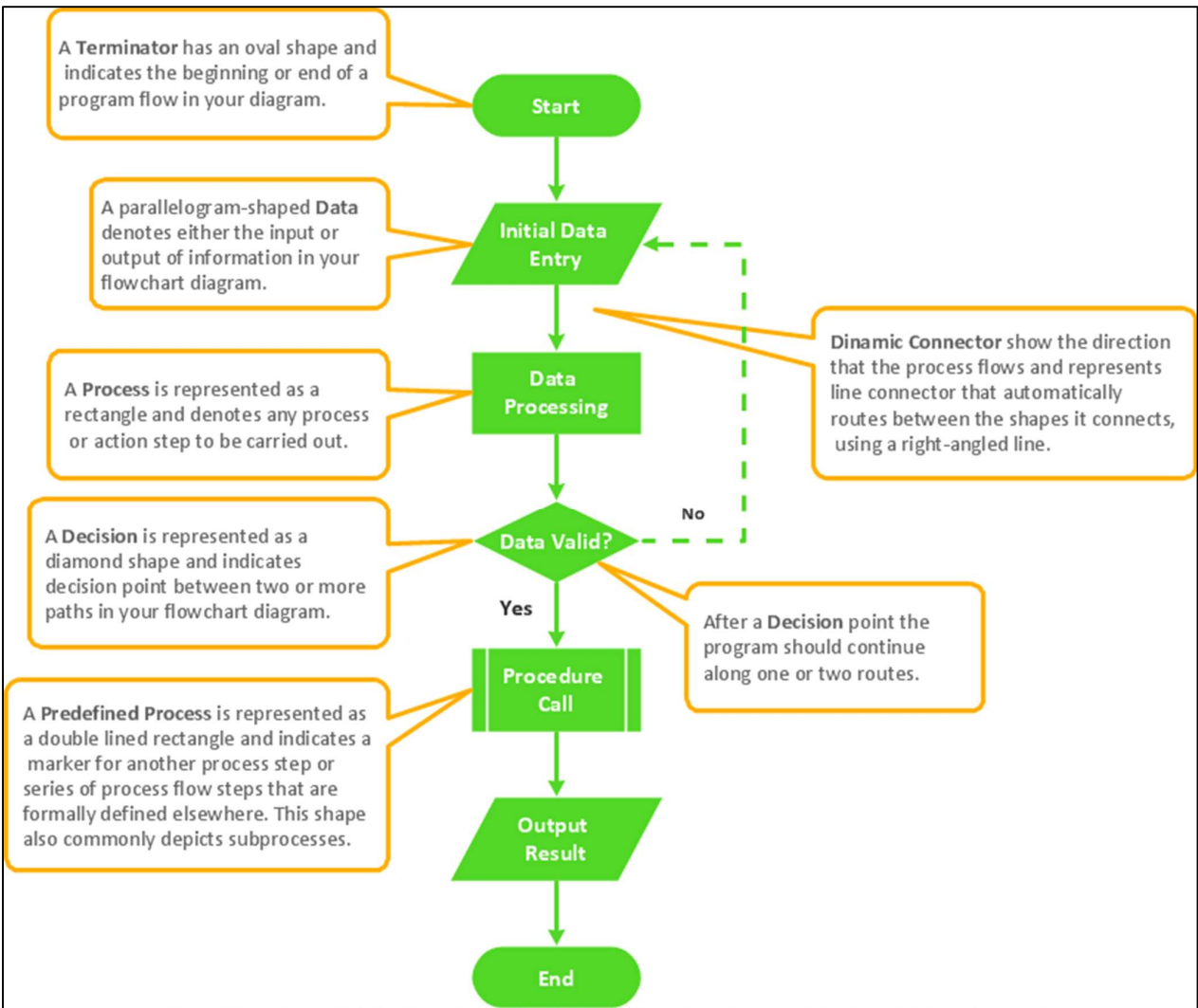
unnecessary steps. Though flow charts are relatively old design tools, they remain popular among computer programmers working on systems analysis and design. In recent years, many software programs have been developed to assist businesspeople in creating flow charts.

Constructing Flow Charts

Flow charts typically utilize specialized symbols. Some of the main symbols that are used to construct flow charts include:

- A round-edged rectangle to represent starting and ending activities, which are sometimes referred to as terminal activities.
- A rectangle to represent an activity or step.
- Each step or activity within a process is indicated by a single rectangle, which is known as an activity or process symbol.
- A diamond to signify a decision point. The question to be answered or decision to be made is written inside the diamond, which is known as a decision symbol. The answer determines the path that will be taken as a next step.
- Flow lines show the progression or transition from one step to another.

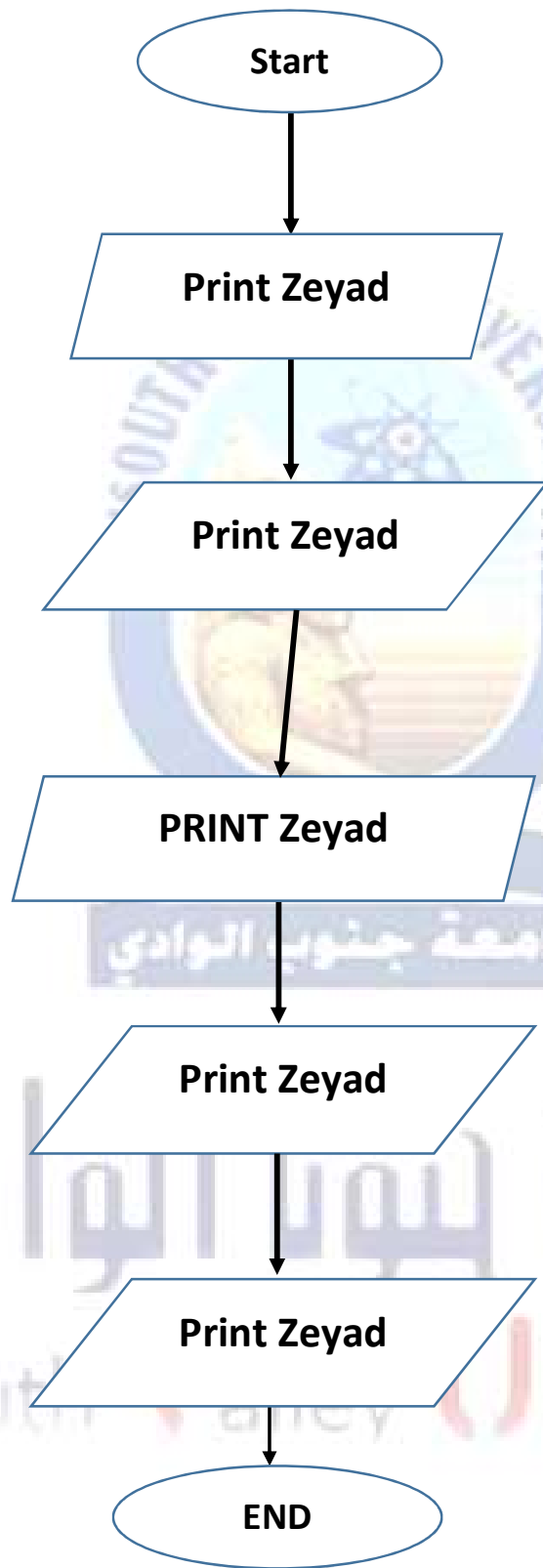
Constructing a flow chart involves the following main steps: (1) Define the process and identify the scope of the flow diagram; (2) Identify project team members that are to be involved in the construction of the process flow diagram; (3) Define the different steps involved in the process and the interrelationships between the different steps (all team members should help develop and agree upon the different steps for the process); (4) Finalize the diagram, involving other concerned individuals as needed and making any modifications necessary; and (5) Use the flow diagram and continuously update it as needed.



Flow Chart EXAMPLES

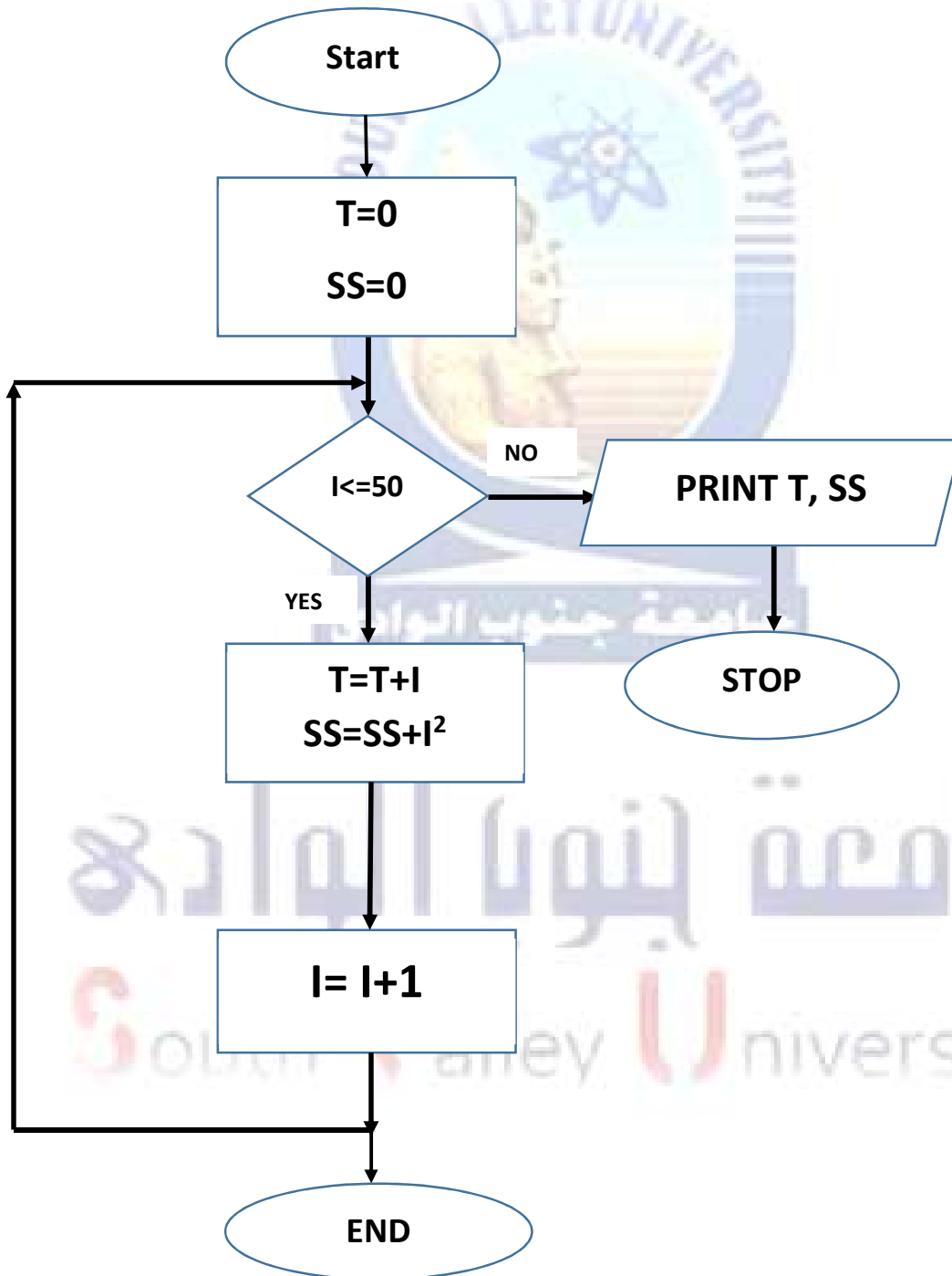
EXAMPLE (1)

Draw a flow chart to print your name 5 times.



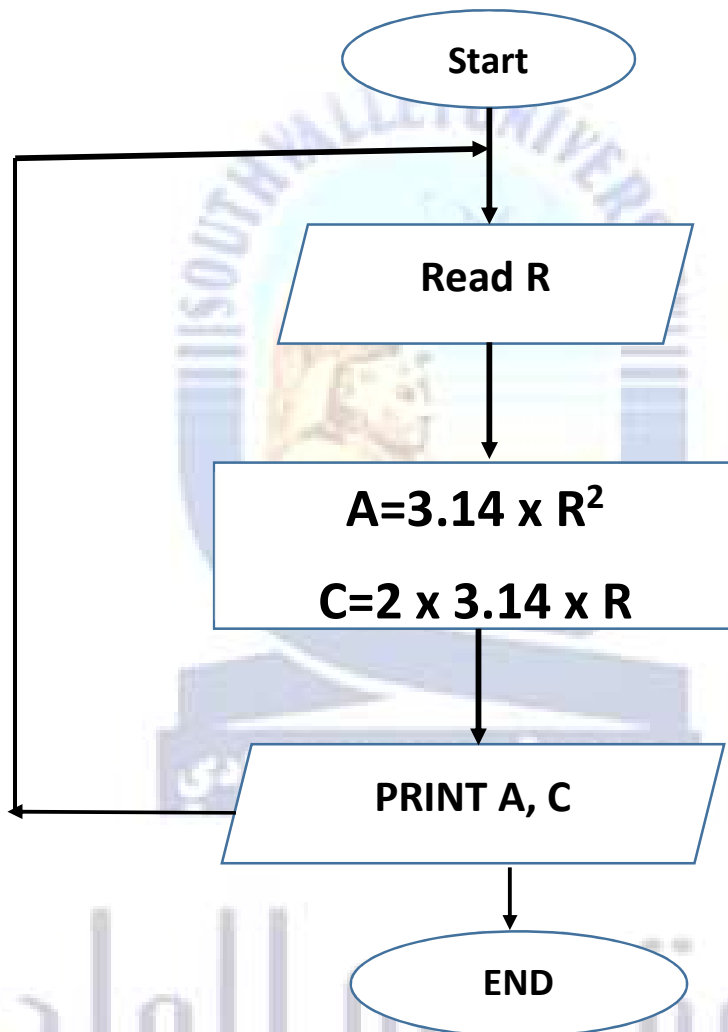
EXAMPLE (2)

Draw a flow chart to find the summation of the numbers from 1 to 50.



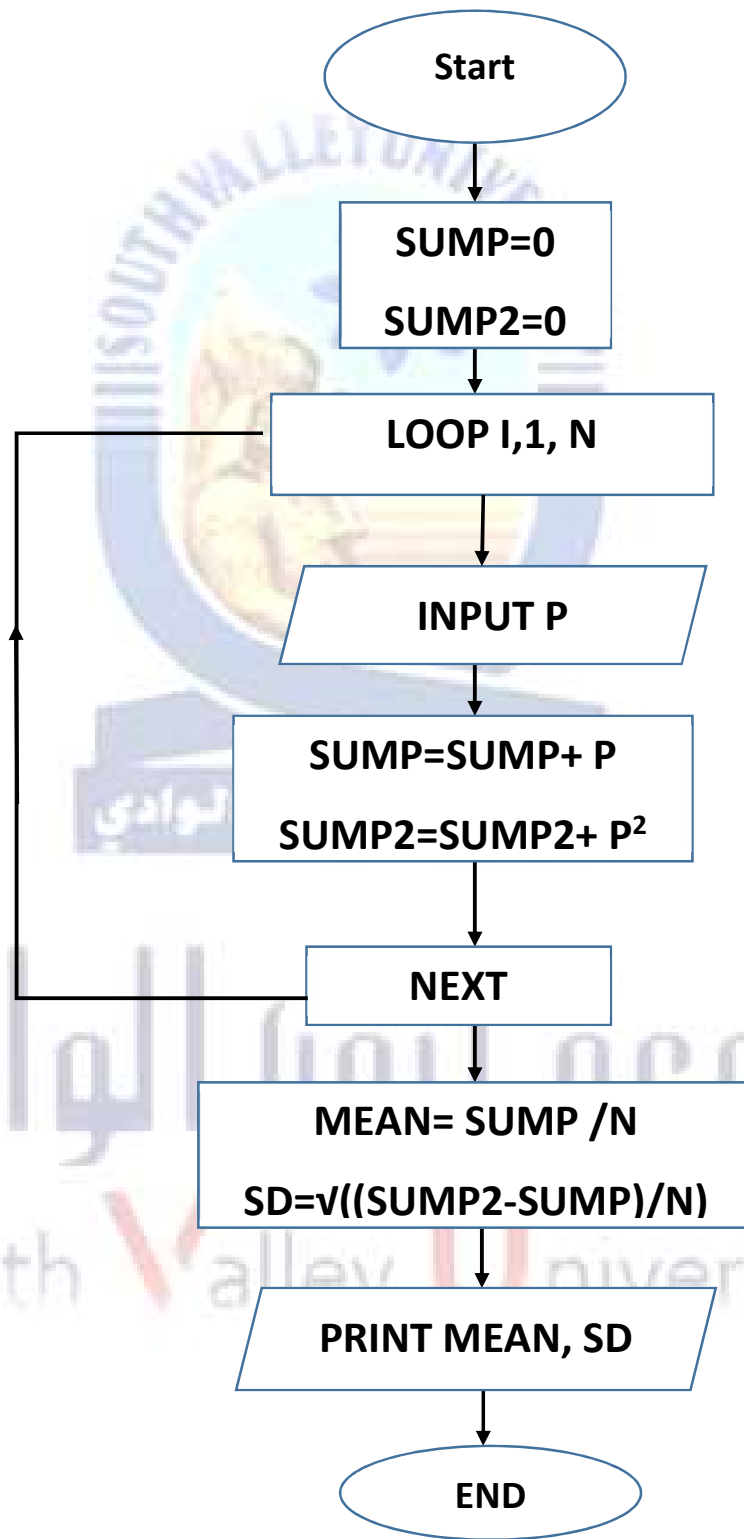
EXAMPLE (3)

Draw a flow chart to find the area and circumference of a circle.



EXAMPLE (4)

Draw a flow chart to find the mean and standard deviation for a group of N numbers.



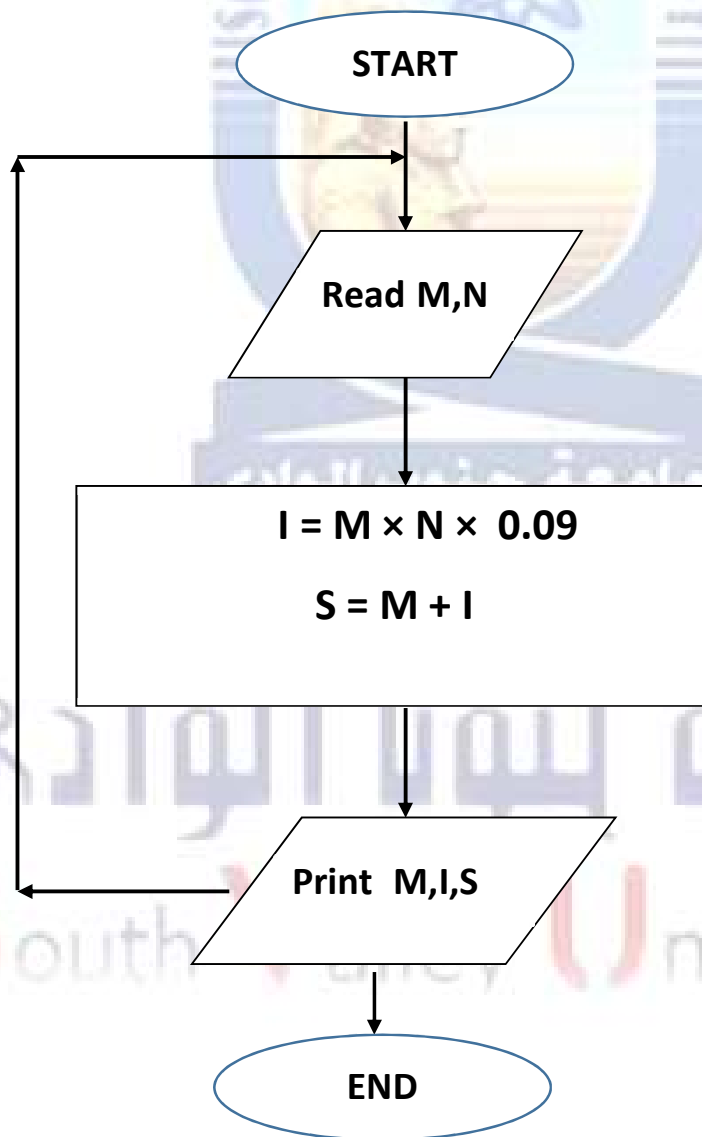
EXAMPLE (5)

Draw a flow chart to find the compound interest on a depositor money, known that the interest is 9%.

Suppose that the original amount of money is **M**.

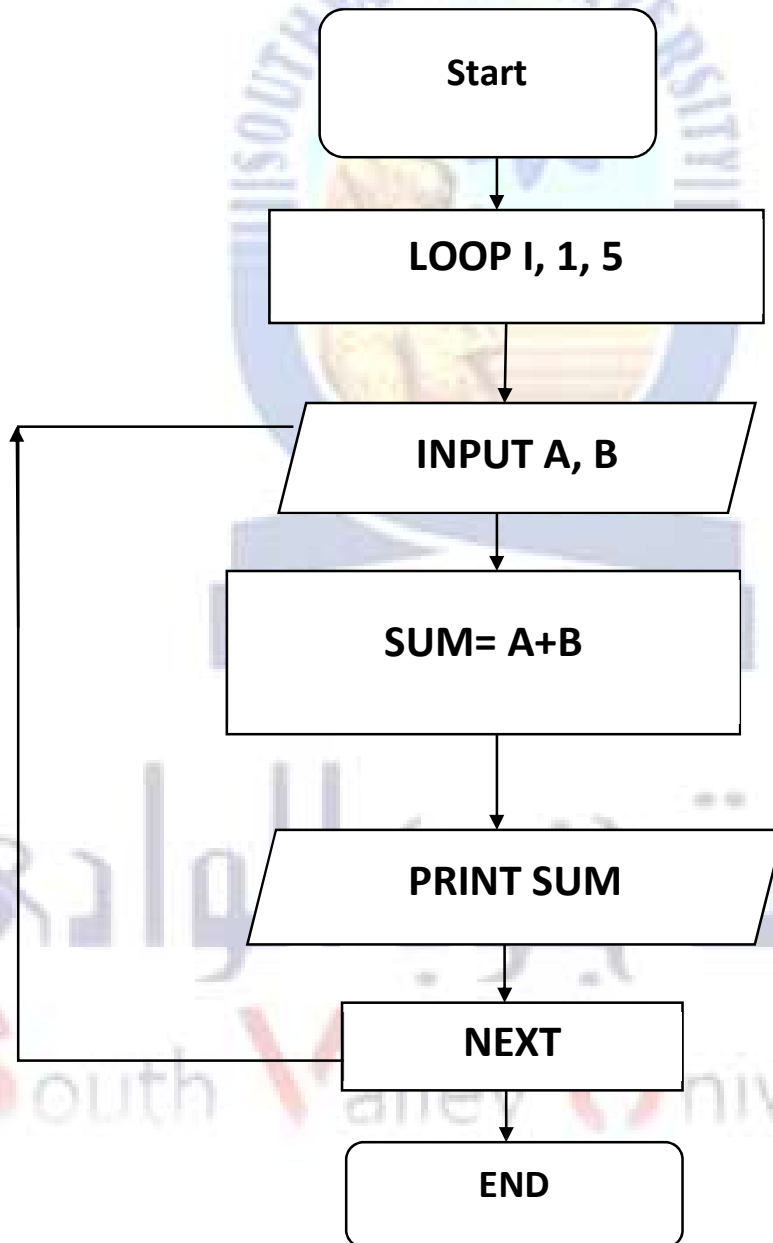
The number of years is **N**.

The interest **$I = M \times N \times 9\%$** .



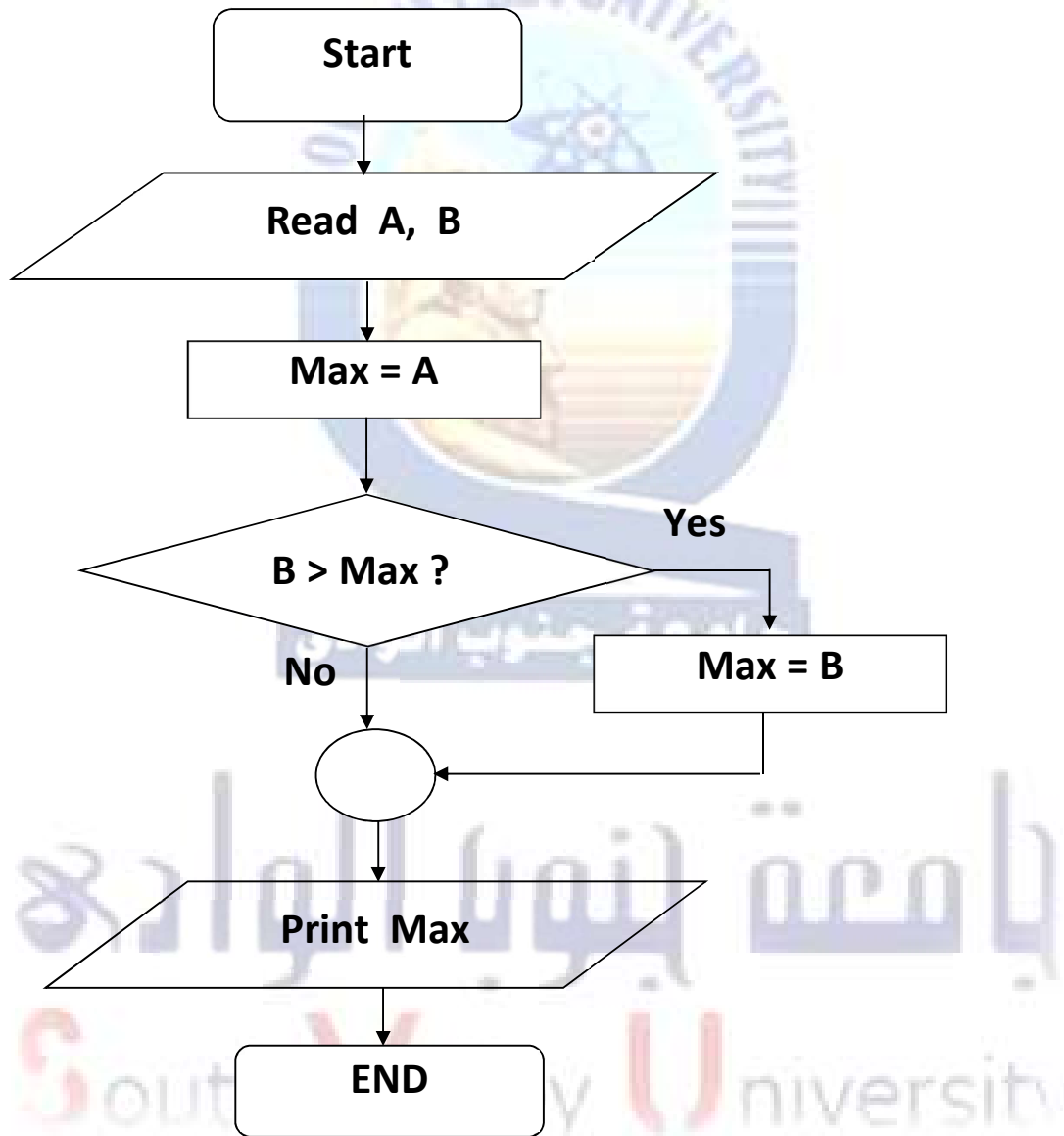
EXAMPLE (6)

Draw a flow chart to find the sum of five different number pairs, using the loops technique.



EXAMPLE (7)

Draw a flow chart to find the bigger value among two different numbers A and B.



EXAMPLE (8)

A certain company would like to buy a group of new computers. A couple of suppliers presented some offers with specific prices. Draw a flow chart that find the offer with the minimum price.

Presume that the offers are **N**, and they are listed as follows:

X(1),

X(2),

X(3),

X(4),

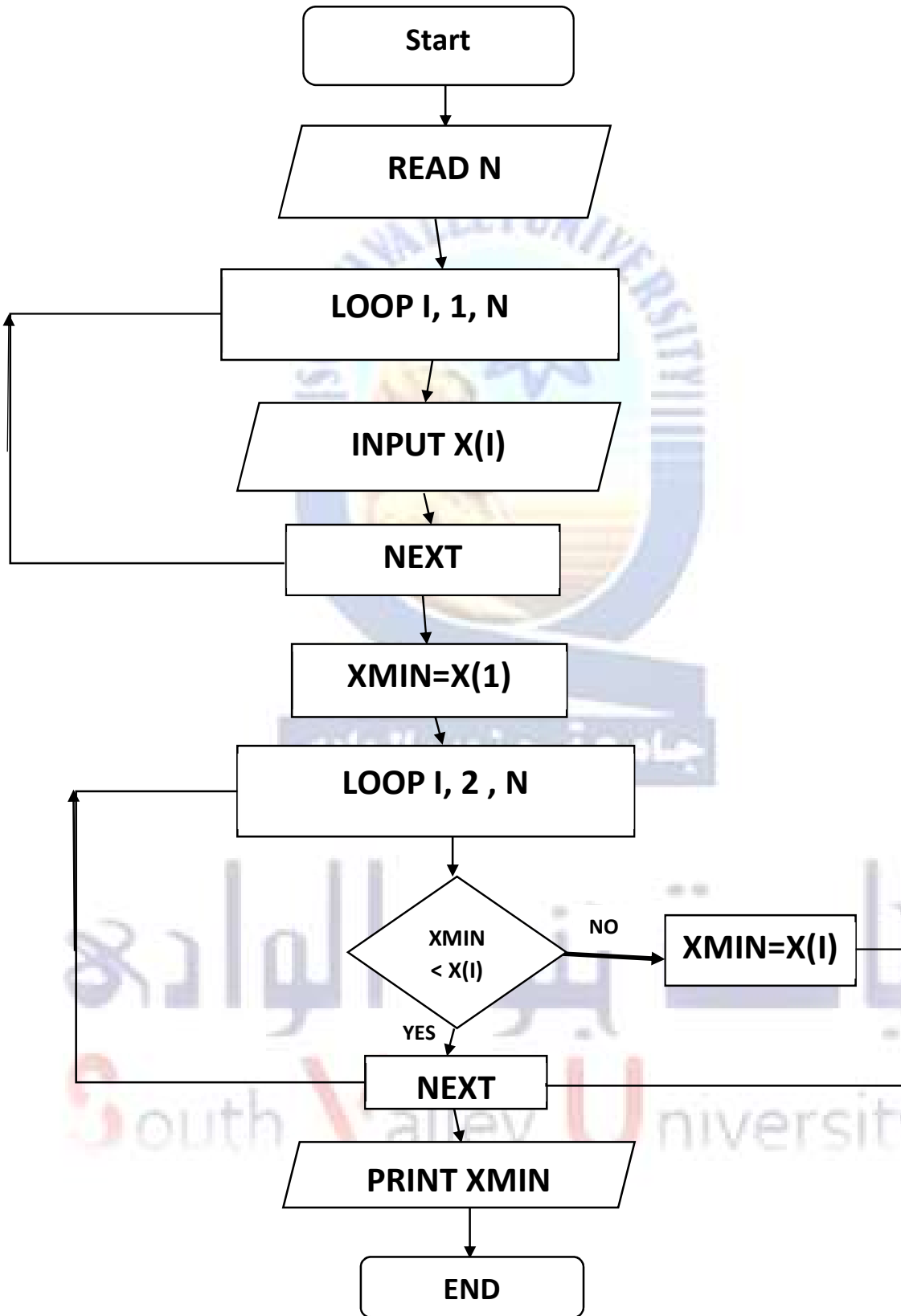
X(5),

X(6),

.....,

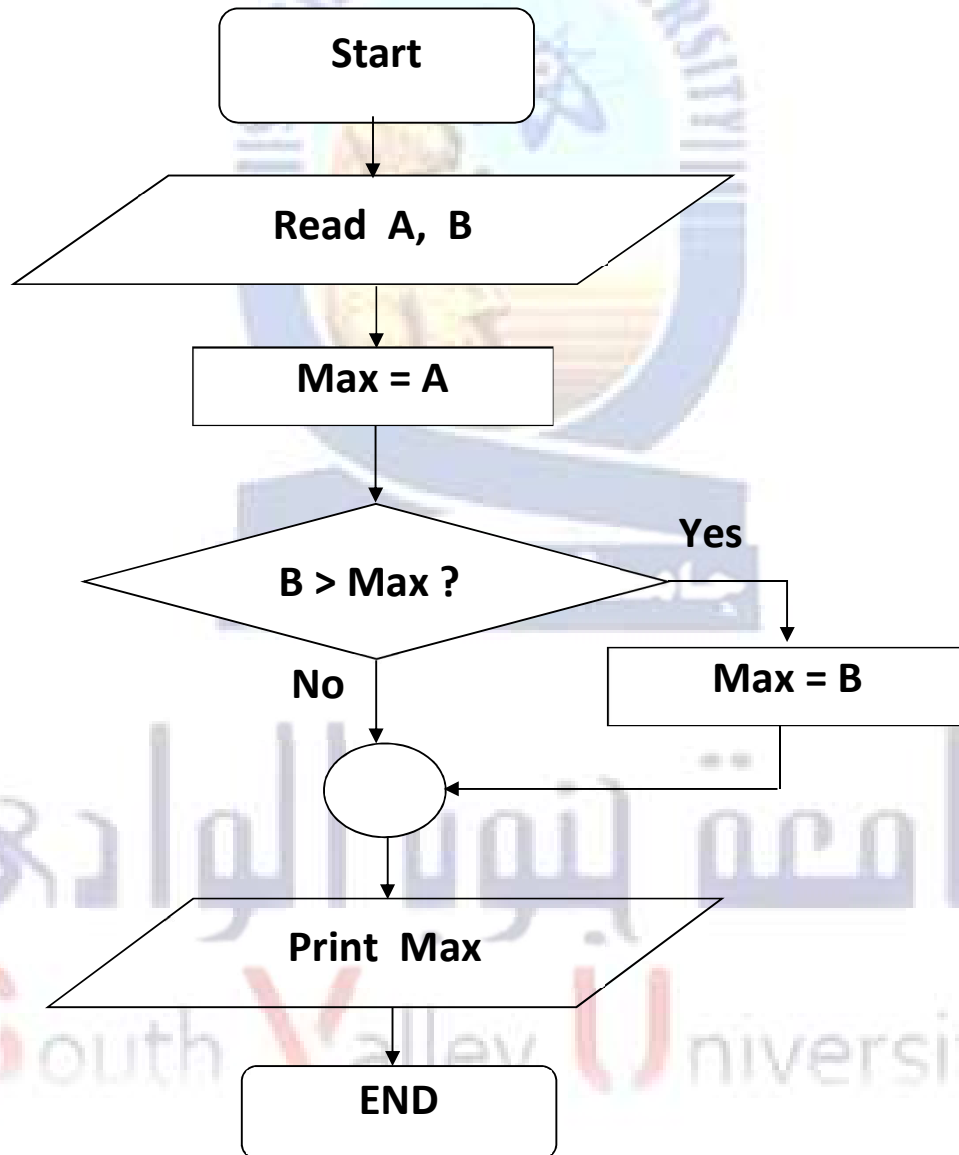
X(N)

Suppose that the minimum offer price is **XMIN**

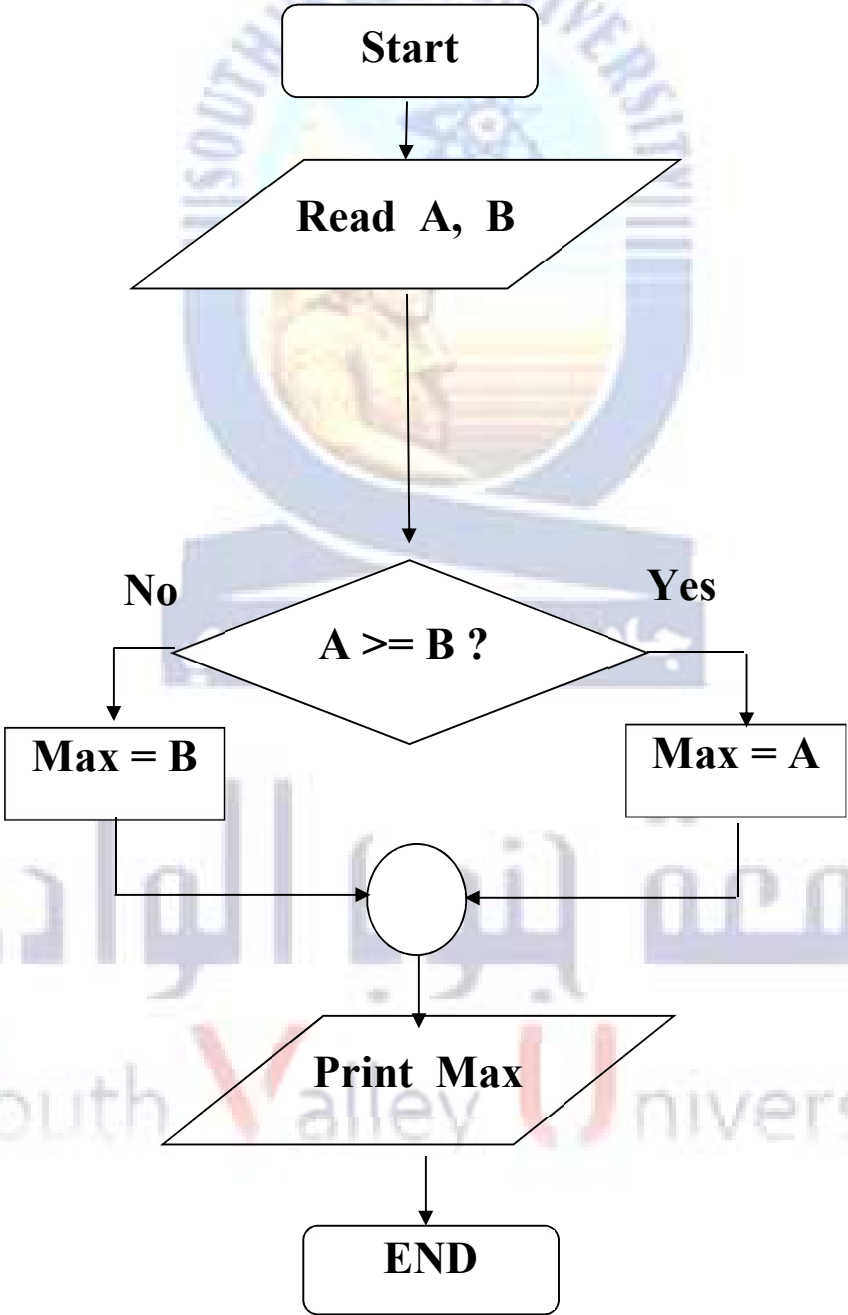


EXAMPLE (9)

Draw a flow chart that finds the minimum of two numbers.



Another solution:



Exercises

1- Draw a flow chart that implements the following function:

$$Abs(x) = |x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

2- Draw a flow chart that implements a program that reads an input number **N** and examines if the number is odd or even.

3- Draw a flow chart that implements the following function:

$$f(x) = \begin{cases} 2X^2 + 3 & X > 0 \\ X + 8 & X = 0 \\ 3X - 9 & X < 0 \end{cases}$$

4- Draw a flow chart that find the odd numbers between 1 and 500.

5- Draw a flow chart to determine if a given number is prime or not.

6- Draw a flow chart that finds the solution of a quadratic equation.

Exam question 2019

Study the flowchart in (figure 1) then answer questions from 12 to 18:

[12] The number of inputs in this flowchart:

- (A) 1
- (B) 0
- (C) 2
- (D) Unknown

[13] The flowchart prints the numbers range :

- (A) 0 to 13
- (B) -1 to 13
- (C) 0 to 14
- (D) -1 to 12

[14] The number of iterations the LOOP executes are:

- (A) 0
- (B) No correct answer.
- (C) 14
- (D) 15

[15] If the LOOP STEP changed to 4, the number of iterations of the LOOP will be:

- (A) 4
- (B) 5
- (C) 3
- (D) 0

[16] If the LOOP STEP changed to 5, the number of iterations of the LOOP will be:

- (A) 4
- (B) 5
- (C) 3
- (D) 0

[17] The last number to be printed by the flowchart is

- (A) 12
- (B) 13
- (C) 11
- (D) 14

[18] The last Number to be printed be the LOOP is:

- (A) 12
- (B) 13
- (C) 11
- (D) 14

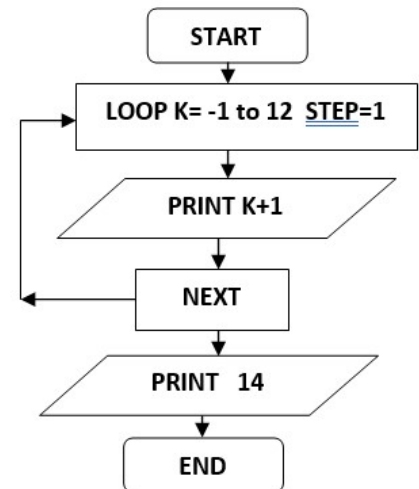


Figure (1)

Part 2



Chapter 1

C++ Programming Basics

What are computer programming languages?

A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. The term programming language usually refers to high-level languages, such as BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal. Each language has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

High-level programming languages, while simple compared to human languages, are more complex than the languages the computer actually understands, called machine languages. Each different type of CPU has its own unique machine language.

Lying between machine languages and high-level languages are languages called assembly languages. Assembly languages are similar to machine languages, but they are much easier to program in because they allow a programmer to substitute names for numbers. Machine languages consist of numbers only.

Lying above high-level languages are languages called fourth-generation languages (usually abbreviated 4GL). 4GLs are far removed from machine languages and represent the class of computer languages closest to human languages.

Regardless of what language you use, you eventually need to convert your program into machine language so that the computer can understand it. There are two ways to do this:

- **Compile the program**
- **Interpret the program**

The question of which language is best is one that consumes a lot of time and energy among computer professionals. Every language has its strengths and weaknesses. For example, FORTRAN is a particularly good language for

processing numerical data, but it does not lend itself very well to organizing large programs. Pascal is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features, but it is complex and difficult to learn.

The choice of which language to use depends on the type of computer the program is to run on, what sort of program it is, and the expertise of the programmer. However, in this course we will target C++ programming language.

History of C++ :

C++ is a general-purpose programming language created by Bjarne **Stroustrup** as an extension of the C programming language, or "C with Classes". The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms.

C++ was designed with a bias toward system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights. C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications, including desktop applications, video games, servers (e.g. e-commerce, Web search, or SQL servers), and performance-critical applications (e.g. telephone switches or space probes).

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2020 as ISO/IEC 14882:2020 (informally known as C++20). The C++ programming language was initially standardized in 1998 as ISO/IEC 14882:1998, which was then amended by the C++03, C++11, C++14, and C++17 standards. The current C++20 standard supersedes these with new features and an enlarged standard library. Before the initial standardization in 1998, C++ was developed by Danish computer scientist Bjarne **Stroustrup** at Bell Labs since 1979 as an extension of the C language; he wanted an efficient and flexible language similar to C that also provided high-level features for program organization. Since 2012, C++ is on a three-year release schedule, with C++23 the next planned standard.

Your first C++ program

Before you can develop even the most basic programmes in any language, you must first learn some fundamentals. Three such fundamentals are introduced in this chapter: basic programme building, variables, and input/output (I/O). It also covers comments, arithmetic operators, the increment operator, data conversion, and library functions, among other language features.

Although these topics are not theoretically tough, you may find that C++'s style is more austere than, say, BASIC or Pascal. A C++ programme may look more like a mathematical formula than a computer programme until you discover what it's all about. This isn't something to be concerned about. As you become more comfortable with C++, you'll notice that it appears less intimidating, whereas other languages appear too complicated and verbose.

Let us look at a very simple C++ program called **BASIC.cpp**. It simply prints a sentence on the screen. Here it is:

```
1 #include <iostream>
2 //BASIC.cpp
3
4 using namespace std;
5 int main()
6 {
7     cout<< "HELLO Third Year Students \n";
8     return 0;
9 }
```

Despite its small size, this program demonstrates a great deal about the construction of C++ programs. Let us examine it in detail.

Functions in C++ Programs

Functions are one of the fundamental building blocks of C++. The **BASIC** program consists almost entirely of a single function called **main()**. The only parts of this program that are not part of the function are the first two lines—the ones that start with **#include** and **using**.

Function Name

A function is distinguished by the parenthesis that follow the word **main**. The compiler would assume that **main** refers to a variable or another programme element without the parenthesis. We'll use the same convention as C++ when discussing functions in the text: Following the

function name, we'll use parentheses. The term **int** before the function name denotes that the function's return value is of the type int.

Function body is always delimited with Braces

A function's body is encircled by braces (sometimes called curly brackets). In several languages, these braces serve the same purpose as the BEGIN and END keywords: They are used to encircle or delimit a group of programme statements. This pair of braces must be used around the function body in every function. The function body in this example contains only two statements: the line beginning with **cout** and the line beginning with return. A function body, on the other hand, can include a large number of statements.

main() must be in every program

The first statement performed when you run a C++ programme is at the start of a function called **main()**. (At least, that's what the console mode programmes in this book are like.) Although the programme may have a large number of functions, classes, and other programme pieces, control is always passed to **main** on startup. If you don't have a function called **main()** in your programme, you'll get an error when you run it. As we'll see later, **main()** in most C++ programmes invokes member functions in various objects to carry out the program's actual job. Other standalone functions may be called from the **main()** function.

There exist two statements in the **BASIC** program: the line

```
cout << "Hello Third Year Students \n";
```

and the return statement

```
return 0;
```

These are called program statements. The first statement tells the computer to display the quoted phrase. Most statements tell the computer to do something. In this respect, statements in C++ are similar to statements in other languages.

The statement comes to a close with a **semicolon**. This is an important component of the syntax, yet it's easy to overlook. The end of a statement is signalled by the end of the line in certain languages, but not in C++. If you omit the semicolon, the compiler will usually give you an error message.

Terminating a program

return 0; is the last statement in the function body. This instructs **main()** to **return 0** to the caller, which in this case is the operating system or compiler (normal program termination) .

Programming and Whitespace

The end of a line isn't significant to a C++ compiler, as we previously stated. Whitespace is virtually completely ignored by the compiler. Spaces, carriage returns, linefeeds, tabs, vertical tabs, and form feeds are all examples of whitespace. The compiler is oblivious to these characters. You can insert many statements on a single line, separated by any number of spaces or tabs, or run a statement across two or more lines. To the compiler, it's all the same. As a result, the **BASIC** programme may be written as follows:

```
1 #include <iostream>
2 using
3 namespace std;
4 int main () { cout
5     <<
    "Hello Third Year Students \n"
```

```
6 ; return
7 0;}
```

There are several exceptions to the rule that whitespace is invisible to the compiler. The first line of the program, starting with **#include**, is a pre-processor directive, which must be written on one line. Also, string constants, such as “Every age has a language of its own”, cannot be broken into separate lines.

Printing Output Using cout

As you have seen, the statement:

```
cout << “Hello Third Year Students \n”;
```

causes the phrase in quotation marks to be displayed on the screen. The identifier **cout** (pronounced “**C out**”) is actually an object. It is predefined in C++ to correspond to the standard output stream. A stream is an abstraction that refers to a flow of data. The standard output stream normally flows to the screen display—although it can be redirected to other output devices. The operator << is called the **insertion** or put to operator. It directs the contents of the variable on its right to the object on its left. In **BASIC** it directs the string constant “Hello Third Year Students \n” to cout, which sends it to the display.

Although the concepts behind the use of **cout** and << may be unclear at this point, using them is easy. They’ll appear in almost every example program. Figure 1 shows the result of using **cout** and the insertion operator <<.

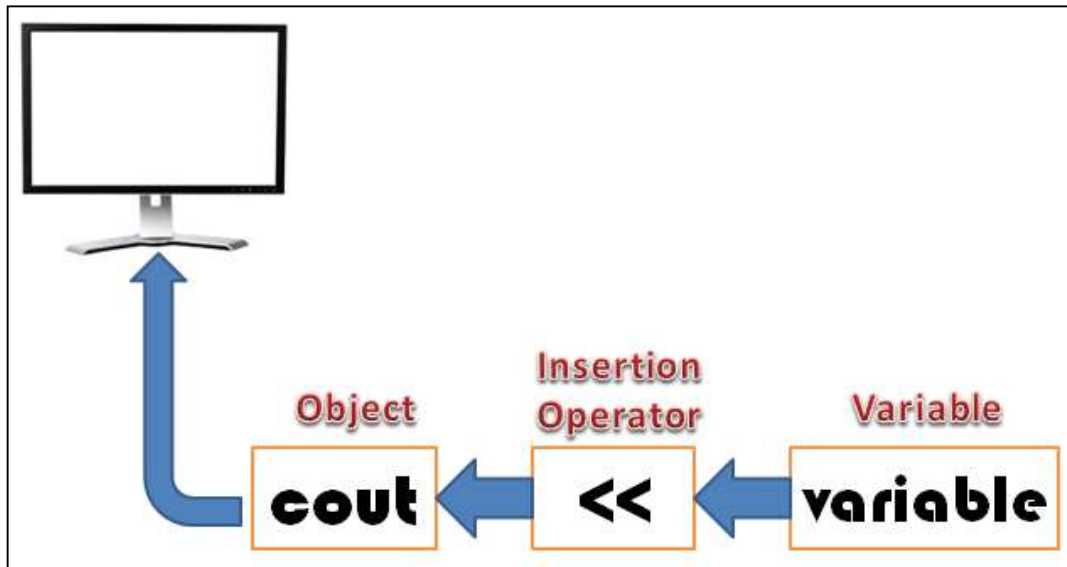


Figure 1 Output with cout in C++ program.

C++ and String Constants

The phrase **"Hello Third Year Students \n"** which is enclosed in quotation marks, is an example of a string constant. A constant, unlike a variable, cannot be given a new value while the programme is running. Its value is determined at the time the programme is developed and remains constant during the program's life.

Escape sequence

The **'\n'** character at the end of the string constant is an example of an escape sequence. It causes the next text output to be displayed on a new line.

"Press any key to continue"

The string "Press any key to continue," is inserted by some compilers for display after the program terminates. You cannot delete it.

Directives

The two lines that begin the **BASIC** program are directives. The first is a **pre-processor** directive, and the second is a using directive. They occupy a sort of gray area: They're not part of the basic C++ language, but they're necessary anyway.

Pre-processor Directives

The first line of the **BASIC** programme **#include <iostream>** is a programme statement, but it is not. It's not part of a function body, and it doesn't end with a semicolon like programme statements must. Instead, a number sign (#) is used as the first character. A **pre-processor directive** is what it's called. Remember that programme statements are computer instructions for doing things like adding two numbers or printing a sentence. On the other hand, a pre-processor directive is a compiler command. The compiler has a section called the **pre-processor** deals with these directives before it begins the real compilation process. The **pre-processor** directive **#include** tells the compiler to insert another file into your source file. In effect, the **#include** directive is replaced by the contents of the file indicated. **Using an #include directive to insert another file into your source file** is similar to pasting a block of text into a document with your word processor. **#include** is only one of many **pre-processor** directives, all of which can be identified by the initial # sign. The use of **pre-processor** directives is not as common in C++ as it is in C, but we'll look at a few additional examples as we go along. The type file usually included by **#include** is called a header file.

Header Files in C++ Language

In the **BASIC** example, the **pre-processor** directive **#include** tells the compiler to add the source file **IOSTREAM** to the **BASIC.CPP** source file

before compiling. IOSTREAM is concerned with basic input/output operations, and contains declarations that are needed by the cout identifier and the << operator. Without these declarations, the compiler won't recognize **cout** and will think << is being used incorrectly. Try running the program with omitting the **#include <iostream>**

The using Directive

Try searching the internet for the meaning of the **using namespace std;** statement ?

How to use Comments

Comments are parts of the source code disregarded by the compiler. They simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code. C++ supports two ways to insert comments:

```
// line comment
```

```
/* block  
comment */
```

Let us rewrite our **BASIC** program, incorporating comments into our source file.

1	// demonstrates comments	
2	#include <iostream>	//pre-processor directive
3		
4	using namespace std;	//”using” directive
5	int main()	//function name “main”
6	{	//start function body
7	cout << “Hello Third Year Students \n”;	
8		

```
9     return 0; //statement
10  }
```

Variables in C++ : Integer Variables

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Integer variables represent integer numbers like 1, 9000, and -29. Unlike floating-point numbers, integers have no fractional part. Integer variables exist in several sizes, but the most commonly used is type `int`. The amount of memory occupied by the integer types is system dependent. However, it is mostly **4 bytes**. Figure 2 shows an integer variable in memory.

```
1 // illustrates integer variables
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     int var1; //define var1
7     int var2; //define var2
8     var1 = 150; //assign value to var1
9     var2 = var1 + 11; //assign value to var2
10    cout << "var1+11 is "; //output text
11    cout << var2 << endl; //output value of var2
12    return 0;
13 }
```

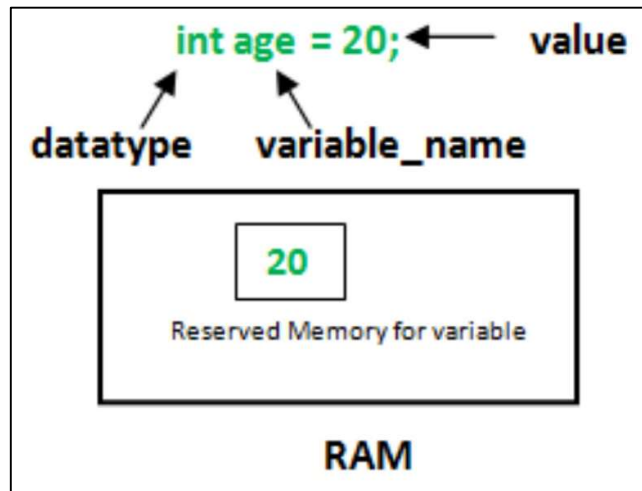



Figure 2 Variable of type int in computer memory.

```
int var1;
int var2;
```

The previous two lines define two integer variables, `var1` and `var2`. The keyword **int** indicates the type of the variable. Before you can use a variable, you must first declare it. Variable declarations, on the other hand, can be placed anywhere in a programme.

Declarations and Definitions : What is the difference?

A declaration presents a variable's name (such as `var1`) into a program and specifies its type (such as **int**). However, if a declaration also sets aside memory for the variable, it is also called a definition. The statements

```
int var1;
int var2;
```

are definitions, as well as declarations, because they set aside memory for `var1` and `var2`. We will be concerned mostly with declarations that are also

definitions, but later on we'll see various kinds of declarations that are not definitions.

Naming variables

A variable name can consist of alphabets (both upper and lower case), numbers and the underscore `_` character. However, the name must not start with a number.

You cannot use a C++ keyword as a variable name. A keyword is a predefined word with a special meaning. `int`, `class`, `if`, and `while` are examples of keywords. A variable's name should make clear to anyone reading the listing the variable's purpose and how it is used.

Assignment Statements

The statements

```
var1 = 150;  
var2 = var1 + 11;
```

assign values to the two variables. The equal sign (`=`), causes the value on the right to be assigned to the variable on the left. In the first line shown here, `var1`, which previously had no value, is given the value 150.

The statement

```
cout << "var1+11 is ";
```

displays a string constant, as we've seen before. The next statement

```
cout << var2 << endl;
```

displays the value of the variable var2. As you can see in your console output window, the output of the program is var1+11 is 161. It's important to note that cout and the operator can handle an integer and a string in different ways. They print strings as text if we send them a string. They print an integer as a number if we send them an integer.

As you can see, the output of the two **cout** statements appears on the same line on the output screen. No linefeed is inserted automatically. If you want to start on a new line, you must insert a linefeed yourself. We've seen how to do this with the `'\n'` escape sequence. Now we'll see another way: using something called a manipulator.

\n versus the endl Manipulator

The **endl** manipulator produces a newline character, exactly as the insertion of `'\n'` does, but it also has an additional behaviour when it is used with buffered streams: the buffer is flushed. Anyway, cout will be an unbuffered stream in most cases, so you can generally use both the `\n` escape character and the endl manipulator in order to specify a new line without any difference in its behaviour.

Character Variables

Type char stores integers that range in value from -128 to 127. variables of this type occupy only **1 byte (eight bits)** of memory. Character variables are used to store ASCII characters. Note the difference between characters and strings as follows :

```
'z'  
'p'  
"Hello world"  
"How do you do?"
```

The following program further illustrates the character variables :

```
1 // Shows character variables
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     char var_1 = 'A';    //define char variable as character
7
8     cout << var_1;      //display character
9     char var_2 = 'B';  //set char variable to char constant
10    cout << var_2;     //display character
11    cout << '\n';      //display newline character
12    return 0;
13 }
```

Standard input with (cin)

The standard input device is usually the keyboard. Handling the standard input in C++ is done by applying the extraction (>>) on the **cin** stream. The operator must be followed by the variable that will store the data that is going to be extracted from the stream. **cin** can only process the input from the keyboard once the **ENTER** key has been pressed. Therefore, even if you request a single character, the extraction from **cin** will not process the input until the user presses **ENTER** key after the character has been introduced. You must always consider the type of the variable that you are using as a container with **cin** extractions. If you request an **integer** you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters. The following program is a simple demonstration of the input with **cin**.

```
1 #include <iostream>
2 using namespace std;
3 int main()
```

```
4 {
5 cout << "Please enter an integer value: ";
6 cin >> i;
7 cout << "The value you entered is " << i;
8 cout << " and its double is " << i*2 << ".\n";
9 return 0;
10 }
```

The statement

```
cin >> i;
```

causes the program to wait for the user to type in a number. The resulting number is placed in the variable `i`. Figure 3 shows input using `cin` and the extraction operator `>>`.

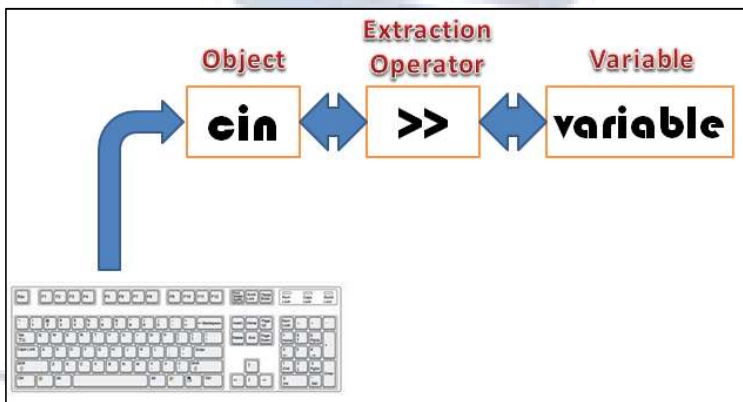


Figure 3 Input with cin.

It is worth mentioning that variables could be defined anywhere in the program whenever you need them.

Cascading << variables

Carefully study the following program, could you inspect what is cascading is ?

```
1 #include <iostream>
2 using namespace std;
```

```

3 int main()
4 {
5     cout << "Please enter an integer value: ";
6     cin >> i;
7     cout << i<<i*2 << ".\n";
8 }
9 return 0;

```

Operator Precedence

Operator precedence determines the grouping of terms in an expression. The associativity of an operator is a property that determines how operators of the same precedence are grouped in the absence of parentheses. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7. The following figure further illustrates such issue (Figure 4).

Precedence	Operator	Description
<i>Higher</i> ↓ <i>Lower</i>	()	Function call
	+	Positive
	-	Negative
	*	Multiplication
	/	Division
	%	Modulus (remainder)
	+	Addition
	-	Subtraction
	=	Assignment

Figure 4 Operator precedence in C++

Floating Point variable types

We've talked about type `int` and type `char`, both of which represent numbers as integers—that is, numbers without a fractional part. Now let us examine a different way of storing numbers as floating-point variables.

Floating-point variables represent numbers with a decimal place—like 3.1497, 0.0025, and -10.2. They have both an integer part, to the left of the decimal point, and a fractional part, to the right. There are three kinds of floating-point variables in C++: type `float`, type `double`, and type `long double`. Let us start with the smallest of these, type `float`.

Type (float)

Type `float` stores numbers in the range of about 3.4×10^{-38} to 3.4×10^{38} , with a precision of seven digits. It occupies 4 bytes (32 bits) in memory. The following example program prompts the user to type in a floating-point number representing the radius of a circle. It then calculates and displays the circle's area.

```
1 //Empty Line
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     float R; //variable of type float
7     const float PI = 3.14; //type const float
8     cout << "Enter radius of circle: "; //prompt
9     cin >> R; //get radius
10    float area = PI * R * R; //find area
11    cout << "Area is " << area << endl; //display answer
12    return 0;
13 }
```

Here is a sample interaction with the program:

```
Enter radius of circle: 1.0  
Area is 3.14
```

Type (double)

The larger floating-point types, **double**, are similar to float except that it requires more memory space and provide a wider range of values and more precision. Type double requires 8 bytes of storage.

Defining constant variables: The const qualifier

Besides demonstrating variables of type float, the **AREA** example also introduces the qualifier const. It's used in the statement

```
const float PI = 3.14;           //type const float
```

A variable's data type is preceded by the keyword **const** (for constant). It declares that a variable's value will remain constant throughout the programme. The compiler will throw an error if you try to change the value of a variable defined with this qualifier.

Variable Type bool

A boolean data type is declared with the bool keyword and can only take the values true or false. When the value is returned, true = 1 and false = 0. It only needs one byte of storage.

```
bool program_ended=1           //1 Means True
```


C++ Variable Type Summary

When programming, we store the variables in our computer's memory, but the computer has to know what kind of data we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers. Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one.

Table 1 Variable types and sizes in C++

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)

However, the values of the columns *Size* and *Range* depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems.

unsigned Data Types: When to use them?

You can modify the range of the character and integer types to start at 0 and include only positive numbers by removing the sign. They can now represent numbers that are twice as large as the signed kind. The unsigned versions are shown in Table 2. The unsigned types are used when the quantities represented are always positive

Table 2 Unsigned Integer Types

<i>Keyword</i>	<i>Numerical Range</i>		<i>Bytes of Memory</i>
	<i>Low</i>	<i>High</i>	
unsigned char	0	255	1
unsigned short	0	65,535	2
unsigned int	0	4,294,967,295	4
unsigned long	0	4,294,967,295	4

Study the following C++ program that demonstrates the usage of signed and unsigned variables.

```
1 //Empty Line.....
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     int signedVar = 1900000000;
7     unsigned int unsignVar = 1900000000;
8     signedVar = (signedVar * 2) / 3;           // out of range
9     unsignVar = (unsignVar * 2) / 3;          // within range
10    cout << "signedVar = " << signedVar << endl;    //wrong
11    cout << "unsignVar = " << unsignVar << endl;    //Right
12    return 0;
13 }
```

The program multiplies both variables by 2, then divides them by 3. Although the result is smaller than the original number, the intermediate calculation is larger than the original number. Here is the output:

```
signedVar = -431,655,764  
unsignVar = 1,000,000,000
```

The signed variable now displays an incorrect answer, while the unsigned variable, which is large enough to hold the intermediate result of the multiplication, records the result correctly.

Type Conversion

Consider the following **CONVERSION** program:

```
1 // Useless Line  
2 #include <iostream>  
3 using namespace std;  
4 int main()  
5 {  
6     int count = 8;  
7     float avgWeight = 160.5;  
8     double totalWeight = count * avgWeight;  
9     cout << "totalWeight=" << totalWeight << endl;  
10    return 0;  
11 }
```

Here a variable of type **int** is multiplied by a variable of type float to yield a result of type double. This program compiles without error; the compiler considers it normal that you want to multiply (or perform any other arithmetic operation on) numbers of different types.

Automatic Conversions

Conversion between types is logical only when the types are related. In such a case, we can use an object or value of one type as an operand in place of an actual operand type that is expected. The compiler does the conversion implicitly, without any direct programmer intervention. For example, consider the following expression.

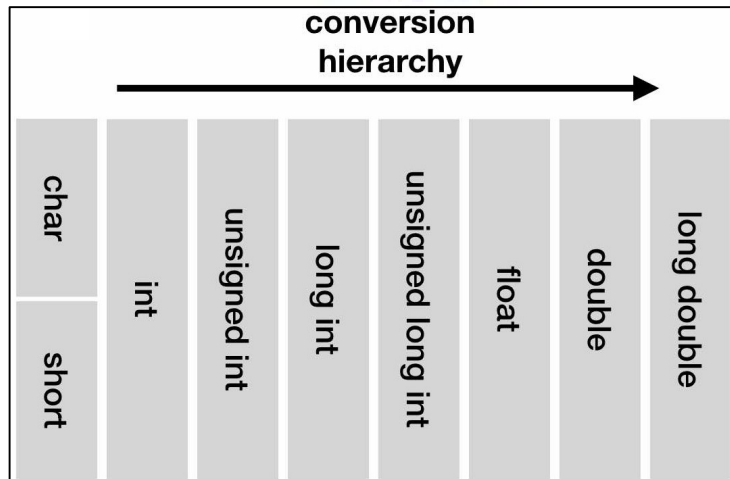
```
int i = 10;  
double d = 1.14159;  
int sum = d + i;
```

Observe that operands of the expressions are two different types: double and integer. Before adding the two values of two different types, the compiler uses a conversion function to transform the operands to a common type. Even a simple looking expression has many tacit functions associated with it.

Observe that the expression has both integer and floating-point operands; the compiler prior to addition transforms the lower type (int) to its associated higher type (floating-point) before performing the function signified by the operator (addition). Therefore, after addition the result actually becomes 11.14159. Now, because initialization happens next (with = operator), the type we are initializing to prevail over any type we want to assign and the floating-point result are re-converted back to the dominant type (which is integer in this case). Therefore, the final content of the sum is 11, discarding the fractional part as the sum variable which is declared as an integer. The conversion between fundamental types thus is defined to do the transformation from lower type (as depicted in table 3) to higher type to preserve precision in the context of a mathematical expression. The

conversion from double to int shall always truncate the fractional part, delimiting the precision.

Table 3 Order of Data Types in conversion.



The Remainder Operator

There is an important arithmetic operator that works only with integer variables. It is called the remainder operator, and is represented by the percent symbol (%). This operator (also called the modulus operator) finds the remainder when one number is divided by another. The following program demonstrates the effect.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     cout << 6 % 8 << endl           // 6
6     << 7 % 8 << endl               // 7
7     << 8 % 8 << endl               // Guess
8     << 9 % 8 << endl               // ?
9     << 10 % 8 << endl;             // ?
10    return 0;
11 }
```

SMART coding: Compound Assignment Operators

Consider the following arithmetic expressions and their corresponding abbreviations :

<u>Compound Assignment Operators in C++</u>	
<u>Simple assignment operator</u>	<u>Compound assignment operator</u>
X = X + 1;	X += 1;
Y = Y - 1;	Y -= 1;
Z = Z + X;	Z += X;
P = P * item ;	P *= item ;
N = N * (x + 1) ;	N *= (x + 1) ;
Total = Total / (X+Y);	Total /= (X+Y);
Hours = Hours % 13;	Hours %= 13;

The compound-assignment operators combine the simple-assignment operator with another binary operator. Compound-assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. For example, a compound-assignment expression such as. expression1 += expression2.

There are arithmetic assignment operators corresponding to all the arithmetic operations: **+=**, **-=**, ***=**, **/=**, and **%=** (and some other operators as well). The following example shows the arithmetic assignment operators in use:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
```

```

5   int ans = 20;
6   ans += 10;           //same as: ans = ans + 10;
7   cout << ans << ", ";
8   ans -= 7;           //same as: ans = ans - 7;
9   cout << ans << ", ";
10  ans *= 2;           //same as: ans = ans * 2;
11  cout << ans << ", ";
12  ans /= 3;           //same as: ans = ans / 3;
13  cout << ans << ", ";
14  ans %= 3;           //same as: ans = ans % 3;
15  cout << ans << endl;
16  return 0;
17 }

```

Run the previous program and see what is the output ?

C++ Increment Operators

Here is an even more specialized operator. You often need to add 1 to the value of an existing variable. You can do this the “normal” way:

```
count = count + 1;           // adds 1 to “count”
```

Or you can use an arithmetic assignment operator:

```
count += 1;                 // adds 1 to “count”
```

But there is an even more condensed approach:

```
++count;                   // adds 1 to “count”
```

The `++` operator increments (adds 1 to) its argument.

Increment and decrement operators

Increment Operators: The increment operator is used to increment the value of a variable in an expression. In the Pre-Increment, value is first incremented and then used inside the expression. Whereas in the Post-Increment, value is first used inside the expression and then incremented.

Syntax:

```
// PREFIX  
++m
```

```
// POSTFIX  
m++
```

Decrement Operators: The decrement operator is used to decrement the value of a variable in an expression. In the Pre-Decrement, value is first decremented and then used inside the expression. Whereas in the Post-Decrement, value is first used inside the expression and then decremented.

Syntax:

```
// PREFIX  
--m
```

```
// POSTFIX  
m--
```

Differences between Increment and Decrement Operators:

The following figures illustrates the difference between prefix and postfix operators:

Increment Operators	Decrement Operators
Increment Operator adds 1 to the operand.	Decrement Operator subtracts 1 from the operand.
Postfix increment operator means the expression is evaluated first using the original value of the variable and then the variable is incremented(increased).	Postfix decrement operator means the expression is evaluated first using the original value of the variable and then the variable is decremented(decreased).
Prefix increment operator means the variable is incremented first and then the expression is evaluated using the new value of the variable.	Prefix decrement operator means the variable is decremented first and then the expression is evaluated using the new value of the variable.

Figure 5 Prefix and postfix in C++

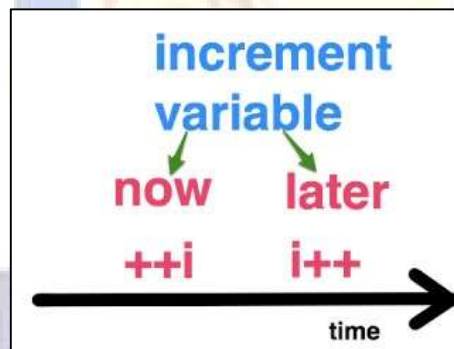


Figure 6 The increment operator

Here is an example that shows both the prefix and postfix versions of the increment operator:

```

1 // Demonstrates the increment operator
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     int ASD = 10;
7     cout << "ASD =" << ASD << endl;    //displays 10
8     cout << "ASD =" << ++ ASD << endl;    //displays 11 (prefix)
9 
```

```

10 cout << "ASD =" << ASD << endl; //displays 11
11 cout << "ASD =" << ASD ++ << endl; //displays 11 (postfix)
12 cout << "ASD =" << ASD << endl; //displays 12
13 return 0;
14 }

```

Here is the program's output:

```

ASD =10
ASD =11
ASD =11
ASD =11
ASD =12

```

Library Functions in C++

Many activities in C++ are carried out by library functions. These functions perform file access, mathematical computations, and data conversion, among other things. The next example, SQRT, uses the library function `sqrt()` to calculate the square root of a number entered by the user.

```

1 #include <iostream> //Guess for what?
2 #include <cmath> //for sqrt()
3 using namespace std;
4 int main()
5 {
6     double number, answer;
7     cout << "Please enter a number: ";
8     cin >> number; //get the number
9     answer = sqrt(number); //find square root
10    cout << "Square root is "
11        << answer << endl; //display it
12
13 }

```

```
14     return 0;
    }
```

The program first obtains a number from the user. This number is then used as an argument to the `sqrt()` function, in the statement

```
answer = sqrt(number);
```

Here is some output from the program:

```
Enter a number: 100
Square root is 10
```

What Are Header Files?

A header file is a file containing declarations to be shared between several source files. You request the use of a header file in your program by including it, with the pre-processing directive `#include`. Header files serve two purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries, like `<iostream>`
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results as copying the header file into each source file that needs it. Such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.



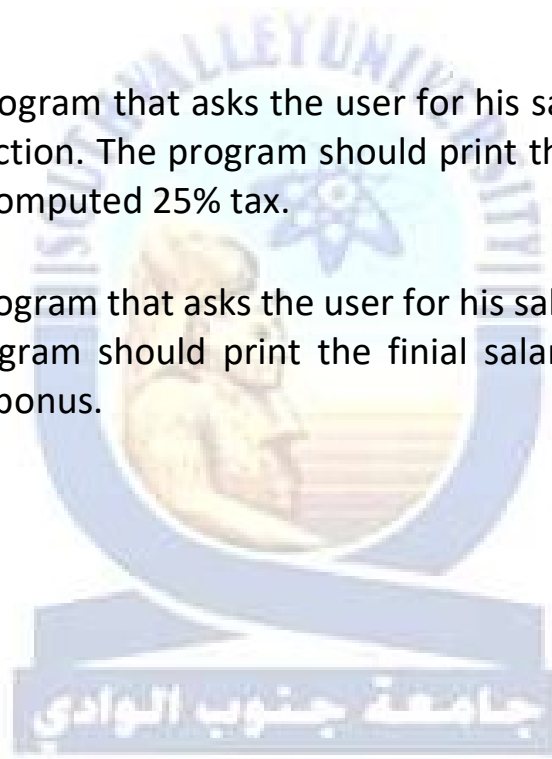
جامعة جنوب الوادي
South Valley University

Exercises

1. Write an example of a normal C++ comment and an example of an old-fashioned `/*` comment.
2. How many bytes are occupied by the following data types in a normal 32-bit system:
 - a. Type `int`
 - b. Type `float`
 - c. Type `long`
3. True or false: A variable of type `char` can hold the value 501.
4. True or false: In an assignment statement, the value on the left of the equal sign is always equal to the value on the right.
5. What header file must you `#include` with your source file to use `cout`?
6. Write a statement that gets a numerical value from the keyboard and places it in the variable `temp`.
7. The expression `11%5` evaluates to _____.
8. Write a program that generates the following table: Use a single `cout` statement for all the output.

1990	199
1991	72950
1992	1135500
1993	16200
9. Write a program that accepts a sequence of 6 numbers and outputs their average and standard deviation.

10. Write a program that accepts 3 numbers and prints out their sum, difference, division and multiplication results.
11. Write a program that accepts an integer number and prints its square value.
12. Write a program that asks the user for his salary and computes a 25% tax deduction. The program should print the final salary after deduction the computed 25% tax.
13. Write a program that asks the user for his salary and adds a 30% bonus. The program should print the final salary after adding the computed 30% bonus.



جامعة جنوب الوادي
South Valley University

Programmer (noun.)

A person who fixed a problem that
you don't know you have,
in a way you don't understand.

جامعة جنوب الوادي

Chapter 2

Using Microsoft visual studio.

South Valley University

Putting C++ into action

In the previous chapter we learnt the basic structure of a C++ program. In this chapter we will learn how to use Microsoft visual studio to code and run C++ programs and see their results.

Microsoft Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs for Microsoft Windows, as well as web sites, web applications and web services. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms, Windows Presentation Foundation, Windows Store and Microsoft Silverlight. It can produce both native code and managed code.

Visual Studio includes a code editor supporting IntelliSense (the code completion component) as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a forms designer for building GUI applications, web designer, class designer, and database schema designer. It accepts plug-ins that enhance the functionality at almost every level including adding support for source-control systems (like Subversion) and adding new toolsets like editors and visual designers for domain-specific languages or toolsets for other aspects of the software development lifecycle (like the Team Foundation Server client: Team Explorer).

Microsoft Visual Studio

There exists more than 10 version of visual studio, since the first release in 1997. Each version has a different and advanced feature following their chronological sequence as listed in Table 4 below.

Table 4 Microsoft visual studio versions

Product name	Codename	Version number	Supported .NET Framework versions	Release date
Visual Studio 97	Boston	5.0	N/A	February 1997
Visual Studio 6.0	Aspen	6.0	N/A	June 1998
Visual Studio .NET (2002)	Rainier	7.0	1.0	February 13, 2002
Visual Studio .NET 2003	Everett	7.1	1.1	April 24, 2003
Visual Studio 2005	Whidbey	8.0	2.0, 3.0	November 7, 2005
Visual Studio 2008	Orcas	9.0	2.0, 3.0, 3.5	November 19, 2007
Visual Studio 2010	Dev10/Rosario	10.0	2.0 – 4.0	April 12, 2010
Visual Studio 2012	Dev11	11.0	2.0 – 4.5.2	September 12, 2012
Visual Studio 2013	Dev12	12.0	2.0 – 4.5.2	October 17, 2013

Visual Studio 2015	Dev14	14.0	2.0 – 4.6	July 20, 2015
Visual Studio 2017	Dev15	15.0	2.0 – 4.6.2; Core 1.0	TBA
Visual Studio 2019				

Any of the different visual studio versions could be used to build and run all of the C++ programs provided in this book. However, this chapter will target visual studio 2013 as a target case (it's start screen is depicted in Figure 11).

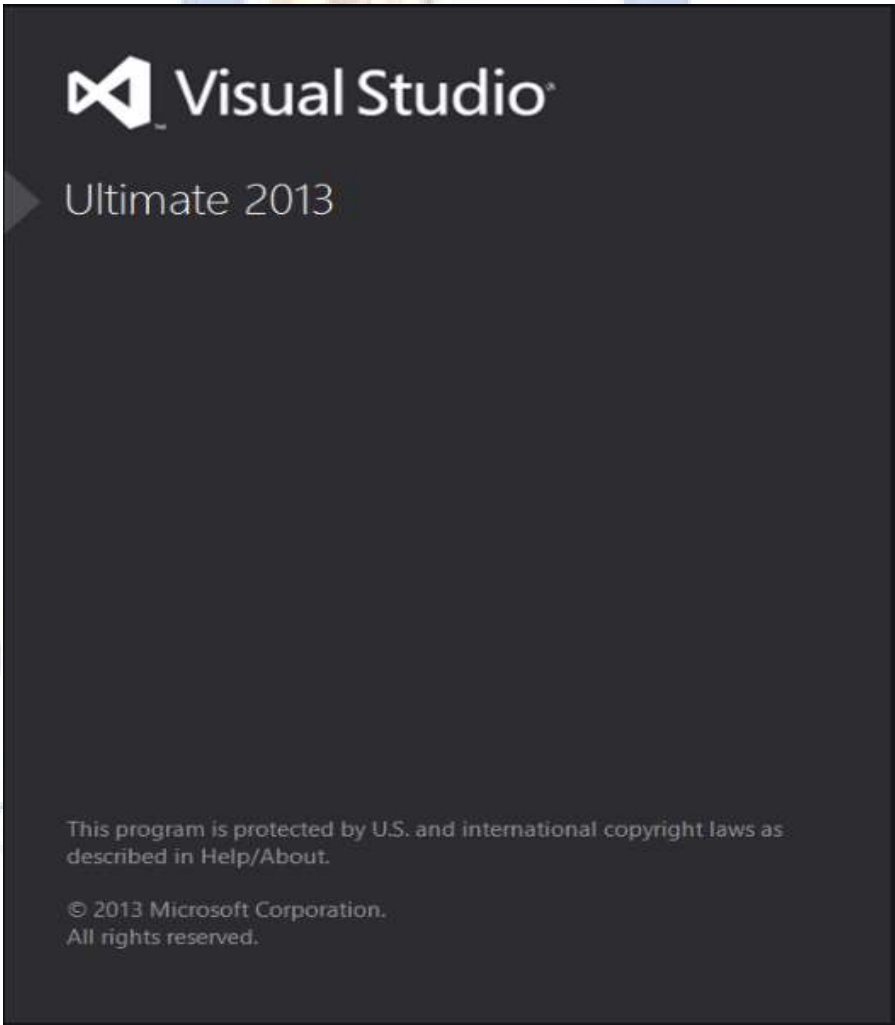


Figure 7 Microsoft visual studio 2013 start screen.

How to get Visual studio?

Fortunately, Microsoft believes in the “code for all principle”, hence, it released the visual studio for free. You can visit the below link and download the community edition for free.

<https://www.visualstudio.com/downloads/>

Your First Visual Studio Run

Once you have installed visual studio 2013 (or whatever version), double click its shortcut from the desktop as shown in Figure 12 below:

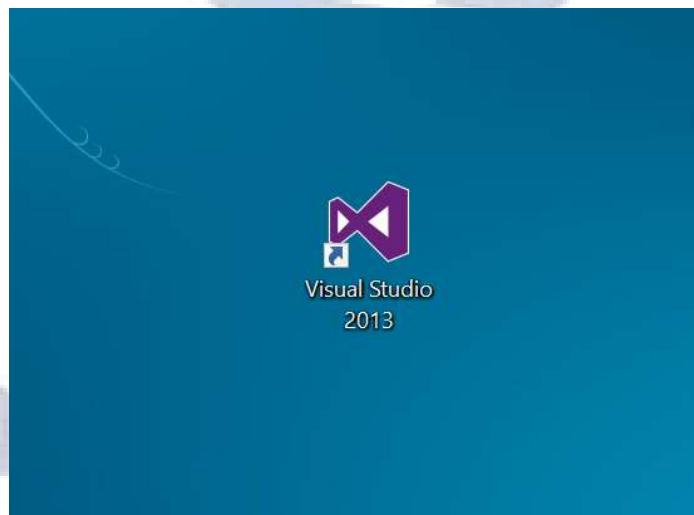


Figure 8 Visual studio desktop shortcut

Depending on your computer speed the program will run shortly after it performs some initialization steps to adjust some settings for the first run. However, do not worry the start-up time would be less next time unless you bought a high spec computer. The next Figure 13 displays the start-up screen of the visual studio compiler.

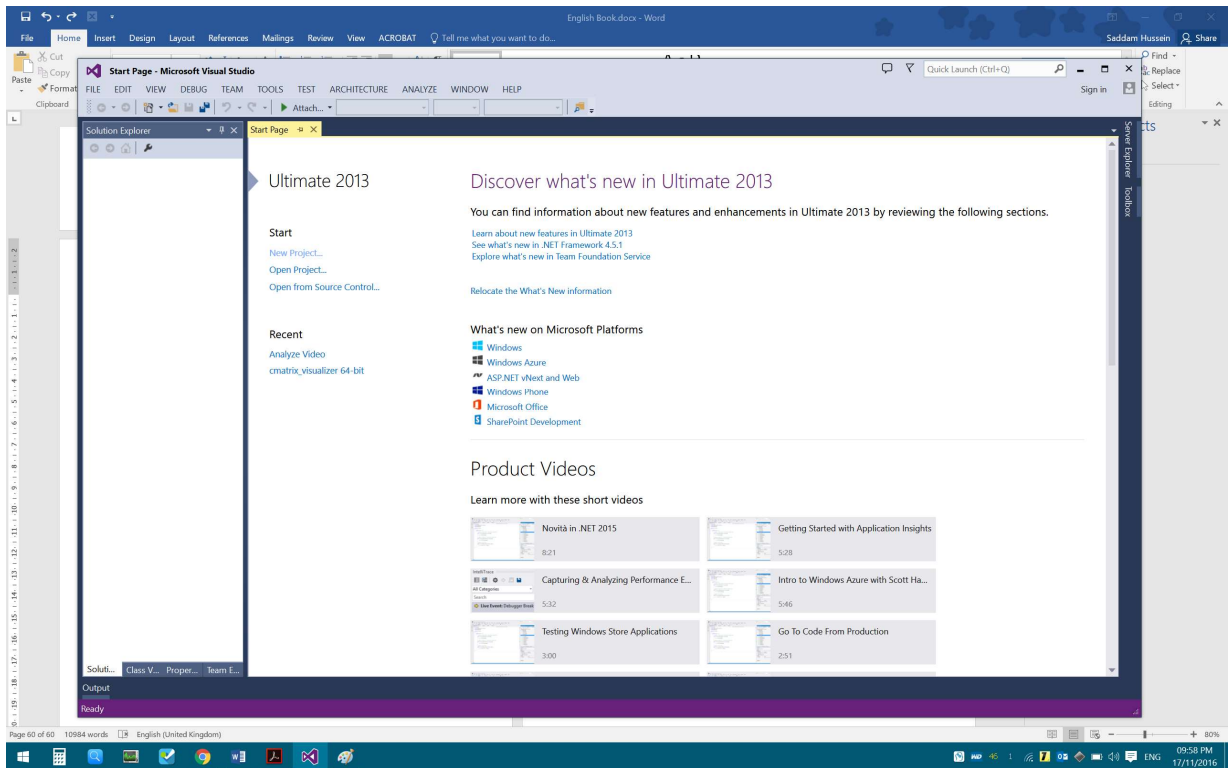
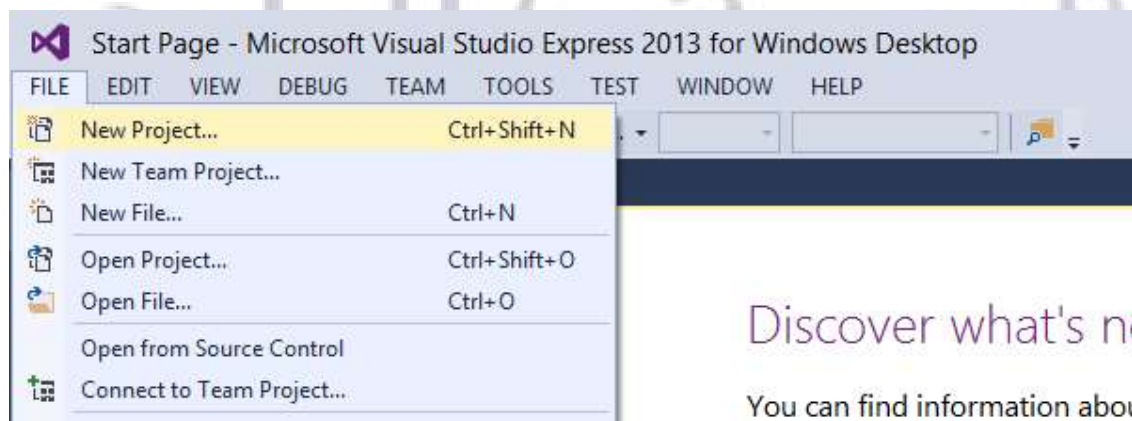


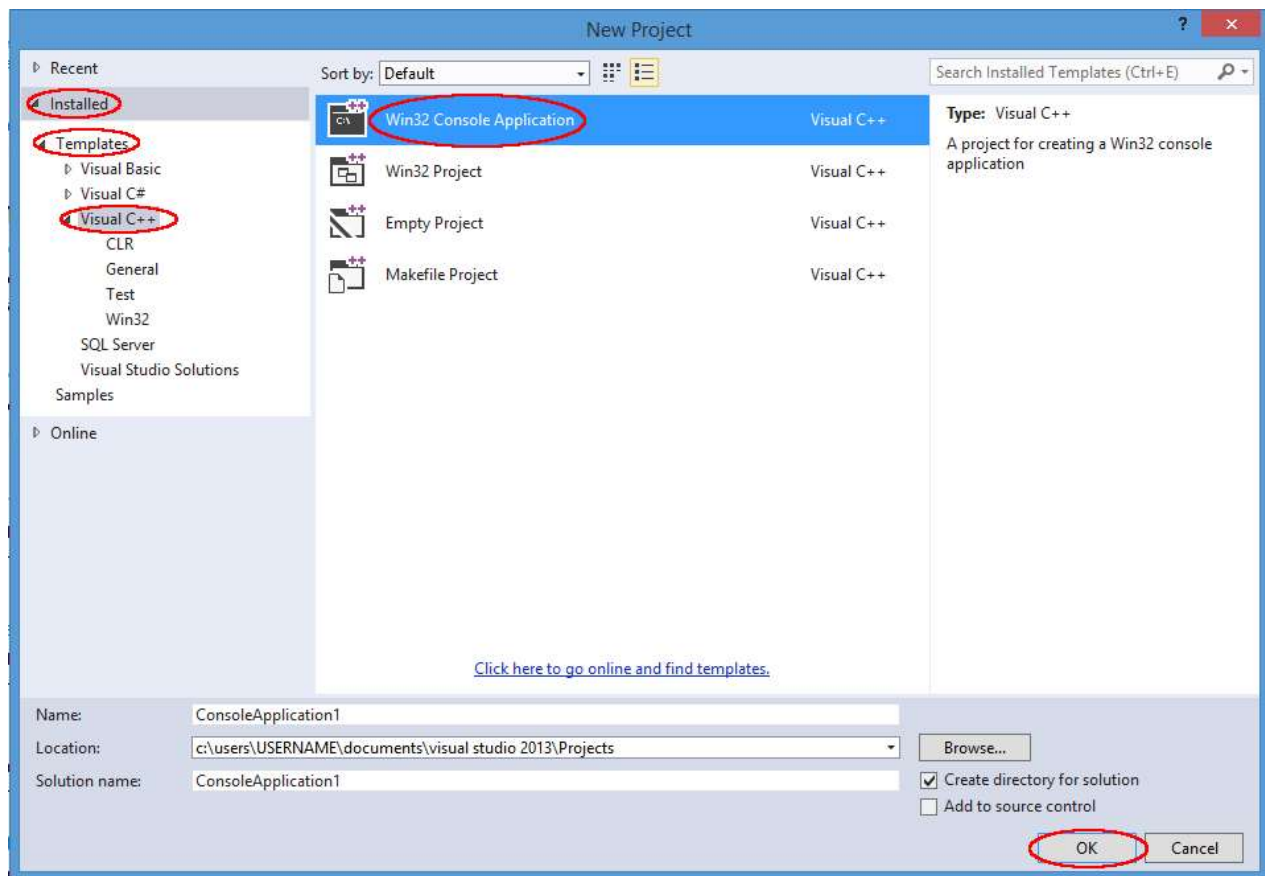
Figure 9 The start-up screen of Microsoft visual studio.

The next step is starting a new project:

File -> New Project...



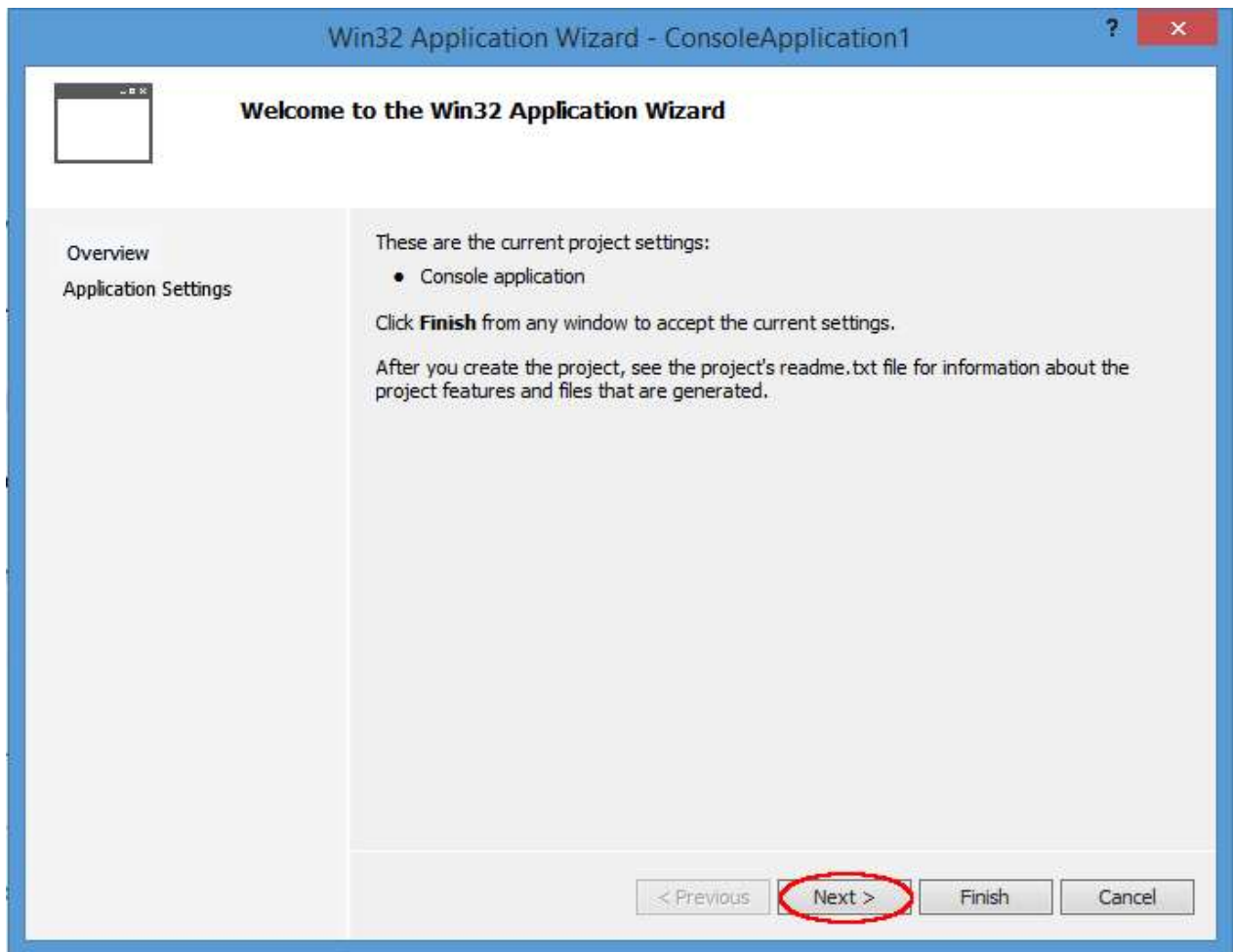
Here, on the left-hand side, select Templates -> Visual C++. Then, on the central part, select Win32 Console Application:



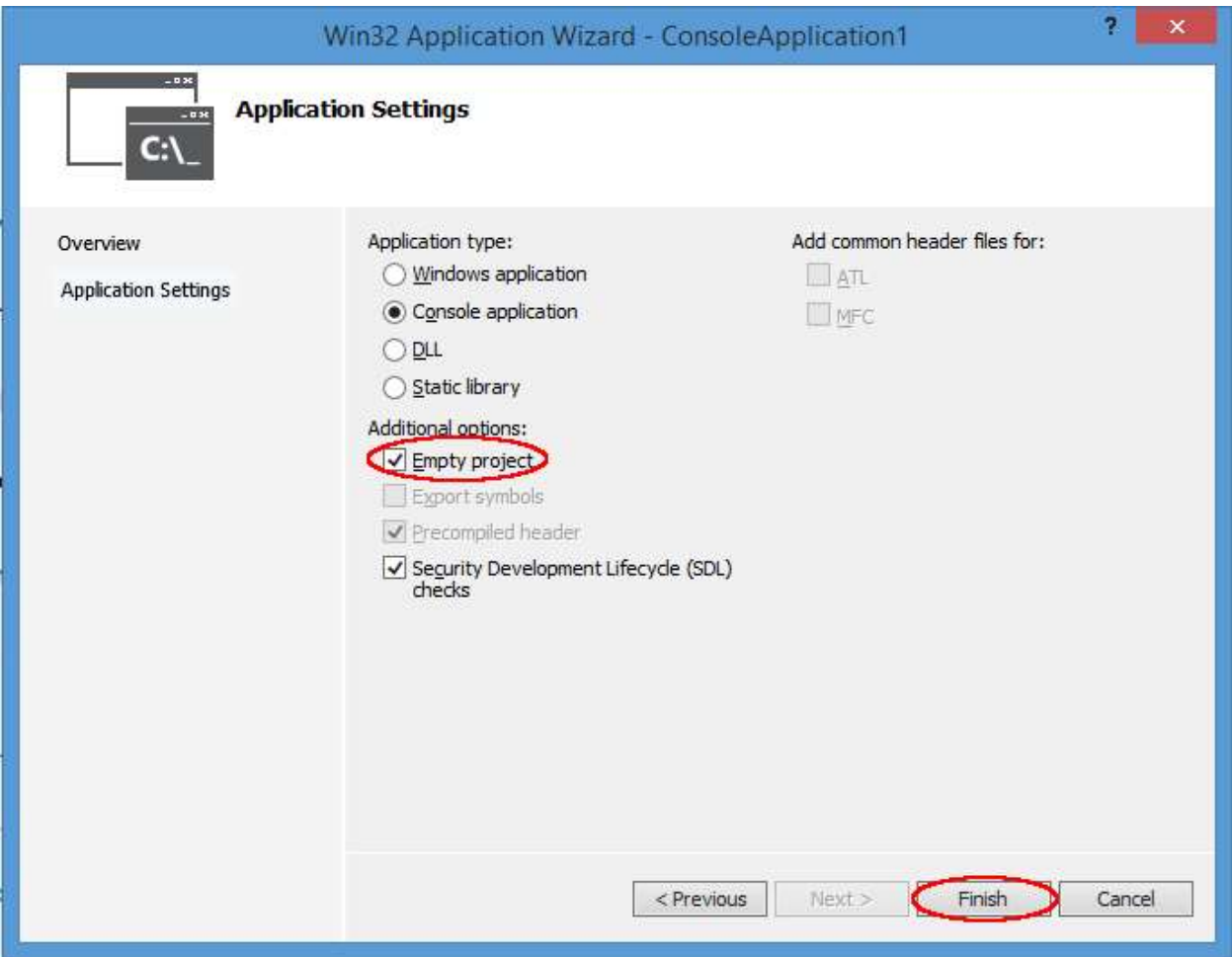
Then, type whatever project name and location on your hard drive, where the files will be stored. The default options are fine, but you can also change them to better fit your needs.

Now click [OK]

This will open the Win32 Application Wizard:

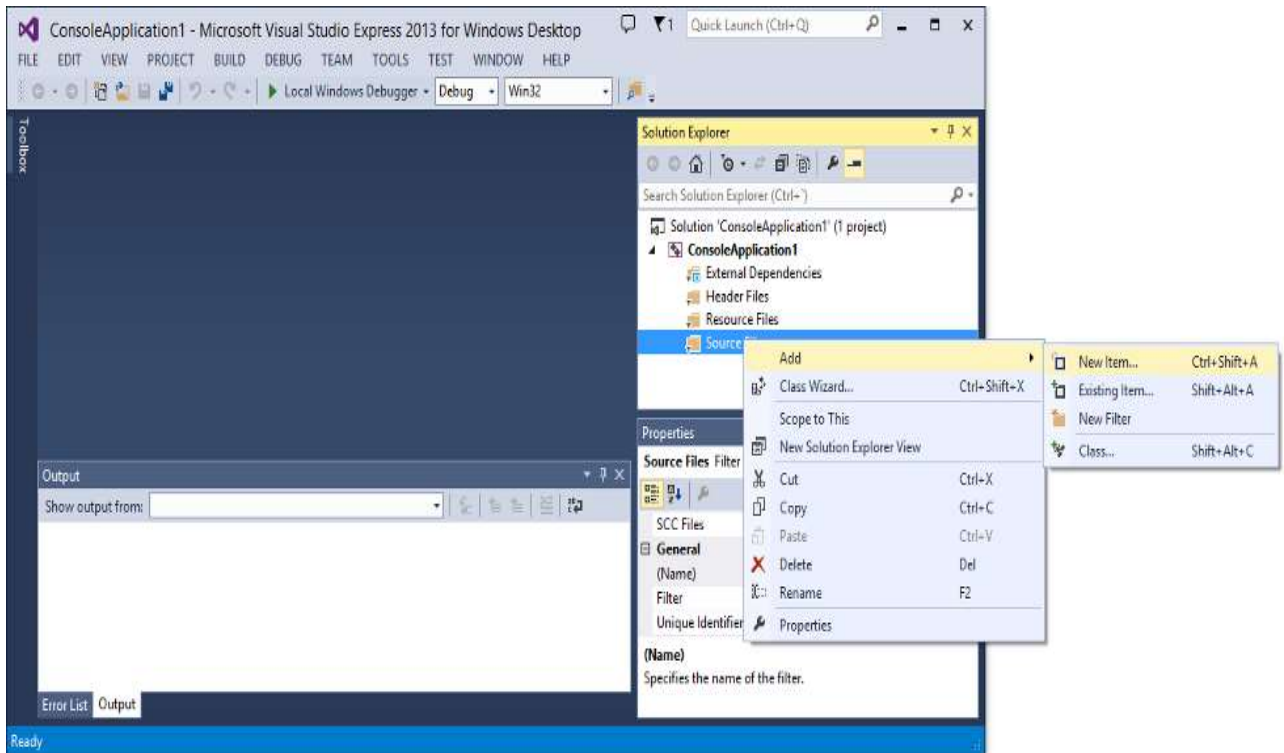


Click [Next].

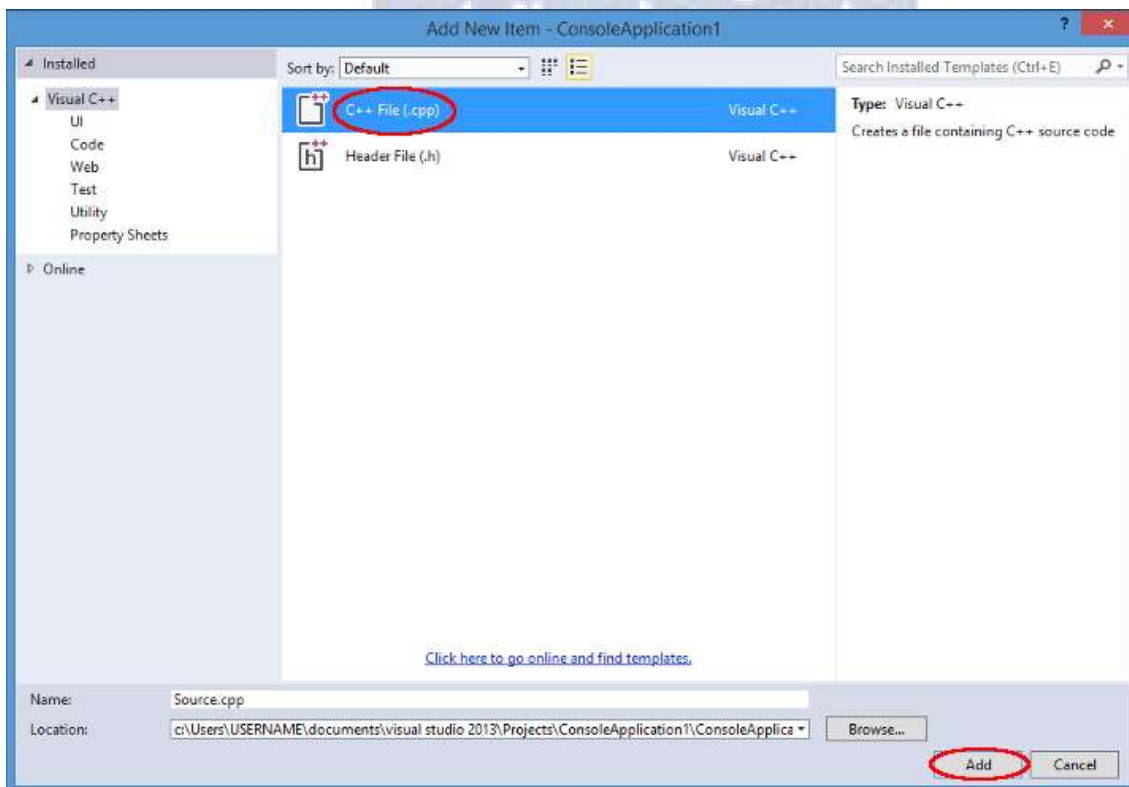


Leave "Console application" selected, and in Additional options select Empty project. Other options are not needed, but won't bother either.

Now we have an empty project. We need to add a file to it. For that: On the Solution Explorer at right, look for Source Files under your application. Right-click -> **Add -> New Item...**



Here, add a new C++ file:

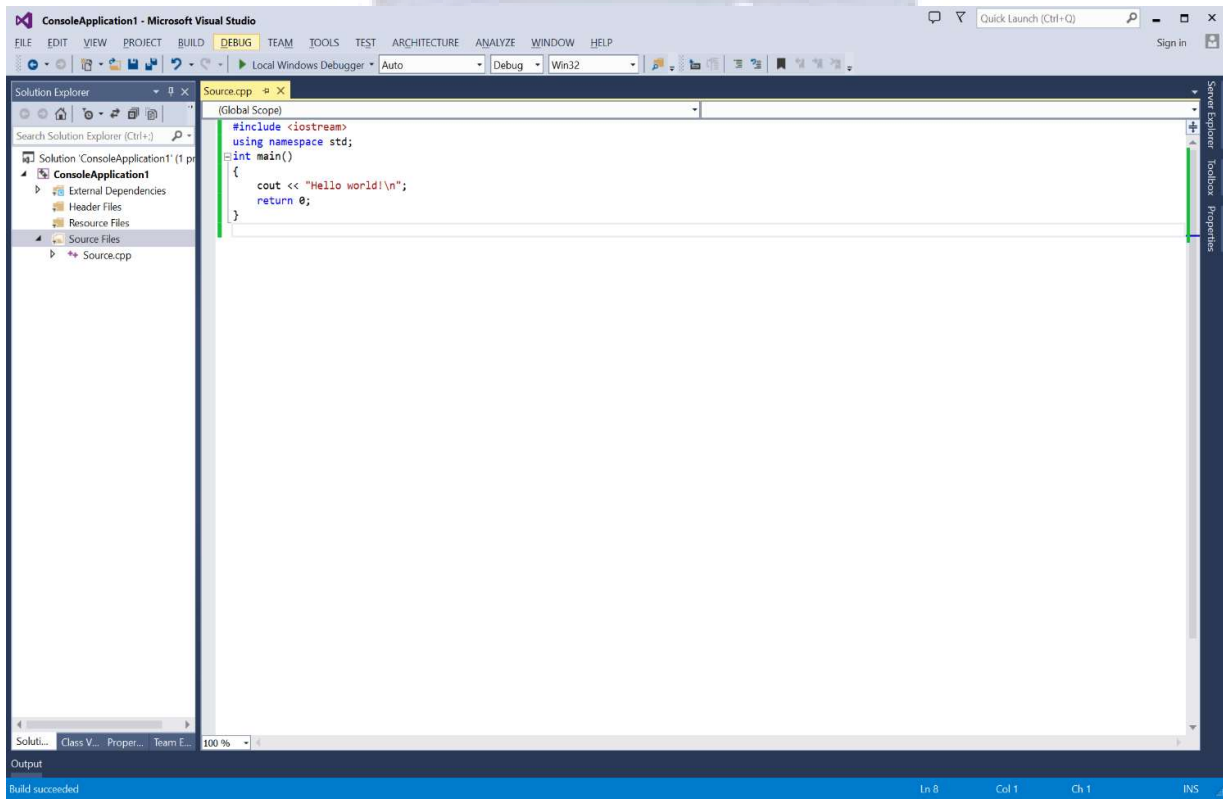


you can give it any name you want with a .cpp extension, such as example.cpp. After clicking OK, the main window will display the editor to edit this new C++ file.

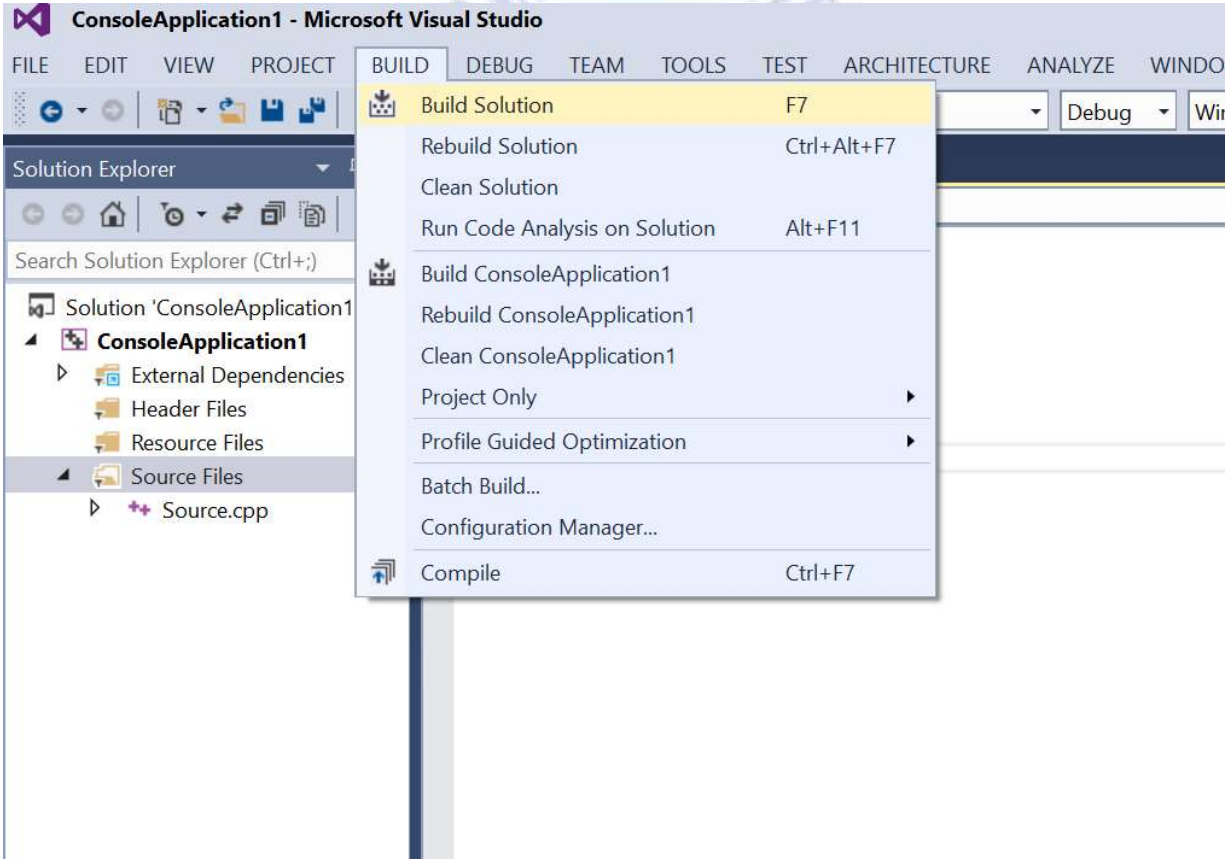
Write the following code in it:

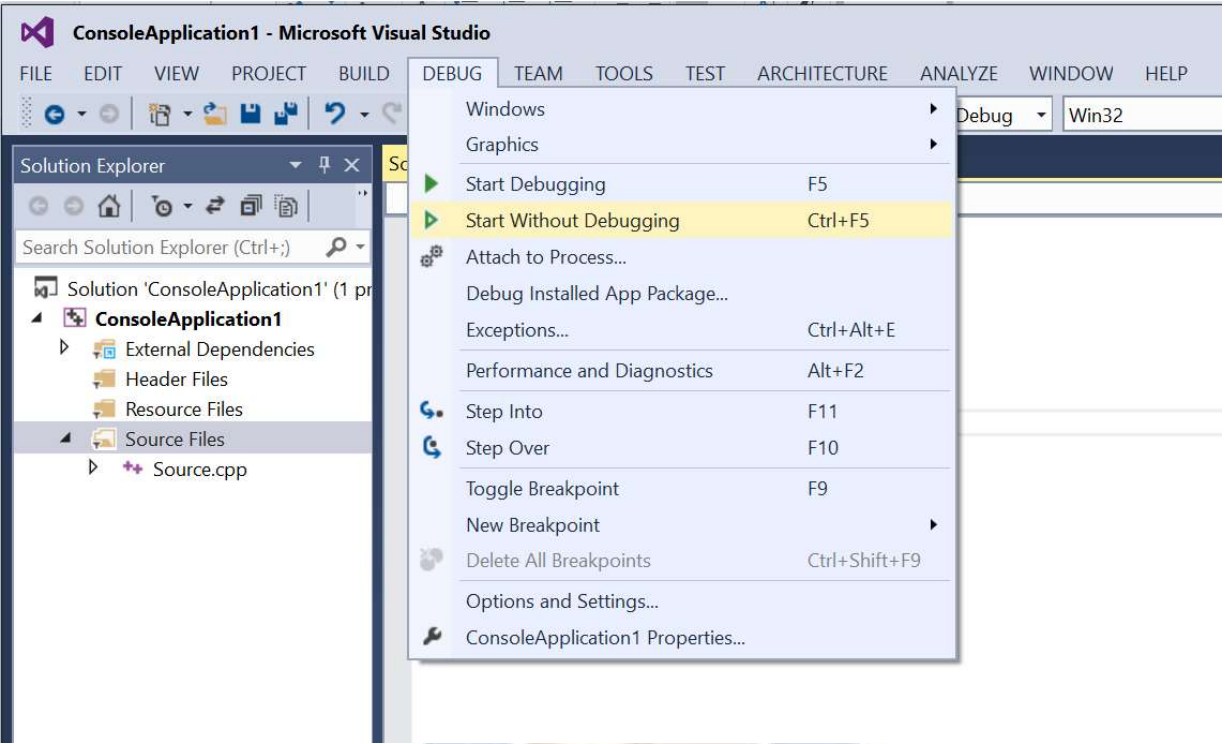
```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world! \n";
    return 0;
}
```

The visual studio window should look like the following screen:

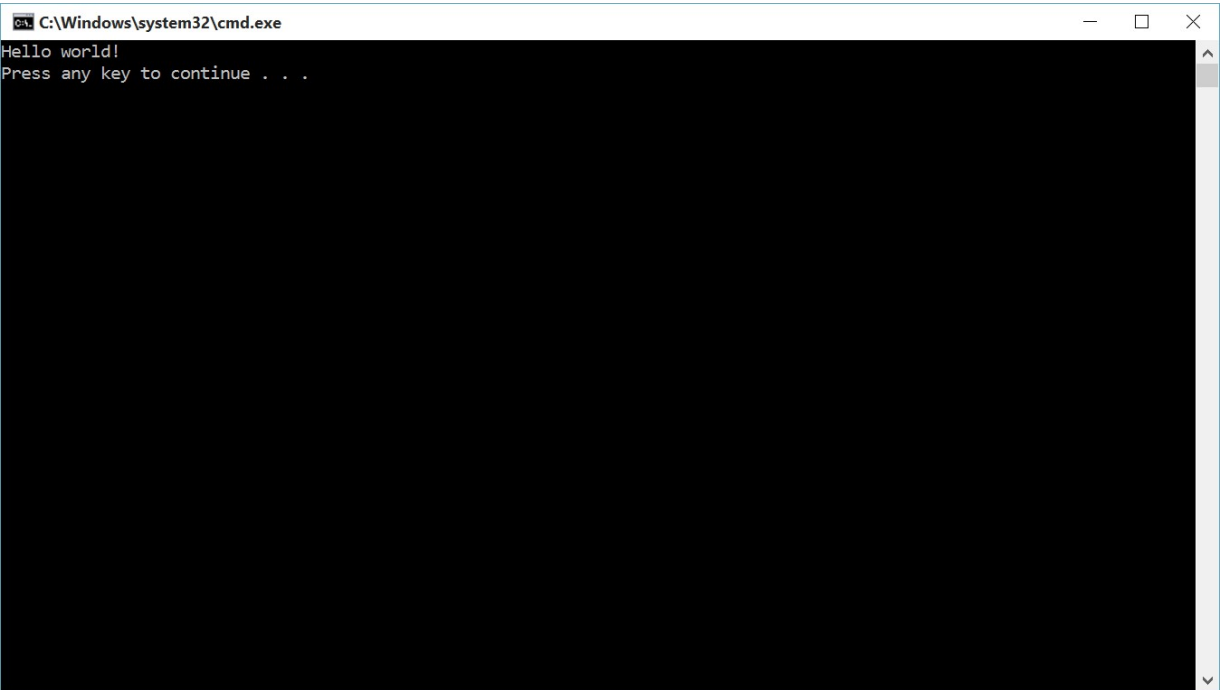


Then, to compile and run this application simply press Ctrl+F5. Or alternatively you can select BUILD -> Build Solution. Then DEBUG -> Start without debugging. This is depicted in the below screens:





Finally, after you execute the previous steps, you will be able to see the result as shown in the next screen:



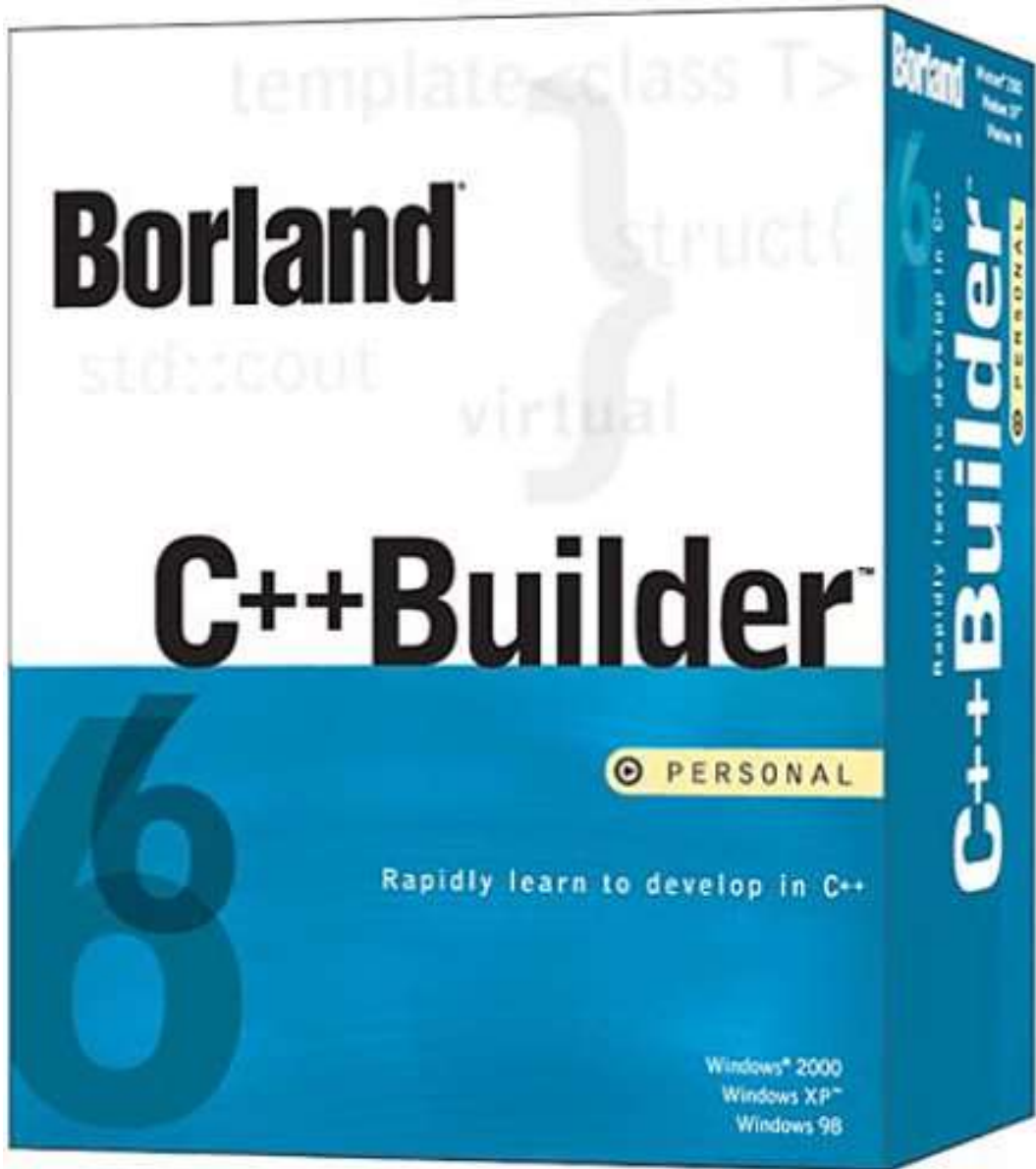
Other C++ compilers

Microsoft visual studio is not the only software package that is used to compile and run C++ code. There are numerous available packages, some of them are listed below:

1- XCode on the MAC OS platform

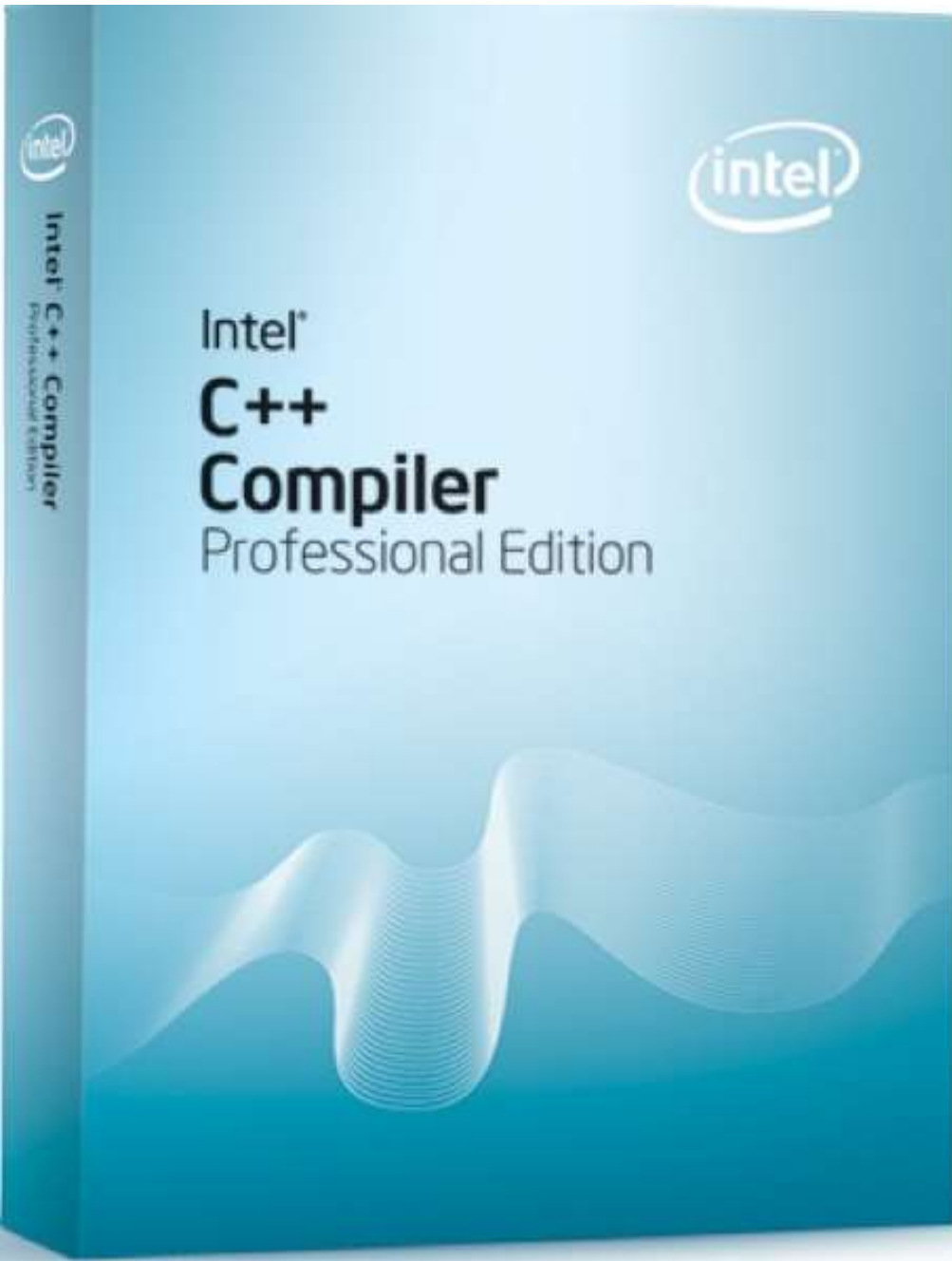


2- Borland C++ builder



South Valley University

3- Intel C++ compiler



4- Salford C++ Compiler.

5- KAI C++ Compiler.

6- Solaris Studio Express

Exercises

What is the result of the following C++ program `cout` lines?

Program 1

```
1 int x=0,y=5,z=0;
2 x=y*z;
3 cout<<x;
4
5
6 x=y+y*10;
7 cout<<x
8
9 z=50;
10 z++;
11 cout<<z;
12 cout<<z++;
13 cout<<++z;
14 cout<<z-1;
```

Program 2

```
1 int m=5;
2 float f=5.0;
3 cout<<m+f;
4 f+=0.5;
5 cout<<f;
6 cout<<(int)f;
7 m=f;
8 cout<<m;
```

Program 3

```
1 int x1=0,x2=5,x3=0;
2 float x3=1.3,x4=1.7;
3
4 cout<<x3+x4;
5 cout<<(int) x3+x4;
6 cout<<(int) (x3+x4);
7
8
9 x1=x3+x4;
10 cout<<x1;
11
12 int x5=x3+x4;
13 cout<<x5;
14
```

جامعة جنوب الوادي

Program 4

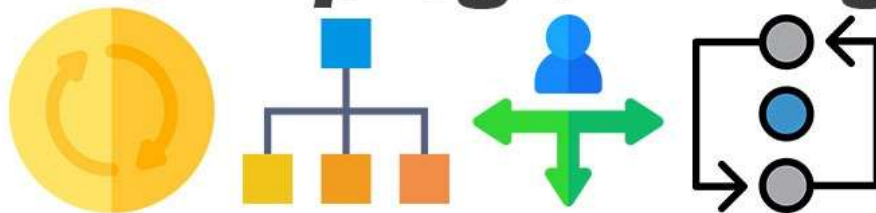
```
1 int x1=10,x2=25,x3=30;
2 cout<<x1+x2+x3;
3 cout<<(x1+x2)+x3;
4 cout<<x1%2;
5 cout<<x1/2;
6 cout<<x2%30;
7
```




Chapter 3

Loops and Decisions in C++.

Control Structures in C++ programming



Few programmes execute all of their statements in the same order from start to finish. Most programmes, like many humans, make decisions based on changing conditions. Depending on the computations completed in the programme, the flow of control jumps from one area of the programme to another. Control statements are programme statements that generate such jumps. Loops and decisions are the two major categories.

The number of times a loop is executed, or whether a choice causes a portion of code to be executed, is determined by whether particular expressions are true or false. A relational operator is a type of operator that compares two values and is commonly used in these expressions. Since the operation of loops and decisions is so closely involved with these operators, we'll examine them first.

Relational Operators

Relational operators are used for comparing two or more numerical values. C++ has different types of Operators which are used for carrying out numerous functions in the program. One such type used is the Relational Operators. We use Relational Operators for the decision-making process. This section contains information about the different types of Relational Operators, their uses, and their program examples. Have a look at the following program the further illustrates these operators.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int numb;
6     cout << "Enter a number: ";
7     cin >> numb;
8     cout << "numb<5 is " << (numb < 5) << endl;
```

```

9   cout << "numb>5 is " << (numb > 5) << endl;
10  cout << "numb==5 is " << (numb == 5) << endl;
11  return 0;
12  }

```

This program performs three kinds of comparisons between 5 and a number entered by the user. Here is the output when the user enters 20:

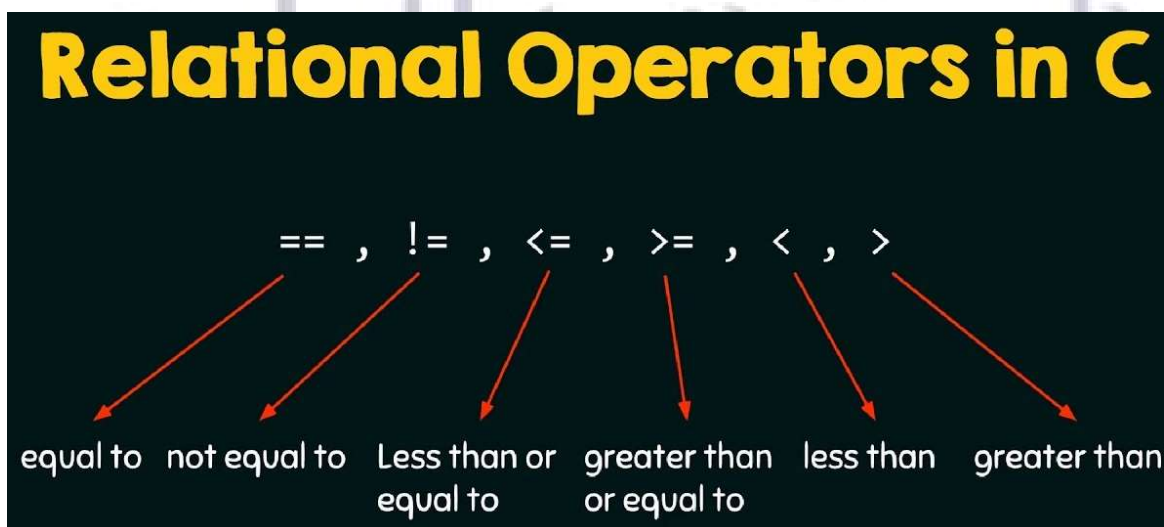
```

Enter a number: 20
numb<5   is  0
numb>5   is  1
numb==5  is  0

```

If numb is less than 10, the first expression holds true. If numb is larger than 10, the second expression is true, and if numb is equal to 10, the third expression is true. As you can see from the output, the C++ compiler assigns a value of 1 to a true expression and a value of 0 to a false expression.

Here is the complete list of C++ relational operators:



Loops and repeating code

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. For example, when you are displaying number from 1 to 100 you may want set the value of a variable to 1 and display it 100 times, increasing its value by 1 on each loop iteration. There are three kinds of loops in C++:

- **for loop**
- **while loop**
- **do loop**

The for Loop

The for loop is the simplest C++ loop to grasp (at least for many people). All of its loop control parts are in one location, however in other loop architectures, they are dispersed across the programme, making it more difficult to figure out how these loops function. The for loop repeats a portion of code for a predetermined number of times. It's frequently (but not always) utilised when you know how many times you want to run the code before entering the loop. The squares of the integers 0 to 14 are displayed in the following example:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int j; //define a loop variable
6     for(j=0; j<15; j++) //loop from 0 → 14,
7         cout << j * j << " "; //displaying the square of j
8
9     cout << endl;
10    return 0;
11 }
```

Here is the output:

```
0 1 4 9 16 25 36 49 64 81 100 121 144 169 196
```

How does this work? The for statement controls the loop. It consists of the keyword for, followed by parentheses that contain three expressions separated by semicolons:

```
for(j=0; j<15; j++)
```

These three expressions are the initialization expression, the test condition, and the increment expression, as shown in Figure 10.

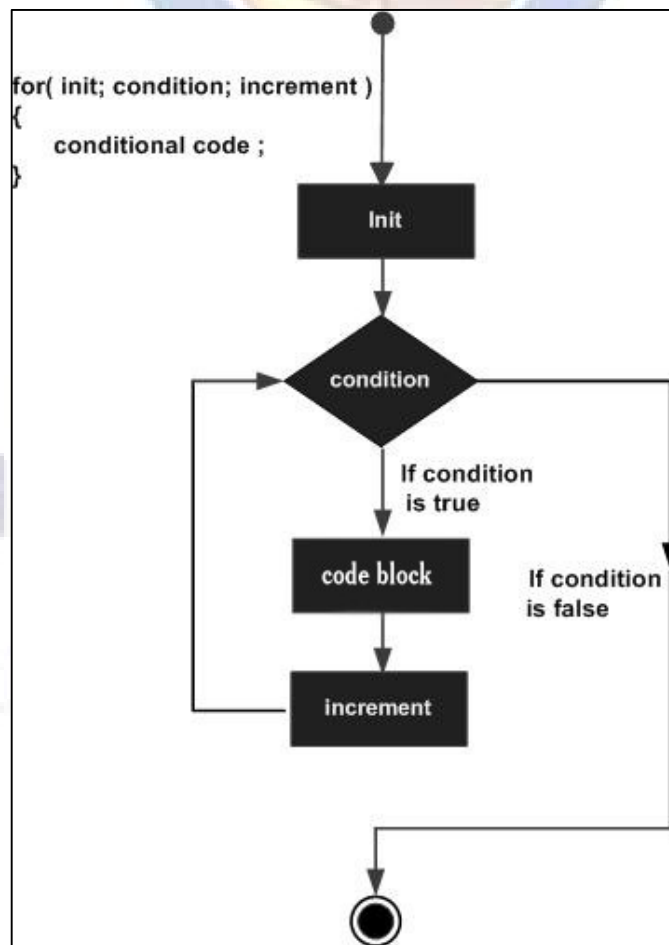


Figure 10 Structure of the for loop

These three expressions usually (but not always) involve the same variable, which we call the loop variable. In the previous example the loop variable is `j`. It's defined before the statements within the loop body start to execute.

The code that will be run each time the loop is repeated is the body of the loop. The loop's *raison d'être* is to repeat this code. The loop body in this example is made up of just one statement:

```
cout << j * j << " ";
```

It's worth noting that the `for` statement isn't preceded by a semicolon. Because the `for` statement and the loop body are both considered programme statements, this is the case. This is a crucial detail. If you use a semicolon after the `for` statement, the compiler will believe there is no loop body, and the programme will behave in unexpected ways. Let us see how the three expressions in the **for** statement control the loop.

The Initialization Expression

When the loop first starts, the initialization statement is only run once. It assigns a starting value to the loop variable. In the previous example it sets `j` to 0.

The Test Expression

A relational operator is frequently used in the test expression. It is assessed every time the loop is run, shortly before the loop's body is executed. It decides whether or not the loop will be repeated. If the test expression returns true, the loop is repeated. If it's false, the loop is terminated, and control is passed to the statements that follow it. The statement in the previous example was

```
cout << endl;
```

is executed following the completion of the loop.

The Increment Expression

The increment expression changes the value of the loop variable, often by incrementing it. It is always executed at the end of the loop, after the loop body has been executed. Here the increment operator ++ adds 1 to j each time through the loop. Figure 11 shows a flowchart of a for loop's operation.

A different for loop example:

```
for (count=0; count<200; count++)  
    // loop body
```

How many times will the loop body be repeated here? Exactly 199 times, with count going from 0 to 199.

How Many Times The loop is Repeated?

The loop in the previous example executes exactly 15 times. The first time, j is 0. This is ensured in the initialization expression. The last time through the loop, j is 14. This is determined by the test expression $j < 15$. When j becomes 15, the loop terminates; the loop body is not executed when j has this value. The arrangement shown is commonly used to do something a fixed number of times: start at 0, use a test expression with the less-than operator and a value equal to the desired number of iterations, and increment the loop variable after each iteration.

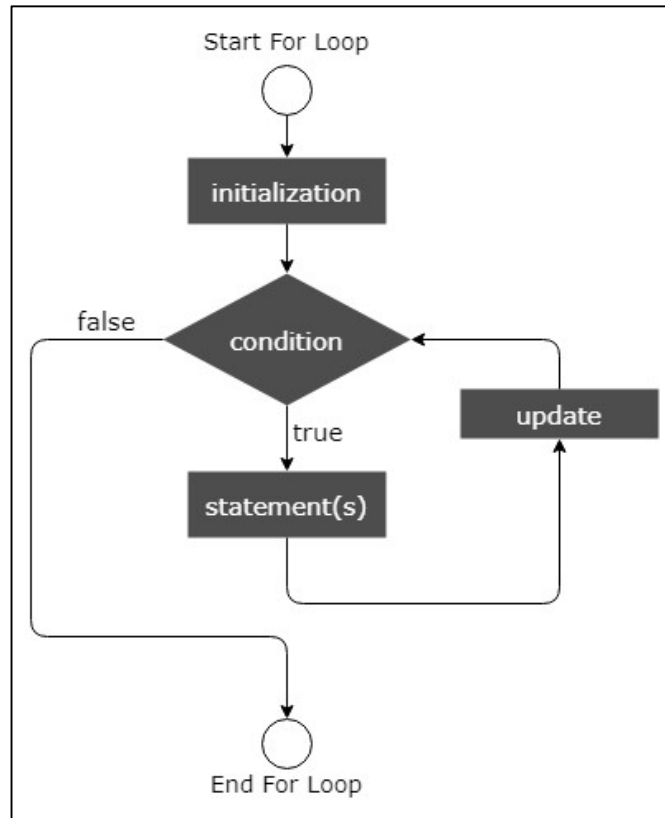


Figure 11 Operation of the for loop.

Multiple Statements in the For Loop Body

Of course, you may want to use the loop body to execute multiple statements. Braces, like functions, are used to separate several statements. Although there are semicolons after the different statements in the loop body, there is no semicolon after the final brace of the loop body. In the next example, the loop body has three statements. It uses a two-column format to print the cubes of the digits 1 to 10.

```

1 #include <iostream>
2 #include <iomanip>           //for setw
3 using namespace std;
4 int main()
5 {
6 int numb;                   //define loop variable

```



```

7 for(numb=1; numb<=11; numb++) //loop from 1 to 10
8 {
9     cout << setw(4) << numb; //display 1st column
10    int cube = numb*numb*numb; //calculate cube
11    cout << setw(6) << cube << endl; //display 2nd column
12 }
13 return 0;
14 }

```

Here is the output from the program:

```

1 1
2 8
3 27
4 64
5 125
6 216
7 343
8 512
9 729
10 1000
11 1331

```

We've made another change in the program to show there is nothing immutable about the format used in the last example. The loop variable is initialized to 1, not to 0, and it ends at 11, by virtue of `<=`, the less-than-or-equal-to operator. The effect is that the loop body is executed 11 times, with the loop variable running from 1 to 11.

Variable Visibility in Code Blocks

The loop body, which consists of braces delimiting several statements, is called a block of code. One important aspect of a block is that a variable defined inside the block is not visible outside it. Visible means that program

statements can access or “see” the variable. In previous program we define the variable cube inside the block, in the statement

```
int cube = numb*numb*numb;
```

This variable is only visible within the braces and cannot be accessed outside of the block. As a result, if you put the statement after the loop body, the compiler will complain since the variable cube is undefined outside the loop. One benefit of limiting variable accessibility is that the same variable name can be used in multiple blocks within the same programme.

Variations of the for Loop

The increment expression isn't required to increment the loop variable; it can do whatever it wants with it. The loop variable is decremented in the next example. The below program below, factorial, prompts the user to enter a number, after which it calculates the factorial of that number. (To compute the factorial, multiply the original number by all positive integers lower than itself.) $5*4*3*2*1$, or 120, is the factorial of 5.)

```
1 // Factorial program
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     unsigned int number;
7     unsigned long fact=1;           //long for larger numbers
8     cout << "Enter a number: ";
9     cin >> number;                 //get number
10    for(int j=number; j>0; j--)     //multiply 1 by
11        fact *= j; //numb, number-1, ..., 2, 1
```

```
12     cout << "Factorial is " << fact << endl;  
13     return 0;  
14 }
```

Notice that we used type unsigned long for the factorial because even small values have big factorials. On 32-bit systems like Windows, int is the same as long, however on 16-bit systems, long provides more space. Even for small input numbers, the following output shows how big factorials may be:

```
Enter a number: 12  
Factorial is 479001600
```

The largest number you can use for input is 12 (why?). You won't get an error message for larger inputs, but the results will be wrong, as the capacity of type long will be exceeded.

Variables Defined inside the for Statements

Notice how the variable j is defined inside the for loop, at the initialization expression:

```
for(int j=numb; j>0; j--)
```

Endless for loop

This is a common C++ construct and, in most circumstances, the ideal way to handle loop variables. It defines the variable as close as feasible to its point of use in the listing. Variables declared in the loop statement in this manner are only visible in the loop body. Actually, you can omit some or all of the expressions if you choose. The statement :

```
for (;;)
```

The while Loop

The for loop repeats something a certain number of times. **What if you don't know how many times you want to do something before starting the loop?** In this scenario, a second type of loop, the while loop, may be utilised.

The next example, asks the user to input a sequence of numbers. When the number entered is 0, the loop terminates.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int n = 88;           // make sure n isn't initialized to 0
6     while( n != 0 )      // loop until n is 0
7         cin >> n;       // read a number into n
8     cout << endl;
9     return 0;
10 }
```

Here is some sample interaction with the program. As long as the test expression is true, the loop continues to be executed.

```
1
270
303
1404
90
0
```

The while loop looks like a simplified version of the for loop. It contains a test expression but no initialization or increment expressions. Figure 12

shows the syntax of the while loop, and Figure 13, shows the operation of a while loop.

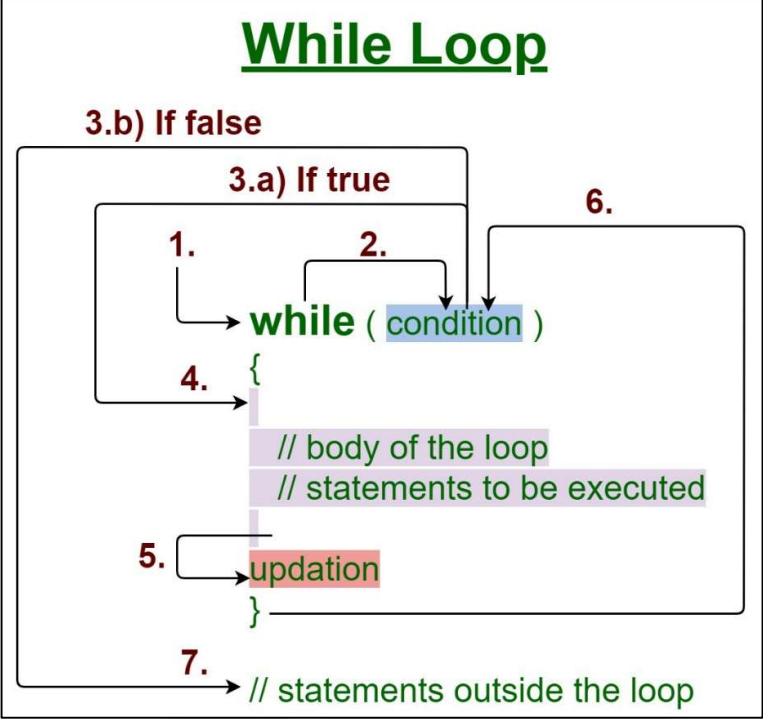


Figure 12 Syntax of the while loop.

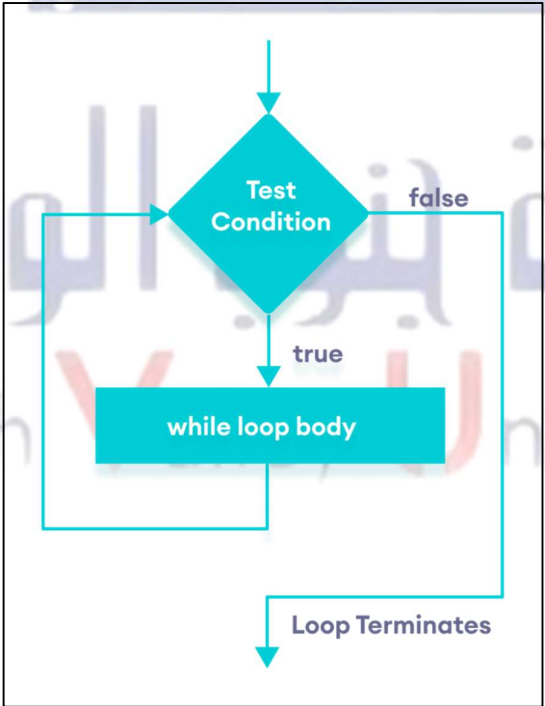


Figure 13 Operation of the while loop.

Multiple Statements inside a while Loop

The following example employs numerous statements within a while loop. It's a for loop-based variation on the cube programme presented before, except instead of the cube, it computes the fourth power of a series of integers. Assume that with this programme, it is critical to display the results in a four-digit column. To ensure that the results fit inside this column width, we must end the loop before the results exceed 9999. We don't know what number will produce a result of this size without prior calculation, so we let the programme figure it out. The while statement's test expression terminates the programme before the powers become too great.

```
1 #include <iostream>
2 #include <iomanip>           //for setw
3 using namespace std;
4 int main()
5 {
6     int pow=1;               //power initially 1
7     int numb=1;             //numb goes from 1 to ???
8     while( pow<1000 )       //loop while power <= 3 digits
9     {
10        cout << setw(2) << numb;
11        cout << setw(5) << pow << endl;
12        ++numb;
13        pow = numb*numb*numb;
14    }
15    cout << endl;
16    return 0;
17 }
```

We simply increase the third power (cubic) of numb by itself three times to discover it. We increment numb each time we run the loop. However, we

do not use `numb` in the `while` test statement; instead, the resulting value of `pow` determines when to end the loop. Here is the result:

```
1      1
2     16
3     81
4    256
5    625
```

The next number would be 1296, which is too wide for our three-digit column; but by this time the loop has terminated.

The do-while Loop

You know that, the test expression is evaluated at the beginning of a `while` loop. The loop body will not be executed if the test expression is false when the loop is entered. In certain cases, this is exactly what you want. However, **there are situations when you want to ensure that the loop body gets run at least once**, regardless of the initial state of the test expression. In this scenario, the **do-while** loop should be used, which places the test expression at the end of the loop.

Our next example, `divide_reminder`, asks the user to enter two numbers: a dividend (numerator) and a divisor (denominator). It then calculates the quotient (the answer) and the remainder, using the `/` and `%` operators, and prints out the result.

```

1 // Empty useless line
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     long dividend, divisor;
7     char ch;
8     do
9     {
10         cout << "Enter dividend: "; cin >> dividend;
11         cout << "Enter divisor: "; cin >> divisor;
12         cout << "Quotient is " << dividend / divisor;
13         cout << ", remainder is " << dividend % divisor;
14         cout << "\nDo another? (y/n): "; //do it again?
15         cin >> ch;
16     }
17     while( ch != 'n' ); //loop condition
18     return 0;
19 }
20
21
22

```

The do loop contains the majority of this programme. First, the term do denotes the start of the loop. The body of the loop is then delimited by braces, as with the other loops. Finally, a while statement completes the loop by providing the test expression. Except for its position at the end of the loop and the fact that it finishes with a semicolon (which is easy to miss!), this while statement appears quite similar to the one in a while loop. Figure 14 depicts the syntax of the do-while loop.

Do - While Loop

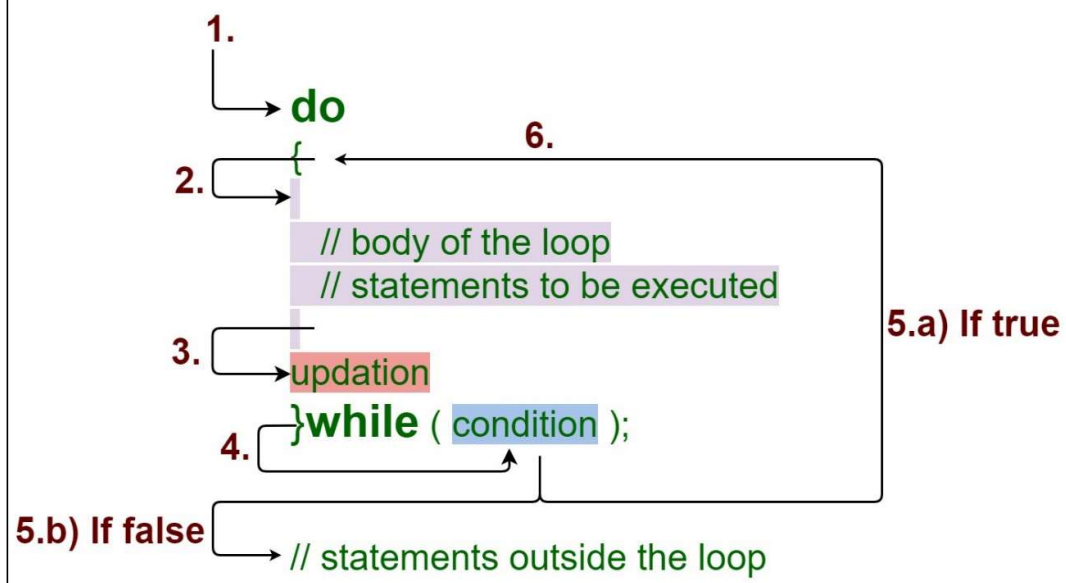


Figure 14 Syntax of the do loop.

Following each computation, `divde_reminder` asks if the user wants to do another. If so, the user enters a 'y' character, and the test expression

```
ch != 'n'
```

remains true. If the user enters 'n', the test expression becomes false and the loop ends. Figure 15 shows the process of the do loop. Here is an example of `divde_reminder` output:

```
Enter dividend: 11
Enter divisor: 3
Quotient is 3, remainder is 2
Do another? (y/n): y
Enter dividend: 10
Enter divisor: 2
Quotient is 5, remainder is 0
Do another? (y/n): n
```

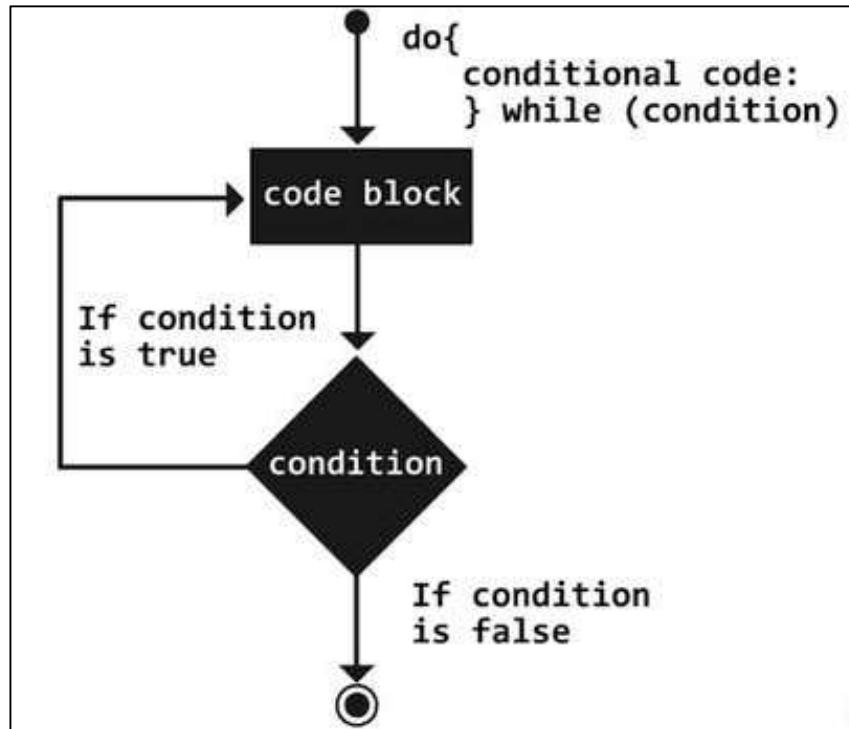


Figure 15 Operation of the do loop.

When to use which of the three Loops

We've made some broad statements regarding how loops work. When you know how many times the loop will be executed, the for loop is acceptable. The while and do loops are used when you don't know when the loop will end (the while loop when you're not sure if you want to execute the loop body even once, and the do loop when you're certain you want to execute the loop body at least once). These standards are somewhat arbitrary. Which loop type to use is a question of taste rather than hard and fast laws. Any of the loop types can be made to function in practically any situation. You should select the type that will make your software the clearest and easiest to understand.

Control Structures and Decisions in C++ Language

C++ conditional statements allow you to make a decision, based upon the result of a condition. These statements are called Decision Making Statements or Conditional Statements. So far, we have seen that all set of statements in a C++ program gets executed sequentially in the order in which they are written and appear. This occurs when there is no jump based statements or repetitions of certain calculations. But some situations may arise where we may have to change the order of execution of statements depending on some specific conditions. This involves a kind of decision making from a set of calculations. This type of structure requires that the programmers indicate several conditions for evaluation within a program. The statement(s) will get executed only if the condition becomes true and optionally, alternative statement or set of statements will get executed if the condition becomes false. The flowchart of the Decision-making technique in C++ can be expressed as follows:

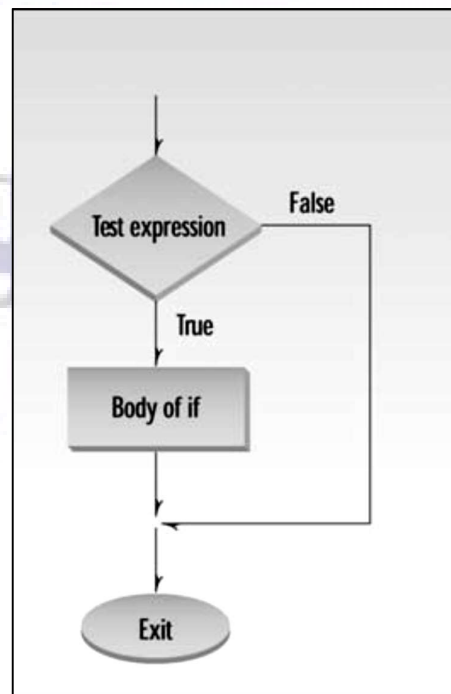


Figure 16 Operation of the if statement

The if Statement

The if statement is the simplest C++ decision statements. The next program, provides an example.

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int x;
6     cout << "Enter a number: ";
7     cin >> x;
8     if( x > 90 )
9         cout << "That number is greater than 90\n";
10    return 0;
11 }
12 }
```

The if keyword is followed by a parenthesized test expression. Figure 16 depicts the syntax of the if statement. As can be seen, the syntax of if is extremely similar to that of while. The distinction is that the statements following the if are only run once if the test expression is true, whereas the statements following the while are done continually until the test expression is false.

Here is an example of the previous program's output when the number entered by the user is greater than 90:

```
Enter a number: 2000
That number is greater than 90
```

If the number entered is not greater than 90, the program will terminate without printing the second line.



Figure 17 Syntax of the if statement (single line).

Multiple Statements in the if Body

As in loops, the code in an if body can consist of a single statement—as shown in the previous example—or a block of statements delimited by braces. You can include more than one single statement in the if block, as depicted in the following figure and program.

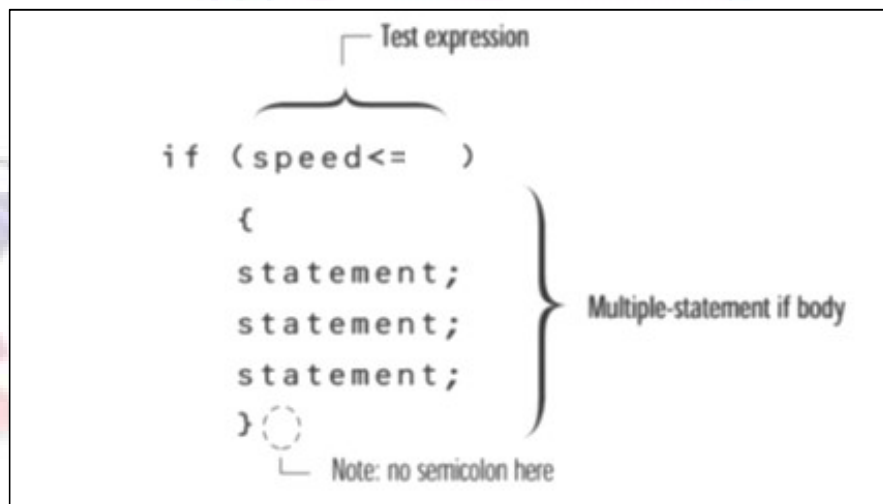


Figure 18 Syntax of the if statement (multi-line).

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int x;
6     cout << "Enter a number: ";
7     cin >> x;
8     if( x > 90 )
9     {
10         cout << "The number " << x;
11         cout << " is greater than 90\n";
12     }
13     return 0;
14 }

```

Here is some output from IF2:

```

Enter a number: 905
The number 905 is greater than 100

```

Embedding ifs Inside Loops

So far, we've seen that loop and decision structures can be layered inside of one another. Ifs can be nestled within loops, loops within ifs, ifs within ifs, and so on. The following is an example of nesting an if within a for loop. This example determines whether a number entered is a **prime number**. (Prime numbers are integers that can only be divided by themselves and 1.) (2, 3, 5, 7, 11, 13, 17) are the first few primes.

```

1 #include <iostream>
2 using namespace std;
3 #include <process.h> //for exit()

```

```

4 int main()
5 {
6 unsigned long n, j;
7 cout << "Enter a number: ";
8 cin >> n;
9 for(j=2; j <= n/2; j++)           //divide by every integer
10 if(n%j == 0)                     //if remainder is 0,
11 {                                 //it's divisible by j
12     cout << "It's not prime; divisible by " << j << endl;
13     exit(0);                       //exit from the program
14 }
15 cout << "It's prime\n";
16 return 0;
17 }

```

In this example, the user enters a number that is allocated to n. The program then uses a for loop to divide n by all the numbers from 2 to n/2. The loop variable, j, is the divisor. If any value of j divides evenly into n, then n is not prime. When a number divides evenly into another, the remainder is 0; we use the remainder operator % in the if statement to test for this condition with each value of j. If the number is not prime, we notify the user and exit the programme. There are no braces around the body of the loop. This is due to the fact that the if statement and the statements in its body are treated as a single statement. You can add braces for readability even if the compiler does not require them.

The following are the results of three separate invocations of the programme:

```

Enter a number: 7
It's prime
Enter a number: 22229

```

It's prime

Enter a number: 22231

It's not prime; divisible by 11

The `exit()` Library Function

When the previous programme discovers that an input number is not prime, it terminates immediately because proving a number is not prime more than once is pointless. This is performed through the use of the library function `exit()`. This function causes the programme to exit, regardless of where it is in the listing. It does not have a return value. When the programme exits, its sole argument, 0 in our case, is returned to the operating system. (This value is important in batch files, where the `ERRORLEVEL` value can be used to query the return value returned by `exit()`). Normally, the value 0 indicates a successful termination; other numbers indicate faults.)

Extra Decision: The `if...else` Statement

If a condition is true, the `if` statement allows you to do something. Nothing happens if it isn't true. However, suppose we wish to do one thing if a condition is true and another if it is false. This is when the `if...else` statement comes into play. It consists of an `if` statement, a statement or block of statements, the keyword `else`, and another statement or group of statements. Figure 19 depicts the syntax.

If - else statement

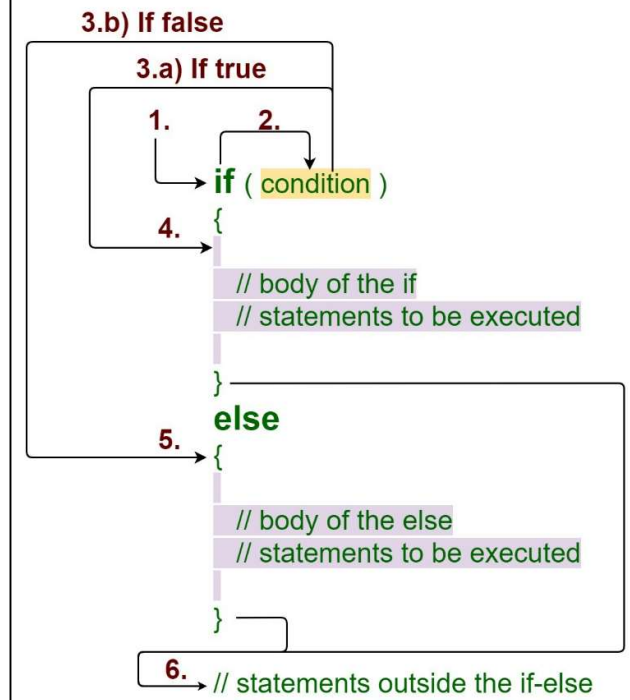


Figure 19 Syntax of the if...else statement.

Here is a variation of the original if-statement example, with an else added to the if:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int x;
6     cout << "\n Enter a number: ";
7     cin >> x;
8     if( x > 90 )
9         cout << "That number is greater than 90\n";
10    else
11        cout << "That number is not greater than 90\n";
12    return 0;
13 }
```

Here is output from two different invocations of the program:

```
Enter a number: 300
That number is greater than 100
Enter a number: 3
That number is not greater than 100
```

The operation of the if...else statement is shown in Figure 20.

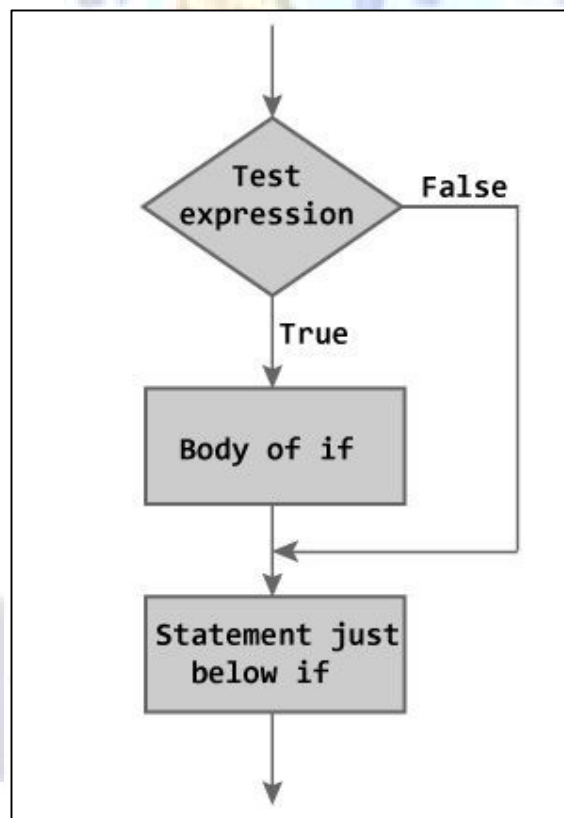


Figure 20 Operation of the if...else statement.

Matching the else with the right if statement

A potential issue with nested if...else statements is that you may mistakenly match an else with the incorrect if. The following program is an example.

```

1 // shows ELSE matched with the wrong IF
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     int a, b, c;
7     cout << "Enter three numbers, a, b, and c:\n";
8     cin >> x >> y >> z;
9     if( x==y )
10        if( y==z )
11            cout << "x, y, and z are the same\n";
12        else
13            cout << "x and y are different\n";
14    return 0;
15 }

```

We utilised several values with a **single cin**. Following each value you enter, press Enter; the three values will be allocated to x, y, and z.

What happens if you type 2, 3, and 3 again? Variable x equals 2, and variable y equals 3. Because they differ, the first test statement is false, and you would expect the else to be executed, because printed x and y differ. However, nothing is printed. What's the mistake? Because else is matched with the incorrect if. The indentation would lead you to believe that the else corresponds to the first if, **but it actually corresponds to the second if**. The rule is as follows: If the last else does not have its own else, it is matched with the last else.

This is the corrected version of the previous program:

```

if(x==y)
    if(y==z)
        cout << "x, y, and z are the same\n";
    else
        cout << "y and z are different\n";

```

We altered the indentation as well as the phrase written by the else body. Nothing will be printed if you type 2, 3, 3. However, typing 2, 2, 3 will result in different output y and z. If you truly want to connect an else with an earlier if, you can use braces around the inner if:

```

if(x==y)
{
    if(y==z)
        cout << "x, y, and z are the same";
}
else
    cout << "x and y are different";

```

Remember, brackets are the solution for matching else with the right if.

The else...if Construction

You can also use else if construction in the same line, carefully study the following program:

```

1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int time = 0;

```

```

7   cin>>time;
8   if (time < 10)
9   {
10      cout << "Good morning.";
11   }
12   else if (time < 20)
13      cout << "Good day.";
14   else
15      cout << "Good evening.";
16  return 0;
    }                                     //end main

```

More Decision Making: The switch Statement

Switch case statement is used when we have multiple conditions and we need to perform different action based on the condition. When we have multiple conditions and we need to execute a block of statements when a particular condition is satisfied. In such case either we can use lengthy if..else-if statement or switch case. The problem with lengthy if..else-if is that it becomes complex when we have several conditions. The switch case is a clean and efficient method of handling such scenarios.

```

1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int DAY;                               //day of the week
6  cout << "\nEnter 1, 2, or 3: ";
7  cin >> DAY;                             //user enters day

```

```

8  switch(DAY) //selection based on Day
9  {
10     case 1: //user entered 1
11         cout << "Saturday\n";
12         break;
13     case 2: //user entered 2
14         cout << "Sunday\n";
15         break;
16     case 3: //user entered 3
17         cout << "Monday\n";
18         break;
19 }
20 return 0;
21 }

```

This program prints one of three possible week days, depending on whether the user inputs the number 1, 2, or 3. The keyword **switch** is followed by a switch variable in parentheses. Figure 21 shows the syntax of the switch statement.

```
switch(Day)
```

Braces then delimit a number of case statements. Each case keyword is followed by a constant, which is not in parentheses but is followed by a colon.

```
case 1:
```

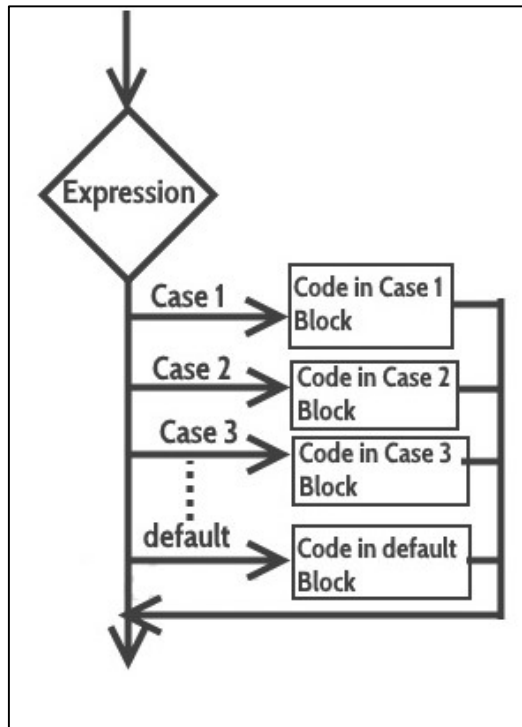


Figure 21 Syntax of the switch statement.

The switch variable should be assigned a value before the programme enters the switch. Typically, this value will match a constant in one of the case statements. When this happens, the statements that come after the keyword case are performed until a break is reached. Here's an example of the result:

Enter 1, 2, or 3:

1

Saturday

The break keyword importance

Break statements are used when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the execution flow would directly come out of the switch,

ignoring rest of the cases. This is why you must end each case block with the break statement.

switch Versus if...else

A switch statement is usually more efficient than a set of nested ifs. Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing.

- Check the Testing Expression: An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object. As shown below:

```
if( Pressure*Factor > 57 )
    // statements
else if( Voltage + 200 < 25000)
    // statements
else if( day==Monday )
    // statements
else
    // statements
```

- Switch better for Multi way branching: When compiler compiles a switch statement, it will inspect each of the case constants and create a “jump table” that it will use for selecting the path of execution depending on the value of the expression. Therefore, if we need to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses. The compiler can do this because it knows that the case constants are all the same type and simply must be compared for equality with the switch expression, while in case of if expressions, the compiler has no such knowledge.

- if-else better for boolean values: If-else conditional branches are great for variable conditions that result into a boolean, whereas switch statements are great for fixed data values.
- Speed: A switch statement might prove to be faster than ifs provided number of cases are good. If there are only few cases, it might not effect the speed in any case. Prefer switch if the number of cases are more than 5 otherwise, you may use if-else too. If a switch contains more than five items, it's implemented using a lookup table or a hash list. This means that all items get the same access time, compared to a list of ifs where the last item takes much more time to reach as it has to evaluate every previous condition first.
- Clarity in readability: A switch looks much cleaner when you have to combine cases.

The Conditional Operator

The conditional operator is also known as a ternary operator. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., `?` and `:`. As conditional operator works on three operands, so it is also known as the ternary operator. The behaviour of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement. For example, here is an **if...else** statement that gives the variable min the value of **num1** or the value of **num2**, depending on which is smaller:

```
if( num1 < num2 )
    min = num1;
else
    min = num2;
```

Here is the equivalent of the same program fragment, using a conditional operator:

```
min = (num1 < num2) ? num1 : num2;
```

the below figures depict the syntax and corresponding flow chart of the conditional operator.

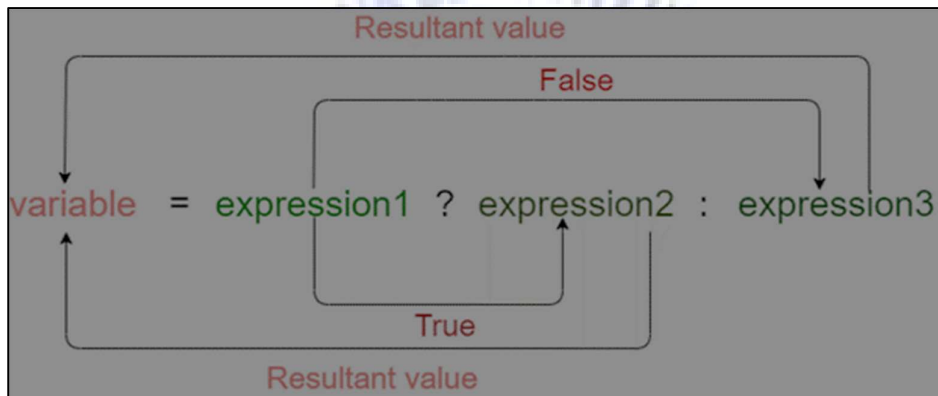


Figure 22 Syntax of the conditional operator.

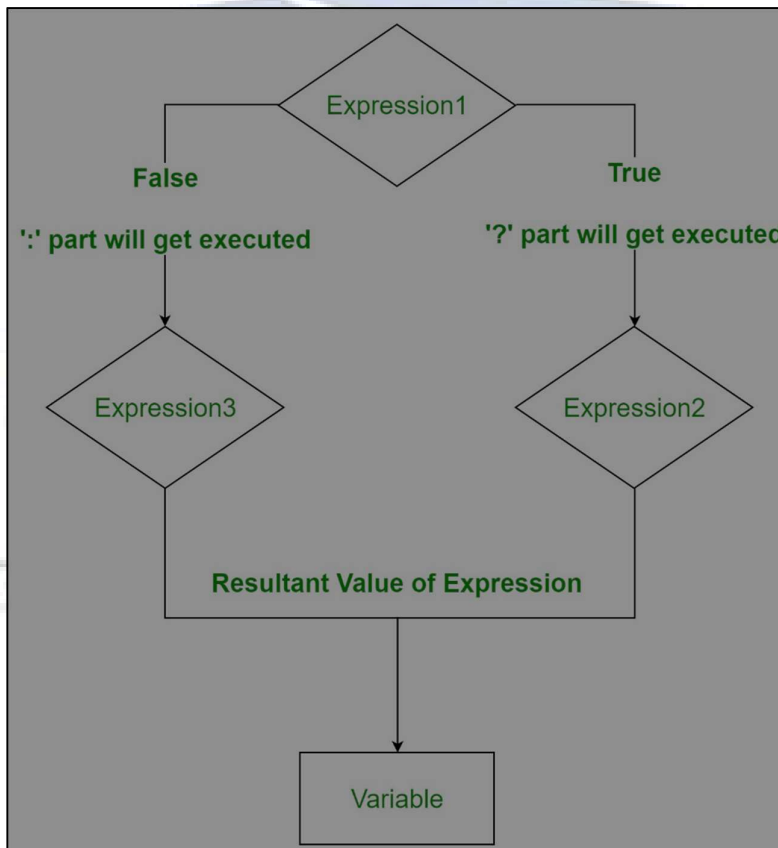


Figure 23 Operation of the conditional operator.

Here is another example: a statement that employs the conditional operator to determine the absolute value of the variable n (A number's absolute value is the number with any negative sign removed, therefore it is always positive.)

```
absval = n<0 ? -n : n;
```

If n is smaller than zero, the equation is -n, which is a positive number. If n is greater than zero, the expression remains n. The absolute value of n is allocated to **absval** as a result.

C++ programmers adore this kind of thing: getting a lot of bang for their buck with very little code. However, you are not required to try for concise code if you do not choose to.

Logical Operators in C++

C++ uses logical operators to check whether an expression is true or false. If the expression is true, it returns 1 whereas if the expression is false, it returns 0. There are three logical operators in C++, depicted in the below figure:

<i>Operator</i>	<i>Effect</i>
&&	Logical AND
	Logical OR
!	Logical NOT

Figure 24 C++ logical operators

Logical AND Operator

Here is an example, that uses the and logical operator .

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int x, y;
6     cout<<"Please enter two numbers" ;
7     cin>>x>>y;
8     if( x==8 && y==12 )
9     {
10         cout << " x is 8 and y is 12 \n";
11     }
12     Return 0;
13 }

```

The key to this program is the if statement

```
if( x==8 && y==12 )
```

The test expression will be true only if x is 8 and y is 12 at the same time. The logical AND operator && joins the two relational expressions to achieve this result. Notice that parentheses are not necessary around the relational expressions.

```
( (x==8) && (y==12) ) // inner parentheses not necessary
```

This is because relational operators take precedence over logical operators. Here is some interaction as the user arrives at these coordinates:

```
Please enter two numbers
```

```
8
```

```
12
```

```
x is 8 and y is 12
```

Logical OR Operator

Check the following program that illustrates the logical OR operator:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int x, y;
6     cout<<" Please enter two numbers" ;
7     cin>>x>>y;
8     if( x==5 || y==19 )
9     {
10         cout << " x is 8 OR y is 12 \n";
11     }
12     Return 0;
13 }
```

The expression:

```
x==5 || y==19
```

is true whenever either x is 5 or y is 19. Again, the `||` operator has lower precedence than the relational operators `<` and `>`, so no parentheses are needed in this expression.


Logical NOT Operator

The logical NOT operator `!` is a unary operator—that is, it takes only one operand. The effect of the `!` is that the logical value of its operand is reversed: If something is true, `!` makes it false; if it is false, `!` makes it true.

(It would be nice if life were so easily manipulated.) For example, $(x==8)$ is true if x is equal to 8, but $!(x==8)$ is true if x is not equal to 8.

Summary of Operator Precedence

Let us recap the precedence situation for the operators we've examined thus far. The operators at the top of the list take precedence over those at the bottom. Higher precedence operators are evaluated before lower precedence operators. The precedence of operators on the same row is the same. By enclosing an expression in parenthesis, you can force it to be evaluated first.

<code>() []</code>	Operators within parenthesis are performed first	Higher	
<code>++, --</code>	Postfix increment / decrement		
<code>++, --</code>	Prefix increment / decrement		
<code>*, /, %</code>	Multiplication, Division, Modulus		
<code>+, -</code>	Addition, Subtraction		
<code><, <=, >, >=</code>	Less than, Less than or equal to, Greater than, Greater than or equal to		
<code>==, !=</code>	Equal to, Not equal to		
<code>&&</code>	Logical AND		
<code> </code>	Logical OR		
<code>?:</code>	Conditional Operator		
<code>=</code>	Simple Assignment		
<code>+=, -=, *=, /=</code>	Shorthand operators		
<code>,</code>	Comma operator		Lower

General C++ programming Exercises

Write a C++ program that computes the volume of a room after inputting the dimensions of the room :

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     float volume, height, length, width;
6
7     cout <<"Enter height, length, width: ";
8     cin >> height>> length >> width;
9     volume = height * length * width;
10    cout << "volume = "<<volume<<endl;
11    return 0;
12 }
```

Write a C++ program that computes the area of the circle:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     double area, radius;
6     cout <<"Enter the radius ==> ";
7     cin >> radius;
8     area = 3.14 * radius * radius;
9     cout << "Area = "<< area <<endl;
10    return 0;
11 }
```

Write a program that allows inputting three real numbers, then computes their average:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     float average, num1, num2, num3;
6
7     cout <<"Enter three numbers ";
8     cin >> num1>>num2>>num3;
9     average = (num1+num2+num3) / 3;
10    cout << "Average = "<< average <<endl;
11    return 0;
12 }
```

Write a program that allows inputting three integer numbers, then prints the smallest number:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int smallest, num1, num2, num3;
6     cout <<"Enter three integer numbers ==> ";
7     cin >> num1 >> num2 >> num3;
8     smallest = num1;
9     if(num2 < smallest) smallest = num2;
10    if(num3 < smallest) smallest = num3;
11    cout << "The smallest is "<< smallest <<endl;
12    return 0;
13 }
```


Write a program that read a number n, then computes the summation of numbers from 1 to n:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int sum, n;
6     cout <<"Enter value for n ==> ";
7     cin >> n;
8     sum = n*(n+1)/2;
9     cout << "sum = "<< sum <<endl;
10
11     return 0;
12 }
```

Write a program that prints the statement "Hello world" 10 times on the screen:

```
1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     for(int i=0; i<10; i++)
6         cout<<"Hello world";
7
8     return 0;
9 }
```

Write a program that asks the user for his age in years and prints the equivalent in days:

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     float years, days;
6     cout<<"please enter your age";
7     cin>>years;
8     days= years * 365;
9     cout<<"your age in days is"<<days;
10    return 0;
11 }

```

Write a program that reads a group of n numbers and prints their average:

```

1 #include <iostream>
2 using namespace std;
3 int main()
4 {
5     int n;
6     float sum=0, Average=0, val;
7     cout<<"How many numbers?";
8     cin>>n;
9     for(int i=1; i<=n; i++)
10    {
11        cin>>val;
12        sum+=val;
13    }
14    Average=sum/n;
15    cout<<"Average is "<<Average;
16    return 0;
17 }

```

Exercises

1. Write an expression that uses a relational operator to return true if the variable `ASD` is not equal to `TXX`.
2. Is `-1` true or false?
3. The increment expression in a for loop can decrement the loop variable: True or false?
4. Write a for loop that displays the numbers from 150 to 110.
5. A block of code is delimited by
6. Write a while loop that displays the numbers from 190 to 195.
7. Relational operators have a higher precedence than arithmetic operators: True or false?
8. Write a program that asks the user to type all the integers between 9 and 15 (both included) using a do-while loop.
9. Write a program that asks the user to type 15 integers and writes the sum of these integers.
10. Write a program that asks the user to type 9 integers and writes the smallest value.
11. Write a program that asks the user to type positive numbers until either a zero or a negative is typed, and then show the user how many positives were typed in.
12. Write a program that asks the user to enter an integer number and determines if this number is odd or even.

Exam question 2019

Study the C++ program in (figure 3) then answer questions from 24 to 35:

[24] The value of h1 at line 6 is:

- (A) 0
- (B) 50
- (C) Undefined
- (D) No correct answer

[25] The cout output at line 7 is:

- (A) 25
- (B) 25.3
- (C) 20.25
- (D) No correct answer

[26] The cout output at line 10 is:

- (A) 50.6
- (B) 25
- (C) 50
- (D) 0

[27] The cout output at line 11 is:

- (A) 49
- (B) 48
- (C) 50
- (D) 51

[28] How many variables in the whole program :

- (A) 2 floats
- (B) 4 variables

[29] The value of h1 at line 12 is:

- (A) 2
- (B) 1

[30] The cout output at line 13 is:

- (A) No correct answer
- (B) 49

[31] How many time the loop at line 16 is repeated:

- (A) 8
- (B) 7

```
1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     float k1=10.1, k2=30.4;
6     int h1, h2=50;
7     cout<< k1+k2/2;
8     h1=k1+k2/2;
9     h1+=h1;
10    cout<< h1++;
11    cout<< --h1;
12    h1%=3;
13    cout<< h1+ (h2++);
14    cout<<(k1+k2)/h1;
15    h2/=6;
16    while(h2>0 && h2!=0)
17        cout<<h2--;
18    cout<<h2++;
19    return 0;
20 }
```

Figure (3)

(C) 4 integers

(D) 2 integers

(C) 0

(D) No correct answer

(C) 52

(D) 50

(C) Zero

(D) 9

Part 3

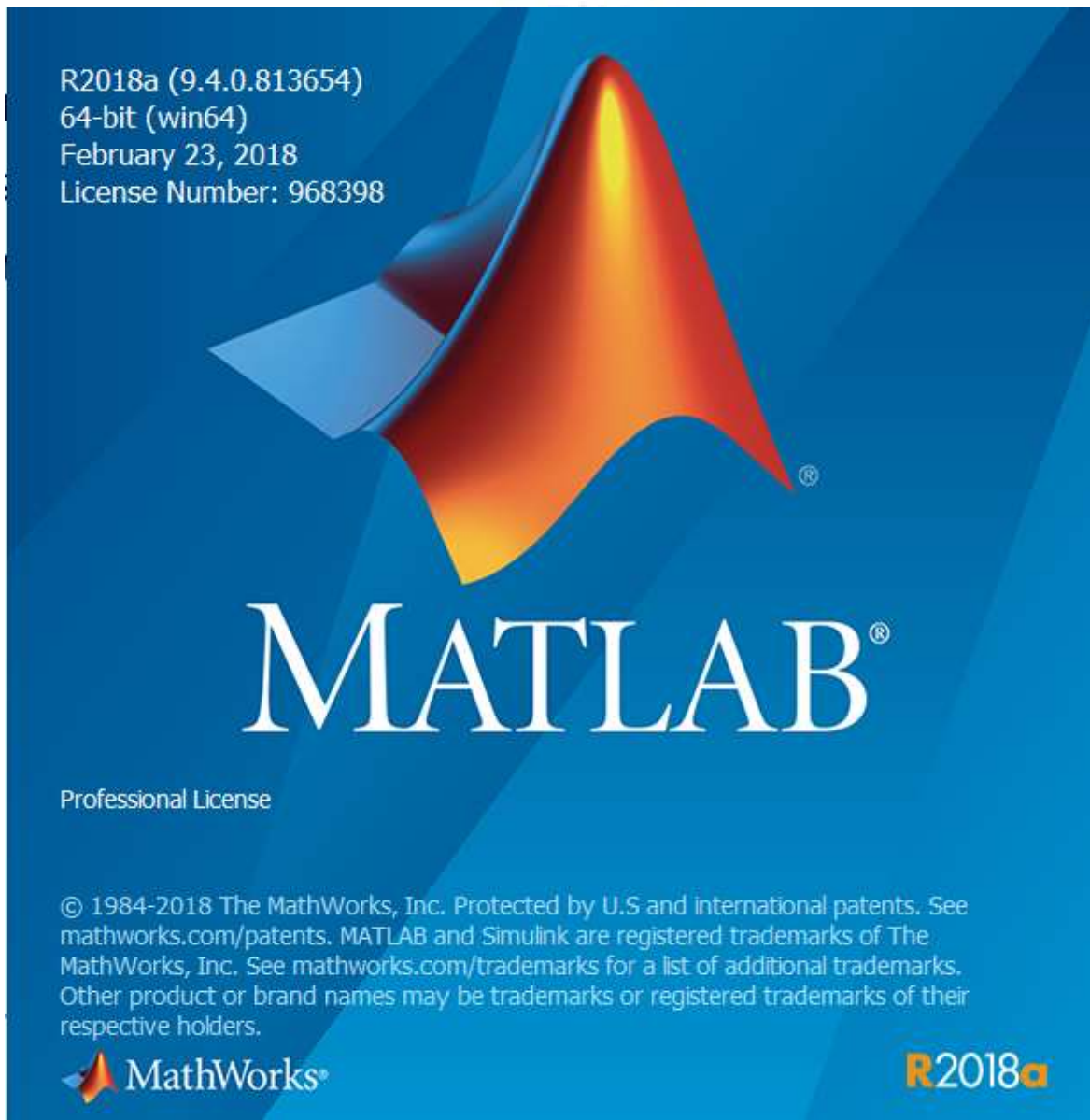


Chapter 1

MATLAB programming: An introduction



What is MATLAB


The image shows the front cover of the MATLAB R2018a software box. The background is a vibrant blue with a 3D surface plot in the center, colored with a gradient from blue to red to yellow. The plot has a sharp peak and a smooth valley. In the top left corner, white text provides version and license information. The word 'MATLAB' is prominently displayed in the center in a large, white, serif font. Below it, 'Professional License' is written in a smaller white font. At the bottom left is the MathWorks logo, and at the bottom right is the 'R2018a' version identifier.

R2018a (9.4.0.813654)
64-bit (win64)
February 23, 2018
License Number: 968398

MATLAB®

Professional License

© 1984-2018 The MathWorks, Inc. Protected by U.S and international patents. See mathworks.com/patents. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

 MathWorks®

R2018a

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Statistical computation
- Math computation
- Algorithm development
- Modelling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including Graphical User Interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C or Fortran.

The name MATLAB stands for matrix laboratory. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state-of-the-art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called toolboxes. Very important to most users of MATLAB, toolboxes allow you to learn and apply specialized technology. Toolboxes are comprehensive collections of

MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include statistical and mathematical computations, signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

The MATLAB system consists of five main parts:

The MATLAB language.

This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both "programming in the small" to rapidly create quick and dirty throw-away programs, and "programming in the large" to create complete large and complex application programs.

The MATLAB working environment.

This is the set of tools and facilities that you work with as the MATLAB user or programmer. It includes facilities for managing the variables in your workspace and importing and exporting data. It also includes tools for developing, managing, debugging, and profiling M-files, MATLAB's applications.

Handle Graphics.

This is the MATLAB graphics system. It includes high-level commands for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level commands that allow you to fully customize the appearance of

graphics as well as to build complete Graphical User Interfaces on your MATLAB applications.

The MATLAB statistical and mathematical function library.

This is a vast collection of computational algorithms ranging from elementary functions like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

The MATLAB Application Program Interface (API).

This is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.

Why to use MATLAB

MATLAB has several advantages over other methods or languages:

- MATLAB's functionality can be greatly expanded by the addition of toolboxes. These are sets of specific functions that provided more specialized functionality. Ex: Excel link allows data to be written in a format recognized by Excel, Statistics Toolbox allows more specialized statistical manipulation of data (Anova, Basic Fits, etc)
- Its basic data element is the matrix. A simple integer is considered an matrix of one row and one column. Several mathematical operations that work on arrays or matrices are built-in to the Matlab

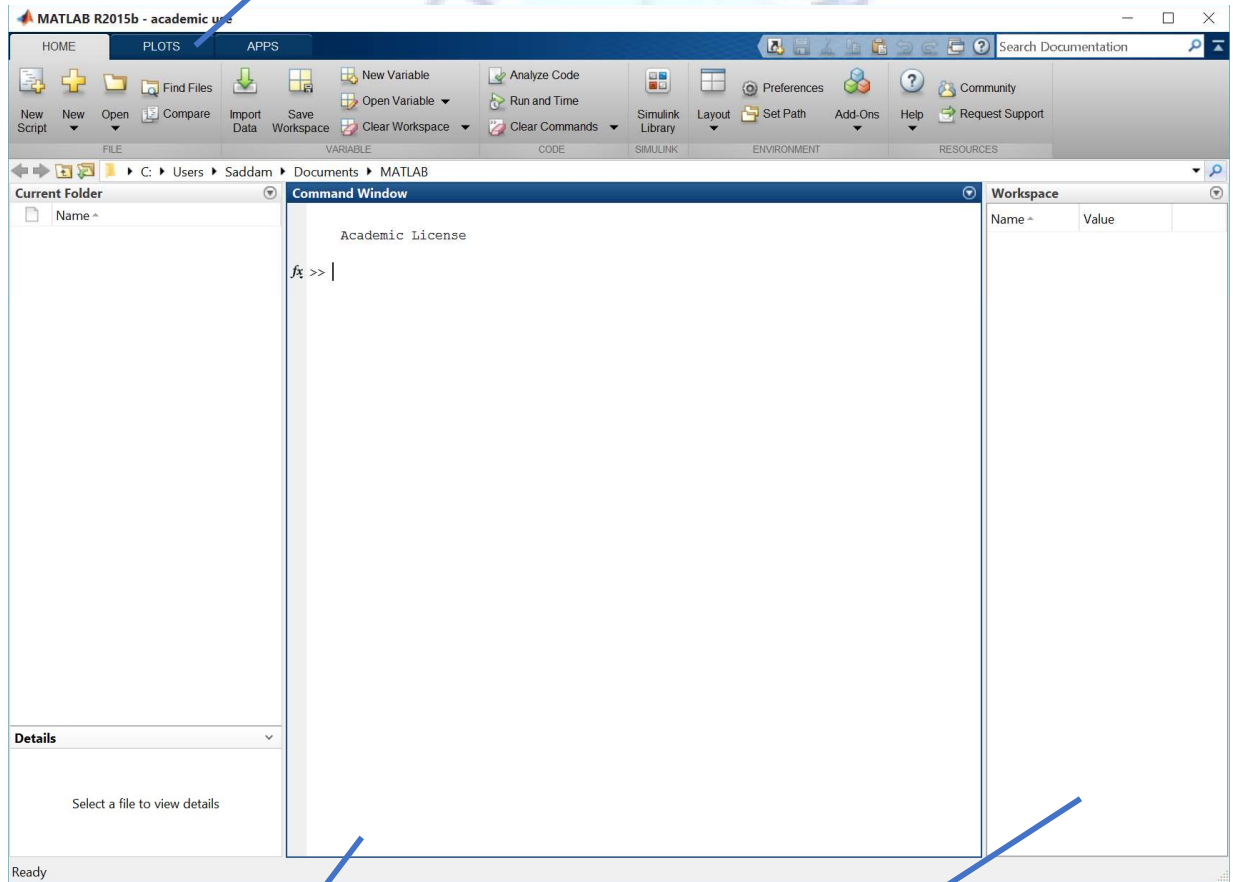
environment. For example, cross-products, dot-products, determinants, inverse matrices.

- Vectorized operations. Adding two arrays together needs only one command, instead of a for or while loop.
- The graphical output is optimized for interaction. You can plot your data very easily, and then change colors, sizes, scales, etc, by using the graphical interactive tools.
- MATLAB has a long history of refinement, and Has very good user documentation and a helpful support community.
- MATLAB has a very large library of built-in pre-written functions for many common numerical computing tasks.
- MATLAB is quite an informal language, allowing newcomers to get going and get results quickly.

Starting MATLAB

Once you finished installing MATLAB the below screen will appear. It's the main MATLAB interface and gives you access to all of its functionality. Study this window carefully to gain some insight about its components, as it would be the start point to inject your MATLAB code and run it.

Menu/Control bar



Command Window

Workspace Window

You are now faced with the MATLAB desktop on your computer, which contains the prompt (`>>`) in the Command Window. Once, you see the command prompt, it means you are ready to execute your Matlab code.

Using MATLAB as a calculator

As an example of a simple interactive calculation, just type the expression you want to evaluate. Let's start at the very beginning. For example, let us suppose you want to calculate the expression, $1 + 2 * 3$. You type it at the prompt command (`>>`) as follows,

```
>> 1+2*3
ans =
7
```

You will have noticed that if you do not specify an output variable, MATLAB uses a default variable **ans**, short for answer, to store the results of the current calculation. Note that the variable **ans** is created (or overwritten, if it is already existed). To avoid this, you may assign a value to a variable or output argument name. For example,

```
>> x = 1+2*3
x =
7
```

will result in **x** being given the value $1 + 2 * 3 = 7$. This variable name can always be used to refer to the results of the previous computations. Therefore, computing $4x$ will result in

```
>> 4*x
ans =
28.0000
```

Before we conclude this minimum session, Table 6 gives the partial list of arithmetic operators.

Table 5 Basic arithmetic operators

SYMBOL	OPERATION	EXAMPLE
+	Addition	$2 + 3$
-	Subtraction	$2 - 3$
*	Multiplication	$2 * 3$
/	Division	$2/3$

Quitting MATLAB

To end your MATLAB session, type quit in the Command Window, or select File -> Exit MATLAB in the desktop main menu.

```
>> quit
```

Getting started

After learning the minimum MATLAB session, we will now learn to use some additional operations.

Creating MATLAB variables

MATLAB variables are created with an assignment statement. The syntax of variable assignment is:

```
variable name = a value (or an expression)
```

For example,

```
>> x = expression
```

where expression is a combination of numerical values, mathematical operators, variables, and function calls. On other words, expression can involve:

- manual entry.
- built-in functions.
- user-defined functions.

Overwriting variable

Once a variable has been created, it can be reassigned. In addition, if you do not wish to see the intermediate results, you can suppress the numerical output by putting a semicolon (;) at the end of the line. Then the sequence of commands looks like this:

```
>> t = 5;  
>> t = t+1  
t =  
6
```

Error messages

If we enter an expression incorrectly, MATLAB will return an error message. For example, in the following, we left out the multiplication sign, *, in the following expression:

```
>> x = 10;
>> 5x
??? 5x
   |
Error: Unexpected MATLAB expression.
```

Making corrections

To make corrections, we can, of course retype the expressions. But if the expression is lengthy, we make more mistakes by typing a second time. A previously typed command can be recalled with the up-arrow key \uparrow . When the command is displayed at the command prompt, it can be modified if needed and executed.

Controlling the hierarchy of operations or precedence

Let's consider the previous arithmetic operation, but now we will include parentheses. For example, $1 + 2 * 3$ will become $(1 + 2) * 3$

```
>> (1+2)*3
ans =
     9
```

and, from previous example

```
>> 1+2*3
ans =
    7
```

By adding parentheses, these two expressions give different results: 9 and 7. The order in which MATLAB performs arithmetic operations is exactly that taught in high school algebra courses. Exponentiations are done first, followed by multiplications and divisions, and finally by additions and subtractions. However, the standard order of precedence of arithmetic operations can be changed by inserting parentheses. For example, the result of $1+2*3$ is quite different than the similar expression with parentheses $(1+2)*3$. The results are 7 and 9 respectively. Parentheses can always be used to overrule priority, and their use is recommended in some complex expressions to avoid ambiguity.

Therefore, to make the evaluation of expressions unambiguous, MATLAB has established a series of rules. The order in which the arithmetic operations are evaluated is given in Table 7. MATLAB arithmetic operators obey the same precedence rules as those in most computer programs. For operators of equal precedence, evaluation is from left to right.

Table 6 Hierarchy of arithmetic operations

PRECEDENCE	MATHEMATICAL OPERATIONS
First	The contents of all parentheses are evaluated first, starting from the innermost parentheses and working outward.
Second	All exponentials are evaluated, working from left to right
Third	All multiplications and divisions are evaluated, working from left to right
Fourth	All additions and subtractions are evaluated, starting from left to right

Now, consider another example:

$$\frac{1}{2+3^2} + \frac{4}{5} \times \frac{6}{7}$$

In MATLAB, it becomes

```
>> 1/(2+3^2)+4/5*6/7  
ans =  
    0.7766
```

or, if parentheses are missing,

```
>> 1/2+3^2+4/5*6/7  
ans =  
    10.1857
```

So here what we get: two different results. Therefore, we want to emphasize the importance of precedence rule in order to avoid ambiguity.

Controlling the appearance of floating point number

MATLAB by default displays only 4 decimals in the result of the calculations, for example -163.6667, as shown in above examples. However, MATLAB does numerical calculations in double precision, which is 15 digits. The command format controls how the results of computations are displayed. Here are some examples of the different formats together with the resulting outputs.

```
>> format short  
>> x=-163.6667
```

If we want to see all 15 digits, we use the command format long

```
>> format long
>> x= -1.636666666666667e+002
```

To return to the standard format, enter `format short`, or simply `format`. There are several other formats. For more details, see the MATLAB documentation, or type `help format`. Note - Up to now, we have let MATLAB repeat everything that we enter at the prompt (`>>`). Sometimes this is not quite useful, in particular when the output is pages en length. To prevent MATLAB from echoing what we type, simply enter a semicolon (`;`) at the end of the command. For example,

```
>> x=-163.6667;
```

and then ask about the value of `x` by typing,

```
>> x
x =
    -163.6667
```

Managing the workspace

The contents of the workspace persist between the executions of separate commands. Therefore, it is possible for the results of one problem to have an effect on the next one. To avoid this possibility, it is a good idea to issue a **clear** command at the start of each new independent calculation.

```
>> clear
```

The command `clear` or `clear all` removes all variables from the workspace. This frees up system memory. In order to display a list of the variables currently in the memory, type

```
>> who
```

while, **whos** will give more details which include size, space allocation, and class of the variables.

Keeping track of your work session

It is possible to keep track of everything done during a MATLAB session with the diary command.

```
>> diary
```

or give a name to a created file,

```
>> diary FileName
```

where FileName could be any arbitrary name you choose.

The function diary is useful if you want to save a complete MATLAB session. They save all input and output as they appear in the MATLAB window. When you want to stop the recording, enter diary off. If you want to start recording again, enter diary on. The file that is created is a simple text file. It can be opened by an editor or a word processing program and edited to remove extraneous material, or to add your comments. You can use the function type to view the diary file or you can edit in a text editor or print. This command is useful, for example in the process of preparing a homework or lab submission.

Entering multiple statements per line

It is possible to enter multiple statements per line. Use commas (,) or semicolons (;) to enter more than one statement at once. Commas (,) allow multiple statements per line without suppressing output.

```
>> a=7; b=cos(a), c=cosh(a)
b =
    0.6570
```

```
c =  
548.3170
```

Miscellaneous commands

Here are few additional useful commands:

- To clear the Command Window, type `clc`
- To abort a MATLAB computation, type `ctrl-c`
- To continue a line, type `...`

Getting help

To view the online documentation, select MATLAB Help from Help menu or MATLAB Help directly in the Command Window. The preferred method is to use the Help Browser. The Help Browser can be started by selecting the **?** icon from the desktop toolbar. On the other hand, information about any command is available by typing

```
>> help Command
```

Another way to get help is to use the **lookfor** command. The **lookfor** command differs from the help command. The help commands search for an exact function name match, while the **lookfor** command searches the quick summary information in each function for a match. For example, suppose that we were looking for a function to take the inverse of a matrix. Since MATLAB does not have a function named `inverse`, the command `help inverse` will produce nothing. On the other hand, the command **lookfor** `inverse` will produce detailed information, which includes the function of interest, `inv`.

```
>> lookfor inverse
```

Note - At this particular time of our study, it is important to emphasize one main point. Because MATLAB is a huge program; it is impossible to cover all the details of each function one by one. However, we will give you information how to get help. Here are some examples:

- Use on-line help to request info on a specific function

```
>> help sqrt
```

- In the current version, the doc function opens the on-line version of the help manual. This is very helpful for more complex commands.

```
>> doc plot
```

- Use **lookfor** to find functions by keywords. The general form is

```
>> lookfor FunctionName
```

Mathematical functions

MATLAB offers many pre-defined mathematical functions for technical computing which contains a large set of mathematical functions. Typing help elfun and help specfun calls up full lists of elementary and special functions respectively.

There is a long list of mathematical functions that are built into MATLAB. These functions are called built-ins. Many standard mathematical functions,

such as $\sin(x)$, $\cos(x)$, $\tan(x)$, e^x , $\ln(x)$, are evaluated by the functions `sin`, `cos`, `tan`, `exp`, and `log` respectively in MATLAB.

Table 8 lists some commonly used functions, where variables x and y can be numbers, vectors, or matrices.

Table 7 lists some commonly used functions, where variables x and y can be numbers,

<code>cos(x)</code>	Cosine
<code>sin(x)</code>	Sine
<code>tan(x)</code>	Tangent
<code>acos(x)</code>	Arc cosine
<code>asin(x)</code>	Arc sine
<code>atan(x)</code>	Arc tangent
<code>exp(x)</code>	Exponential
<code>sqrt(x)</code>	Square root
<code>log(x)</code>	Natural logarithm
<code>log10(x)</code>	Common logarithm
<code>abs(x)</code>	Absolute value
<code>sign(x)</code>	Signum function
<code>max(x)</code>	Maximum value
<code>min(x)</code>	Minimum value
<code>ceil(x)</code>	Round towards $+\infty$
<code>floor(x)</code>	Round towards $-\infty$
<code>round(x)</code>	Round to nearest integer
<code>rem(x)</code>	Remainder after division
<code>angle(x)</code>	Phase angle
<code>conj(x)</code>	Complex conjugate

In addition to the elementary functions, MATLAB includes a number of predefined constant values. A list of the most common values is given in Table 9.

Table 8 Predefined constant values

pi	The π number, $\pi = 3.14159\dots$
i, j	The imaginary unit $i, \sqrt{-1}$
Inf	The infinity, ∞
NaN	Not a number

Examples

We illustrate here some typical examples which related to the elementary functions previously defined.

As a first example, the value of the expression $y = e^{-a} \sin(x) + 10\sqrt{y}$ for $a = 5$, $x = 2$, and $y = 8$ is computed by

```
>> a = 5; x = 2; y = 8;  
>> y = exp(-a)*sin(x)+10*sqrt(y)  
y =  
    28.2904
```

The subsequent examples are

```
>> log(142)  
ans =  
    4.9558
```

```
>> log10(142)  
ans =  
    2.1523
```

Note the difference between the natural logarithm $\log(x)$ and the decimal logarithm (base 10) $\log_{10}(x)$.

To calculate $\sin(\pi/4)$ and e^{10} , we enter the following commands in MATLAB,

```
>> sin(pi/4)
ans =
    0.7071
>> exp(10)
ans =
    2.2026e+004
```

Notes:

- Only use built-in functions on the right-hand side of an expression. Reassigning the value to a built-in function can create problems.
- There are some exceptions. For example, i and j are pre-assigned to $\sqrt{-1}$. However, one or both of i or j are often used as loop indices.
- To avoid any possible confusion, it is suggested to use instead ii or jj as loop indices.

Exercises

1- Evaluate the following MATLAB expressions by hand and use MATLAB to check the answers

- a. $2 / 2 * 3$
- b. $6 - 2 / 5 + 7 ^ 2 - 1$
- c. $10 / 2 \setminus 5 - 3 + 2 * 4$
- d. $3 ^ 2 / 4$
- e. $3 ^ 2 ^ 2$
- f. $2 + \text{round}(6 / 9 + 3 * 2) / 2 - 3$
- g. $2 + \text{floor}(6 / 9 + 3 * 2) / 2 - 3$
- h. $2 + \text{ceil}(6 / 9 + 3 * 2) / 2 - 3$

2- Write down the MATLAB expression(s) that will

a. Compute the length of the hypotenuse of a right triangle given the lengths of the sides (try to do this for a vector of side-length values).

b. Compute the length of the third side of a triangle given the lengths of the other two sides, given the cosine rule

$$c^2 = a^2 + b^2 - 2(a)(b)\cos(t)$$

where t is the included angle between the given sides. Assume any arbitrary lengths for a and b.

3- Write down the MATLAB expression that will compute the average of the following numbers sequence 10, 11, 15, 0, 40 and 24.

4- Write down the MATLAB expression that computes the standard deviation of the previous numbers in exercise 3.

Chapter 2

Plotting in MATLAB



جامعة جنوب الوادي
South Valley University

Basic plotting

MATLAB has an excellent set of graphic tools. Plotting a given data set or the results of computation is possible with very few commands. You are highly encouraged to plot mathematical functions and results of analysis as often as possible. Trying to understand mathematical equations with graphics is an enjoyable and very efficient way of learning mathematics. Being able to plot mathematical functions and data freely is the most important step, and this section is written to assist you to do just that.

Creating simple plots

The basic MATLAB graphing procedure, for example in 2D, is to take a vector of x coordinates, $x = (x_1, \dots, x_N)$, and a vector of y-coordinates, $y = (y_1, \dots, y_N)$, locate the points (x_i, y_i) , with $i = 1, 2, \dots, n$ and then join them by straight lines. You need to prepare x and y in an identical array form; namely, x and y are both row arrays or column arrays of the same length.

The MATLAB command to plot a graph is **plot(x,y)**. The vectors $x = (1, 2, 3, 4, 5, 6)$ and $y = (3, -1, 2, 4, 5, 1)$ produce the picture shown in Figure 34.

```
>> x = [1 2 3 4 5 6];  
>> y = [3 -1 2 4 5 1];  
>> plot(x,y)
```

Note: The plot functions have different forms depending on the input arguments. If y is a vector plot(y) produces a piecewise linear graph of the elements of y versus the index of the elements of y. If we specify two vectors, as mentioned above, **plot(x,y)** produces a graph of y versus x.

For example, to plot the function $\sin(x)$ on the interval $[0; 2\pi]$, we first create a vector of x values ranging from 0 to 2π , then compute the sine of these values, and finally plot the result:

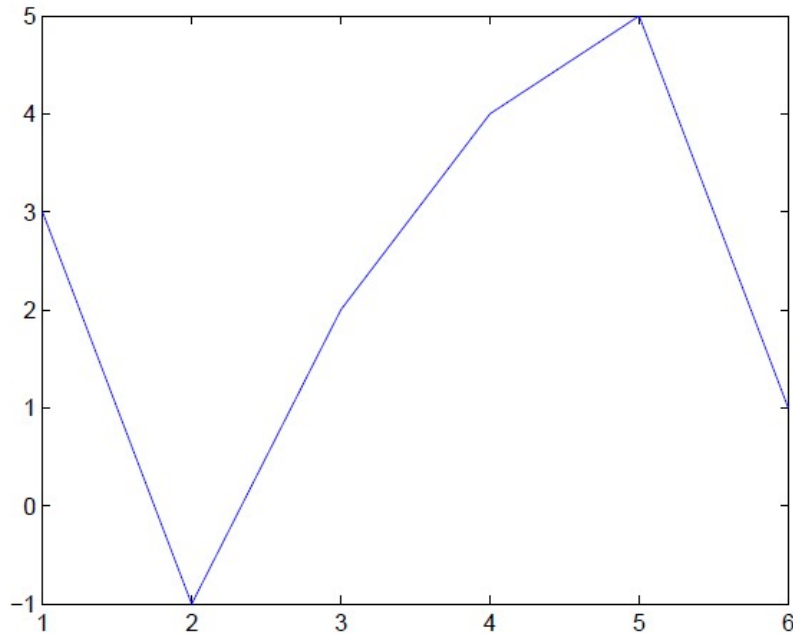


Figure 25 Plot for the vectors x and y

```
>> x = 0: pi/100:2*pi;  
>> y = sin(x);  
>> plot(x,y)
```

Notes:

- $0: \pi/100:2\pi$ yields a vector that
 - starts at 0,
 - takes steps (or increments) of $\pi=100$,
 - stops when 2π is reached.

- If you omit the increment, MATLAB automatically increments by 1.

Adding titles, axis labels, and annotations

MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an x- and y-axis labels.

Now label the axes and add a title. The character `\pi` creates the symbol π . An example of 2D plot is shown in Figure 35.

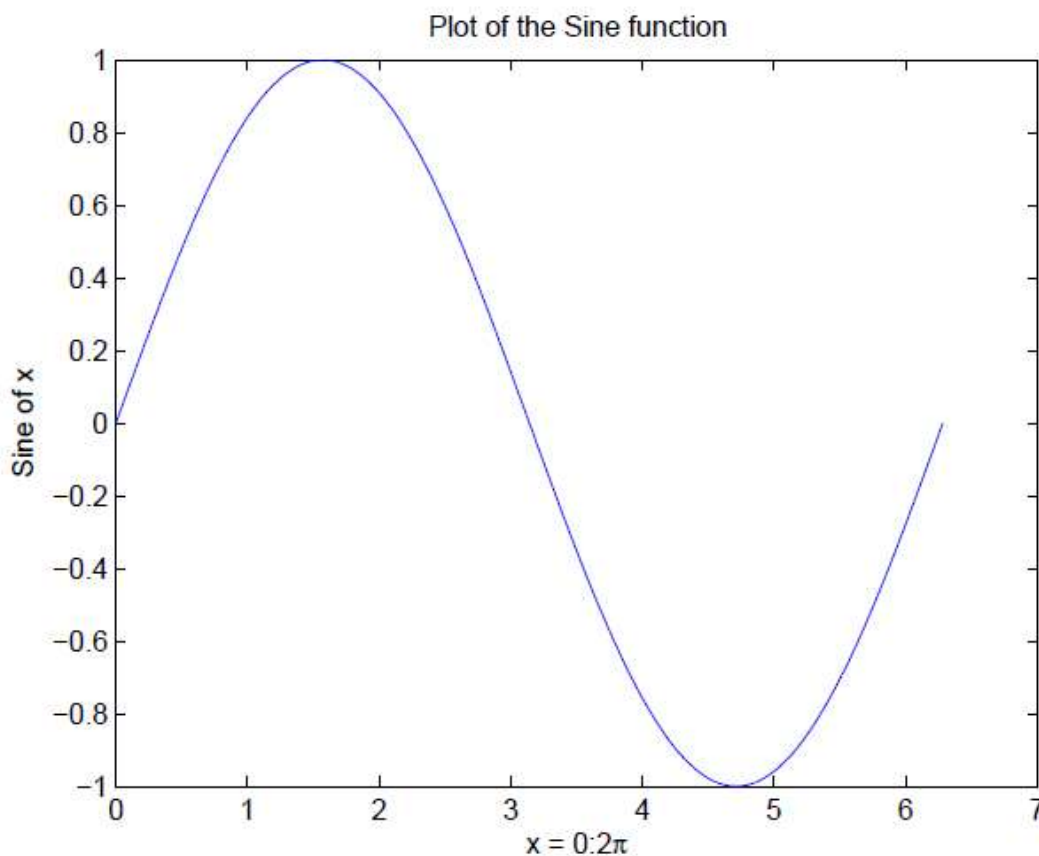


Figure 26 Plot of the Sine function

```
>> xlabel('x = 0:2\pi')  
>> ylabel('Sine of x')  
>> title('Plot of the Sine function')
```

The color of a single curve is, by default, **blue**, but other colors are possible. The desired color is indicated by a third argument. For example, **red** is selected by `plot(x,y,'r')`. Note the single quotes, ' ', around r.

Multiple data sets in one plot

Multiple (x; y) pairs arguments create multiple graphs with a single call to plot. For example, these statements plot three related functions of x: $y_1 = 2 \cos(x)$, $y_2 = \cos(x)$, and $y_3 = 0.5 * \cos(x)$, in the interval $0 \leq x \leq 2\pi$.

```
>> x = 0:pi/100:2*pi;
>> y1 = 2*cos(x);
>> y2 = cos(x);
>> y3 = 0.5*cos(x);
>> plot(x,y1,'--',x,y2,'-',x,y3,':')
>> xlabel('0 \leq x \leq 2\pi')
>> ylabel('Cosine functions')
>> legend('2*cos(x)', 'cos(x)', '0.5*cos(x)')
>> title('Typical example of multiple plots')
>> axis([0 2*pi -3 3])
```

The result of multiple data sets in one graph plot is shown in Figure 36.

By default, MATLAB uses line style and color to distinguish the data sets plotted in the graph. However, you can change the appearance of these graphic components or add annotations to the graph to help explain your data for presentation.

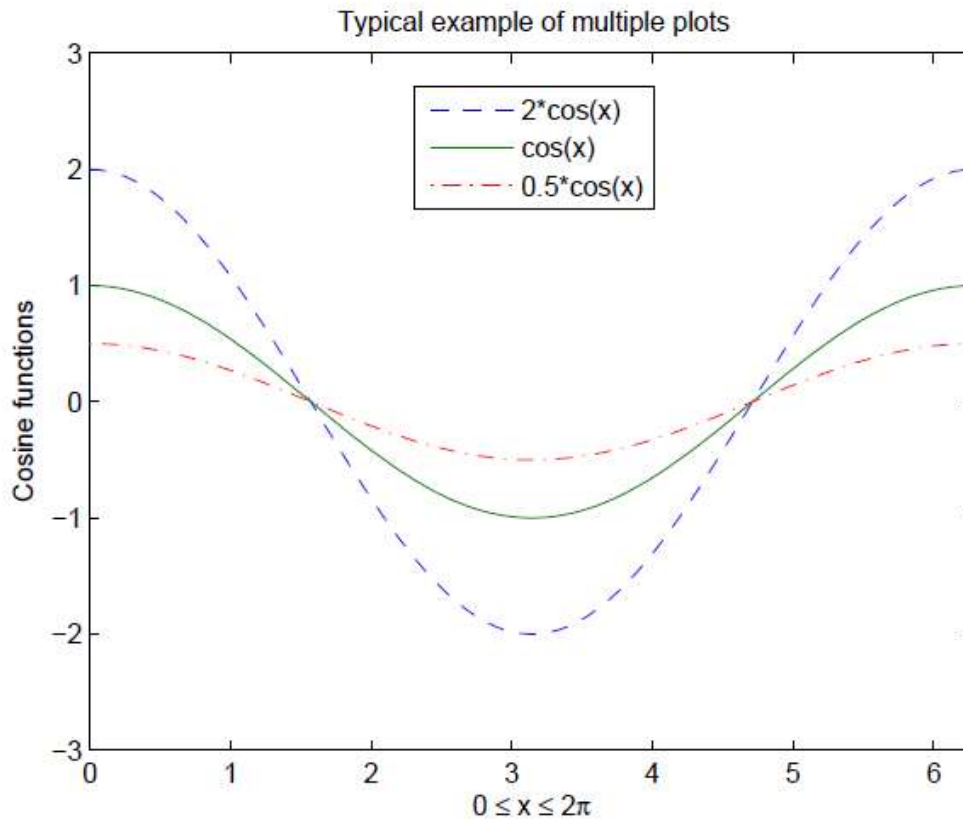


Figure 27 Typical example of multiple plots

Specifying line styles and colors

It is possible to specify line styles, colors, and markers (e.g., circles, plus signs, . . .) using the plot command:

```
plot(x,y,'style_color_marker')
```

where `style_color_marker` is a triplet of values from Table 10. To find additional information, type help `plot` or `doc plot`.

Table 9 Attributes for plot

SYMBOL	COLOR	SYMBOL	LINE STYLE	SYMBOL	MARKER
k	Black	—	Solid	+	Plus sign
r	Red	--	Dashed	o	Circle
b	Blue	:	Dotted	*	Asterisk
g	Green	-.	Dash-dot	.	Point
c	Cyan	none	No line	×	Cross
m	Magenta			s	Square
y	Yellow			d	Diamond



جامعة جنوب الوادي
South Valley University

Exercises

1. Using MATLAB plot the function of a straight line:

$$y = a x + b$$

where a is the slope of the line and b is the intercept part of the Y-axis, you can assume $x = [0, \dots, 10]$ and the slope is 2 with an intercept of zero value.

2. Execute the following MATLAB code, and try to guess what does the output plot statistically represents:

```
>> x = [-3:.1:3];  
norm = normpdf(x,0,1);  
figure;  
plot(x,norm)
```

3. Use MATLAB to sketch the following function

$$y = (x + 2)^2 - 3$$

assume the range $x = [-4, \dots, 4]$.

What does this equation called, and what does it represent?

4. Execute the following code in MATLAB and analyse the output graph

```
>> cx = input('Enter x coordinate: ');  
cy = input('Enter y coordinate: ');  
  
n = 50;  
R = 1;  
angle = 0:2*pi/n:2*pi;  
x = cx+R*cos(angle); y = cy+R*sin(angle);  
plot(x,y);  
axis equal;  
grid on;
```

Chapter 3

Statistical usages of Vectors and matrices in MATLAB



جامعة جنوب الوادي
South Valley University

Introduction

Matrices/vectors are the basic elements of the MATLAB environment. A matrix is a two-dimensional array consisting of m rows and n columns. Special cases are column vectors ($n = 1$) and row vectors ($m = 1$).

In this chapter we will illustrate how to apply different operations on matrices. The following topics are discussed: vectors and matrices in MATLAB, the inverse of a matrix, determinants, and matrix manipulation.

MATLAB supports two types of operations, known as matrix operations and array operations. Matrix operations will be discussed first, as vectors are special case from the matrices (One dimensional matrix).

Vectors in MATLAB

A vector is a special case of a matrix. The purpose of this section is to show how to create vectors and matrices in MATLAB. As discussed earlier, an array of dimension $1 \times n$ is called a row vector, whereas an array of dimension $m \times 1$ is called a column vector. The elements of vectors in MATLAB are enclosed by square brackets and are separated by spaces or by commas. For example, to enter a row vector, v , type

```
>> v = [1 4 7 10 13]
v =
    1  4  7 10 13
```

Column vectors are created in a similar way, however, semicolon (;) must separate the components of a column vector,

```
>> w = [1;4;7;10;13]
w =
     1
     4
     7
    10
    13
```

```
4
7
10
13
```

On the other hand, a row vector is converted to a column vector using the transpose operator. The transpose operation is denoted by an apostrophe or a single quote (').

```
>> w = v'  
w =  
1  
4  
7  
10  
13
```

Thus, $v(1)$ is the first element of vector v , $v(2)$ its second element, and so forth. Furthermore, to access blocks of elements, we use MATLAB's colon notation (:). For example, to access the first three elements of v , we write,

```
>> v(1:3)  
ans =  
1 4 7
```

Or, all elements from the third through the last elements,

```
>> v(3:end)  
ans =  
7 10 13
```

where end signifies the last element in the vector. If v is a vector, writing

```
>> v(:)
```

produces a column vector, whereas writing

```
>> v(1:end)
```

produces a row vector.

Creating a matrix

A matrix is an array of numbers. To type a matrix into MATLAB you must:

- Begin with a square bracket, [
- Separate elements in a row with spaces or commas (,)
- Use a semicolon (;) to separate rows
- End the matrix with another square bracket,].

Here is a typical example. To enter a matrix A, such as,

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

type,

```
>> A = [1 2 3; 4 5 6; 7 8 9]
```

MATLAB then displays the 3 x 3 matrix as follows,

```
A =  
    1    2    3  
    4    5    6  
    7    8    9
```

Note that the use of semicolons (;) here is different from their use mentioned earlier to suppress output or to write multiple commands in a single line. Once we have entered the matrix, it is automatically stored and remembered in the Workspace. We can refer to it simply as matrix A. We can then view a particular element in a matrix by specifying its location. We write,

```
>> A(2,1)
ans =
4
```

A(2,1) is an element located in the second row and first column. Its value is 4.

Matrix indexing

We select elements in a matrix just as we did for vectors, but now we need two indices. The element of row i and column j of the matrix A is denoted by $A(i,j)$. Thus, $A(i,j)$ in MATLAB refers to the element A_{ij} of matrix A . The first index is the row number and the second index is the column number. For example, $A(1,3)$ is an element of first row and third column. Here, $A(1,3)=3$.

Correcting any entry is easy through indexing. Here we substitute $A(3,3)=9$ by $A(3,3)=0$. The result is:

```
>> A(3,3) = 0
A =
 1  2  3
 4  5  6
 7  8  0
```

Single elements of a matrix are accessed as $A(i,j)$, where $i \geq 1$ and $j \geq 1$. Zero or negative subscripts are not supported in MATLAB.

Colon operator

The colon operator will prove very useful and understanding how it works is the key to efficient and convenient usage of MATLAB. It occurs in several different forms.

Often we must deal with matrices or vectors that are too large to enter one element at a time. For example, suppose we want to enter a vector x consisting of points $(0, 0.1, 0.2, 0.3, \dots, 5)$. We can use the command

```
>> x = 0:0.1:5;
```

The row vector has 51 elements.

Linear spacing

On the other hand, there is a command to generate linearly spaced vectors: **linspace**. It is similar to the colon operator (:), but gives direct control over the number of points. For example,

```
y = linspace(a,b)
```

generates a row vector y of 100 points linearly spaced between and including a and b .

```
y = linspace(a,b,n)
```

generates a row vector y of n points linearly spaced between and including a and b . This is useful when we want to divide an interval into a number of subintervals of the same length. For example,

```
>> theta = linspace(0,2*pi,101)
```

divides the interval $[0, 2\pi]$ into 100 equal subintervals, then creating a vector of 101 elements.

Colon operator in a matrix

The colon operator can also be used to pick out a certain row or column. For example, the statement $A(m:n,k:l)$ specifies rows m to n and column k to l . Subscript expressions refer to portions of a matrix. For example,

```
>> A(2,:)
ans =
    4    5    6
```

is the second row elements of A.

The colon operator can also be used to extract a sub-matrix from a matrix A, as in the following example.

```
>> A(:,2:3)
ans =
    2    3
    5    6
    8    0
```

$A(:,2:3)$ is a sub-matrix with the last two columns of A.

A row or a column of a matrix can be deleted by setting it to a null vector, $[]$.

```
>> A(:,2)=[]
ans =
```



```
1 3
4 6
7 0
```

Creating a sub-matrix

To extract a submatrix B consisting of rows 2 and 3 and columns 1 and 2 of the matrix A, do the following:

```
>> B = A([2 3],[1 2])
B =
    4    5
    7    8
```

To interchange rows 1 and 2 of A, use the vector of row indices together with the colon operator.

```
>> C = A([2 1 3],:)
C =
    4    5    6
    1    2    3
    7    8    0
```

It is important to note that the colon operator (:) stands for all columns or all rows. To create a vector version of matrix A, do the following:

```
>> A(:)
ans =
    1
    2
    3
    4
    5
    6
```

```
7
8
0
```

The submatrix comprising the intersection of rows p to q and columns r to s is denoted by $A(p:q,r:s)$.

As a special case, a colon ($:$) as the row or column specifier covers all entries in that row or column; thus

- $A(:,j)$ is the j^{th} column of A .
- $A(i,:)$ is the i^{th} row.
- $A(\text{end},:)$ picks out the last row of A .

The keyword `end`, used in $A(\text{end},:)$, denotes the last index in the specified dimension. Here are some examples.

```
>> A
A =
     1     2     3
     4     5     6
     7     8     9
```

```
>> A(2:3,2:3)
ans =
     5     6
     8     9
```

```
>> A(end:-1:1,end)
ans =
     9
     6
     3
```

```
>> A([1 3],[2 3])
ans =
     2     3
     8     9
```

Deleting row or column

To delete a row or column of a matrix, use the empty vector operator, [].

```
>> A(3,:) = []
A =
     1     2     3
     4     5     6
```

Third row of matrix A is now deleted. To restore the third row, we use a technique for creating a matrix

```
>> A = [A(1,:);A(2,:);[7 8 0]]
A =
     1     2     3
     4     5     6
     7     8     0
```

Matrix A is now restored to its original form.

Matrix Dimensions

To determine the dimensions of a matrix or vector, use the command size. For example,

```
>> size(A)
ans =
     3     3
```

means 3 rows and 3 columns.

Or more explicitly with,

```
>> [m,n]=size(A)
```

Transposing a matrix

The transpose operation is denoted by an apostrophe or a single quote ('). It flips a matrix about its main diagonal and it turns a row vector into a column vector. Thus,

```
>> A'
ans =
     1     4     7
     2     5     8
     3     6     0
```

By using linear algebra notation, the transpose of $m \times n$ real matrix A is the $n \times m$ matrix that results from interchanging the rows and columns of A . The transpose matrix is denoted A^T .

Concatenating matrices

Matrices can be made up of sub-matrices. Here is an example. First, let us recall our previous matrix A .

```
A = [
1   2   3,
4   5   6,
7   8   9]
```

The new matrix B will be,

```
>> B = [A 10*A; -A [1 0 0; 0 1 0; 0 0 1]]
B =
 1 2 3 10 20 30
 4 5 6 40 50 60
 7 8 9 70 80 90
-1 -2 -3 1 0 0
-4 -5 -6 0 1 0
-7 -8 -9 0 0 1
```

Matrix generators

MATLAB provides functions that generates elementary matrices. The matrix of zeros, the matrix of ones, and the identity matrix are returned by the function's zeros, ones, and eye, respectively.

Table 10 Elementary matrices

<code>eye(m,n)</code>	Returns an m-by-n matrix with 1 on the main diagonal
<code>eye(n)</code>	Returns an n-by-n square identity matrix
<code>zeros(m,n)</code>	Returns an m-by-n matrix of zeros
<code>ones(m,n)</code>	Returns an m-by-n matrix of ones
<code>diag(A)</code>	Extracts the diagonal of matrix A
<code>rand(m,n)</code>	Returns an m-by-n matrix of random numbers

For a complete list of elementary matrices and matrix manipulations, type `help elmat` or `doc elmat`. Here are some examples:

```
>> b=ones(3,1)
```

```
b =
```

```
1  
1  
1
```

```
>> eye(3)
```

```
ans =
```

```
1    0    0  
0    1    0  
0    0    1
```

```
>> c=zeros(2,3)
```

```
c =
```

```
0    0    0  
0    0    0
```

In addition, it is important to remember that the three elementary operations of addition (+), subtraction (-), and multiplication (⋈) apply also to matrices whenever the dimensions are compatible.

Two other important matrix generation functions are `rand` and `random`, which generate matrices of (pseudo-)random numbers using the same syntax as `eye`.

In addition, matrices can be constructed in a block form. With `C` defined by `C = [1 2; 3 4]`, we may create a matrix `D` as follows:

```
>> D = [C zeros(2); ones(2) eye(2)]
```

```
D =
```

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

Special matrices

MATLAB provides a number of special matrices (see Table 12). These matrices have interesting properties that make them useful for constructing examples and for testing algorithms.

For more information, see MATLAB documentation.

Table 11 Special Matrices

<code>hilb</code>	Hilbert matrix
<code>invhilb</code>	Inverse Hilbert matrix
<code>magic</code>	Magic square
<code>pascal</code>	Pascal matrix
<code>toeplitz</code>	Toeplitz matrix
<code>vander</code>	Vandermonde matrix
<code>wilkinson</code>	Wilkinson's eigenvalue test matrix

Array operations

MATLAB has two different types of arithmetic operations: matrix arithmetic operations and array arithmetic operations. We have seen matrix arithmetic operations in the previous lab. Now, we are interested in array operations that will facilitate for the statistical analysis.

As we mentioned earlier, MATLAB allows arithmetic operations: +, i, x, and ^ to be carried out on matrices. Thus,

- A+B or B+A is valid if A and B are of the same size
- A*B is valid if A's number of column equals B's number of rows
- A^2 is valid if A is square and equals A*A
- α *A or A* α multiplies each element of A by α

On the other hand, array arithmetic operations or array operations for short, are done element-by-element. The period character, ., distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition (+) and subtraction (i), the character pairs (.+) and (.-) are not used. The list of array operators is shown below in Table 13. If A and B are two matrices of the same size with elements $\mathbf{A} = [a_{ij}]$ and $\mathbf{B} = [b_{ij}]$, then the command

Table 12 Array operations

.*	Element-by-element multiplication
./	Element-by-element division
.^	Element-by-element exponentiation

```
>> C = A.*B
```

produces another matrix C of the same size with elements $c_{ij} = a_{ij} b_{ij}$. For example, using the same 3 x 3 matrices,

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}$$

we have,

```
>> C = A.*B
C =
    10    40    90
   160   250   360
   490   640   810
```

To raise a scalar to a power, we use for example the command 10^2 . If we want the operation to be applied to each element of a matrix, we use 2 . For example, if we want to produce a new matrix whose elements are the square of the elements of the matrix A, we enter

```
>> A.^2
ans =
     1     4     9
    16    25    36
    49    64    81
```

The relations below summarize the above operations. To simplify, let us consider two vectors U and V with elements $U = [u_i]$ and $V = [v_j]$.

$U.*V$	produces	$[u_1v_1 \ u_2v_2 \ \dots \ u_nv_n]$
$U./V$	produces	$[u_1/v_1 \ u_2/v_2 \ \dots \ u_n/v_n]$
$U.^V$	produces	$[u_1^{v_1} \ u_2^{v_2} \ \dots \ u_n^{v_n}]$

Matrix inverse

Let's consider the same matrix A.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Calculating the inverse of A manually is probably not a pleasant work. Here the hand calculation of A^{-1} gives as a final result:

$$A^{-1} = \frac{1}{9} \begin{bmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{bmatrix}$$

In MATLAB, however, it becomes as simple as the following commands:

```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> inv(A)
ans =
-1.7778    0.8889   -0.1111
 1.5556   -0.7778    0.2222
-0.1111    0.2222   -0.1111
```

which is similar to:

$$A^{-1} = \frac{1}{9} \begin{bmatrix} -16 & 8 & -1 \\ 14 & -7 & 2 \\ -1 & 2 & -1 \end{bmatrix}$$

and the determinant of A is

```
>> det(A)
ans =
    27
```

Matrix functions

MATLAB provides many matrix functions for various matrix/vector manipulations; see Table 14 for some of these functions. Use the online help of MATLAB to find how to use these functions.

Table 13 Matrix functions

<code>det</code>	Determinant
<code>diag</code>	Diagonal matrices and diagonals of a matrix
<code>eig</code>	Eigenvalues and eigenvectors
<code>inv</code>	Matrix inverse
<code>norm</code>	Matrix and vector norms
<code>rank</code>	Number of linearly independent rows or columns



Exercises

1. Write the Matlab code that converts temperature in degrees Fahrenheit (F) to degrees Centigrade (C). Use input and **fprintf** commands to display a mix of text and numbers. Recall the conversion formulation, $C = \frac{5}{9} * (F - 32)$.
2. Write the Given $x = [3 \ 1 \ 5 \ 7 \ 9 \ 2 \ 6]$, explain what the following commands "mean" by summarizing the net result of the command.
 - a. $x(3)$
 - b. $x(1:7)$
 - c. $x(1:end)$
 - d. $x(1:end-1)$
 - e. $x(6:-2:1)$
 - f. $x([1 \ 6 \ 2 \ 1 \ 1])$
 - g. $\text{sum}(x)$
3. Given the array $A = [\ 2 \ 4 \ 1 \ ; \ 6 \ 7 \ 2 \ ; \ 3 \ 5 \ 9]$, provide the commands needed to
 - a. assign the first row of A to a vector called x1
 - b. assign the last 2 rows of A to an array called y
 - c. compute the sum over the columns of A
 - d. compute the sum over the rows of A
 - e. compute the standard error of the mean of each column of A (NB. the standard error of the mean is defined as the standard deviation divided by the square root of the number of elements used to compute the mean.)

4. Give the following commands to create an array called F:

```
>> randn('seed',123456789)
F = randn(5,10);
```

- Compute the mean of each column and assign the results to the elements of a vector called avg.
- Compute the standard deviation of each column and assign the results to the elements of a vector called s.
- Compute the vector of t-scores that test the hypothesis that the mean of each column is no different from zero.
- If $\Pr(|t| > 2.132) = 0.1$ with 4 degrees of freedom, are any of the mean values in the vector avg statistically different from 0?

(hint for question 4, look at the below Matlab code)

```
N = size(F,1)
avg = mean(F)
s = std(F)
tscore = (avg - 0)./(s/sqrt(N))
None were different at (all < 2.132).
```

- Create a vector x containing integer numbers from 1 to 100. Create a vector y containing numbers 1, 0.9, 0.8, 0.7, . . . 0.1, 0 in this order.
- Create matrix 3x3 with all ones. Create matrix 8 by 1 with all zeros. Create matrix 5 by 2 with all elements equal to 0.37.
- Given $A = [1 \ 1; 2 \ 2; 3 \ 3; 4 \ 100]$ Run the MATLAB code:
average = mean(A)
med = median(A)
dev = std(A)



Chapter 4
Statistical Applications of
MATLAB

جامعة بنو الوادي
South Valley University

You may need to summarize large, complex data sets—both numerically and visually—to convey their essence to the data analyst and to allow for further processing. This chapter focuses on numerical summaries.

Measures of Central Tendency

Measures of central tendency locate a distribution of data along an appropriate scale. The following table lists the functions that calculate the measures of central tendency.

The following table lists the MATLAB functions that calculate the measures of central tendency.

Function Name	Description
geomean	Geometric mean
harmmean	Harmonic mean
mean	Arithmetic average
median	50th percentile
mode	Most frequent value
trimmean	Trimmed mean

The **average** is a simple and popular estimate of location. If the data sample comes from a normal distribution, then the sample mean is also optimal.

Unfortunately, outliers, data entry errors, or glitches exist in almost all real data. The sample mean is sensitive to these problems. One bad data value can move the average away from the centre of the rest of the data by an arbitrarily large distance.

The **median** and trimmed mean are two measures that are resistant (robust) to outliers. The median is the 50th percentile of the sample, which will only change slightly if you add a large perturbation to any value. The idea behind the trimmed mean is to ignore a small percentage of the highest and lowest values of a sample when determining the center of the sample.

The **geometric mean** and **harmonic mean**, like the average, are not robust to outliers. They are useful when the sample is distributed lognormal or heavily skewed.

The following example shows the behaviour of the measures of location for a sample with one outlier.

```
>>x = [ones(1,6) 100]
x =
1 1 1 1 1 1 100
```

```
>>locate = [geomean(x) harmmean(x) mean(x) median(x)...
trimmean(x,25)]
locate =
1.9307 1.1647 15.1429 1.0000 1.0000
```

You can see that the mean is far from any data value because of the influence of the outlier. The median and trimmed mean ignore the outlying value and describe the location of the rest of the data values.

Measures of Dispersion

The purpose of measures of dispersion is to find out how spread out the data values are on the number line. Another term for these statistics is measures of spread.

The table gives the function names and descriptions.

Function Name	Description
iqr	Interquartile range
mad	Mean absolute deviation
moment	Central moment of all orders
range	Range
std	Standard deviation
var	Variance

The **range** (the difference between the maximum and minimum values) is the simplest measure of spread. But if there is an outlier in the data, it will be the minimum or maximum value. Thus, the range is not robust to outliers.

The **standard deviation** and the **variance** are popular measures of spread that are optimal for normally distributed samples. The sample variance is. The standard deviation is the square root of the variance and has the desirable property of being in the same units as the data. That is, if the data is in meters, the standard deviation is in meters as well. The variance is in meters², which is more difficult to interpret.

Neither the standard deviation nor the variance is robust to outliers. A data

value that is separate from the body of the data can increase the value of the statistics by an arbitrarily large amount.

The mean absolute deviation (**MAD**) is also sensitive to outliers. But the MAD does not move quite as much as the standard deviation or variance in response to bad data.

The interquartile range (**IQR**) is the difference between the 75th and 25th percentile of the data. Since only the middle 50% of the data affects this measure, it is robust to outliers.

The following example shows the behaviour of the measures of dispersion for a sample with one outlier.

```
>>x = [ones(1,6) 100]
x =
     1     1     1     1     1     1    100
stats = [iqr(x) mad(x) range(x) std(x)]
stats =
     0    24.2449    99.0000    37.4185
```

Measures of Shape

Quantiles and percentiles provide information about the shape of data as well as its location and spread.

The quantile of order p ($0 \leq p \leq 1$) is the smallest x value where the cumulative distribution function equals or exceeds p . The function `quantile` computes quantiles as follows:

1. n sorted data points are the $0.5/n, 1.5/n, \dots, (n-0.5)/n$ quantiles.
2. Linear interpolation is used to compute intermediate quantiles.
3. The data min or max are assigned to quantiles outside the range.
4. Missing values are treated as NaN, and removed from the data.

Percentiles, computed by the `prctile` function, are quantiles for a certain percentage of the data, specified for $0 \leq p \leq 100$.

The following example shows the result of looking at every quartile (quantiles with orders that are multiples of 0.25) of a sample containing a mixture of two distributions.

```
x = [normrnd(4,1,1,100) normrnd(6,0.5,1,200)];
p = 100*(0:0.25:1);
y = prctile(x,p);
z = [p;y]
z =
0          25.0000   50.0000   75.0000   100.0000
1.8293    4.6728    5.6459    6.0766    7.1546
```

A box plot helps to visualize the statistics:

```
boxplot(x)
```

The box plot figure is depicted in Figure 37 below.

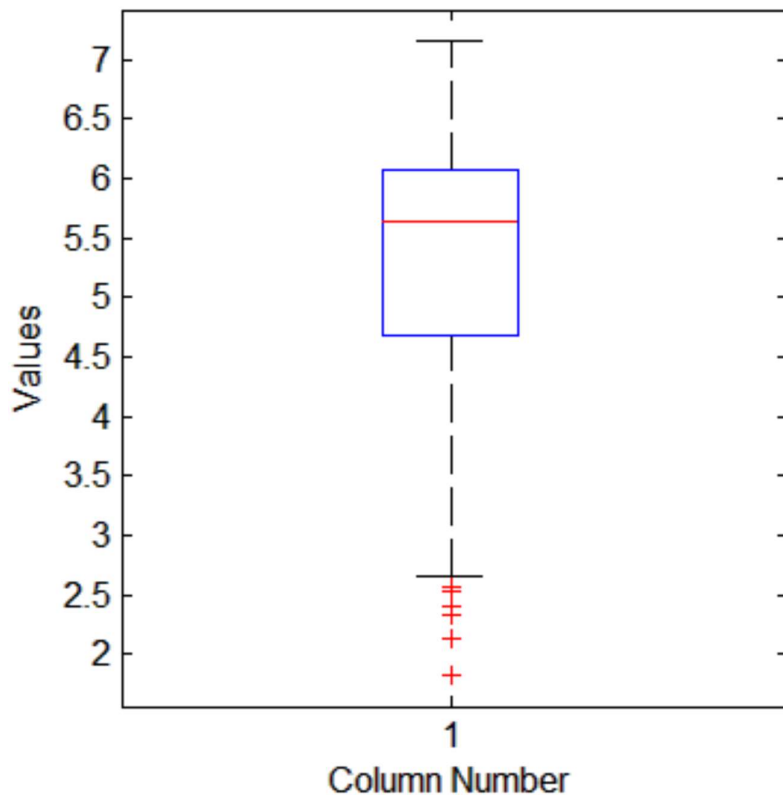


Figure 28 Box plot

The long lower tail and plus signs show the lack of symmetry in the sample values. For more information on box plots.

The shape of a data distribution is also measured by the Statistics Toolbox functions skewness, kurtosis, and, more generally, moment.

Data with Missing Values

Many data sets have one or more missing values. It is convenient to code missing values as **NaN** (Not a Number) to preserve the structure of data sets across multiple variables and observations.

For example:

```
X = magic(3);
X([1 5]) = [NaN NaN]
X =
     NaN     1     6
     3     NaN     7
     4     9     2
```

Normal MATLAB arithmetic operations yield NaN values when operands are NaN:

```
s1 = sum(X)
s1 =
NaN     NaN     15
```

Removing the NaN values would destroy the matrix structure. Removing the rows containing the NaN values would discard data. Statistics Toolbox functions in the following table remove NaN values only for the purposes of computation.

Function	Description
nancov	Covariance matrix, ignoring NaN values
nanmax	Maximum, ignoring NaN values
nanmean	Mean, ignoring NaN values
nanmedian	Median, ignoring NaN values
nanmin	Minimum, ignoring NaN values
nanstd	Standard deviation, ignoring NaN values
nansum	Sum, ignoring NaN values
nanvar	Variance, ignoring NaN values

For example:

```
s2 = nansum(X)
s2 =
7 10 15
```

Other Statistics Toolbox functions also ignore NaN values. These include `iqr`, `kurtosis`, `mad`, `prctile`, `range`, `skewness`, and `trimmean`.

Statistical Visualization

Statistics Toolbox data visualization functions add to the extensive graphics capabilities already in MATLAB.

- Scatter plots are a basic visualization tool for multivariate data. They are used to identify relationships among variables. Grouped versions of these plots use different plotting symbols to indicate group membership. The `gname` function is used to label points on these plots with a text label or an observation number.
- Box plots display a five-number summary of a set of data: the median, the two ends of the interquartile range (the box), and two extreme values (the whiskers) above and below the box. Because they show less detail than histograms, box plots are most useful for side-by-side comparisons of two distributions.
- Distribution plots help you identify an appropriate distribution family for your data. They include normal and Weibull probability plots, quantile-quantile plots, and empirical cumulative distribution plots.

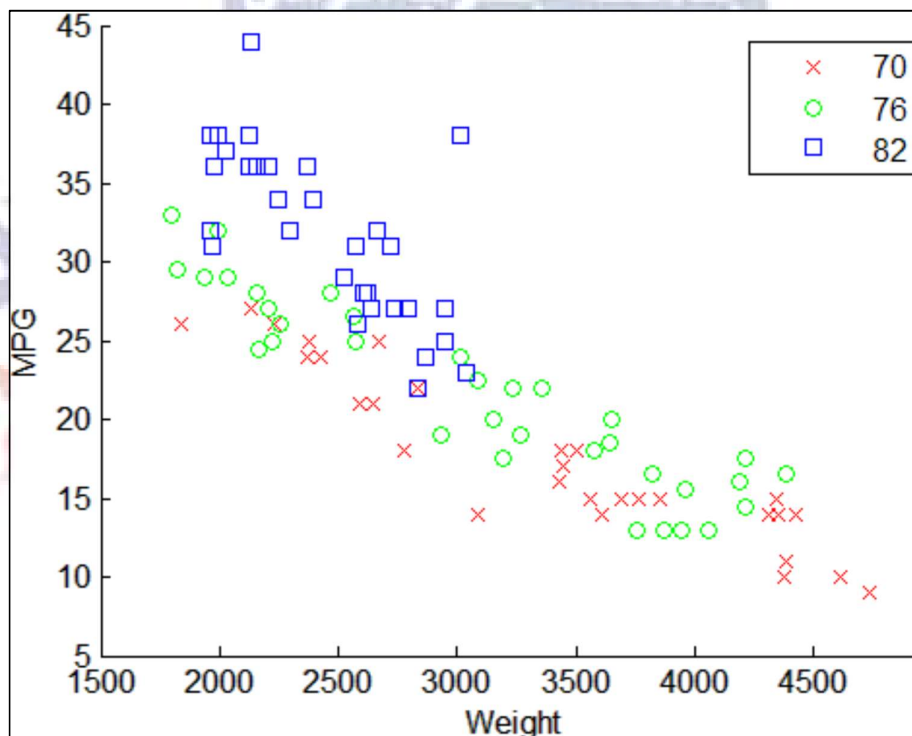
Scatter Plots

A scatter plot is a simple plot of one variable against another. The MATLAB functions `plot` and `scatter` produce scatter plots. The MATLAB function `plotmatrix` can produce a matrix of such plots showing the relationship between several pairs of variables.

Statistics Toolbox functions `gscatter` and `gplotmatrix` produce grouped versions of these plots. These are useful for determining whether the values of two variables or the relationship between those variables is the same in each group.

Suppose you want to examine the weight and mileage of cars from three different model years.

```
load carsmall
gscatter(Weight,MPG,Model_Year,'xos')
```



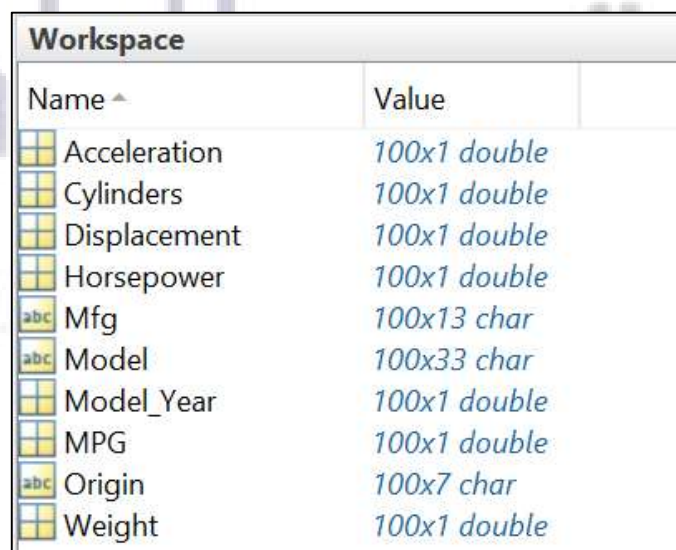
This shows that not only is there a strong relationship between the weight of a car and its mileage, but also that newer cars tend to be lighter and have better gas mileage than older cars.

The default arguments for **gscatter** produce a scatter plot with the different groups shown with the same symbol but different colors. The last two arguments above request that all groups be shown in default colors and with different symbols.

what is carsmall dataset?

Carsmall is a built in dataset in the MATLAB that is available to try the various statistical functions and study their behaviour. After you load it in the MATLAB workspace, have a look at its different variables. (A snapshot is depicted in Figure 38).

This data contains a total of 100 cars, each defined by its Acceleration, Cylinders, Displacement, Horsepower, Mfg, Model, Model_Year, MPG, Origin and Weight.

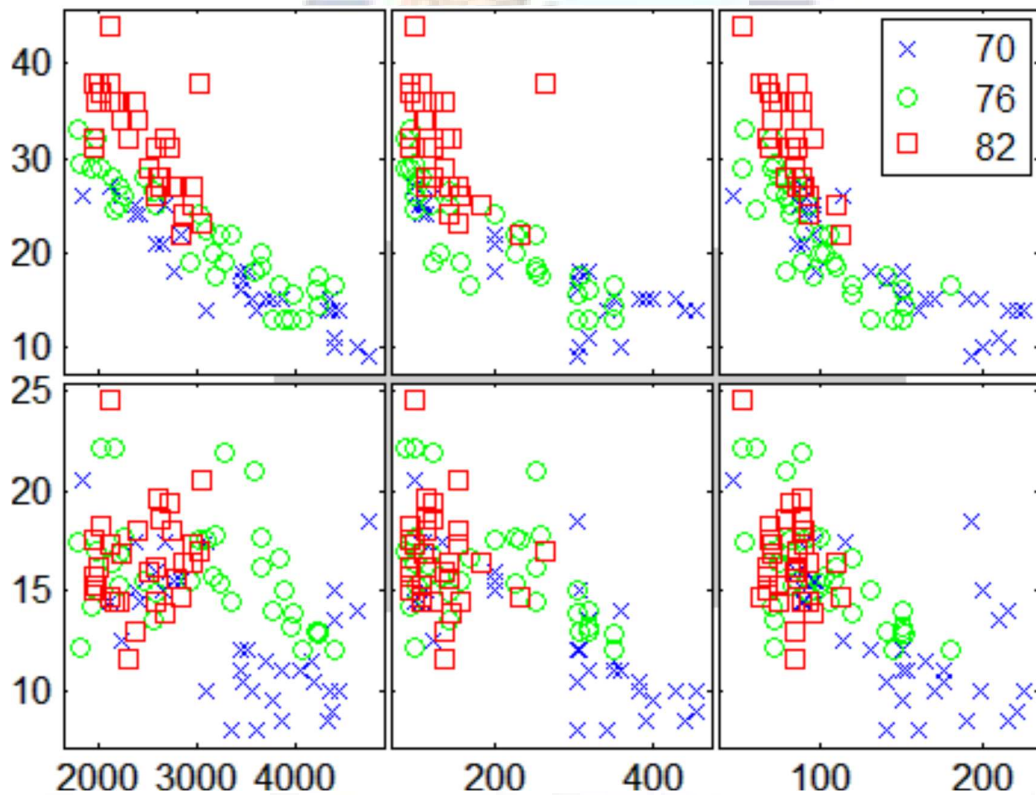


Name ^	Value
Acceleration	100x1 double
Cylinders	100x1 double
Displacement	100x1 double
Horsepower	100x1 double
Mfg	100x13 char
Model	100x33 char
Model_Year	100x1 double
MPG	100x1 double
Origin	100x7 char
Weight	100x1 double

Figure 29 variables of the MATLAB carsmall dataset

The **carsmall** data set contains other variables that describe different aspects of cars. You can examine several of them in a single display by creating a grouped plot matrix.

```
xvars = [Weight Displacement Horsepower];  
yvars = [MPG Acceleration];  
gplotmatrix(xvars,yvars,Model_Year,"','xos')
```



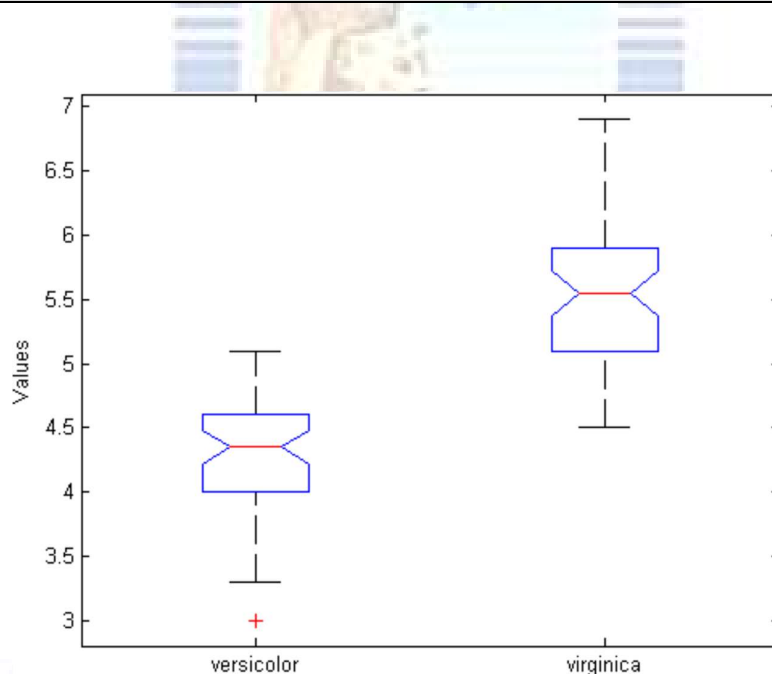
The upper right subplot displays MPG against Horsepower, and shows that over the years the horsepower of the cars has decreased but the gas mileage has improved.

The **gplotmatrix** function can also graph all pairs from a single list of variables, along with histograms for each variable.

Box Plots

The graph below, created with the boxplot command, compares petal lengths in samples from two species of iris.

```
load fisheriris
s1 = meas(51:100,3);
s2 = meas(101:150,3);
boxplot([s1 s2],'notch','on',...
'labels',{'versicolor','virginica'})
```



This plot has the following features:

- The tops and bottoms of each “box” are the 25th and 75th percentiles of the samples, respectively. The distances between the tops and bottoms are the interquartile ranges.
- The line in the middle of each box is the sample median. If the median is not centred in the box, it shows sample skewness.

- The whiskers are lines extending above and below each box. Whiskers are drawn from the ends of the interquartile ranges to the furthest observations within the whisker length (the adjacent values).
- Observations beyond the whisker length are marked as outliers. By default, an outlier is a value that is more than 1.5 times the interquartile range away from the top or bottom of the box, but this value can be adjusted with additional input arguments. Outliers are displayed with a red + sign.
- Notches display the variability of the median between samples. The width of a notch is computed so that box plots whose notches do not overlap (as above) have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. Comparing box-plot medians is like a visual hypothesis test, analogous to the t-test used for means.

what is fisheriris dataset?

fisheriris is another built in dataset in the MATLAB that is available to try the various statistical functions and study their behaviour. After you load it in the MATLAB workspace, have a look at its different variables.

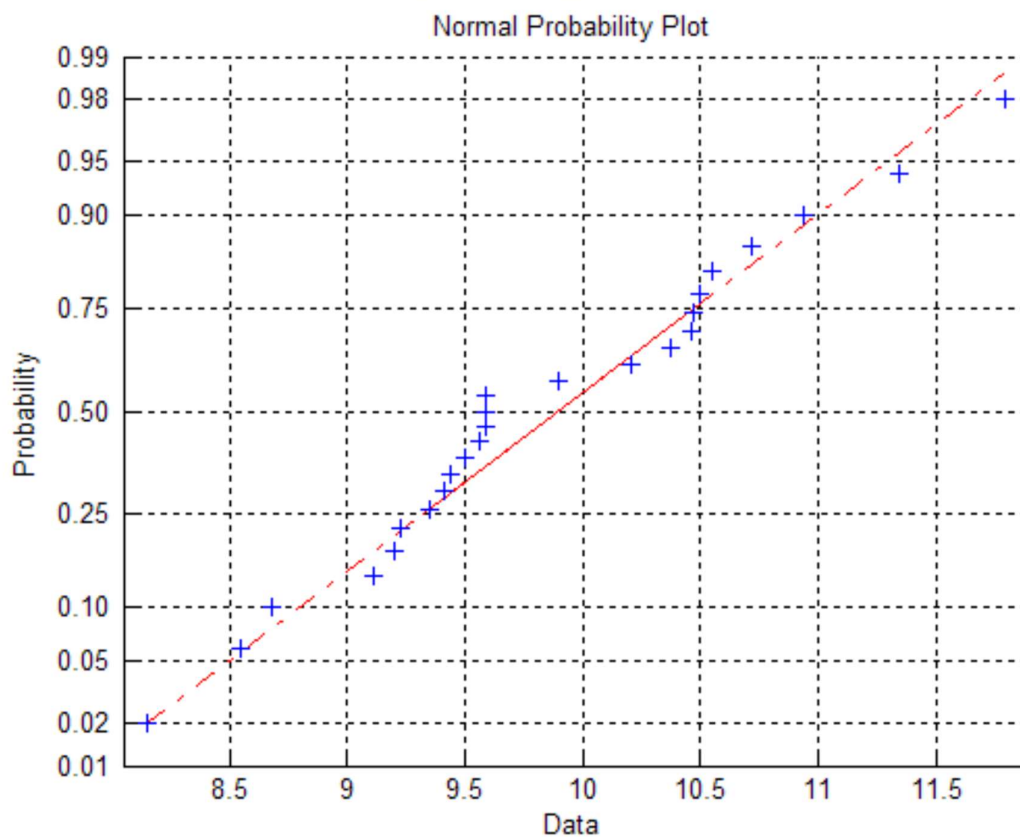
This data contains a total of 150 iris, each defined by its species and meas. For further information about this dataset load it into MATLAB and search the unknown terms in google.

Normal Probability Plots

Normal probability plots are used to assess whether data comes from a normal distribution. Many statistical procedures make the assumption that an underlying distribution is normal, so normal probability plots can provide some assurance that the assumption is justified, or else provide a warning of problems with the assumption.

The following example shows a normal probability plot created with the `normplot` function.

```
x = normrnd(10,1,25,1);  
normplot(x)
```

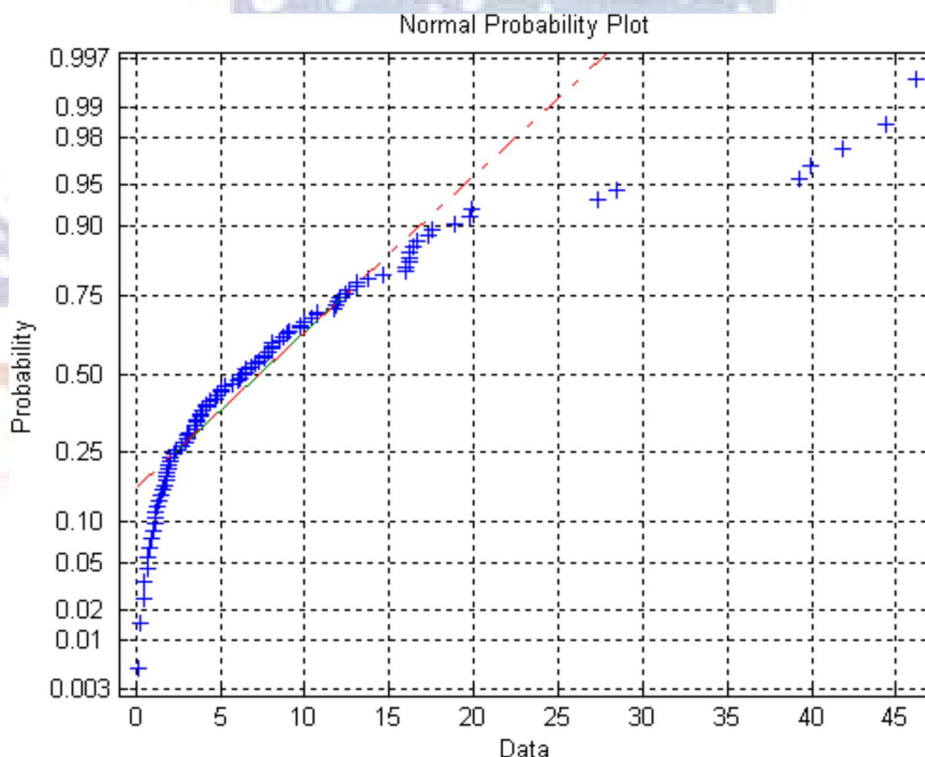


The plus signs plot the empirical probability versus the data value for each point in the data. A solid line connects the 25th and 75th percentiles in the data, and a dashed line extends it to the ends of the data. The y-axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks on the y-axis matches the distance between the quantiles of a normal distribution. The quantiles are close together near the median (probability = 0.5) and stretch out symmetrically as you move away from the median.

In a normal probability plot, if all the data points fall near the line, an assumption of normality is reasonable. Otherwise, the points will curve away from the line, and an assumption of normality is not justified.

For example:

```
x = exprnd(10,100,1);  
normplot(x)
```



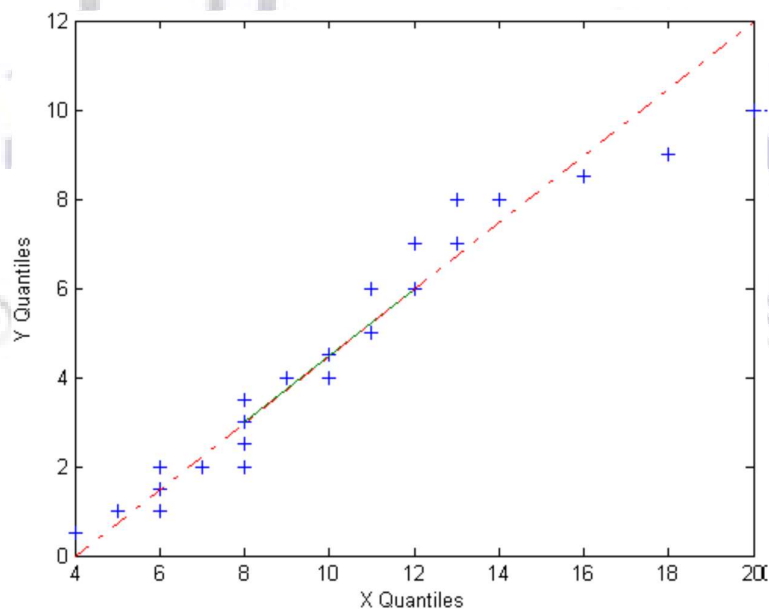
The plot is strong evidence that the underlying distribution is not normal. Do you recall how does the normal distribution looks like?

Quantile-Quantile Plots

Quantile-quantile plots are used to determine whether two samples come from the same distribution family. They are scatter plots of quantiles computed from each sample, with a line drawn between the first and third quartiles. If the data falls near the line, it is reasonable to assume that the two samples come from the same distribution. The method is robust with respect to changes in the location and scale of either distribution.

To create a quantile-quantile plot, use the `qqplot` function. The following example shows a quantile-quantile plot of two samples from Poisson distributions.

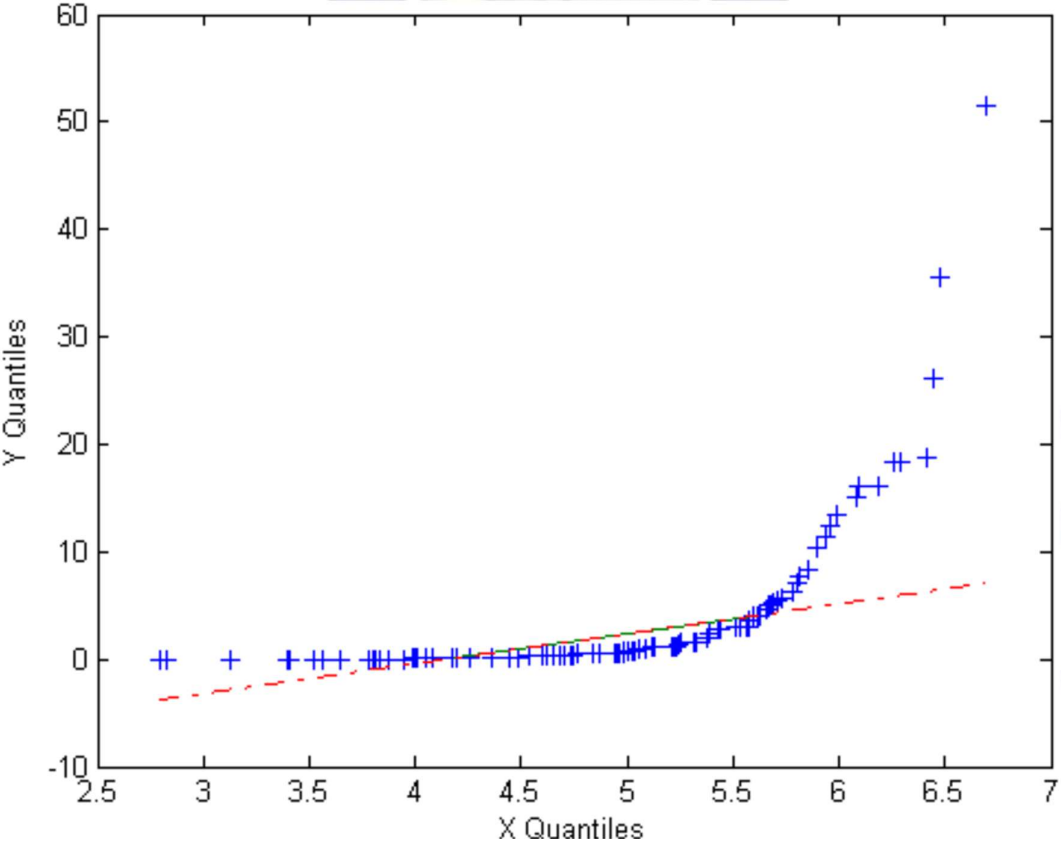
```
x = poissrnd(10,50,1);  
y = poissrnd(5,100,1);  
qqplot(x,y);
```



Even though the parameters and sample sizes are different, the approximate linear relationship suggests that the two samples may come from the same distribution family. For statistical procedures that depend on the two samples coming from the same distribution, however, a linear quantile-quantile plot is often sufficient.

The following example shows what happens when the underlying distributions are not the same.

```
x = normrnd(5,1,100,1);  
y = wblrnd(2,0.5,100,1);  
qqplot(x,y);
```



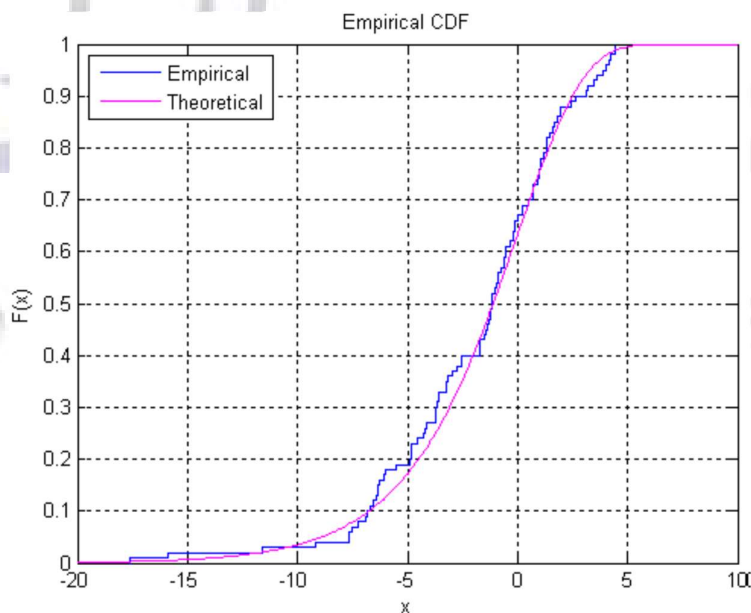
These samples clearly are not from the same distribution family.

Cumulative Distribution Plots

An empirical cumulative distribution function (cdf) plot shows the proportion of data less than each x value, as a function of x . The scale on the y -axis is linear; in particular, it is not scaled to any particular distribution. Empirical cdf plots are used to compare data cdfs to cdfs for particular distributions.

To create an empirical cdf plot, use the `cdfplot` function (or `ecdf` and `stairs`). The following example compares the empirical cdf for a sample from an extreme value distribution with a plot of the cdf for the sampling distribution. In practice, the sampling distribution would be unknown, and would be chosen to match the empirical cdf.

```
y = evrnd(0,3,100,1);
cdfplot(y)
hold on
x = -20:0.1:10;
f = evcdf(x,0,3);
plot(x,f,'m')
legend('Empirical','Theoretical','Location','NW')
```



Part 4



Chapter 1



An introduction to SPSS

What is SPSS?

SPSS is a Windows based program that can be used to perform data entry and analysis and to create tables and graphs. SPSS is capable of handling large amounts of data and can perform all of the analyses covered in the text and much more. SPSS is commonly used in the Social Sciences and in the business world, so familiarity with this program should serve you well in the future. SPSS is updated often. This document was written around an earlier version, but the differences should not cause any problems. If you want to go further and learn much more about SPSS, I strongly recommend Andy Field's book (Field, 2009, *Discovering statistics using SPSS*). Those of us who have used software for years think that we know it all and do not pay a lot of attention to new features. I learned a huge amount from Andy's book.

Opening SPSS

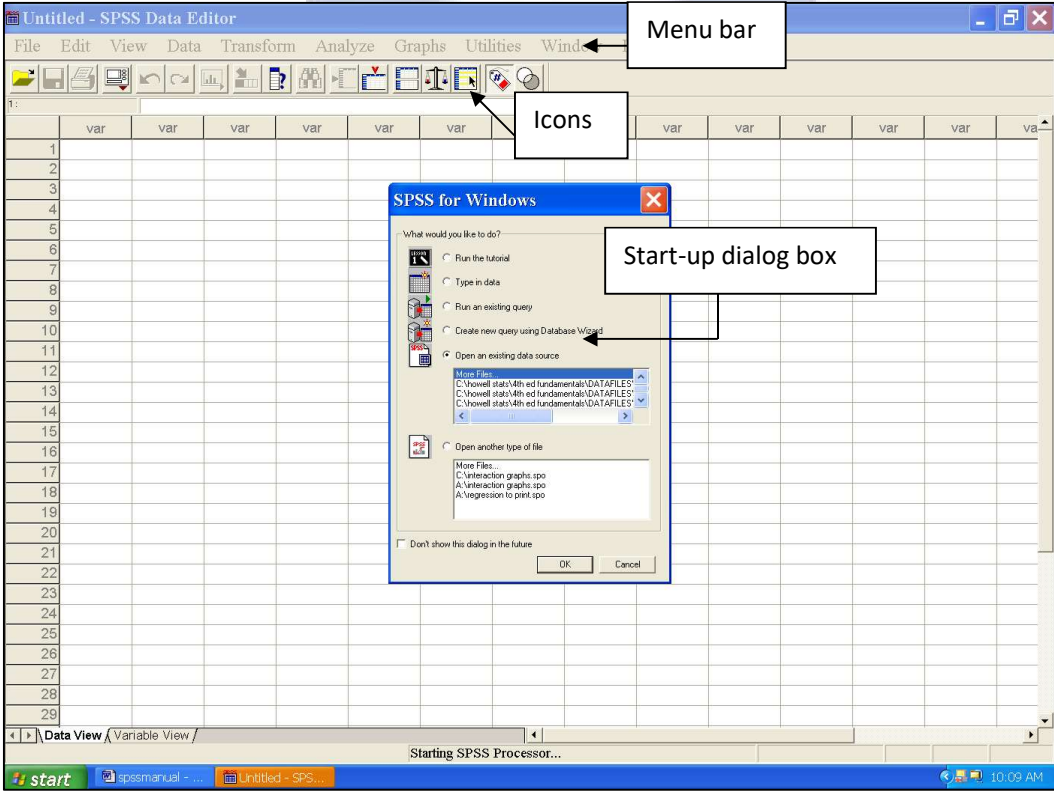
Depending on how the computer you are working on is structured, you can open SPSS in one of two ways.

1. If there is an SPSS shortcut like this  on the desktop, simply put the cursor on it and double click the left mouse button.
2. Click the left mouse button on the  button on your screen, then put your cursor on Programs or All Programs and left click the mouse. Select SPSS 17.0 for Windows by clicking the left mouse button. The version number may change by the time you read this.) Either approach will launch the program.

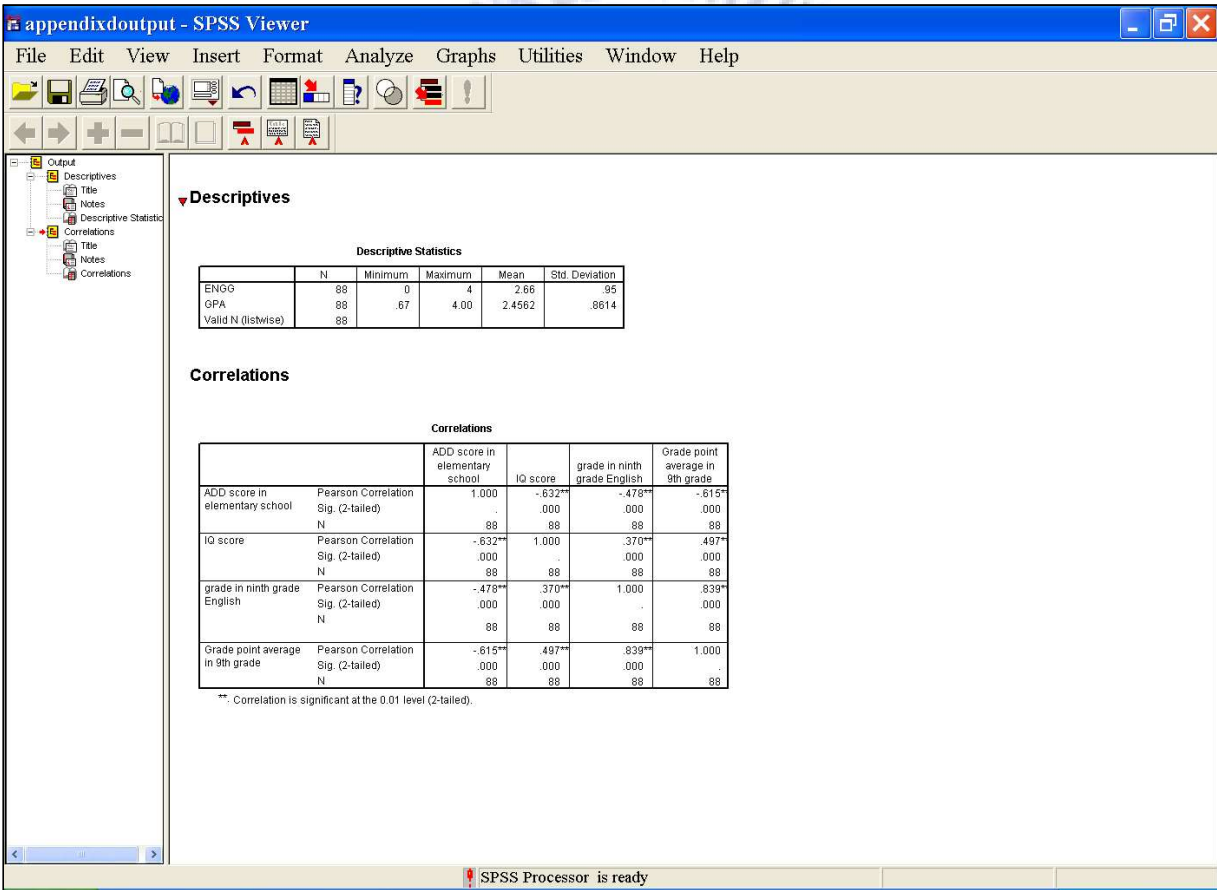
You will see a screen that looks like the image on the next page. The dialog box that appears offers choices of running the tutorial, typing in data, running queries, or opening an existing data source. The window behind this is the Data Editor window which is used to display the data from whatever file you are using. You could select any one of the options on the start-up dialog box and click OK, or you could simply hit Cancel. If you hit Cancel, you can either enter new data in the blank Data Editor or you could open an existing file using the File menu bar as explained later.

Layout of SPSS

The Data Editor window has two views that can be selected from the lower left hand side of the screen. Data View is where you see the data you are using. Variable View is where you can specify the format of your data when you are creating a file or where you can check the format of a pre-existing file. The data in the Data Editor is saved in a file with the extension **.sav**.



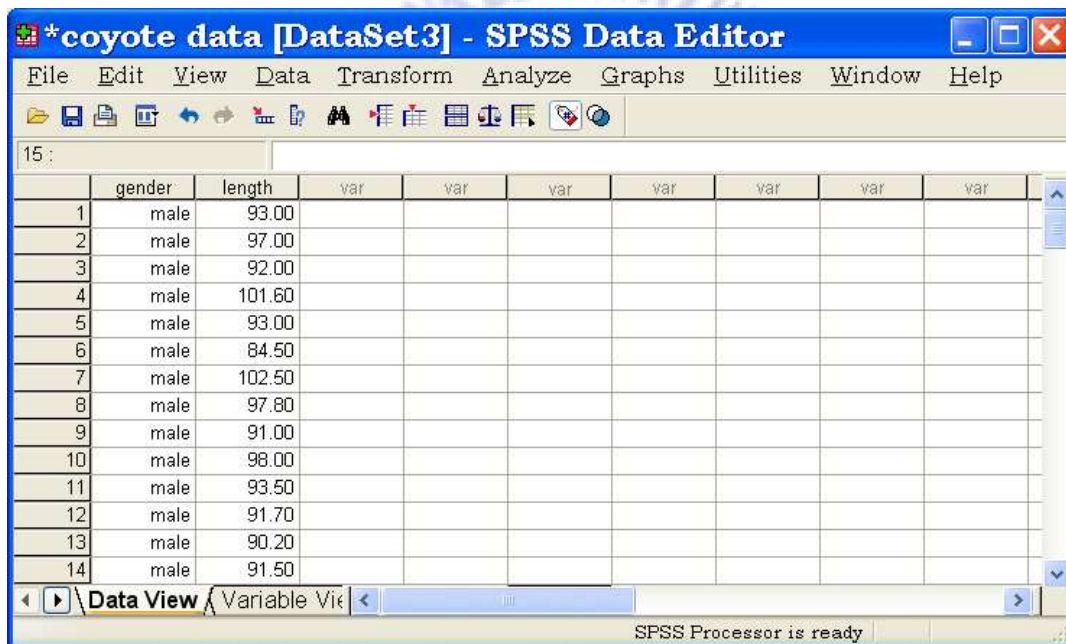
The other most commonly used SPSS window is the SPSS Viewer window which displays the output from any analyses that have been run and any error messages. Information from the Output Viewer is saved in a file with the extension **.spo**. Let us open an output file and look at it.



Finally, there is the Syntax window which displays the command language used to run various operations. Typically, you will simply use the dialog boxes to set up commands, and would not see the Syntax window. The Syntax window would be activated if you pasted the commands from the dialog box to it, or if you wrote your own syntax--something we will not focus on here. Syntax files end in the extension **.sps**.

Data View in SPSS

Unlike in Excel, SPSS files have 2 “sides”: the **Data view** which looks very much like an Excel file and a **Variable view** which is a kind of “behind the scenes” thing.

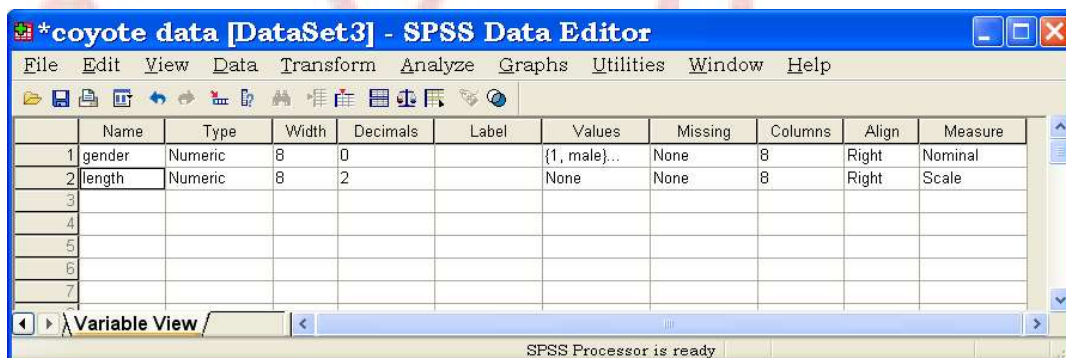


The screenshot shows the SPSS Data Editor window titled '*coyote data [DataSet3] - SPSS Data Editor'. The window is in 'Data View' mode. The main area displays a table with 14 rows of data. The first two columns are 'gender' and 'length'. The 'gender' column contains the value 'male' for all rows. The 'length' column contains numerical values ranging from 90.20 to 102.50. There are several empty columns labeled 'var'.

	gender	length	var	var	var	var	var	var	var
1	male	93.00							
2	male	97.00							
3	male	92.00							
4	male	101.60							
5	male	93.00							
6	male	84.50							
7	male	102.50							
8	male	97.80							
9	male	91.00							
10	male	98.00							
11	male	93.50							
12	male	91.70							
13	male	90.20							
14	male	91.50							

In Data View, columns represent variables (e.g. gender, length), and rows represent cases (observations such as the gender and the length of coyote (an American wolf species)).

Variable view



The screenshot shows the SPSS Data Editor window in 'Variable View' mode. The main area displays a table with 7 rows of variable definitions. The first two rows are for 'gender' and 'length'. The 'gender' variable is defined as a Nominal variable with a width of 8 and 0 decimals. The 'length' variable is defined as a Scale variable with a width of 8 and 2 decimals. There are several empty rows for other variables.

	Name	Type	Width	Decimals	Label	Values	Missing	Columns	Align	Measure
1	gender	Numeric	8	0		{1, male}...	None	8	Right	Nominal
2	length	Numeric	8	2		None	None	8	Right	Scale
3										
4										
5										
6										
7										

Variable view is where you define the variables you will be using: to define/modify a property of a given variable, you click on the cell containing the property you want to define/modify.

You can modify:

- the **name** and the type of your variable,
- the **width**, which corresponds to the number of characters you can have in a cell,
- the **decimals**, which corresponds to the number of decimals recorded,
Tip: when importing data from Excel, SPSS would sometimes give extravagant number of decimals. Do not forget to check that before you start drawing graphs or analysing your data, otherwise you will be unable to read some of analysis outputs and you will get ugly graphs.

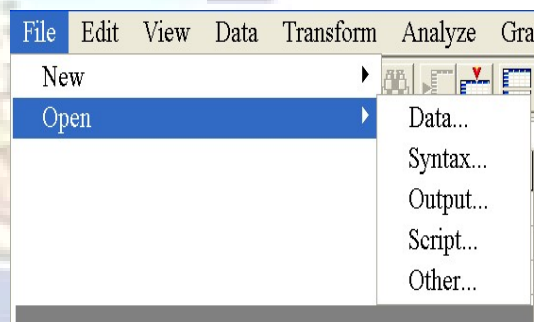
- The label is used when you want to define a variable more accurately or to describe it. In the example above, the label “length” could be “length of the body”.
- The values: useful for categorical data (e.g. gender: male=1 and female=2). This is quite an important characteristic:
 - some analyses will not accept a string variable as a factor,
 - when you draw a graph from your data, if you have not defined any values, you will only see numerical values on the x-axis. For example, you measure the level of a substance in 5 types of cell and you plot it. If you have not specified any values you’ll get a x-axis with numbers from 1 to 5 instead of having the names of the types of cell.
 - you will need to remember that you decided that male=1 and female=2!

- missing: useful for epidemiological questionnaires,
- column (see width),
- align: like Excel: right, left or centre,
- measure: scale (e.g. weight: quantitative variable), ordinal (e.g. no, a little, a lot) or nominal (e.g. male or female: qualitative variable).

SPSS Menus and Icons

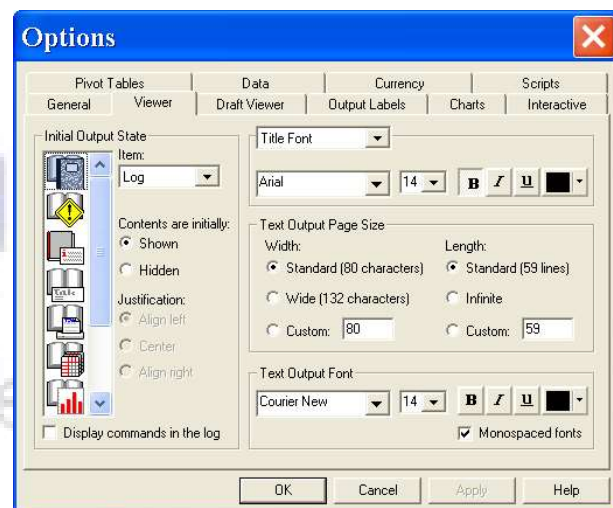
Now, let us review the menus and icons.

File includes all of the options you typically use in other programs, such as open, save, exit. Notice, that you can open or create new files of multiple types as illustrated to the right.



Edit includes the typical cut, copy, and paste commands, and allows you to specify various options for displaying data and output.

Click on **Options**, and you will see the dialog box to the left. You can use this to format the data, output, charts, etc. These choices are rather overwhelming, and you can simply take the default options for now. The author of your text (me) was too dumb to even know these options could easily be set.



View allows you to select which toolbars you want to show, select font size, add or remove the gridlines that separate each piece of data, and to select whether or not to display your raw data or the data labels.

Data allows you to select several options ranging from displaying data that is sorted by a specific variable to selecting certain cases for subsequent analyses.

Transform includes several options to change current variables. For example, you can change continuous variables to categorical variables, change scores into rank scores, add a constant to variables, etc.

Analyze includes all of the commands to carry out statistical analyses and to calculate descriptive statistics. Much of this book will focus on using commands located in this menu.

Graphs includes the commands to create various types of graphs including box plots, histograms, line graphs, and bar charts.


Utilities allows you to list file information which is a list of all variables, their labels, values, locations in the data file, and type.

Add-ons are programs that can be added to the base SPSS package. You probably do not have access to any of those.


Window can be used to select which window you want to view (i.e., Data Editor, Output Viewer, or Syntax). Since we have a data file and an output file open, let us try this.

- ✓ Select **Window/Data Editor**. Then select **Window/SPSS Viewer**.

Help has many useful options including a link to the SPSS homepage, a statistics coach, and a syntax guide. Using **topics**, you can use the index option to type in any key word and get a list of options, or you can view the categories and subcategories available under contents. This is an excellent tool and can be used to troubleshoot most problems. The Icons directly under the Menu bar provide shortcuts to many common commands that are available in specific menus. Take a moment to review these as well.

Place your cursor over the Icons for a few seconds, and a description of the underlying command will appear. For example, this icon  is the shortcut for Save. Review the others yourself.

Exiting SPSS

To close SPSS, you can either left click on the close button  located on the upper right hand corner of the screen or select **Exit** from the **File** menu.

A dialog box like the one below will appear for every open window asking you if you want to save it before exiting. You almost always want to save data files. Output files may be large, so you should ask yourself if you need to save them or if you simply want to print them.



Click **No** for each dialog box since we do not have any new files or changed files to save.

Entering and Editing Data Using the Data Editor

The Data Editor provides a convenient spreadsheet-like facility for entering, editing, and displaying the contents of your data file. A Data Editor window opens automatically when you start an SPSS session. Instruction on Using the Data Editor to enter data is given in the SPSS Help Tutorials. Note that if you are already familiar with entering data into a different spreadsheet program (e.g., MS Excel), you might find it easy to enter your data in the program you are familiar with and then read the data into SPSS.

Entering Data.

Basic data entry in the Data Editor is simple:

Step 1. Create a new (empty) Data Editor window. At the start of an SPSS session a new (empty) Data Editor window opens automatically. During an SPSS session you can create a new Data Editor window by

1. Choose **File**
2. Choose **New**
3. Choose **Data**

Step 2. Move the cursor to the first empty column.

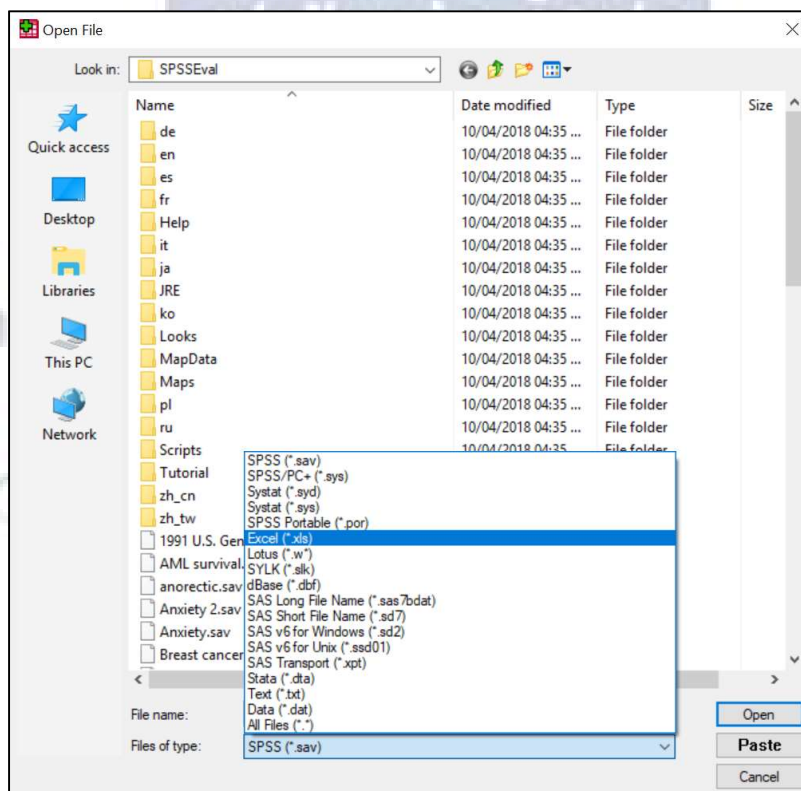
Step 3. Type a value into the cell. As you type, the value appears in the cell editor at the top of the Data Editor window. Each time you press the Enter key, the value is entered in the cell and you move down to the next row. By entering data in a column, you automatically create a variable and SPSS gives it the default variable name var00001.

Step 4. Choose the first cell in the next column. You can use the mouse to click on the cell or use the arrow keys on the keyboard to move to the cell. By default, SPSS names the data in the second column var00002.

Step 5. Repeat step 4 until you have entered all the data. If you entered an incorrect value(s) you will need to edit your data. See the following section on Editing Data.

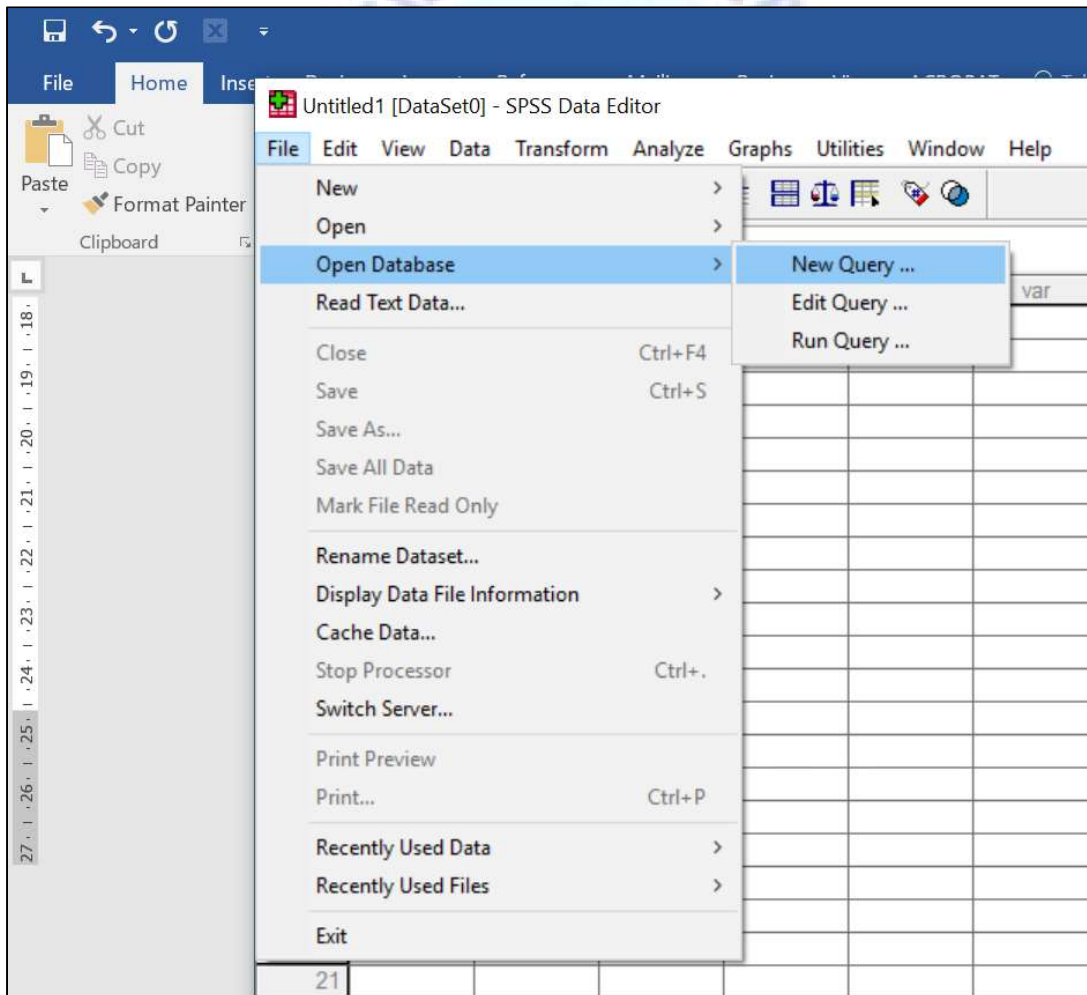
Entering Data from Excel Files

To read Excel files, select *Open* from the *File* menu and *Data* from the sub-menu. Select the file type from the drop-down list. Select a file from the list. If the first row of the Excel file contains column headings or labels, click *Read variable names* from first row of data. For Excel 5 or later, you can also specify the sheet in the Excel file that you want to read. Column headings from the Excel file are used as variable names. Since variable names cannot exceed 8 characters, column headings are truncated at 8 characters. The original column heading is preserved as a variable label. If the column heading cell is blank, a default variable name is assigned. For Excel 5 or later files, if a column contains mixed data types (for example numeric and string), the data values in the column are read as string data.



Entering Data from Database Files with the Database Wizard

To read any database data source, in the Data Editor, select Open Database from the File menu and choose New Query. Click the appropriate data source in the Database Wizard, and then click Next. If the data source you need isn't displayed in the list, click *Add Data Source*. Then use the *ODBC Data Source Administrator* to add the data source.

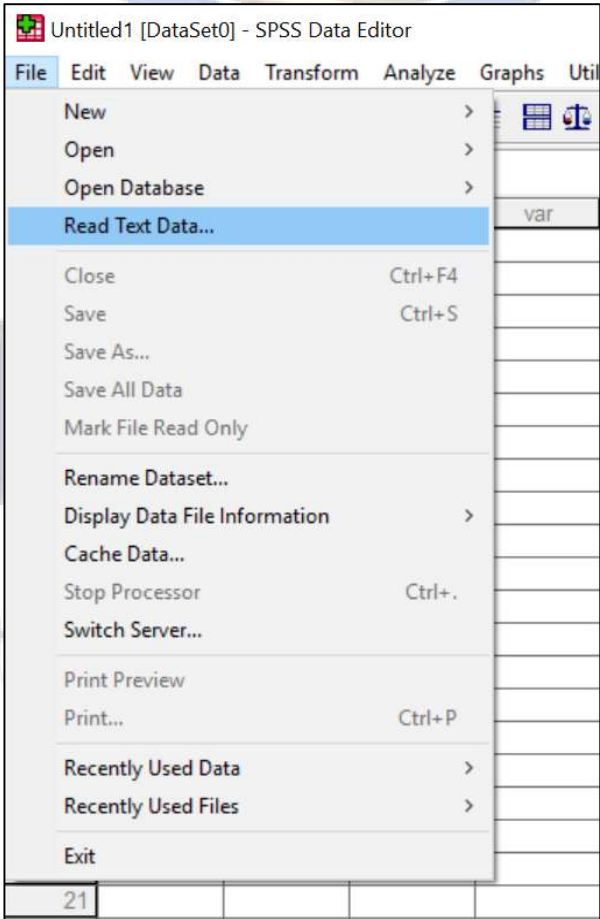


In distributed analysis mode (available with the server version), the Add Data Source button is not available because only the system administrator can add data sources. For some data sources you may also have to select a file. You may also need to supply a username and password. Each table is shown in the *Available Tables* list. Click the plus sign (+) to show all the fields in a table. Drag and drop the table(s) you want to import into the *Retrieve*

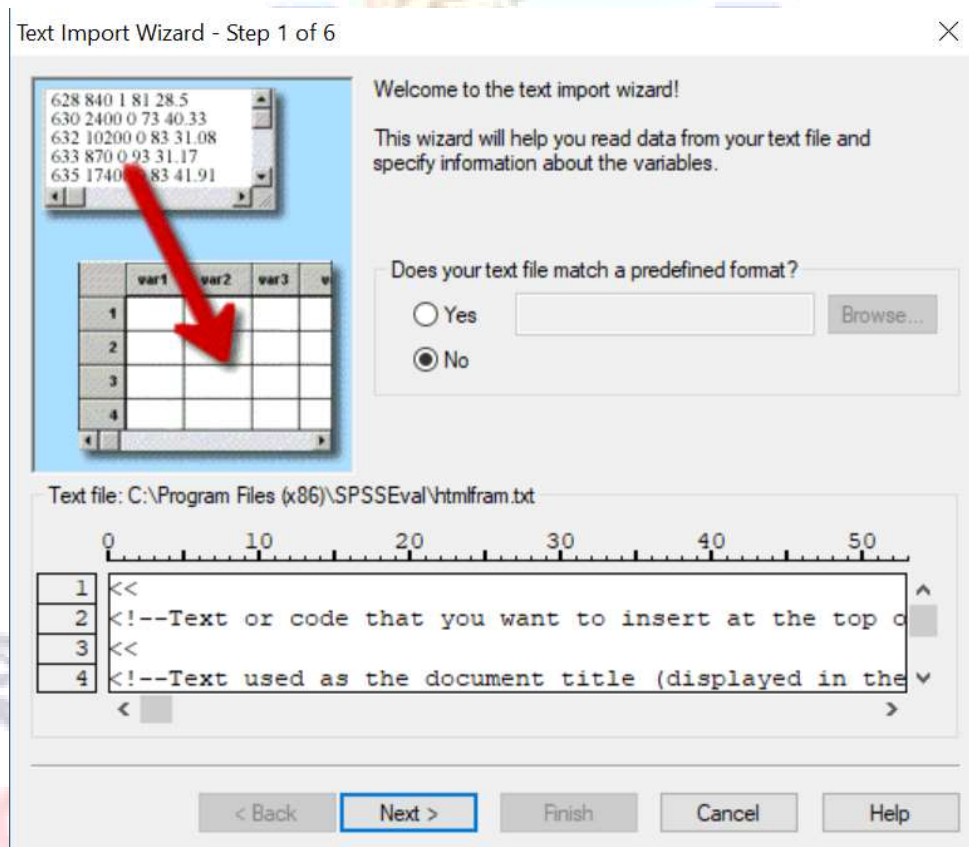
Fields list. Fields become variables in the Data Editor. You can also select a subset of fields. Specify join types if you are importing fields from more than one table. Select a subset of cases based on conditional expressions. Specify user-friendly variable names. Save the query to file for use in other sessions.

Reading Text Data Files

If your raw data are in a simple text file (standard ASCII format), select *Read Text Data* from the File menu. Select a text data file to read. This opens the *Text Wizard*. The data file is displayed in the preview window. In the first step you can apply a predefined format (previously saved in the Text Wizard). In this example, we'll simply click *Next* since we want to define a new format.



Step 2 provides information about variables. A variable is similar to a field in a database. For example, each item in a questionnaire is a variable. Fixed format means each variable is recorded in the same column location for every case. Delimited means that spaces, commas, tabs, or other characters are used to separate variables. The variables are recorded in the same order for each case but not necessarily in the same column locations. In this example, the data file is delimited. It also contains descriptive labels in the first row, which you can use as variable names.



Step 3 provides information about cases. A case is similar to a record in a database. For example, each respondent to a questionnaire is a case. In the previous step we indicated that the top line of the file contains variable names -- so in this step we indicate that the data values start on the second line. The Text Wizard also needs to know how many variables to read for each case. In this example, all the data values for each case are recorded on

a single line. If not all lines contain the same number of data values, the number of variables for each case is determined by the line with the greatest number of data values.

Step 4 displays the Text Wizard's best guess on how to read the data file and allows you to modify the way the Text Wizard will read variables from the data file. In this example the Text Wizard correctly detected that commas are used to delimit data values. The Text Wizard also assumes that consecutive delimiters indicate missing data. In this comma-delimited file, that means that two commas without an intervening data value indicate a variable with a missing value.

Step 5 controls the data format that the Text Wizard will use to read each variable and which variables will be included in the final data file. To change the format of a variable, click the variable in the preview window. Then select a format from the drop-down list. To omit a variable from the imported data file, click the variable in the preview window, and select *Do Not Import* from the drop-down list.

Step 6 is the final step of the Text Wizard. You can save your specifications to read similar text data files. You can also paste and save the underlying command syntax. When you're ready to read the text data file, just click *Finish*.

Editing Data at the Data Editor Screen.

With the Data Editor, you can modify a data file in many ways. For example, you can change values or cut, copy, and paste values, or add and delete cases.

To Change a Data Value:

1. Click on a data cell. The cell value is displayed in the cell editor.
2. Type the new value. It replaces the old value in the cell editor.

3. Press then Enter key. The new value appears in the data cell.

To Cut, Copy, and Paste Data Values

1. Select (highlight) the cell value(s) you want to cut or copy.
2. Pull down the Edit box on the main menu bar.
3. Choose Cut. The selected cell values will be copied, then deleted. Or
4. Choose Copy. The selected cell values will be copied, but not deleted.
5. Select the target cell(s) (where you want to put the cut or copy values).
6. Pull down the Edit box on the main menu bar.
7. Choose Paste. The cut or copy values will be "pasted" in the target cells.

To Delete a Case (i.e., a Row of Data)

1. Click on the case number on the left side of the row. The whole row will be highlighted.
2. Pull down the Edit box on the main menu bar.
3. Choose Clear.

To Add a Case (i.e., a Row of Data)

1. Select any cell in the case from the row below where you want to insert the new case.
2. Pull down the Data box on the main menu bar.
3. Choose Insert.

Defining Variables.

The default name for new variables is the prefix *var* and a sequential five-digit number (e.g., var00001, var00002, var00003). To change the name, format and other attributes of a variable.

1. Double click on the variable name at the top of a column or,
2. Click on the Variable View tab at the bottom of Data Editor Window.

3. Edit the variable name under column labeled Name. The variable name must be eight characters or less in length. You can also specify the number of decimal places (under Decimals), assign a descriptive name (under Label), define missing values (under Missing), define the type of variable (under Measure; e.g., scale, ordinal, nominal), and define the values for nominal variables (under Values).

After the data is entered (or several times during data entering), you will want to save it as an SPSS save file. See the section on Saving Data As An SPSS Save File.

Saving Data as an SPSS Data (.sav) File

To save data as a *new* SPSS Data file onto your computer:

1. Display the Data Editor window (i.e., execute the following commands while in the Data Editor window displaying the data you want to save.)
2. Choose File on the menu bar.
3. Choose Save As...
4. Edit the directory or disk drive to indicate where the data should be saved. SPSS will automatically add the **.sav** suffix to the filename.
5. Choose Save

To save data changes in an *existing* SPSS Save: file.

1. Display the Data Editor window (i.e., execute the following commands while in the Data Editor window displaying the data you want to save.)
2. Choose File box on the menu bar
3. Choose Save

Caution. The Save command saves the modified data by overwriting the previous version of the file.

You can save your data in other formats besides an SPSS save file (e.g., as an ASCII file, Excel file, SAS data set). To save your data with a given format you follow the same steps as saving data in a new SPSS Save file, except that you specify the Save as Type as the desired format.

Saving Your Output (Statistical Results and Graphs)

To save the statistical results and graphs displayed in the Viewer window as a *new* SPSS Output file:

1. Display the Viewer window (i.e., execute the following commands while in the Viewer window displaying the results you want to save.)
2. Choose File on the menu bar.
3. Choose Save As...
4. Edit the directory or disk drive to indicate where the output should be saved. SPSS will automatically add the **.spo** suffix to the filename.
5. Choose Save

To save Viewer changes in an *existing* SPSS Output file.

1. Display the Viewer window (i.e., execute the following commands while in the Viewer window displaying the results you want to save.)
2. Choose File on the menu bar.
3. Choose Save.

Caution. The Save command saves the modified Viewer window by overwriting the previous version of the file. Note that you will *not* be able to open SPSS output that was created with a newer version than the version of SPSS that you are using to open the output. Hence, you may want to avoid this problem you by exporting your output in html or MS word format. Also, charts often do not export properly into a Html or Word file. Usually you need to export charts separately into a window metafile file (.wmf).

Sometimes the output, including charts, can be copied and pasted directly into a Word file.

Exporting SPSS Output

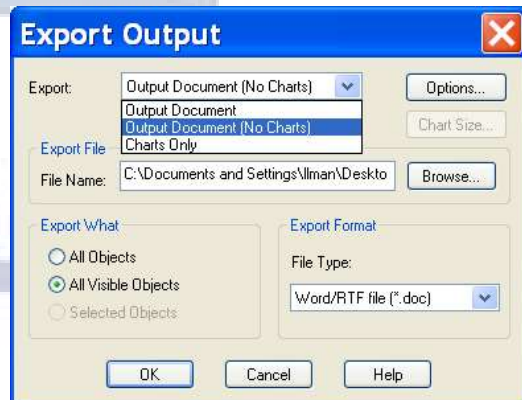
Sometimes you will want to save your SPSS output in a different file format than a SPSS output file, because you want to avoid compatibility problems between different versions of SPSS, you want to further edit your output in a Word document, or you want to include graphs or figures in another document file. The basic steps in exporting SPSS output to another file type are, while in a SPSS (output) Viewer window:

1. Choose File
2. Choose Export

3. Choose what you want to export:

- Output Document – exports all the output
- Output Document (No Charts) – exports only the numerical results
- Charts Only – exports only charts (i.e., graphs & figures)

Note that charts often do not export properly into a Html or Word file. Usually you need to export charts separately into a window metafile file (.wmf).

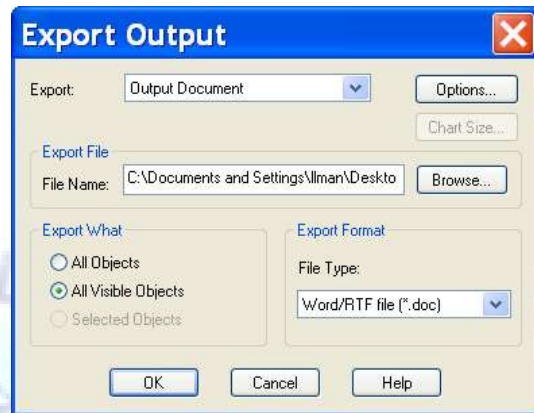


4. Define further what you want to export:

All Objects – this option also exports other extraneous information (rarely useful)

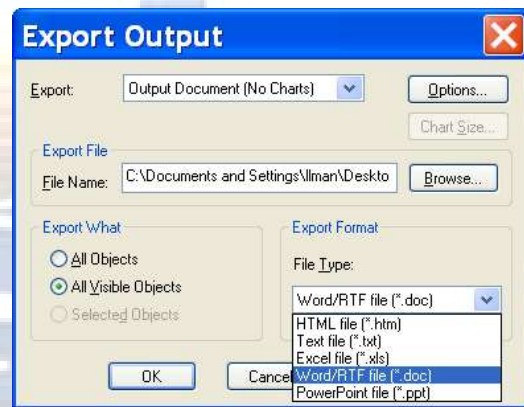
All Visible Objects – use this option to export all the output.

Selected Objects – this allows you to export only the objects you have selected in the Viewer window.



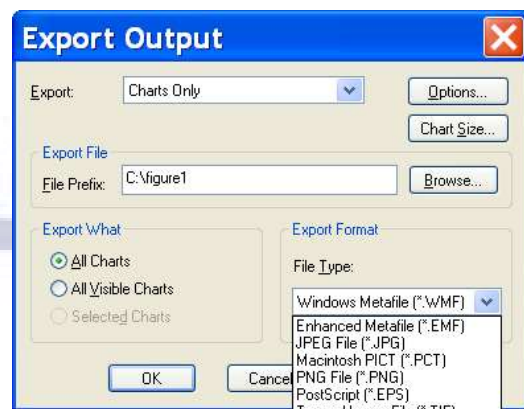
5. Choose the file type

HTML and Word/RTF a good file types for numerical results (no charts).

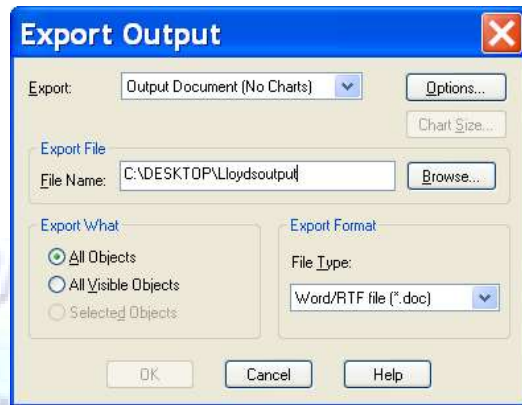


Windows Metafile (.WMF) is a good file type for charts in you want to include figures in a MS Word document.

Note that the file type options are dependent on what you are exporting.



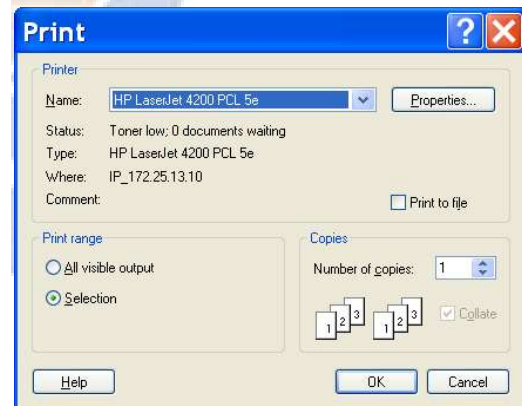
6. Choose the location and file name for the output you want to export.



Printing Your Work in SPSS

To print statistical results and graphs in the Viewer window or data in the Data Editor window:

1. Display the output or data you want to print (i.e., execute the following commands while in a output or data window)
2. Choose File on the menu bar.
3. Choose Print...
4. Choose All visible output or Selection (if you have selected parts of the output). When printing from a data file, the options are All, Selection and Page # to Page #.
5. Choose OK



Exiting SPSS

To exit SPSS:

1. Choose File on the menu bar
2. Choose Exit SPSS

If you have made changes to the data file or the output file since the last time you saved these files, before exiting SPSS you will be asked whether you want to save the contents of the Data Editor window and Viewer window. If you are unsure as to whether you want to save the contents of the data or output window, choose Cancel, then display the window(s) and if you want to save the contents of the window, follow the instructions in this handout for saving data or output windows. SPSS will use the overwrite method when saving the contents of the window.

Creating a New Variable

To create a new variable:

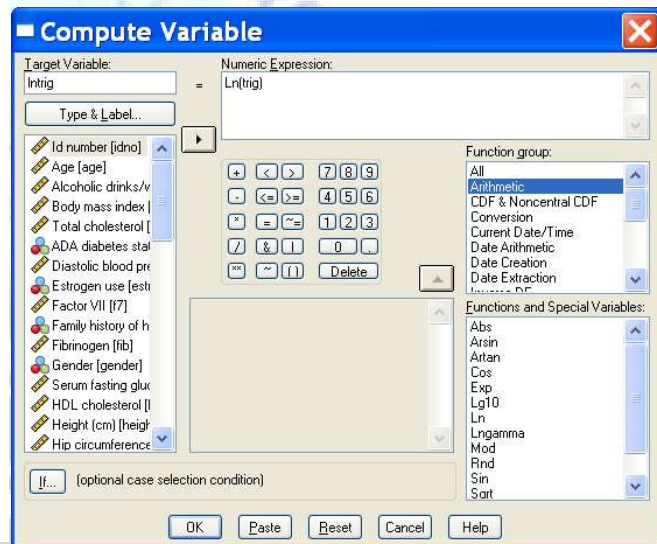
1. Display the **Data Editor window** (i.e., execute the following commands while in the Data Editor window displaying the data file you want to use to create a new variable).
2. Choose **Transform** on the menu bar
3. Choose **Compute...**
4. Enter the new variable name in the Target Variable box.
5. Enter the definition of the new variable in the Numeric Expression box (e.g., $\text{SQRT}(\text{visan})$, $\text{LN}(\text{age})$, or $\text{MEAN}(\text{age})$) or
6. Select variable(s) and combine with desired arithmetic operations and/or functions.
7. Choose **OK**

After creating a new variable(s), you will probably want to save the new variable(s) by re-saving your data using the Save command under File on the menu bar (See Saving Data as an SPSS Save File). Further instructions on creating a new variable are given in the **SPSS Help Tutorials under Modifying Data Values**.

Example: Creating a (New) Transformed Variable

You can use the SPSS commands for creating a new variable to create a transformed variable. Suppose you have a variable indicating triglyceride level, trig, and you want to transform this variable using the natural logarithm to make the distribution less skewed (i.e., you want to create a new variable which is natural logarithm of triglyceride levels).

1. Display the Data Editor window
2. Choose Transform on the menu bar
3. Choose Compute...
4. Enter, say, Intrig, in the Target Variable box.
5. Enter Ln(trig) in the Numeric Expression box.
6. Choose OK



Now, a new variable, **Intrig**, which is the natural logarithm of trig, will be added to your data set. Remember to save your data set before exiting SPSS (e.g., while in the SPSS Data window, choose Save under File or click on the floppy disk icon).

Recoding or Combining Categories of a Variable

To recode or combine categories of a variable:

1. Display the **Data Editor window** (i.e., execute the following commands while in the Data Editor window displaying the data file you want to use to recode variables).
2. Choose **Transform** on the menu bar
3. Choose **Recode**

4. Choose Into Same Variable... or **Into Different Variable...**
5. **Select a variable to recode** from the variable list on the left and then click on the arrow located in the middle of the window. This defines the input variable.
6. If recoding into a different variable, **enter the new variable name** in the box under Name:, then choose Change. This defines the output variable.
7. Choose **Old and New Values...**
8. Choose Value or Range under Old Value and enter old value(s).
9. Choose New Value and enter new value, then choose **Add**.
10. Repeat the process until all old values have been redefined.
11. Choose **Continue**
12. Choose **OK**

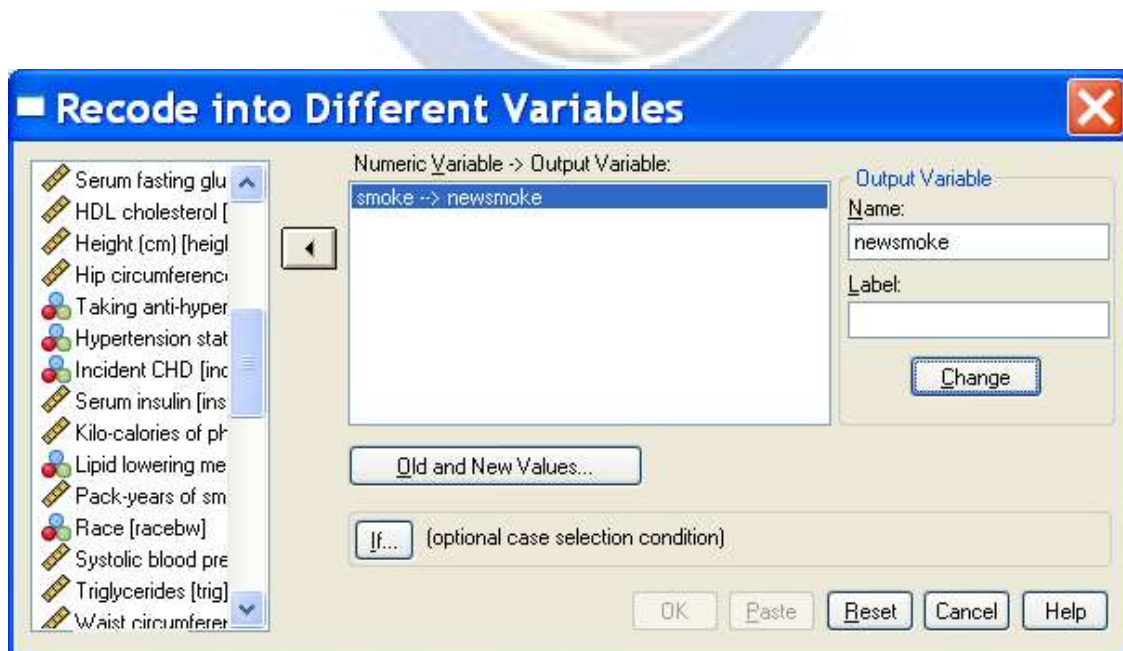
After creating a new variable(s), you will probably want to save the new variable(s) by re-saving your data using the Save command under File box on the menu bar (See Saving Data as an SPSS Save File).

Example: Recoding a Categorical Variable

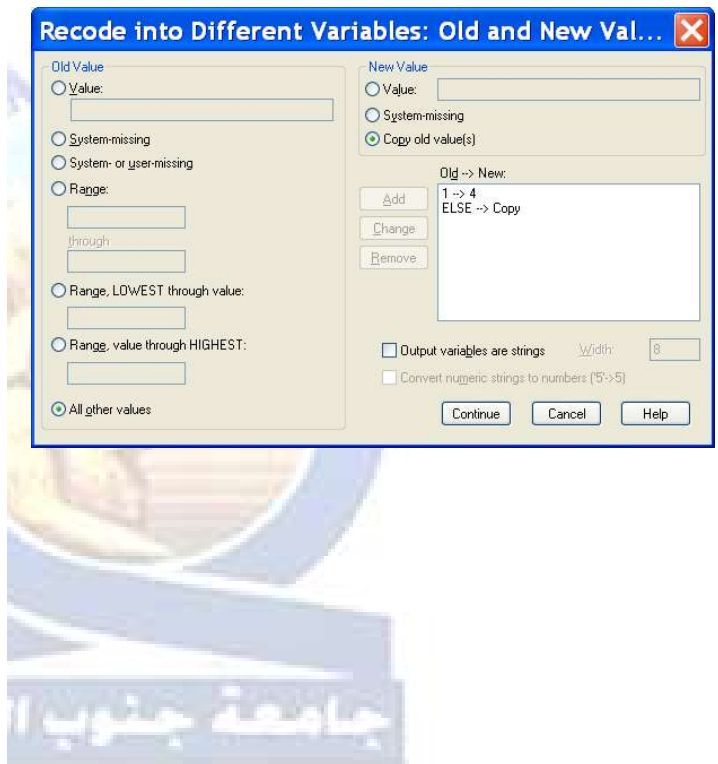
You can use the commands for recoding a variable to change the coding values of a categorical variable. You may want to change a coding value for a particular category to modify which category SPSS uses as the referent category in a statistical procedure. For example, suppose you want to perform linear regression using the ANOVA (or General Linear Model) commands, and one of your independent variables is smoking status, smoke, that is coded 1 for never smoked, 2 for former smoker and 3 for current smoker. By default SPSS will use current smoker as the referent category because current smoker has the largest numerical (code) value. If you want never smoked to be the referent category you need to recode the value for never smoked to a value larger than 3.

Although you can recode the smoking status into the same variable, it is better to recode the variable into a new/different variable, newsmoke, so you do not lose your original data if you make an error while recoding.

1. Display the Data Editor window
2. Choose Transform
3. Choose Recode
4. Choose Into Different Variables...
5. Select the variable smoke as the Input variable
6. Enter newsmoke as the name of the Output variable, and then choose Change.
7. Choose Old and New Values...



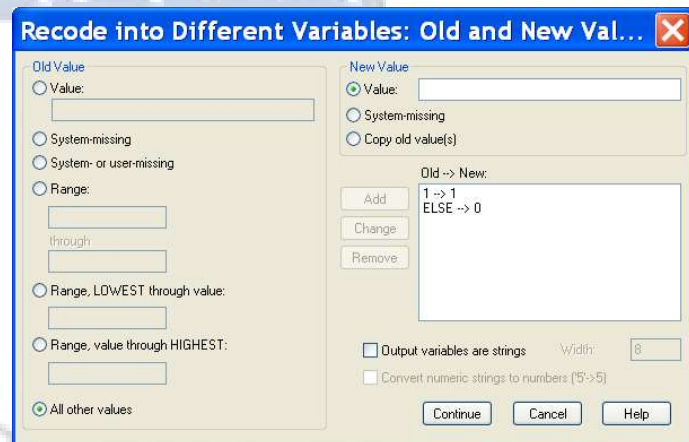
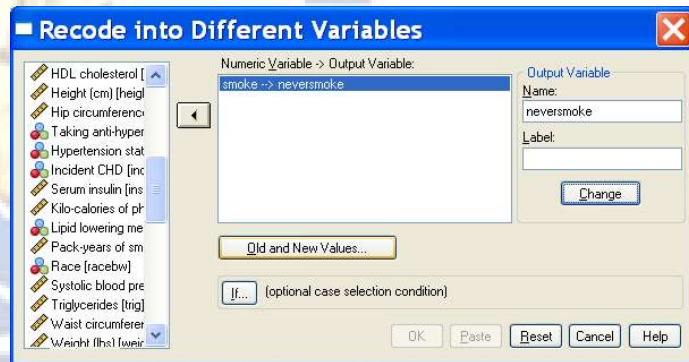
8. Choose Value under Old Value. (It may already be selected.)
9. Enter 1 (code for never smoker)
10. Choose Value under New Value. (It may already be selected.)
11. Enter 4 (or any value greater than 3)
12. Choose Add
13. Choose All Other Values under Old Value.
14. Choose Copy Old Value(s) under New Value.
15. Choose Add
16. Choose Continue
17. Choose OK



Example: Creating Indicator or Dummy Variables

You can use the commands for recoding a variable to create indicator or dummy variables in SPSS. Suppose you have a variable indicating smoking status, smoke, that is coded 1 for never smoked, 2 for former smoker and 3 for current smoker. To create three new indicator or dummy variables for never, former and current smoking:

1. Display the Data Editor window
2. Choose Transform
3. Choose Recode
4. Choose Into Different Variables...
5. Select the variable smoke as the Input variable
6. Enter neversmoke as the name of the Output variable, and then choose Change.
7. Choose Old and New Values...
8. Choose Value under Old Value. (It may already be selected.)
9. Enter 1 (code value for never smoker)
10. Choose Value under New Value. (It may already be selected.)
11. Enter 1 (to indicate never smoker)
12. Choose Add
13. Choose All Other Values under Old Value.
14. Choose Value under New Value.
15. Enter 0
16. Choose Add
17. Choose Continue
18. Choose OK



Now, you have created a binary indicator variable for never smoker (coded 1 if never smoker, 0 if former or current smoker). Next, create a binary indicator variable for former smoker.

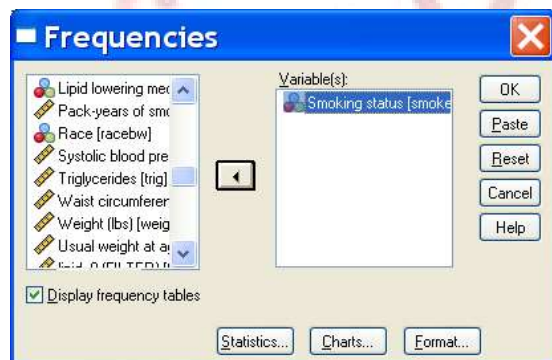
Summarizing Your Data in SPSS

Frequency Tables (& Bar Charts) for Categorical Variables.

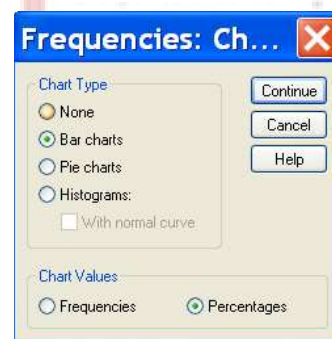
To produce frequency tables and bar charts for categorical variables:

1. Choose **Analyze** from the menu bar
2. Choose **Descriptive Statistics**
3. Choose **Frequencies...**
4. **Variable(s)**: To select the variables you want from the source list on the left, highlight a variable by pointing and clicking the mouse and then click on the arrow located in the middle of the window. Repeat the process until you have selected all the variables you want.
5. Choose **Charts** (Skip to step 7 if you do not want bar charts.)
6. Choose **Bar Chart(s)**
7. Choose **Continue**
8. Choose **OK**

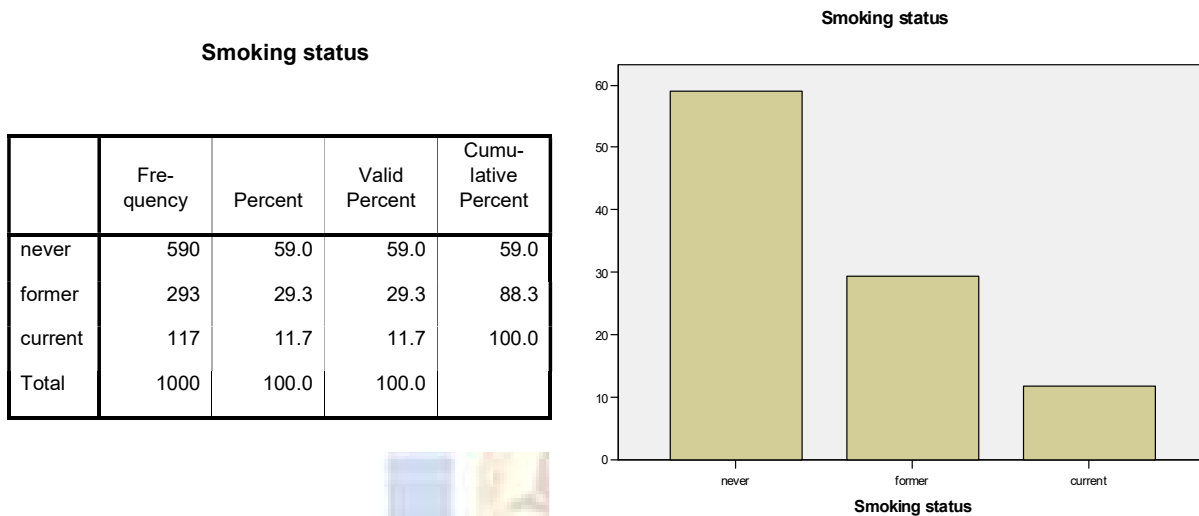
Example: Frequency table and bar chart for the categorical variable, smoking status.



Smoking status is the selected variable(s) and Bar charts under Charts... has been selected.



Frequency table and bar chart of smoking status



Descriptive Statistics (& Histograms) for Numerical Variables.

To produce descriptive statistics and histograms for numerical variables:

1. Choose **Analyze** on the menu bar
2. Choose **Descriptive Statistics**
3. Choose **Frequencies...**
4. **Variable(s)**: To select the variables you want from the source list on the left, highlight a variable by pointing and clicking the mouse and then click on the arrow located in the middle of the window. Repeat the process until you have selected all the variables you want.
5. Choose **Display frequency tables to turn off the option**. Note that the option is turned off when the little box is empty.
6. Choose **Statistics**
7. Choose **summary measures** (e.g., mean, median, standard deviation, minimum, maximum, skewness or kurtosis).
8. Choose **Continue**
9. Choose **Charts** (Skip to step 11 if you do not want histograms.)
10. Choose **Histograms(s)**
11. Choose **Continue**
12. Choose **OK**

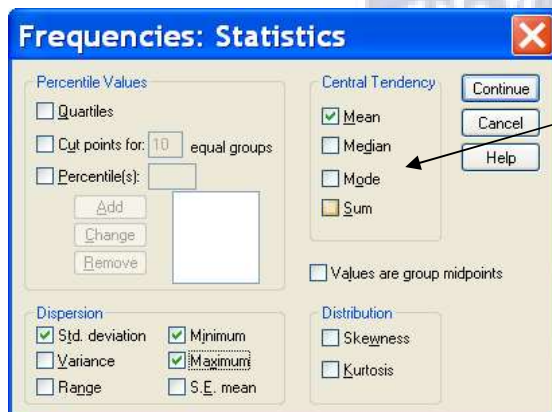
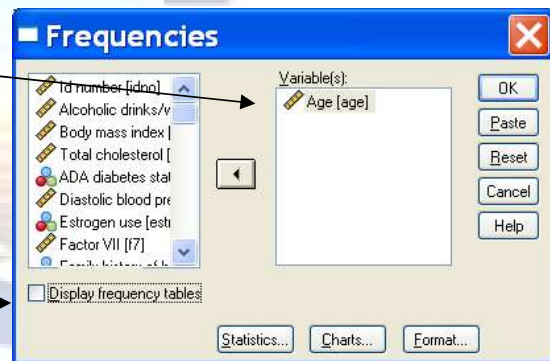
An alternate way to produce only the descriptive statistics is at step 3 to choose **Descriptives...** instead of **Frequencies...**, then, select the variables you want. By default, SPSS computes the mean, standard deviation, minimum and maximum. Choose Options... to select other summary measures.

Example:

Descriptive summaries and histogram for the numerical variable age.

Age is the variable to summarize.
You can select more than one variable to analyze.

Remember to turn off the Display frequency tables option.



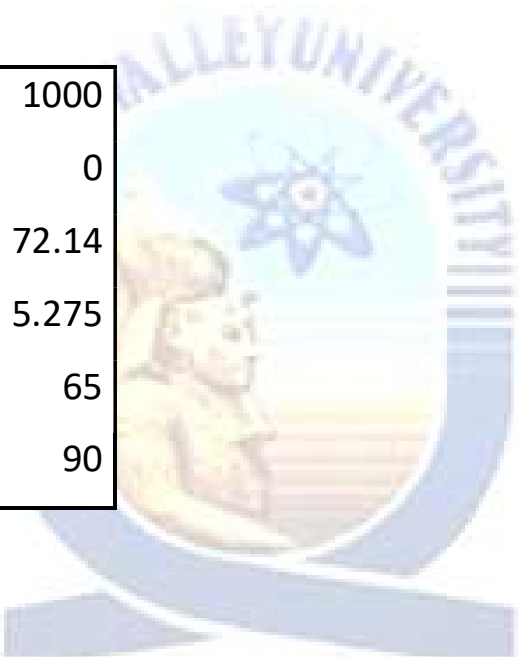
Mean, standard deviation, minimum and maximum were selected under Statistics..., and histogram was selected under Charts...



Summaries for Age

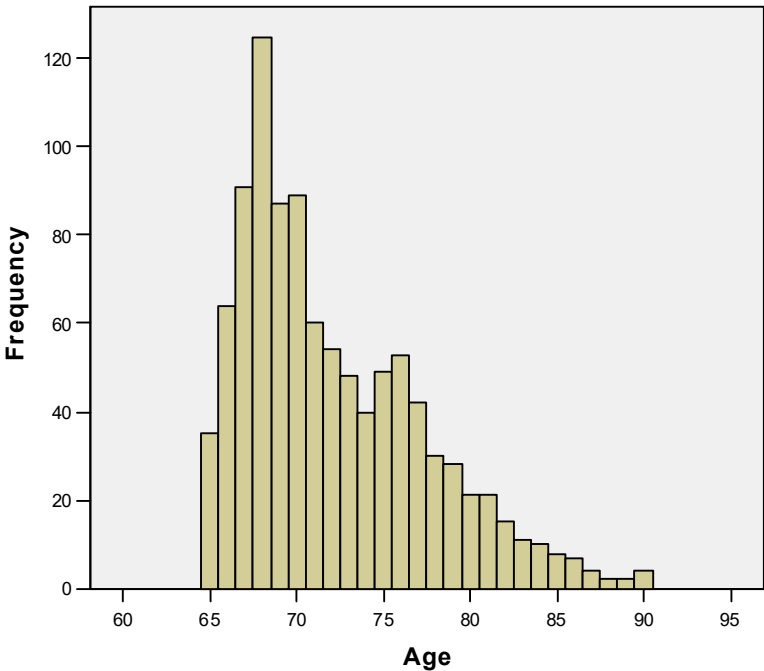
Statistics

Age		
N	Valid	1000
	Missing	0
Mean		72.14
Std. Deviation		5.275
Minimum		65
Maximum		90



Histogram of Age

Histogram



Mean = 72.14
 Std. Dev. = 5.275
 N = 1,000

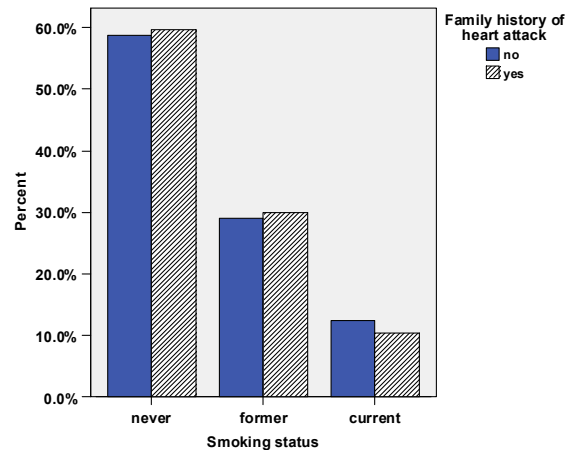
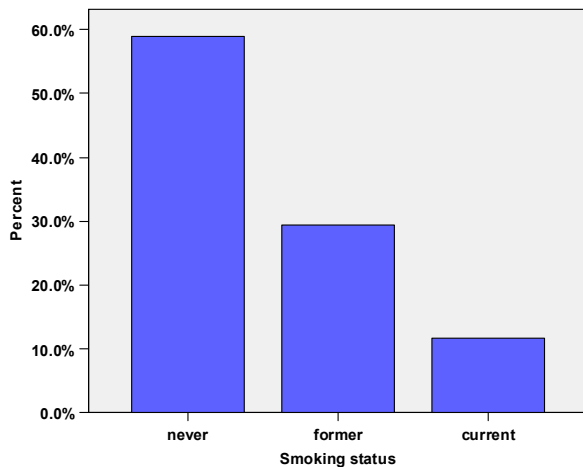
Graphing Your Data

You can produce very fancy figures and graphs in SPSS. Producing fancy figures and graphs is beyond the scope of this handout. Instructions on producing figures and graphs can be found in SPSS Help under Topics → Contents → Chart Galleries, Standard Charts, and Chart Editor, as well as in the SPSS Tutorials under Creating and Editing Charts. The commands for making charts are located under Graphs (and then Legacy Dialogs, if using Version 15) on the menu bar, and the commands for making simple figures and graphs are relatively easy to use and some instruction is given below. The Interactive option under Graphs is another way to produce charts in SPSS interactively, as well as fancier versions of the basic charts (e.g., 3-dimensional bar charts).

Bar Charts

The easiest way to produce simple bar charts is to use the Bar Chart option with the Frequencies... command. See Frequency Tables (& Bar Charts) for Categorical Variables. You can only produce only one bar chart at a time using the Bar command.

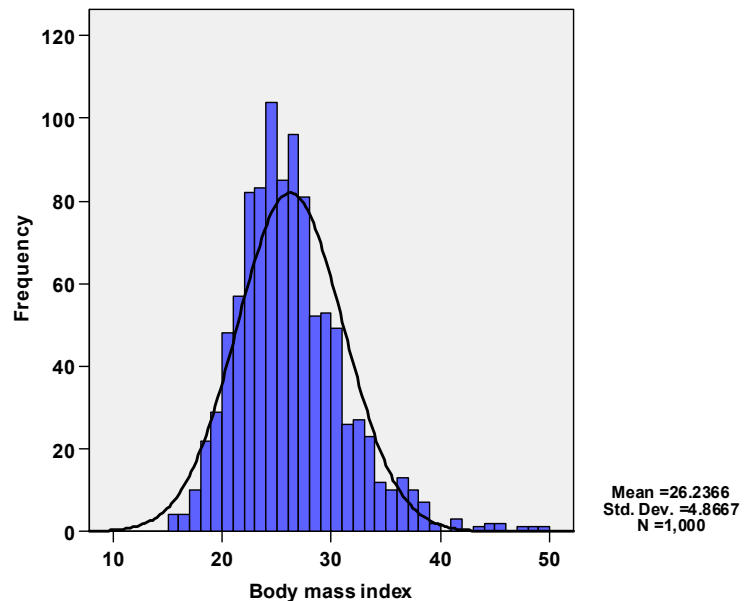
1. Choose **Graphs** (& then **Legacy Dialogs**, if Version 15) from the menu bar.
2. Choose **Bar...**
3. Choose **Simple, Clustered, or Stacked**
4. Choose what the **data in the bar chart** represent (e.g., summaries for groups of cases).
5. Choose **Define**
6. **Select a variable** from the variable list on the left and the click on the arrow next to the Category axis.
7. Choose what the **bars represent** (e.g., number of cases or percentage of cases)
8. Choose **OK**



Histograms

The easiest way to produce simple histograms is to use the Histogram option with the Frequencies... command. See Descriptive Statistics (& Histograms) for Numerical Variables. You can produce only one histogram at a time using the Histogram command.

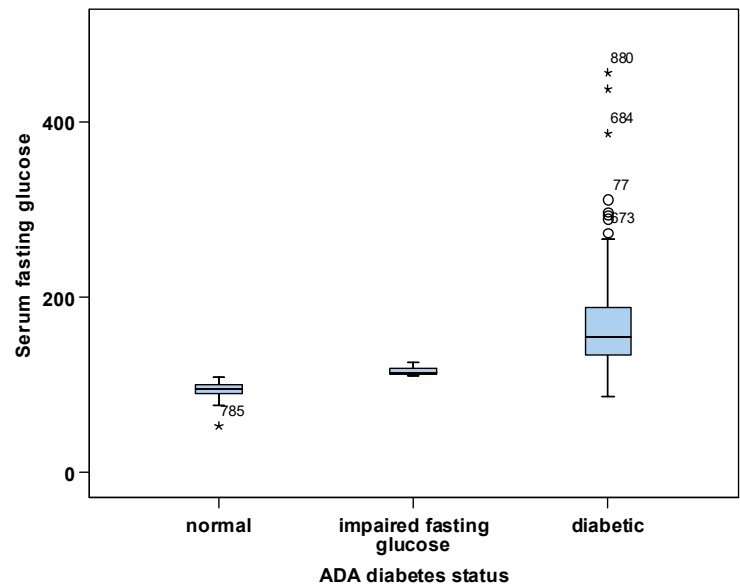
1. Choose **Graphs** (& then **Legacy Dialogs**, if Version 15) from the menu bar
2. Choose **Histogram...**
3. **Select a variable** from the variable list on the left and then click on the arrow in the middle of the window.
4. Choose Display **normal** Curve if you want a normal curve superimposed on the histogram.
5. Choose **OK**



Boxplots

The easiest way to produce simple boxplots is to use the Boxplot option with the Explore... command. See Descriptive Statistics (& Boxplots) By Groups for Numerical Variables. You can produce only one boxplot at a time using the Boxplot command.

1. Choose **Graphs** (& then **Legacy Dialogs**, if Version 15) from the menu bar.
2. Choose **Boxplot...**
3. Choose **Simple or Clustered**
4. Choose what the **data in the boxplots represent** (e.g., summaries for groups of cases).
5. Choose **Define**
6. **Select a variable** from the variable list on the left and then click on the arrow next to the Variable box.
7. **Select the variable** from the variable list that **defines the groups** and then click on the arrow next to Category Axis.
8. Choose **OK**



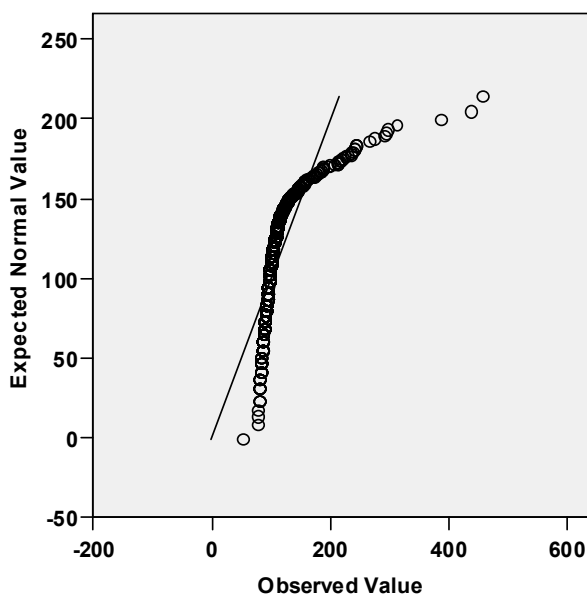
Normal Probability Plots.

To produce Normal probability plots:

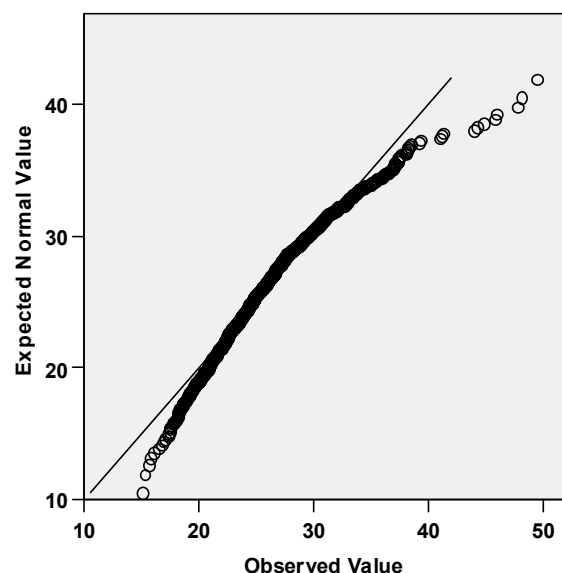
1. Choose **Graphs** (& then **Legacy Dialogs**, if Version 15) from the menu bar.
2. Choose **Q-Q...** to get a plot of the quantiles (Q-Q plot) or choose **P-P...** to get a plot of the cumulative proportions (P-P plot)
3. **Select the variables** from the source list on the left and then click on the arrow located in the middle of the window.
4. **Choose Normal** as the Test Distribution. The Normal distribution is the default Test Distribution. Other Test Distributions can be selected by clicking on the down arrow and clicking on the desired Test distribution.
5. Choose **OK**

SPSS will produce both a Normal probability plot and a detrended Normal probability plot for each selected variable. Usually the Q-Q plot is the most useful for assessing if the distribution of the variable is approximately Normal.

Normal Q-Q Plot of Serum fasting glucose



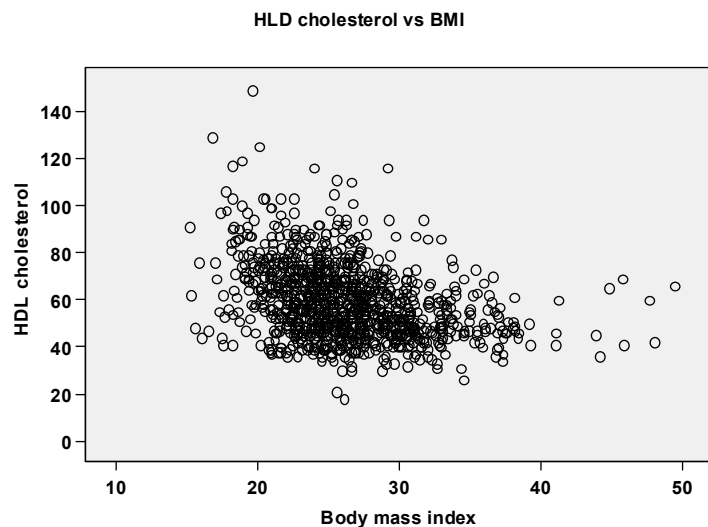
Normal Q-Q Plot of Body mass index



Scatter Plot.

To produce a scatter plot between two numerical variables:

1. Choose **Graphs** (& then **Legacy Dialogs**, if Version 15) on the menu bar.
2. Choose **Scatter/Dot...**
3. Choose **Simple**
4. Choose **Define**
5. **Y Axis:** Select the y variable you want from the source list on the left and then click on the arrow next to the y axis box.
6. **X Axis:** Select the x variable you want from the source list on the left and then click on the arrow next to the x axis box.
7. Choose **Titles...**
8. Enter a title for the plot (e.g., y vs. x).
9. Choose **Continue**
10. Choose **OK**



جامعة جنوب ال

South Valley University

Practical Example (1) DESCRIPTIVE STATISTICS

Consider the smoker data condensed from a study conducted to explore the prevalence and impact of smoking problems on various aspects of people's lives. Staff from a university in Melbourne, Australia were invited to complete a questionnaire containing questions about their smoking status.

The sample consisted of 271 respondents (55% female, 45% male) ranging in age from 18 to 84 years (mean=44yrs). The task is to use SPSS to analyse this data.

The figure below shows the variable view window in the SPSS.

The screenshot shows the SPSS Data Editor window for a dataset named 'sleep.sav'. The 'Variable View' tab is active, displaying a table of variables. The variables listed are 'id', 'sex', 'age', 'weight', 'height', and 'smoke'. Each variable is defined with a name, type (Numeric), width (8), decimals (0), label, values, missing values, column position, alignment (Right), and measure (Scale or Nominal).

	Name	Type	Width	Decimals	Label	Values	Missing	Columns	Align	Measure
1	id	Numeric	8	0	Identification	None	None	8	Right	Scale
2	sex	Numeric	8	0	sex	{0, female}...	None	8	Right	Nominal
3	age	Numeric	8	0	age	None	None	8	Right	Scale
4	weight	Numeric	8	0		None	None	8	Right	Scale
5	height	Numeric	8	0		None	None	8	Right	Scale
6	smoke	Numeric	8	0	do you smoke	{1, yes}...	None	8	Right	Nominal
7										
8										
9										
10										

The figure below shows a partial view of the smoker data.



*sleep.sav [DataSet1] - SPSS Data Editor

File Edit View Data Transform Analyze Graphs Utilities Window Help

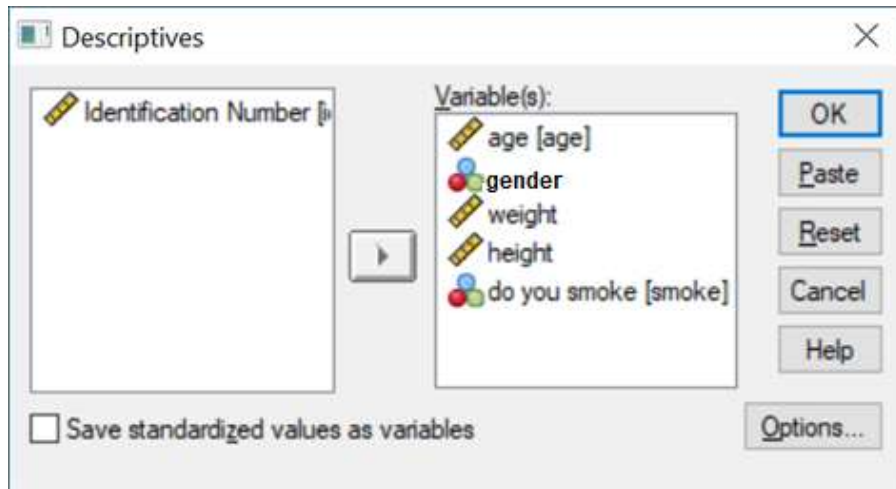
10 : height 187

	id	gender	age	weight	height	smoke
1	83	0	42	52	162	1
2	294	0	54	65	174	2
3	425	1	.	89	170	2
4	64	0	41	66	178	1
5	536	0	39	62	160	2
6	57	0	66	62	165	2
7	251	0	36	62	165	2
8	255	0	35	75	174	2
9	265	1	.	90	180	2
10	290	1	41	75	187	1
11	418	1	.	70	.	2
12	95	1	.	78	178	2
13	77	0	33	67	158	2
14	203	0	.	80	173	1
15	413	0	24	56	162	2
16	69	0	24	49	165	2
17	277	0	35	50	153	2
18	401	1	28	75	170	2
19	7	1	19	60	176	2
20	12	1	31	.	.	2
21	50	1	.	72	173	2
22	67	1	.	95	192	1
23	218	0	41	75	170	2
24	245	1	45	89	175	2
25	258	0	23	68	178	2
26	291	0	48	.	163	2
27	2	1	62	90	188	2
28	40	1	.	94	185	2
29	51	1	.	70	179	1
30	26	1	.	63	174	2
31	249	1	21	.	.	2
32	410	1	27	82	185	2
33	513	0	31	67	165	2
34	525	1	26	80	180	2
35	47	1	25	75	180	2
36	63	1	59	75	182	2
37	68	1	44	90	174	2
38	207	1	.	56	171	2
39	302	1	56	75	180	2
40	409	1	.	75	176	2
41	112	0	36	56	163	2

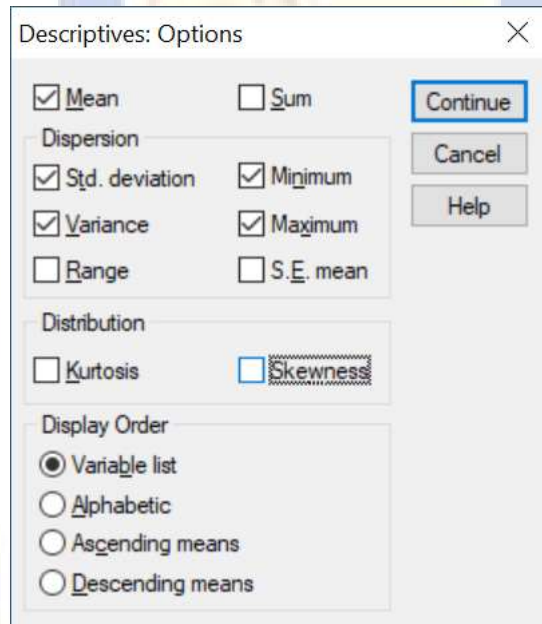
Data View Variable View

SPSS Process

Now to obtain the descriptive statistics from the **Analyze** menu choose **descriptive** statistics, then choose the variables you wish to analyse, as depicted in the following screen shot. Choose the variables age, gender, weight, height and smoke.



Then click options and make sure the following options are selected, as showed in the following screen.



After selecting the previous settings, hit the continue button and the following results will appear as in the following screen.

Descriptive Statistics

	N	Minimum	Maximum	Mean	Std. Deviation	Variance
age	248	18	84	43.87	12.684	160.882
gender	271	0	1	.45	.498	.248
weight	249	43	160	73.38	15.284	233.606
height	246	150	199	170.22	10.275	105.585
do you smoke	270	1	2	1.87	.332	.110
Valid N (listwise)	218					

After a careful examination for the descriptive statistics for the five variables (age, gender, weight, height and smoke). The table depicts six columns in addition to the variable name column. The first column is **N** which indicates the total number of observations for each variable. The other five columns represent the minimum, maximum, mean, standard deviation and variance for each of the five variables.

For a quick refreshment for these terms study the following formulas:

Mean

$$\bar{x} = \frac{(\sum x)}{n}$$

Standard deviation

$$SD = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}$$

Variance

$$\text{Variance} = \frac{\sum(x - \bar{x})^2}{n - 1}$$

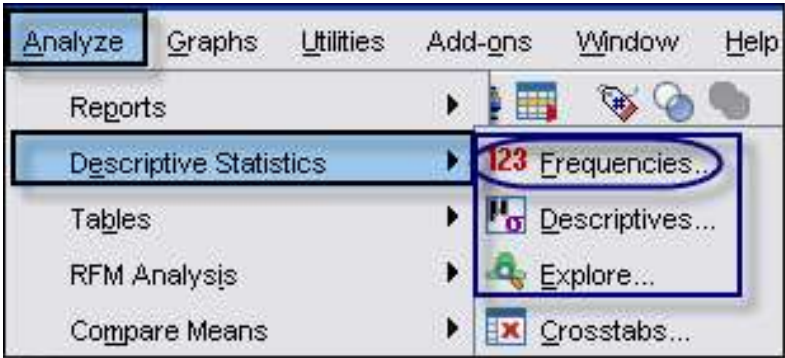
And finally **Min** and **Max** are the highest and lowest values among the variable values.

NOTE. For a given variable there might be missing entries, that's is why not all the number of observations (Column N) are equal in the table.

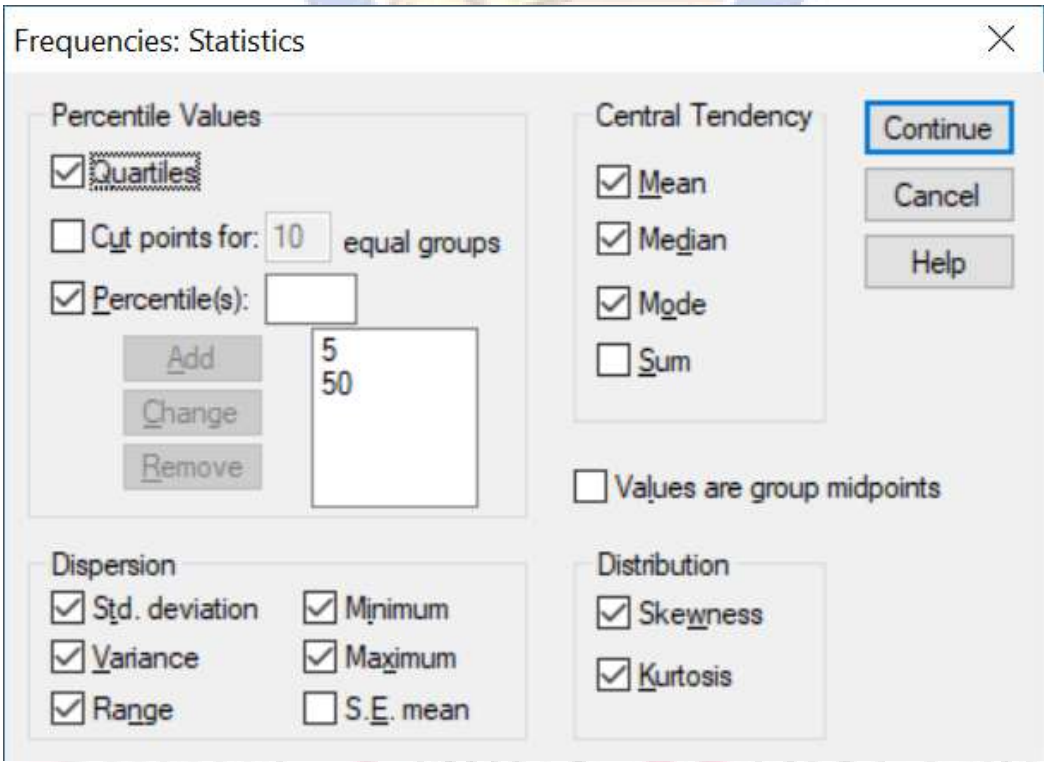
CONTINUED. Practical Example (1) Frequency Analysis

SPSS is rich with various commands to explore the data. For example, you can use the Frequencies command to reorganize your data and sort them ascending or descending according to a specific choice.

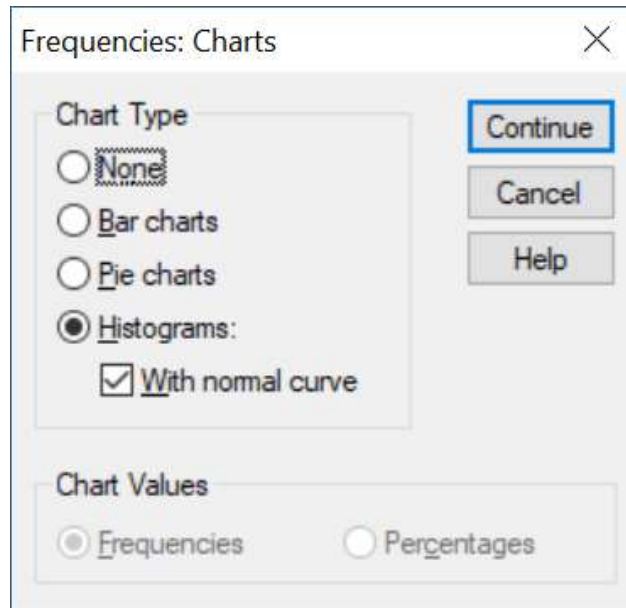
Consider the previous example for the smoke data. To obtain the frequency analysis for the variables from the Analyze menu select frequencies as shown below. **Analyze → Descriptive statistics → Frequencies**



Then select the required statistics you require, as shown below.



The final step is to select the required charts you want to display. For example, select the Histogram chart and check the with normal curve option. This option enables you to check if the data follows the normal distribution or not.



The following results appear upon pressing **continue** in the previous dialogue. This table displays the central tendency measures for each variable of the five variables.

Statistics

		gender	age	weight	height	do you smoke
N	Valid	271	248	249	246	270
	Missing	0	23	22	25	1
Mean		.45	43.87	73.38	170.22	1.87
Median		.00	44.00	72.00	170.00	2.00
Mode		0	35 ^a	75	165	2
Std. Deviation		.498	12.684	15.284	10.275	.332
Variance		.248	160.882	233.606	105.585	.110
Skewness		.216	.075	1.009	.355	-2.268
Std. Error of Skewness		.148	.155	.154	.155	.148
Kurtosis		-1.968	-.566	3.472	-.610	3.166
Std. Error of Kurtosis		.295	.308	.307	.309	.295
Range		1	66	117	49	1
Minimum		0	18	43	150	1
Maximum		1	84	160	199	2
Percentiles	5	.00	23.00	51.50	155.00	1.00
	25	.00	34.00	63.00	162.50	2.00
	50	.00	44.00	72.00	170.00	2.00
	75	1.00	54.00	82.00	178.00	2.00

a. Multiple modes exist. The smallest value is shown

SPSS frequency command also gives a frequency table for each of the variables as shown below.

gender

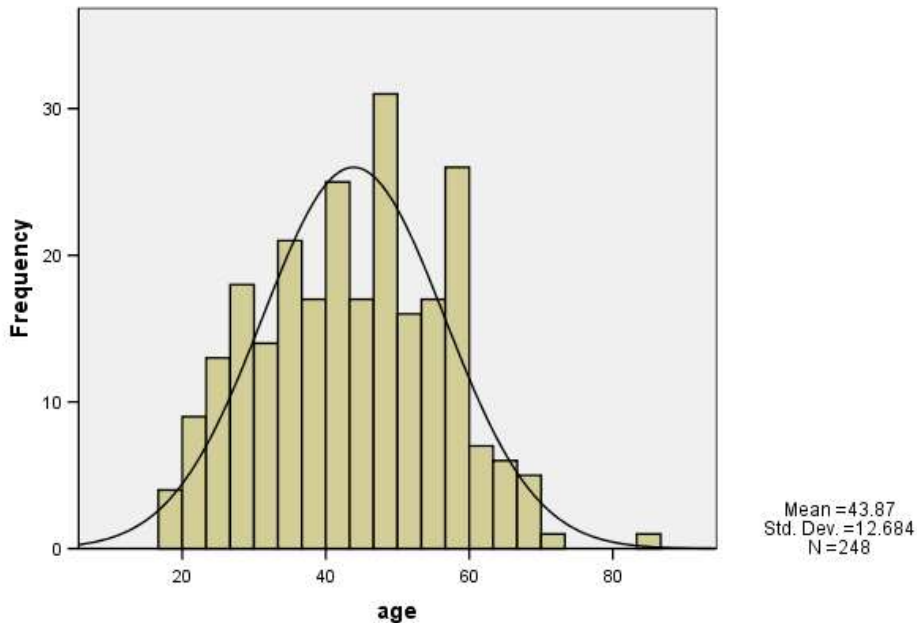
		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	female	150	55.4	55.4	55.4
	male	121	44.6	44.6	100.0
	Total	271	100.0	100.0	

do you smoke

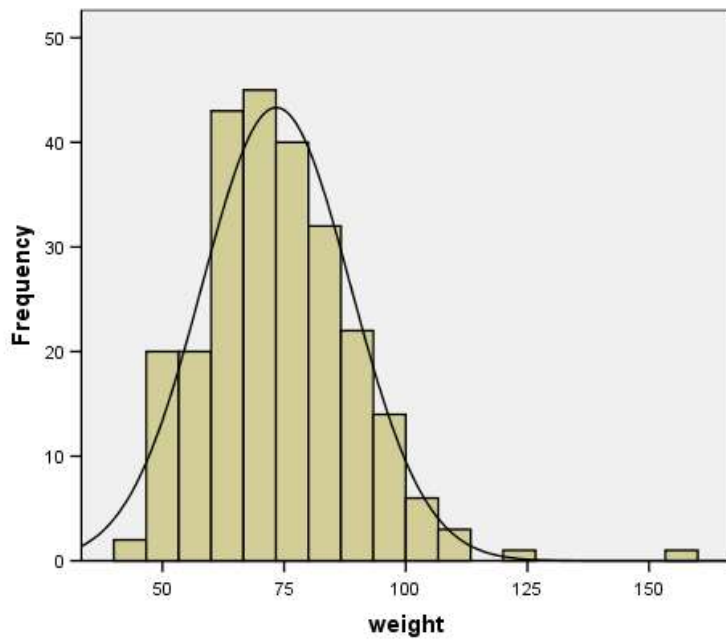
		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	yes	34	12.5	12.6	12.6
	no	236	87.1	87.4	100.0
	Total	270	99.6	100.0	
Missing	System	1	.4		
Total		271	100.0		

To find the rest of the frequencies tables you can apply the example on your computer. The second part is the histograms tables that is displayed per each variable along with the statistics table. As depicted in the following results.

age

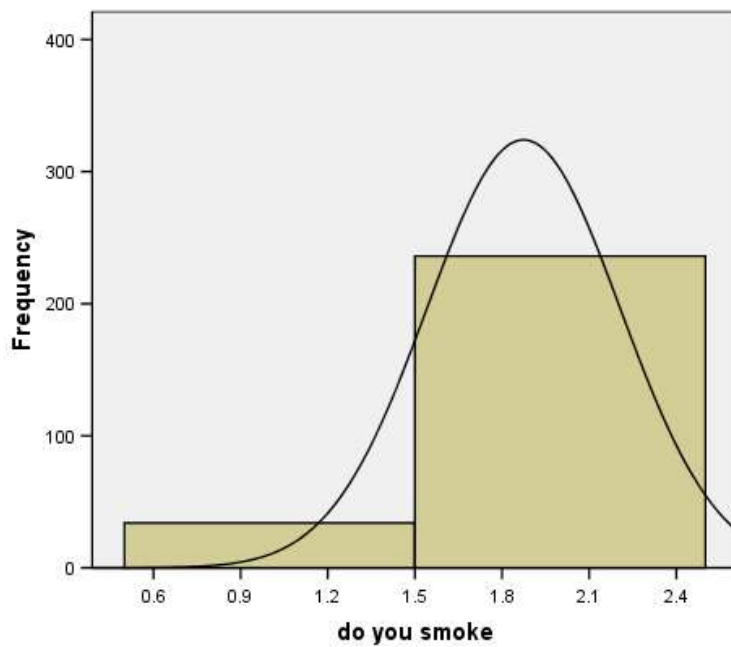


weight



Mean = 73.38
Std. Dev. = 15.284
N = 249

do you smoke



Mean = 1.87
Std. Dev. = 0.332
N = 270

Hypothesis Tests & Confidence Intervals

One-Sample t -Test

The One Sample t -Test determines whether the sample mean is statistically different from a known or hypothesized population mean. The One Sample t -Test is a parametric test. Furthermore, this test can only compare a single sample mean to a specified constant. It cannot compare sample means between two or more groups. If you wish to compare the means of multiple groups to each other, you will likely want to run an Independent Samples t -Test (to compare the means of two groups) or a One-Way ANOVA (to compare the means of two or more groups).

1. Choose **Analyze** from the menu bar.
2. Choose **Compare Means**
3. Choose **One-Sample T Test...**
4. **Test Variable(s)**: Select the variable you want from the source list on the left, highlight variables by pointing and clicking the mouse and then click on the arrow located in the middle of the window.
5. **Edit the Test Value**. The Test Value is the value of the mean under the null hypothesis. The default value is zero.
6. Choose **OK**

Confidence Interval for a Mean (from one sample of data)

1. Choose **Analyze** from the menu bar.
2. Choose **Compare Means**
3. Choose **One-Sample T Test...**
4. **Test Variable(s)**: Select the variable you want from the source list on the left, highlight variables by pointing and clicking the mouse and then click on the arrow located in the middle of the window.
5. **The Test Value should be 0**, which is the default value.

6. By default, a 95% confidence interval will be computed. Choose Options... to change the confidence level.

7. Choose **OK**

EXAMPLE (2)

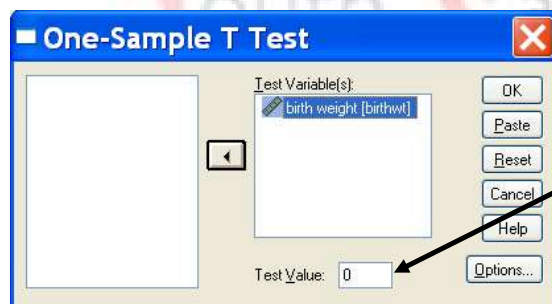
SIDS Example. There were 48 SIDS cases in King County, Washington, during the years 1974 and 1975. The birth weights (in grams) of these 48 cases were:

2466	3941	2807	3118	2098	3175
3317	3742	3062	3033	2353	3515
2013	3515	3260	2892	1616	4423
2750	2807	2807	3005	3374	3572
2722	2495	3459	3374	1984	2495
3005	2608	2353	4394	3232	3062
2013	2551	2977	3118	2637	1503
2722	2863	2013	3232	2863	2438

The mean (and standard deviation) of these measurements is 2891 (623) grams.

We want to know if the mean birth weight in the population of SIDS infant is different from that of normal children, 3300 grams. We could construct a 95% **confidence interval**, to see if the interval contains the value of 3300 grams or we could perform a **one sample t test** to test if the mean in the SIDs population is equal to 3300 (versus not equal to 3300).

To construct a **95% confidence interval**



When computing the interval for a mean make sure the Test Value is 0.

One-Sample Statistics

	N	Mean	Std. Deviation	Std. Error Mean
birth weight	48	2891.1250	623.39177	89.97885

Number of subjects, mean, standard deviation, and standard error of the mean.

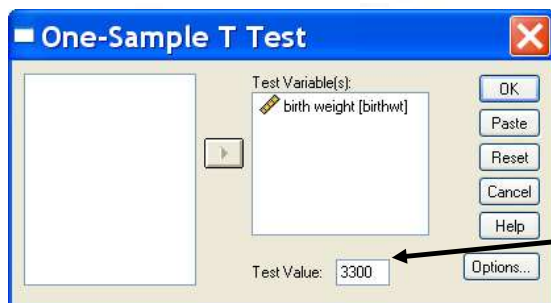
One-Sample Test

	Test Value = 0					
	t	df	Sig. (2-tailed)	Mean Difference	95% Confidence Interval of the Difference	
					Lower	Upper
birth weight	32.131	47	.000	2891.12500	2710.1109	3072.1391

Ignore the t test results (t, df, sig.) because these results are for testing if the mean birth weight is equal to 0 (versus not equal to zero).

95% confidence interval for the mean birth weight is 2710 to 3072 grams

To perform a **one sample t test** to test if the mean in the SIDs population is equal to 3300 versus not equal to 3300.



To run the one-sample t test to test if the mean birth weight is equal to 3300 you need to change the Test Value from the default value of 0 to 3300.

One-Sample Statistics

	N	Mean	Std. Deviation	Std. Error Mean
birth weight	48	2891.1250	623.39177	89.97885

One-Sample Test

	Test Value = 3300					
	t	df	Sig. (2-tailed)	Mean Difference	95% Confidence Interval of the Difference	
					Lower	Upper
birth weight	-4.544	47	.000	-408.87500	-589.8891	-227.8609

Sig. (2-tailed) = two tailed p-value = <.001

t = test statistic value = -4.544

df = degrees of freedom = 47

Ignore the results for 95% confidence interval of the difference, because it is the confidence interval for the mean minus 3300.

Paired t-Test

The Paired Samples *t*-Test compares two means that are from the same individual, object, or related units. The two means typically represent two different times (e.g., pre-test and post-test with an intervention between the two time points) or two different but related conditions or units (e.g., left and right ears, twins). The purpose of the test is to determine whether there is statistical evidence that the mean difference between paired observations on a particular outcome is significantly different from zero. The Paired Samples *t*-Test is a parametric test.

1. Choose **Analyze** from the menu bar.
2. Choose **Compare Means**
3. Choose **Paired-Samples T Test...**

4. **Paired Variable(s):** Select two paired variables you want from the source list on the left, highlight both variables by pointing and clicking the mouse and then click on the arrow located in the middle of the window. Repeat the process until you have selected all the paired variables you want to test.
5. Choose **OK**

Confidence Interval for the Difference Between Means from Paired Sample. By default, a 95% confidence interval for the difference means of the paired samples will be computed when performing a paired t test. Choose Options... to change the confidence level.

EXAMPLE (2)

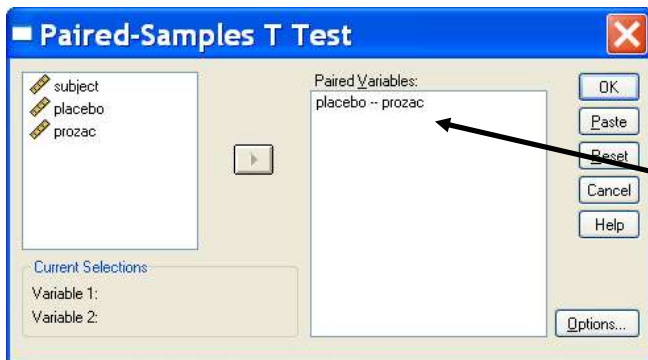
Prozac Example. To compare the effect of Prozac (some kind of medicine) on anxiety 10 subjects are given one week of treatment with Prozac and one week of treatment with a placebo (another medicine). The order of the treatments was randomized for each subject. An anxiety questionnaire was used to measure a subject's anxiety on a scale of 0 to 30. Higher scores indicate more anxiety.

Subject	Placebo	Prozac	Difference
1	22	19	3
2	18	11	7
3	17	14	3
4	19	17	2
5	22	23	-1
6	12	11	1
7	14	15	-1
8	11	19	-8
9	19	11	8
10	7	8	-1

Mean difference, $\bar{d} = 1.3$

Standard deviation, $s_d = 4.5$

Paired t-test and confidence interval for the difference between paired means.



The order of the variables in calculating the difference is determined by the order of the variables in the data set (and not the order in which you select the variables).

Paired Samples Statistics

	Mean	N	Std. Deviation	Std. Error Mean
Pair 1 placebo	16.1000	10	4.95424	1.56667
prozac	14.8000	10	4.68568	1.48174

Summaries for each sample of data (or variable).

Paired Samples Correlations

	N	Correlation	Sig.
Pair 1 placebo & prozac	10	.556	.095

Correlation between the paired values - usually not

Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
				Lower	Upper			
Pair 1 placebo - prozac	1.30000	4.54728	1.43798	-1.95293	4.55293	.904	9	.390

difference = placebo - prozac

mean difference = 1.3

standard deviation of the differences = 4.5

standard error of the differences = 1.4

95% confidence interval for the mean difference is -1.9 to 4.6

Paired t test

Sig. (2 tailed) = two-sided p-value = 0.39

t = test statistic value = .904

df = degrees of freedom

Two-Sample t -Test

The Independent Samples t -Test compares the means of two independent groups in order to determine whether there is statistical evidence that the associated population means are significantly different. The Independent Samples t -Test is a parametric test.

1. Choose **Analyze** on the menu bar.
2. Choose **Compare Means**
3. Choose **Independent-Samples T Test...**
4. **Test Variable(s)**: Select the test variable you want from the source list on the left and then click on the arrow located next to the test variable box. Repeat the process until you have selected all the variables you want.
5. **Grouping Variable**: Select the variable which defines the groups and then click on the arrow located next to the grouping variable box.
6. Choose **Define Groups...**
7. Click on blank box next to Group 1, then **enter the code value (numeric or character/string) for group 1.**
8. Click on blank box next to Group 2, then **enter the code value (numeric or character/string) for group 2.**
9. Choose **Continue**
10. Choose **OK**

Confidence Interval for the Difference Between Means from Independent Samples

By default, a 95% confidence interval for the difference means from two independent samples will be computed when performing a two sample t test. Choose Options... to change the confidence level.

EXAMPLE (3)

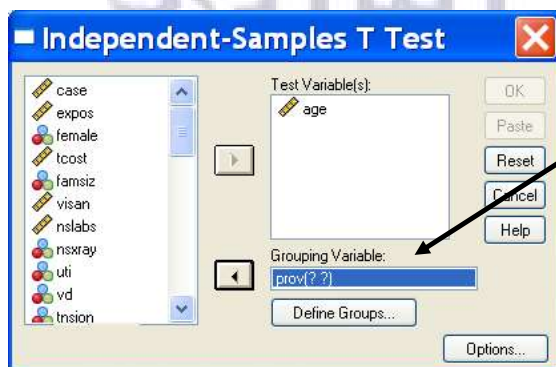
Model Cities Example. Two groups of people were studied - those who had been randomly allocated to a Fee-For-Service medical insurance group and those who had been randomly allocated to a Prepaid insurance group.

We would like to compare the two groups on the quality of health care they received in each group, but first we would like to know how comparable the groups are on other characteristics that might affect medical outcome. For example, we would like to know if the mean age in the two groups is similar. Hopefully, the process of random allocation minimizes this possibility, but there is always a chance that it didn't.

Group	n	Mean	Standard deviation
Prepaid (GHC)	1167	24.0	15.3
Fee-for-service (KCM)	3207	26.4	17.1

We could compare the average age between the two groups using a two sample t-test or a confidence interval for the difference between the average ages of the two groups.

Two sample t test and 95% confidence interval for the difference between means (from independent samples).



After you select the Grouping Variable, SPSS will put in question marks to prompt you to define the code values for the two groups. Select Define Groups... to enter the code values.



In this example the group codes are numeric, 0 (for GHC) and 1 (for KCM)

T-Test

Group Statistics

	prov	N	Mean	Std. Deviation	Std. Error Mean
age	GHC	1167	23.9846	15.30787	.44810
	KCM	3207	26.3676	17.10260	.30200

Summaries for each sample/group.

Independent Samples Test

		Levene's Test for Equality of Variances	
		F	Sig.
age	Equal variances assumed	47.068	.000
	Equal variances not assumed		

SPSS by default tests if the variances are equal using Levene's test. A small p-value (sig.) indicates the variances may be different.

sig. = p-value = <.001

F = test statistic value = 47.0

Independent Samples Test

		t-test for Equality of Means				
		t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference
age	Equal variances assumed	-4.188	4372	.000	-2.38306	.56896
	Equal variances not assumed	-4.410	2293.698	.000	-2.38306	.54037

Two Sample t test. SPSS by default always performs both versions of the two sample t test assuming equal variance and unequal variances

Sig. (2 – tailed) = two sided p-value = <.001 (equal var.), <.001 (unequal var.)

t = test statistic value = -4.2 (equal var.), -4.4 (unequal var.)

df = degrees of freedom = 4372 (equal var.), 2294 (unequal var.)

mean difference = difference between means = -2.4 (equal and unequal var.)

std. error difference = standard error of the difference between means = .6 (equal var.), .5 (unequal var.)

Independent Samples Test

		95% Confidence Interval of the Difference	
		Lower	Upper
age	Equal variances assumed	-3.49851	-1.26760
	Equal variances not assumed	-3.44273	-1.32338

95% confidence interval for the difference between means is

-3.4 to -1.3 (assuming equal variances)

-3.4 to -1.3 (assuming unequal variances)

One-way ANOVA (Analysis of Variance)

The One-Way ANOVA ("analysis of variance") compares the means of two or more independent groups in order to determine whether there is statistical evidence that the associated population means are significantly different. One-Way ANOVA is a parametric test. The variables used in this test are known as:

Dependent variable

Independent variable (also known as the grouping variable, or factor)

This variable divides cases into two or more mutually exclusive levels, or groups.

Towards comparing two or more means from two or more independent samples

1. Choose **Analyze** on the menu bar
2. Choose **Compare Means**
3. Choose **One-Way ANOVA...**
4. **Dependent:** Select the variable from the source list on the left for which you want to use to compare the groups and then click on the arrow next to the dependent variable box. You run multiple one-way ANOVAs by selecting more than one dependent variable.
5. **Factor:** Select the variable from the source list on the left which defines the groups.
6. Choose **OK**

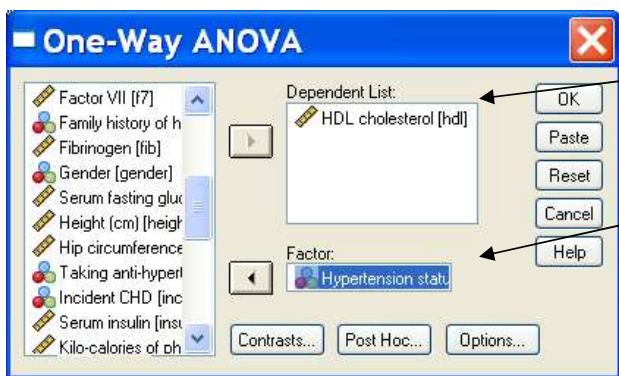
To perform pairwise comparisons to determine which groups are different while controlling for multiple testing use the Post Hoc... option. There are many methods to choose from (e.g., Bonferroni and R-E-G-W-Q).

Other useful options can be found under Options... For example, choose Descriptive to get descriptive statistics for each group (e.g., mean, standard deviation, minimum value, and maximum value). Choose Homogeneity-of-variance to perform the Levene Test to test if the group variances are all equal versus not all equal. A small p-value for the Levene's Test may indicate that the variances are not all equal.

EXAMPLE (4)

CHD Example. We can use one-way ANOVA to compare HDL levels between subjects with different hypertensive status (0=normotensive, 1=borderline, 2=definite)

Hypertensive Group	n	Mean	Standard Deviation
Normotensive	1568	55.8	15.5
Borderline	547	55.7	16.2
Definite	1310	53.5	15.2



You can select 1 or more variables to compare between groups.

The variable selected as the Factor defines the groups. The variable can be numeric or character/string.

Oneway ANOVA

HDL cholesterol

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	4344.834	2	2172.417	9.045	.000
Within Groups	821904.577	3422	240.183		
Total	826249.411	3424			

One-way analysis of variance

Sig. = p-value = <.001

F = test statistic = 9.0; df = degrees of freedom

Sometimes the test statistic and degrees of freedom of the test statistics are reported along with the p-value; in this example, F=9.0 with degrees of freedom 2 and 3422. Sum of squares and mean square are used to compute the test statistic; they are usually not reported.

Descriptive statistics

Under Options you can request **Descriptive** for each group to be computed. This information can be used to describe the differences between the groups.

HDL cholesterol

	N	Mean	Std. Deviation	Std. Error	95% Confidence Interval for Mean		Minimum	Maximum
					Lower Bound	Upper Bound		
normotensive	1568	55.82	15.500	.391	55.05	56.59	21	138
borderline	547	55.67	16.202	.693	54.30	57.03	24	149
definite	1310	53.47	15.192	.420	52.64	54.29	15	129
Total	3425	54.90	15.534	.265	54.38	55.42	15	149

Exercises

1. The following data represents the ages of a sample population of 23 college students. Enter the data in the SPSS and write the SPSS commands to display and explore this data.

AGE: 16 17 19 20 25 30 17 17 18 18 15 14
12 17 19 21 22 23 24 20 20 16 24.

2. The following data represents the weight of a group of students, use the SPSS program to find the mode, mean, variance and standard deviation.

WEIGHT 42 44 48 46 50 52 55 44

3. The administrator at your local hospital states that on weekends the average wait time for emergency room visits is 10 minutes. Based on discussions you have had with friends who have complained on how long they waited to be seeing in the ER over a weekend, you dispute the administrator's claim. You decide to test your hypothesis. Over the course of a few weekends you record the wait time for 40 randomly selected patients. The average wait time for these 40 patients is 11 minutes with a standard deviation of 3 minutes. Use SPSS to examine if you have enough evidence to support your hypothesis that the average ER wait time exceeds 10 minutes? You opt to conduct the test at a 5% level of significance.

Previous Exams

Choose the right answer

(1) The computer language that uses binary symbols:

- (A) Machine Language
- (B) HTML
- (C) High-Level Language
- (D) Assembly Language

(2) A program that converts a program source code to a machine language :

- (A) Microsoft Office
- (B) Compiler
- (C) Operating system
- (D) Visual Studio

(3) The C++ language is considered:

- (A) Assembly language
- (B) High level language
- (C) Procedural language
- (D) Machine language

(4) The extension of an executable C++ program is:

- (A) .obj
- (B) .exe
- (C) .bmp
- (D) .cpp

(5) The extension of a C++ source file is:

- (A) .obj
- (B) .cpp
- (C) .exe
- (D) .link

(6) The stage that precedes the compilation of a C++ program:

- (A) Linking
- (B) Pre-processing
- (C) Errors discovering
- (D) Compilation

(7) The <iostream> header file includes the definitions of:

- (A) String functions
- (B) Input/output commands
- (C) File handling functions
- (D) Math functions

(8) The <cmath> header file includes the definitions of:

- (A) String functions
- (B) Math functions
- (C) Input/output commands
- (D) File handling functions

(9) Each C++ file contains:

- (A) A single cin statement
- (B) A single main() function
- (C) A single #include statement
- (D) A single cout statement

(10) Each C++ line ends with a:

- (A) Insertion operator
- (B) Semicolon
- (C) Extraction operator
- (D) Double quotation

- (11) The **return 0;** statement at the end of each C++ program means:
- (A) Normal program termination (C) An endless program
 (B) The program did not execute yet (D) Problematic program termination
- (12) Which C++ statements are not executed:
- (A) pre-processor directives (C) cout
 (B) comments (D) cin
- (13) The command that defines an integer variable in C++:
- (A) char (C) float
 (B) unsigned long (D) unsigned double
- (14) The C++ datatype that uses one byte of memory
- (A) float (C) unsigned int
 (B) char (D) long
- (15) How many ways to write a comment in a C++ program:
- (A) 4 (C) 1
 (B) 2 (D) 3
- (16) The keyword the forbids changing a variable value:
- (A) assignment (C) fixed
 (B) const (D) int
- (17) The statement **k=k+1;** can be abbreviated to:
- (A) k; (C) k+=2;
 (B) ++k; (D) k++;
- (18) The statement **k=k-2;** is equivalent to:
- (A) k-=1; (C) k-=1;
 (B) k--; (D) k--;
- (19) The expression **k+m/3/y-1;** is equivalent to:
- (A) k+(m/3)/(y--);
 (B) (k+((m/3) /y))-1;
 (C) k+(m/3)/(y-1);
 (D) k+m+3/(y-1);

Study the next C++ program (figure 1) then answer the questions from 20 to 30:

```

1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5     int x1=10, x2=30,
6     x3=20;
7     cout<< x1+x2/2;
8     cout<< x1+(x2/15);
9     cout<< (x1+x2%10);
10    cout<< (x1+x2)/2;
11    cout<< (x1*x2)/2;
12    cout<<x1/x2;
13    ++x1;
14    for(int k=0; k<113;
15    k++)
        cout<<k;
        return 0;
    }
  
```

- (20) The cout output at line 6 is:
- (A) 2.66 (C) 40
(B) 2 (D) 25
- (21) The cout output at line 7 is:
- (A) 2.66 (C) 40
(B) 2 (D) 12
- (22) The cout output at line 8 is :
- (A) 2.66 (C) 40
(B) 0 (D) 10
- (23) The cout output at line 9 is :
- (A) 2.66 (C) 40
(B) 12 (D) 20
- (24) How many variables in the whole program :
- (A) 3 floats (C) 3 integers
(B) 4 floats (D) 4 integers
- (25) The cout output at line 10 is:
- (A) 2.66 (C) 40
(B) 20 (D) 5
- (26) The cout output at line 11 is:
- (A) 0.33 (C) 10
(B) 1 (D) 0
- (27) How many time the loop at line 13 is repeated:
- (A) 100 (C) 112
(B) 110 (D) 113
- (28) The loop at line 13 prints:
- (A) Numbers between 0 and 113 (C) Even numbers between 0 and 113
(B) Odd numbers between 0 and 113 (D) Numbers starting from 0 to 112
- (29) The last even number printed by the loop at line 13 is:
- (A) 113 (C) 110
(B) 111 (D) 112
- (30) The last odd number printed by the loop at line 13 is:
- (A) 111
(B) 112
(C) 113
(D) 110

REFERENCES

1. Meyers, Scott. More effective C++: 35 new ways to improve your programs and designs. 1995.
2. Stroustrup, Bjarne. The design and evolution of C++. Pearson Education India, 1994.
3. Deitel, Paul, and Harvey Deitel. C++ How to Program 7th Edition. Prentice Hall, 2010.
4. Karlsson, Björn. Beyond the C++ standard library: an introduction to boost. Pearson Education, 2005.
5. Dawson, Michael. Beginning C++ Game Programming (Game Development Series). Premier Press, 2004.
6. Lafore, Robert. Object-oriented programming in C++. Pearson Education, 1997.
7. Venugopal, K. R., and Rajkumar Buyya. Mastering C++. Tata McGraw-Hill Education, 2013.
8. Schildt, Herbert. C++: the complete reference. McGraw-Hill/Osborne, 2003.
9. Stroustrup, Bjarne. The C++ programming language. Pearson Education, 2013.
10. S. J. Chapman. MATLAB Programming for Engineers. Thomson, 2004.
11. The MathWorks Inc. MATLAB 7.0 (R14SP2). The MathWorks Inc., 2005.
12. C. F. Van Loan. Introduction to Scientific Computing. Prentice Hall, 1997.
13. Statistics and Machine Learning Toolbox User's Guide, 2016.
14. J. Cooper. A MATLAB Companion for Multivariable Calculus. Academic Press, 2001.
15. J. C. Polking and D. Arnold. ODE using MATLAB. Prentice Hall, 2004.
16. Houcque, David. "Introduction to Matlab for engineering students." Northwestern University(2005).
17. D. Kahaner, C. Moler, and S. Nash. Numerical Methods and Software. Prentice-Hall, 1989.
18. Neter, J., M. H. Kutner, C. J. Nachtsheim, and W. Wasserman. 4th ed. Applied Linear Statistical Models. Irwin Press, 1996.
19. Glenda Francis, INTRODUCTION TO SPSS FOR WINDOWS Version 15.0. 5th edition, 1999.
20. Chaudhuri, Anil Bikas., The Art of Programming Through Flowcharts & Algorithms. Firewall Media, 2005.

Communication with Lecturer

Email:



Website:

