



Computer Programming

University Of Salahaddin
Engineering College
Electrical Engineering
2009-2010





Study Skills Tips

You have to develop many different types of skills to be a successful student. What skills do you want to work on?





Study Skills

1. Find your own quiet place where you can concentrate on your homework.
2. Sit in a comfortable chair (not too comfortable, though or you could end up asleep).
3. Avoid distractions like the TV and try to ignore the telephone. Your friends can leave a message!
4. Play quiet background music. It might help you stay focused.
5. Study with a friend or a group of friends. Compare notes and ask each other questions.
6. Know your learning style and study in a way that best matches it.
7. Take short but frequent breaks.
8. Relate what you're studying to things you already know. This helps you remember information more easily.
9. Start with the most difficult tasks or assignments to focus maximum brainpower on the toughest. Then move on to the easier tasks.



Study Skills (cont)

1. Plan to spend more time (not less) on the subjects that are harder for you.
2. Focus on the quality of your study time. It's much more important than the quantity.
3. Get into the habit of studying every day.
4. Determine your best study time and plan to study at that time every day.
5. Think of homework as practice, not work. It takes practice to get better at sports or music or cheer leading School is the same.
6. Ask questions if you're not sure about something. Asking questions is one of the most effective ways we learn!
7. After each study session, try to recall the main points and as many details as possible.





Organizational Skills

1. Use outlines, charts, or flashcards to help you organize and learn new material. You'll be reviewing the material as you make these tools. And, you'll have them to use later when it's time to study for tests.
2. Create a planner to keep track of homework assignments, tests, and projects. Write in your planner every day so it becomes a habit!
3. Organize your notes and homework assignments by subject in separate notebooks and folders.
4. Keep a "To Do" list. Write down things you need to do. Then decide what you need to get done right away and what can wait until later.





Time Management Skills

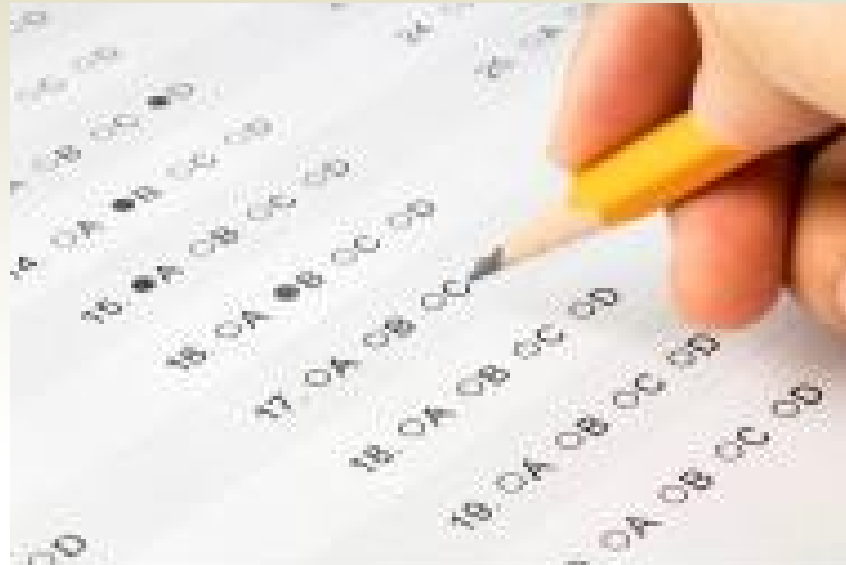
1. Plan ahead and stick to a schedule.
2. Decide what you want to accomplish and how long you will spend on each subject or assignment.
3. Break your workload down into manageable chunks.
4. Don't procrastinate (that's a big word that means putting things off).
5. Be aware of things that distract you or waste your time, and keep them to a minimum.





Test Taking Skills

1. Ask what type of test you'll be taking (essay, multiple choice, true/ false, matching, etc.). It's likely that test questions will be similar to homework you have done.
2. Don't cram. It's OK to spend extra time studying but don't try to learn everything in one night.
3. Get plenty of rest the night before test day.
4. Don't panic. If a question is too hard, skip it and come back to it later.





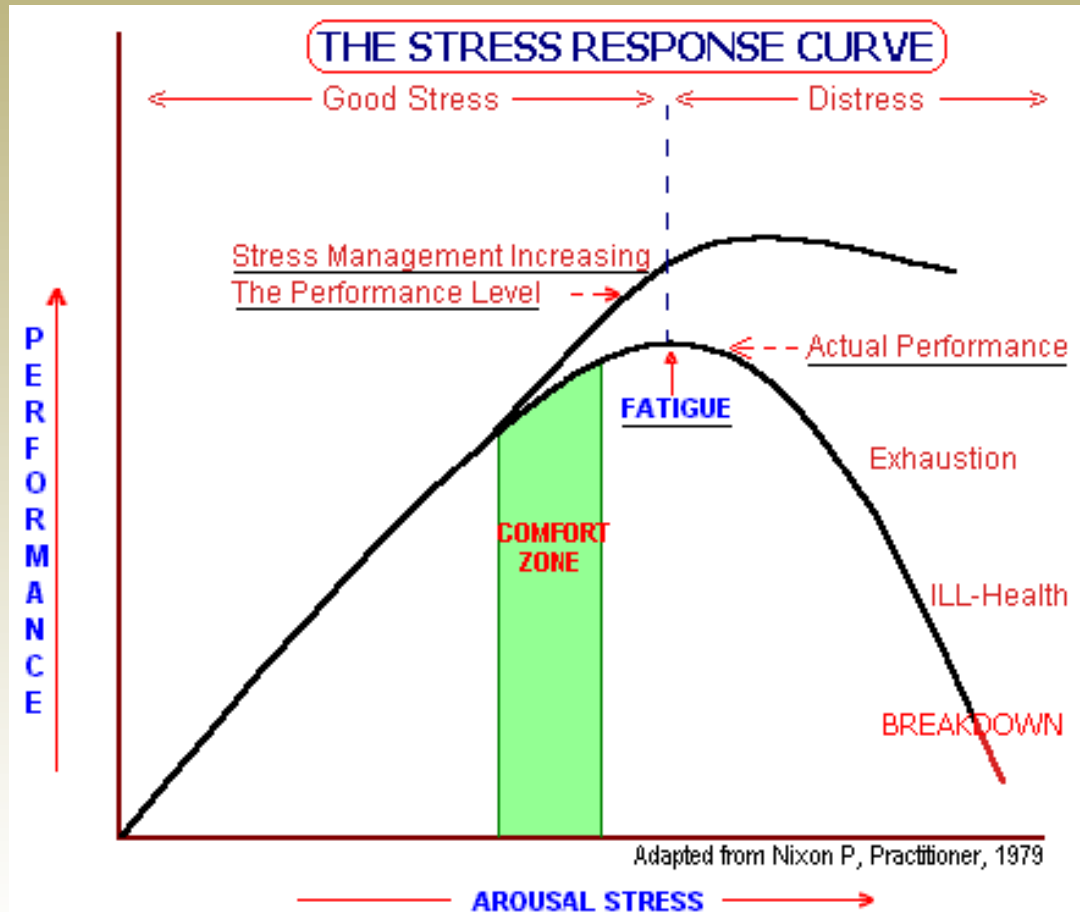
Note Taking Skills

1. Focus on the main ideas. Don't try to write down everything the teacher says.
2. Use your own words.
3. Keep your notes organized. They will be as important as the textbook.
4. Review your notes every day. This will make things easier to remember when it's time to study for the test.



Stress Management Skills

1. Don't sweat the small stuff. Prioritize your activities and focus on the most important ones.
2. Exercise. It takes your mind off things that are bothering you.
3. Take care of yourself. Eat right and get enough sleep.
4. Avoid drugs. They do not reduce stress, they hide it





Take a Moment

If she can. You can. Could you?





What we learn

We will begin the study of the language starting with the fundamentals of the language and simple programs; and as we explore more of the language, we will write increasingly larger programs.

By the end of this year, every student:

- will learn about all the essential programming concepts
- will demonstrate a good familiarity with C++ syntax
- will be able to write reasonably complex procedural programs



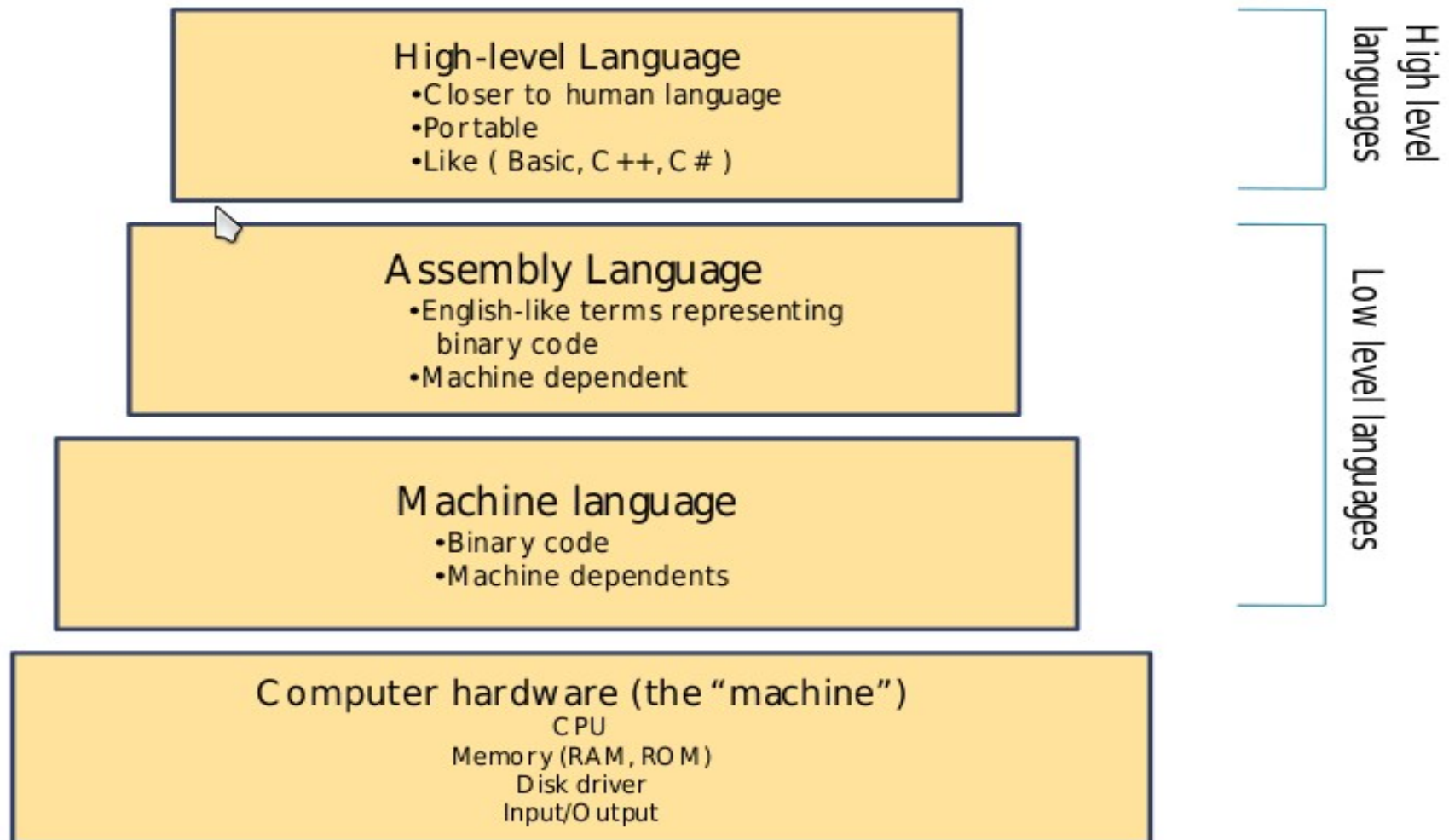


How To Solve Problems

1. Define the problem and see if there is a better way to redefine it.
2. Find the way how human solves it.
3. Write down human's solution
4. Break down this solution to its original steps and do not forget simplest thoughts.
5. match every human activity with computer's tool.
6. If some activity don't exist directly in computer find alternatives.
7. Group these tiny computer actions into logically related functions.
8. Combine these functions in a complete solution.
9. Solution Implementation and Verification



Levels of Programming Languages





Levels of Programming Languages *

High-level language Example

$A = 3 + 2$

Assembly language Example

MOV AL,3
ADD AL,2

Machine language Example

10111000 00000011
00000100 00000010



Levels of Programming Languages *

- Machine Language that the hardware understands because it is written with 1s and 0s.
- At the level above machine language assembly language where the 1s and 0s are represented by English like words.
- Assembly languages are considered low level because they are closely related to machine language and are machine dependent



Levels of Programming Languages *

- Machine dependent mean a given assembly can be used on a specific microprocessor.
- At the level above Assembly language is high level language.
- High-level language is closer to human language further than machine language.
- An advantage of high-level language is portable (Independent).



Introduction to C++

A program is a sequence of instructions that can be executed by a computer. Every program is written in some programming language.

C++ (pronounced “see-plus-plus”) is one of the most powerful programming languages available. It gives the programmer the power to write efficient, structured, object-oriented programs.



Introduction to C++

- To write and run C++ programs, you need to have a text editor and a C++ compiler installed on your computer.
- A text editor is a software system that allows you to create and edit text files on your computer.
- A compiler is a software system that translates programs into the machine language (called binary code) that the computer's operating system can then run.
- That translation process is called compiling the program. A C++ compiler compiles C++ programs into machine language.



C++ Programming Style

Main structure

C++ program start execution at the beginning of the main()function. Since a program can have only one starting point, every C++ language program must contain one and only one main()function.

```
#include <iostream.h>
int main (){
    program statements;
    return 0;
}
```



C++ Programming Style

Comments

Comments are explanatory remarks made within a program. There are two types of comments in C++:

1. Single line comment by using two slashes //
2. Block comment by using slash and astrisc /* */

```
//This is comment
#include <iostream.h>
int main (){
/*This program has
Many comments*/
    program statements;
    return 0;
}
```




C++ Programming Style

Preprocessor Directives

The preprocessor commands are begin with a pound sign (#) and perform some action before the compiler translates the source program into machine code.

```
#include <iostream.h>
```

Variable names

All the variable names should suggest their use. Try to use meaningful names for variables. This is good programming as it makes our programs

- 1.easier to read
- 2.easier to correct, and
- 3.easier to change



C++ Programming Style

Indenting

A program should be laid out so that elements that are naturally considered a group are made to look like a group.

One way to do this is to skip a line between parts that are logically considered separate. indenting can help to make the structure of the program clearer. A statement within a statement should be indented.

Brackets

Also, the brackets `{}` determine a large part of the structure of a program. Placing each on a separate line by itself, makes it easy to find the matching bracket. One pair of brackets is embedded inside another pair, the inner pair should be indented more than the outer pair.



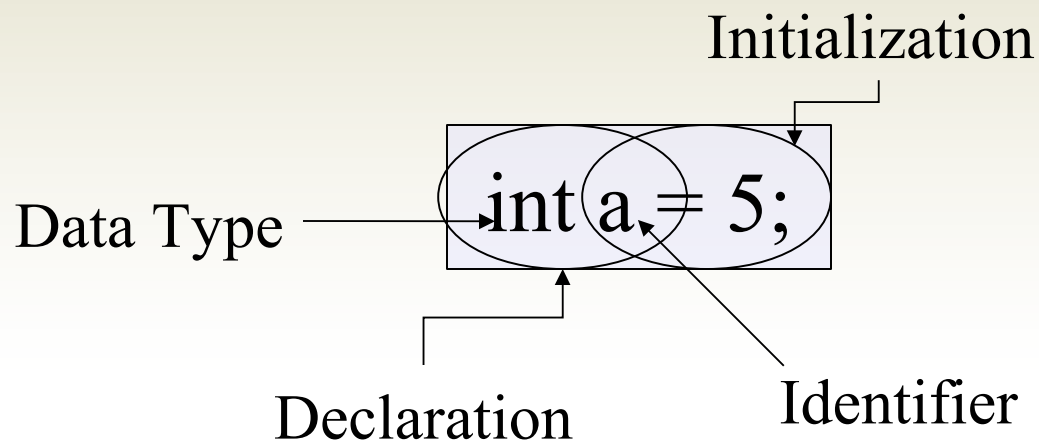
Variables

- In mathematical modeling and computer programming, we introduce variables representing abstract notions and physical objects. Examples are the temperature (Temp), the velocity (V), time (t) , period (T), voltage (V)... etc.
- Program variables correspond to memory spaces reserved for storage.
- Every variable stored in the computer's memory has a name, a value and a type.
- The term Variable is used because the value stored in the variable can change.



Variables *

- In order to use a variable in C++, its specified must first declared which of data types wanted to be.
- It's Better to “give” variables initial values to let the program know what to use as an initial value other wise it will contain a random number and the results will be unexpected





Variable Naming

A variable name is called an *identifier*. An identifier in C++ can be up to 31 characters long. C++ is case sensitive, so count is different than Count

Can not begin with a digit.

(Invalid: 1First)

Can not contain blanks.

(Invalid: num elements)

Can not contain a hyphen, underscore () is OK.

(Invalid: num-elements)

Special symbols are not allowed.

(Invalid: cost\$, cost!)

Reserved words can not be used as identifiers.

(Invalid: int, const, float)



Variable Naming

A reserved word (**keyword**) is a word that has a special meaning in C++. It can not be used as a programmer-defined identifier.

asm	default	float	public	try
auto	delete	for	register	typedef
bool	do	goto	return	typeid
break	double	if	short	typename
case	else	inline	signed	union
catch	enum	int	sizeof	unsigned
char	explicit	Long	static	using
class	export	new	struct	void
const	extern	operator	switch	while
continue	false	Private	true	



Data Types

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.

Type	bytes	Size Range	Precision
bool	1	true or false	N/A
char	1	ASCII codes	N/A
short or short int	2	-32,767 to 32,767	N/A
int	4	-2,147,483,647 to 2,147,483,647	N/A
long or long int	4	-2,147,483,647 to 2,147,483,647	N/A
float	4	10 ⁻³⁸ to 10 ³⁸	7 digits
double	8	-10 ³⁰⁸ to 10 ³⁰⁸	15 digits
long double	10	-10 ⁴⁹³² to 10 ⁴⁹³²	19 digits



Variables

integer Numbers (int)

As we have already mentioned the type int refers to whole numbers like 37 and -45.

```
int num = 55;  
int a = 1, b = -1 , c;  
c = 789;
```

Real Numbers(Double)

Numbers with fractional parts such as 3.56 and 0.112 are of type double or float.(the real numbers in mathematics). Numbers of type double can have a precision of up to 15 digits.

```
double realNum = -12.43;
```



Variables *

Characters (char)

Computers and C++ not only use numerical values but they can also use non-numerical values like characters and strings. Values of type char include letters of the alphabet, digits, symbols and punctuation marks. For example:

```
char letter, symbol;  
letter='A';  
symbol='#';
```

Notice that character values are placed inside single quotes.



Variables *

Boolean expressions (bool)

This type was recently added to the language. Values of type bool are called Boolean expressions and include only two values: true or false. Boolean expressions are used in branching and looping statements.

```
bool test = true, odd = false, even;  
even = false;
```

Important note

As a general rule, you cannot store values of one type in a variable of another type. This is a good rule to follow, even though some C++ compilers do not enforce this type checking.



Constants

Sometimes it is required to have some value unchanged throughout the program execution. Some value that does not change in a C++ program is called a constant. it's common to use capital letters for constants. For example:

```
const double PI = 3.14159265;
```

Or you can use

```
#define PI 3.14159265
```



Arithmetic Operators

Usually, arithmetic operators come in between two expressions as in

expression1 operator expression2

Or operand1 operator operand2

The operand can be either a single variable/number or a composite expression. The main C++ arithmetic operators are the following:

- + for addition
- for subtraction
- * for multiplication
- / for division
- % for division remainder (Mod)



Arithmetic Operators *

For example:

```
int computers=10;  
int price=7340;  
int total=price * computers;
```

operand1 operator operand2

All of the arithmetic operators can be used with numbers of type int, numbers of type double and even with one number of each type.

Division Operator (/)

However, if both operands are of type int, the result is of type int. if one, or both operands are of type double, the result will be of type double.



Arithmetic Operators *

When used with one or both operands of type double, the division Operator, /, behaves as you would expect: $10/4.0=2.5$

But when used with two operands of type int, the division operator / gives only the integer part of the division: $10/3$ is 3 and not 3.333...

Remainder Operator (%)

The % operator can be used with operands of type int to recover the information lost when you use / to do division with operands of type int; for example:

11 divided by 3 is $\rightarrow 11/3 == 3$

with a remainder of $\rightarrow 11\%3 == 2$



Assignment Statements

Values can be assigned or stored in variables with assignment statements:

```
books=34;
```

An assignment statement is an order, to the computer, to assign the value on the right-hand side of the equal sign to the variable on the left-hand side. The sign (=) is called the assignment operator. The value on the right-hand side can also be another variable or expression:

```
books1=books2;
```

in an assignment statement, first the value on the right-hand side is evaluated and then its result is stored or assigned to the variable on the left-hand side.



Assignment Statements *

C++ has shorthand notation that combines the assignment operator (=) and an arithmetic operator. For example:

```
int hours=5;  
hours += 7; //is equivalent to    hours=hours + 7;
```

That is, the new value of the variable hours is equal to its old value plus the number constant 7.

We can use other arithmetic operators too:

```
hours -=2;    hours=hours-2;  
hours /=3;    hours=hours/3;  
hours *=2;    hours=hours*2;  
hours %=8;    hours=hours%8;
```



Precedence Rules

You can specify the order of operations in C++ using parentheses as illustrated in the following expressions:

1: $(x + y) * z$

2: $x + (y * z)$

1: the computer first adds x to y and then multiplies the result by z .

2: the computer first multiplies y by z and then adds the result to x .

If you omit the parentheses, the computer will follow the C++ rules of precedence. So, if you wrote $x + y * z$, the computer would multiply y by z and add the result to x . Because $*$ has higher precedence than $+$ (the same is true for $/$ and $\%$)



Precedence Rules *

The following list contains the precedence rules for some C++ operators:

`()`, `[]`, `..`, `->`, `(postfix)++`, `(postfix)--`
unary `+(unary)`, `-(unary)`, `++(prefix)`, `--(prefix)`, `!`, `sizeof`
binary arithmetic `*`, `/`, `%`
binary arithmetic `+`, `-`
Boolean operators `<`, `>`, `<=`, `>=`
Boolean operators `==`, `!=`
Boolean operator `&&`
Boolean operator `||`
assignment `=`, `+=`, `-=`, `*=`, `/=`, `%=`

For example, consider `(x+1) > 2 || (x+1) < -3`,
this is equivalent to `((x+1) > 2) || ((x+1) < -3)`
because `<` and `>` have higher precedence than `||`.



Performing Output

The values of variables, numerical values and strings of text ,may be output to the screen using cout as in

```
int books=0;  
cout<<books<<endl;  
cout<<72<<endl;  
cout<<"This is the output"<<endl;
```

The double arrow signs (<<) are called the insertion operators.



input using cin

In C++, cin is used to input values into variables. After declaring a variable of type int,

```
int price;  
cout<<"Enter the price:";  
cin>>price;  
cout<<"The price you entered is $"<<price<<endl;
```

in this program extract, first an integer is declared, a message is also output to the screen notifying the user of the input value.

When the program reaches a cin statement, it waits for input to be entered from the keyboard and for this value to be input into the Variable, the user must enter a new line.



Flow of Control

in the simple C++ program the program consisted of a list of program statements which were executed sequentially; one statement after another.

For bigger and more sophisticated programs, you will need some way to vary the order in which statements are executed.

The order in which statements are executed is called the flow of control.

C++ has a number of mechanisms which let you control the flow of program execution.

First, we will study a branching mechanism that allows you to choose between two alternative actions. Then we will discuss loops.



Branching (if-else statement)

There is a C++ statement that chooses between two alternative actions. it is called the if-else statement. The general form of the if-else-statement is as follows:

```
if (Boolean-expression)
    yes-statement
else
    no-statement
```

When program execution reaches the if-else statement only one of the two statements is executed. if the Boolean-expression is true then the yes-statement is executed. if the the Boolean-expression is false then the no-statement is executed.



Branching (if-else statement) *

Example:

Suppose your program asks the user about the amount of time per week he/she spends on practicing C++. And you want the program to decide whether or not this is enough.

```
#include <iostream.h>
main(){
    int hours_per_week=0;
    cout<<"How many hours/week do you practice C++? "<<endl;
    cin>> hours_per_week;
    if (hours_per_week>=4)
        cout<<"That's good "<<endl;
    else
        cout<<"That's not good enough"<<endl;
    return 0;
}
```



Branching (if-else statement) *

Sometime, you want your program to test a condition and if the condition is satisfied the program does something, otherwise it does not do anything. You can do this by omitting the else part from the if-else-statement. For example:

```
if (grade >= 50)
    cout << "The student has passed." << endl;

cout << "....." << endl;
```

This is called the if-statement



Branching (if-else statement) *

You may want to execute more than one statement inside an if-else-statement. To do this, enclose the statements inside brackets { }. For example:

```
if (grade >=50){  
    cout<<" The student has passed"<<endl;  
    passed-students +=1;  
}  
else{  
    cout<<"Student has failed"<<endl;  
    failed-students +=1;  
}
```

A list of statements enclosed inside brackets is called a compound statement.



Branching (if-else statement) *

The Boolean expression in the if-else-statement is:
`hours_per_week >= 4`

Remember that Boolean expressions or variables have only 2 values: true and false.

Here, we use C++ comparison operators. Here is the full list of comparison operators:

Math Symbol	C++ Notation
=	==
≠	!=
<	<
≤	<=
>	>
≥	>=



Boolean Logic

in evaluating Boolean expressions, C++ uses Boolean Logic principles.

You can combine two (or more) comparisons using the Boolean Logic “and” operator. For example, the expression

$(x > 2) \ \&\& \ (x < 10)$

is true only if both $(x > 2)$ and $(x < 10)$ Boolean expressions are true.

X	Y	X Y	X	Y	X && Y
True	True	True	True	True	True
True	False	True	True	False	False
False	True	True	False	True	False
False	False	False	False	False	False

Note: $\&\&$ is equivalent to “AND” and $||$ to “OR”.



Multi-way if-else statements

An if-else statement is a Multi-way branch. it allows a program can choose between more than two alternative actions.

You can do this by nesting if-else statements. For example, suppose we want to write a game-playing program in which the user must guess the value of some number.

```
cout<<"Guess the number: ";  
cin>>guess;  
if (guess > number)  
    cout<<"No, too high";  
else if (guess == number)  
    cout<<"Correct!"<<endl;  
else  
    cout<<"No, too low"<<endl;
```




Switch-statement

Let's look at an example involving a switch-statement:

```
char grade;
cout<<"Enter your grade: ";
cin>>grade;
switch (grade) {
    case 'A':
        cout<<"Excellent."<<endl;
        break;
    case 'B':
        cout<<"Very good."<<endl;
        break;
    case 'C':
        cout<<"Passing"<<endl;
        break;
    case 'D': case 'E':
        cout<<"Too bad, go study"<<endl;
        break;
    default:
        cout<<"This is not a possible grade"<<endl;
```

```
}
```



Switch-statement *

Notice that the constant is followed by a colon. Also note that you cannot have two occurrences of case with the same constant value after them since that would be an ambiguous instruction.

A break-statement consists of the keyword break followed by a semicolon. When the computer executes the statements after a case label, it continues until it reaches a break-statement and this is when the switch-statement ends.

if you omit the break-statement, then after executing the code for one case, it goes on to execute the code for the following case.



Loops

Most programs have some action that is repeated a number of times.

A section of a program repeats a statement or group of statements is called a loop. C++ has a number of ways to create loops. One of them is called a while-statement or while-loop.

```
main() {  
    int num_of_greetings=0;  
    cout<<"How many greetings do you want? " ;  
    cin>>num_of_greetings;  
  
    while (num_of_greetings > 0){  
        cout<<"Hello " ;  
        num_of_greetings=num_of_greetings-1;  
    }  
    return 0;  
}
```



Loops *

while-loop might execute its loop body zero times. if you know that under all circumstances your loop body should be executed at least once, then you can use a do-while loop statement.

The do-while loop statement is similar to the while-loop statement, except that the loop body is executed at least once. The syntax of do-while loop statements is as follows:

```
do {  
    Statement_1;  
    Statement_2;  
    ...  
    Statement_n;  
} while (Boolean-expression);
```

Loop body is executed once first, then the Boolean expression is checked for additional iterations of the loop body.



Loops *

An example involving a do-while loop statement:

```
main(){
    char answer='n';

    do {
        cout<<"Hello\n";
        cout<<"Do you want another greeting?\n"
            <<"Press y for yes, n for no,\n"
            <<"and then press Enter/Return: ";
        cin>>answer;
    } while (answer == 'y' || answer == 'n');

    cout<<"Goodbye\n";
}
```



Infinite Loops

Write positive even numbers less than 12:

```
int x=2;
while ( x != 12){
    cout<<x<<endl;
    x=x+2;
}
```

Write positive odd numbers less than 12:

```
int x=1;
while ( x != 12){
    cout<<x<<endl;
    x=x+2;
}
```

Which is an infinite loop?

To terminate a program use control-C or Ctrl-C on the keyboard.



for - Statement

In performing numeric calculations, it is common to do a calculation with the number one, then with the number two and so forth until some last value is reached.

For example to add one through ten you want the computer to perform the following statement ten times with the value of n equal to 1 the first time and with n increased by one each subsequent time.

$$\text{sum} = \text{sum} + n$$



for – Statement *

The following is one way to accomplish this with a while statement:

```
sum=0;
n=1;
while (n<=10){
    sum=sum + n;
    n++;
}
```

Although a while-loop is OK here, this kind of situation is just what the for-loop was designed for. The following for-loop will neatly accomplish the same thing:

```
sum=0;
for ( n=1; n <= 10; n++)
    sum=sum + n;
```




for – Statement *

An example involving a for-loop:

```
#include <iostream.h>
```

```
main()
```

```
{  
    ← initialization      repeat the loop  
    int sum=0;           as long as this is true
```

```
    for ( int n=1; n <= 10; n++){  
        sum = sum + n; ← done after each  
    }                    loop body iteration
```

```
    cout<<"The sum of the numbers 1 to 10 is :  
    "<<sum<<endl;  
    return 0;  
}
```



break Statement

break-statement can be used to exit a loop. Sometimes you want to exit a loop before it ends in the normal way.

For example, the loop might contain a check for improper input and if some improper input is encountered then you may want to end the loop. To input a list of negative numbers and exit the loop if positive numbers are input:

```
int count=0, sum=0, number=0;
while (++count <= 10) {
    cin>>number;        //input number
    if (number >= 0)
        break;          //exit loop
    sum = sum + number;
}
```



continue Statement

We saw on the previous slide how to use the break-statement to exit from a loop or from a switch-statement case. There is another control statement that you can use in your programs: the continue statement.

The continue statement causes the current iteration of a loop to stop and the next iteration, if there is one, to begin. For example consider the following code fragment:

```
while (true) {  
    cin>>letter;  
    if ( letter== ' ' ) continue;  
    ...  
}
```



Problem Solving

Engineers use their knowledge of science, mathematics, and appropriate experience to find suitable solutions to a problem. Engineering is considered a branch of applied mathematics and science. Creating an appropriate mathematical model of a problem allows them to analyze it (sometimes definitively), and to test potential solutions.

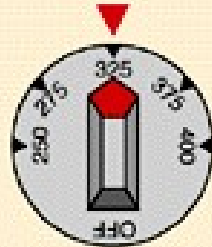
Usually multiple reasonable solutions exist, so engineers must evaluate the different design choices on their merits and choose the solution that best meets their requirements.





An Algorithm: Baking a Cake

Heat oven to 325°F



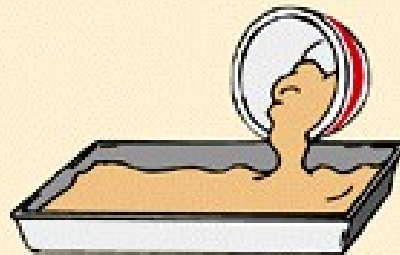
Gather the ingredients



Mix ingredients thoroughly in a bowl



Pour the mixture into a baking pan





An Algorithm: Baking a Cake *

Bake in the oven
50 minutes

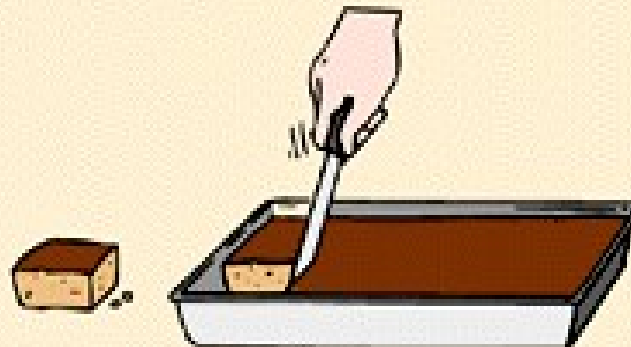


Repeat

Bake 5 minutes more

Until cake top springs back when touched in the center

Cool on a rack before cutting


















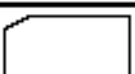
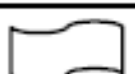

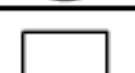
Flowchart

A **flowchart** is a common type of diagram, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows.















Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields

Flowchart Symbol	Name (Alternates)	Description
	Process	An operation or action step.
	Terminator	A start or stop point in a process.
	Decision	A question or branch in the process.
	Delay	A waiting period.
	Predefined Process	A formally defined sub-process.
	Alternate Process	An alternate to the normal process step.
	Data (I/O)	Indicates data inputs and outputs to and from a process.

Flowchart (cont)

	Document	A document or report.
	Multi-Document	Same as Document, except, well, multiple documents.
	Preparation	A preparation or set-up process step.
	Display	A machine display.
	Manual Input	Manually input into a system.
	Manual Operation	A process step that isn't automated.
	Card	A old computer punch card.
	Punched Tape	An old computer punched tape input.
	Connector	A jump from one point to another.
	Off-Page Connector	Continuation onto another page.

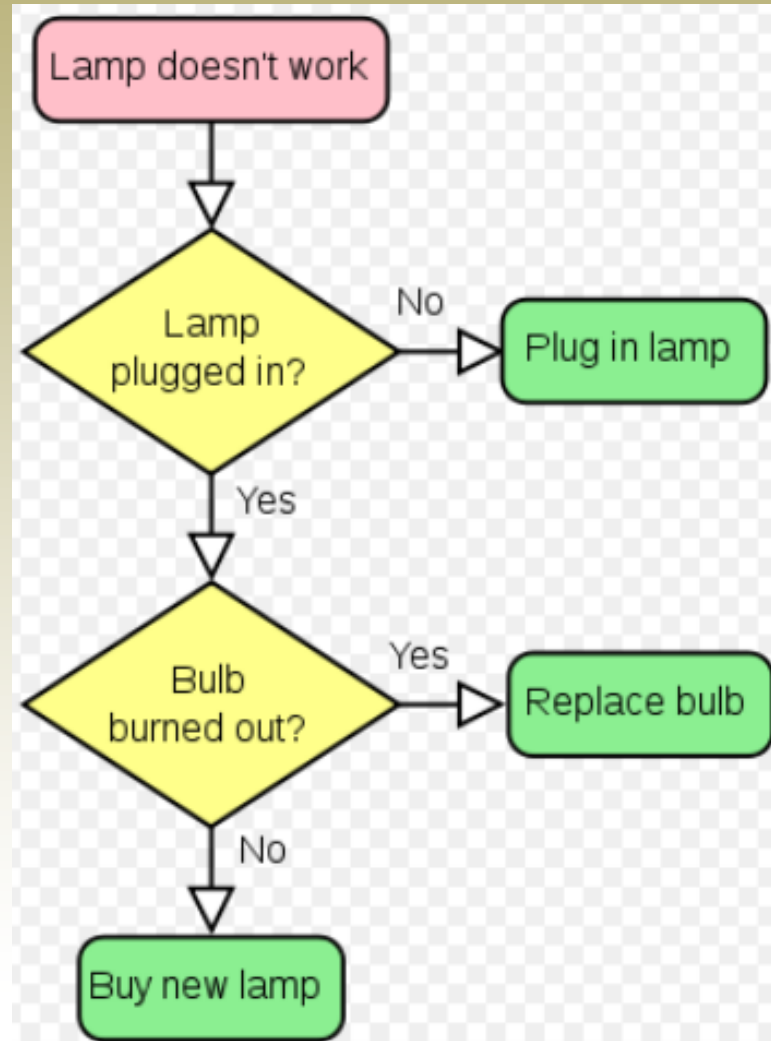
Flowchart (cont)

	Transfer	Transfer of materials.
	Or	Logical OR
	Summing Junction	Logical AND
	Collate	Organizing data into a standard format or arrangement.
	Sort	Sorting of data into some pre-defined order.
	Merge (Storage)	Merge multiple processes into one. Also used to show raw material storage.
	Extract (Measurement) (Finished Goods)	Extract (split processes) or more commonly - a measurement or finished goods.
	Stored Data	A general data storage flowchart symbol.
	Magnetic Disk (Database)	A database.
	Direct Access Storage	Storage on a hard drive.
	Internal Storage	Data stored in memory.
	Sequential Access Storage (Magnetic Tape)	An old reel of tape.
	Callout	One of many callout symbols used to add comments to a flowchart
	Flow Line	Indicates the direction of flow for materials and/or information



Flowchart (cont)

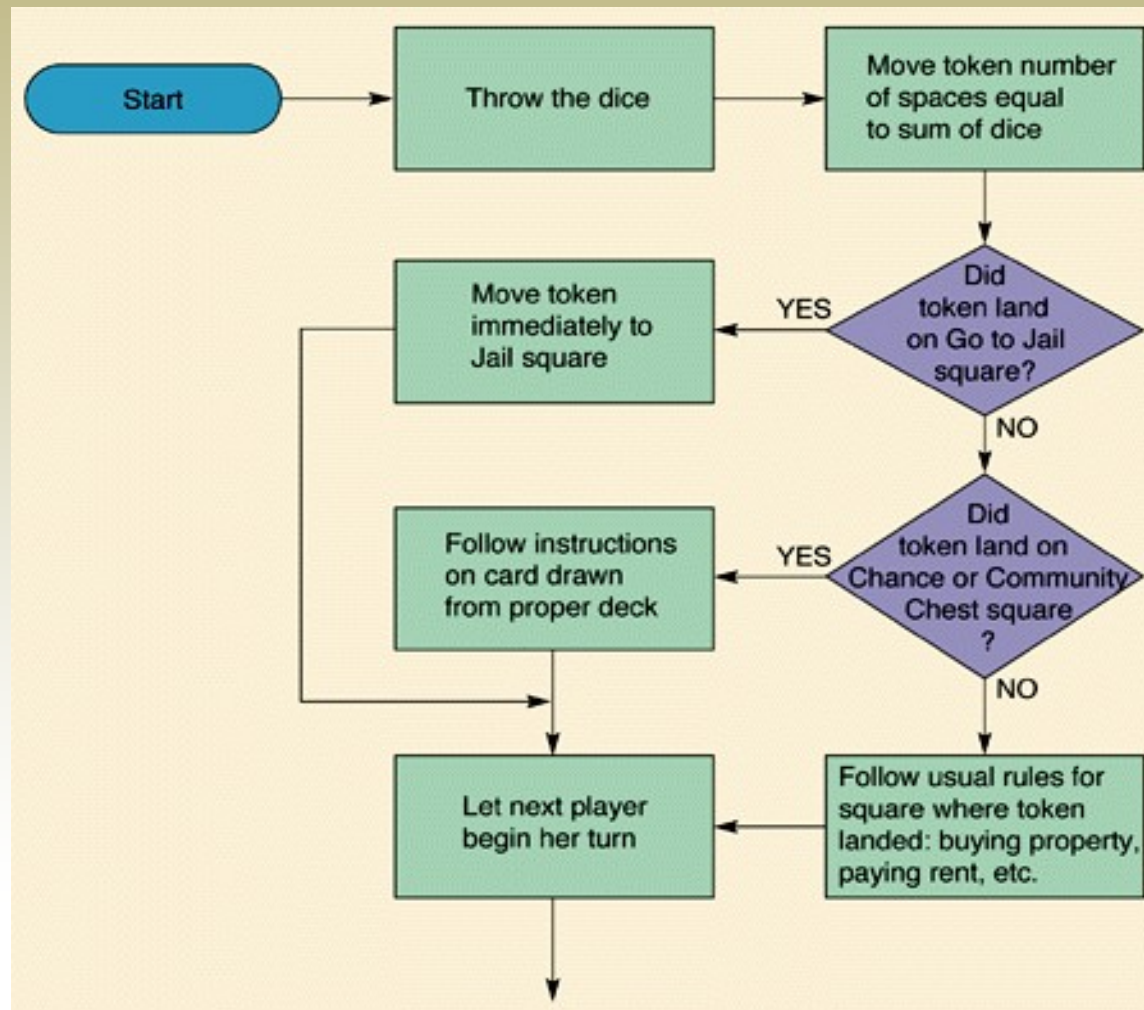
Lamp Fixing Flowchart.





Flowchart (cont)

Game of Monopoly Flowchart.





Pseudocode

This is the pseudocode for a Game of Monopoly, including one person's move as a procedure:

Main Procedure Monopoly_Game

Hand out each player's initial money.

Decide which player goes first.

Repeat

 Call Procedure Monopoly_Move for next player.

 Decide if this player must drop out.

Until all players except one have dropped out.

Declare the surviving player to be the winner.

Procedure Monopoly_Move

Begin one's move.

Throw the dice.

Move the number of spaces on the board shown on the dice.

If the token landed on "Go to Jail,"

 then go there immediately.

Else if the token landed on "Chance" or "Community Chest,"

 then draw a card and follow its instructions.

Else

 follow the usual rules for the square (buying property, paying rent, collecting \$200 for passing "Go", etc.).

End one's move.



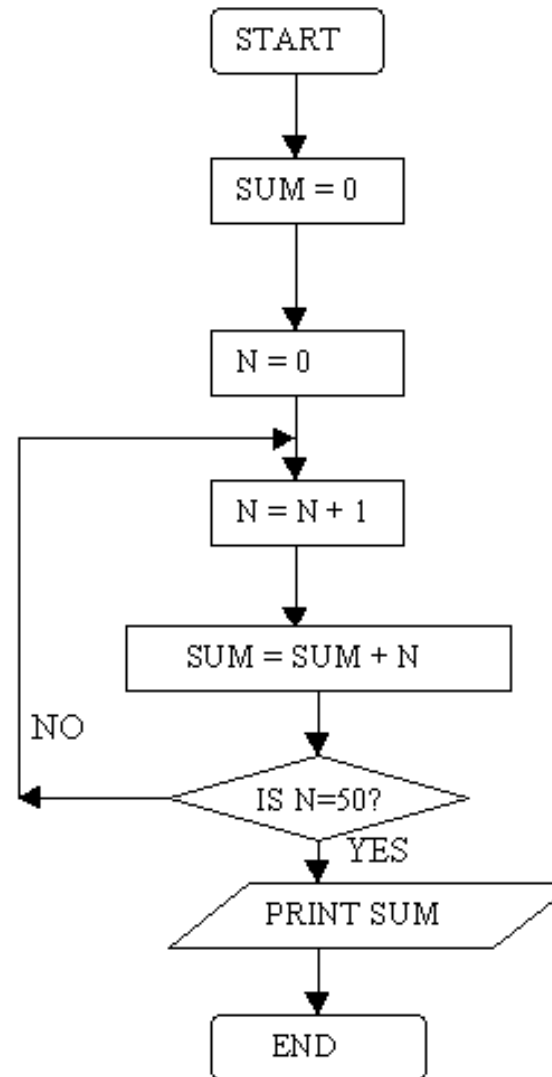
Exercise

1. Find a solution for this equation $x^2 + x - 9 = 18$. using pseudocode and flowchart.
2. How can you replace a broken door.
3. Write a pseudocode that illustrate kids going to the school steps.
4. Draw a flowchart for buying a house.



Flowchart Example #1

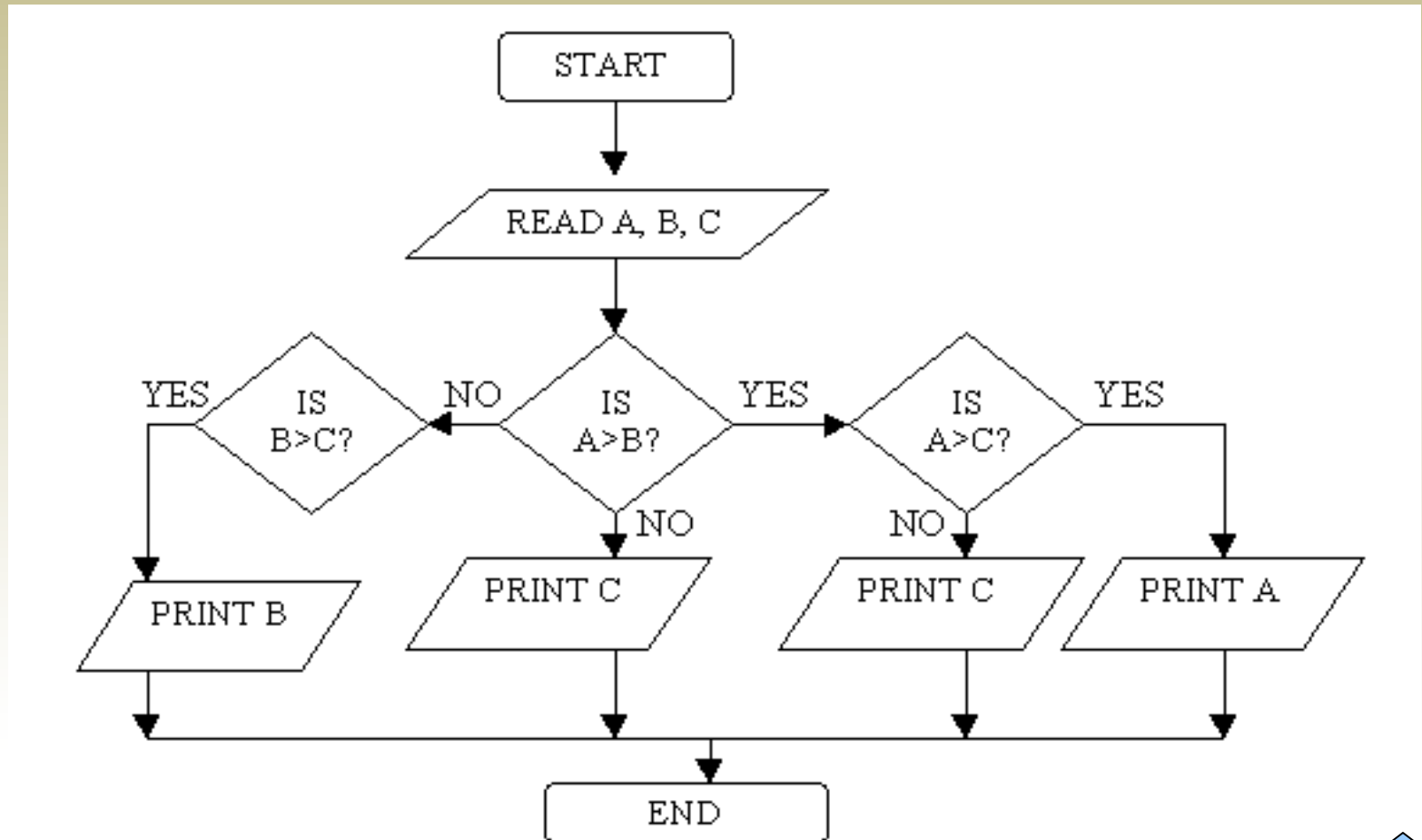
Draw a flowchart to find the sum of first 50 natural numbers





Flowchart Example #2

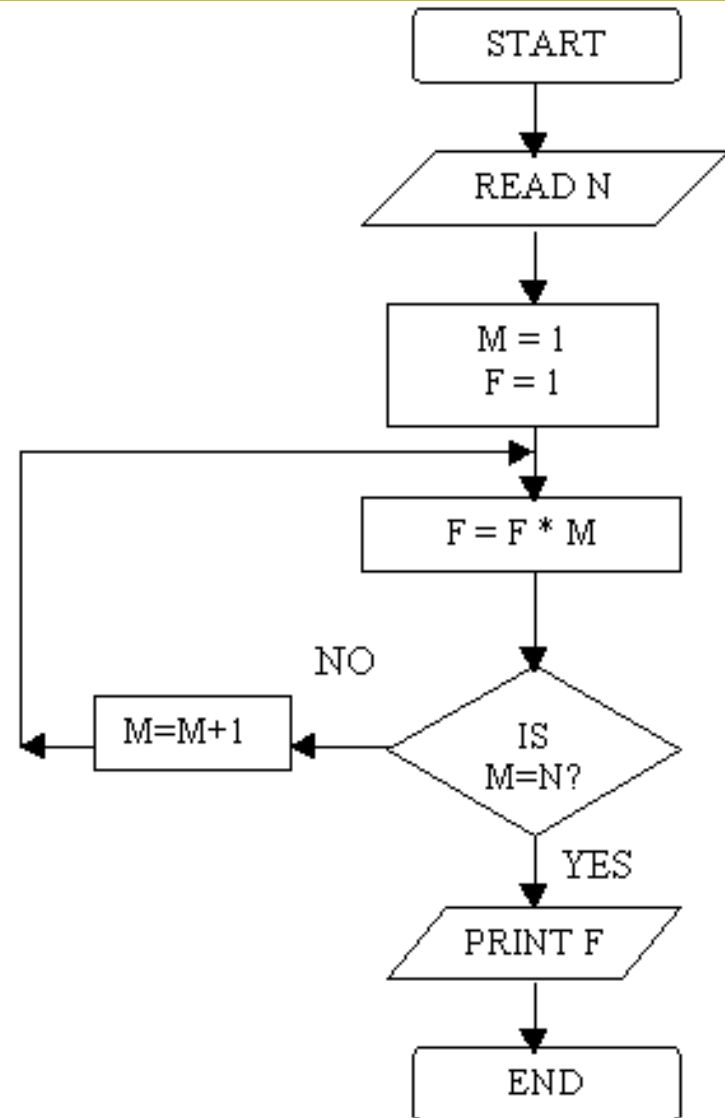
Draw a flowchart to find the largest of three numbers A, B, and C.





Flowchart Example #3

Draw a flowchart for computing factorial N ($N!$)





Arrays

An array is a collection of variables all of the same data type that are referred to by a common name. A specific element in an array is accessed by an index.

in C++ the array elements are stored in contiguous memory locations.

The lowest address refers to the first element and the highest address refers to the last element.

For example, to store a list of exam marks for students we can use an array as in:

```
int mark[5];
```

This array is of type integer and it can hold 5 variables of type integer. 'mark' is the name of this array.



Arrays *

The array declaration (similar to variable declaration) on the previous slide is like the following declarations:

```
int mark1, mark2, ..., mark5;
```

You can see that the array notation is clearer and more elegant. The array

```
int mark[5];
```

declares an array of type integer that can store 5 variables all of type integer.

Array elements are numbered from 0. That is, the index of the first Element in the array is 0 and the index of the last element is one less than the size of the array. (in this example, first element has index 0 and last element has index 4)



Arrays *

You can initialize arrays in this way:

```
int mark[5] = { 87, 67, 90, 89, 100};
```

The size of this array is 5. The size of the array need not be declared if it can be inferred from the values in the initialization:

```
int mark[ ] = { 87, 67, 90, 89, 100};
```

To access array elements we use the array name plus the index of the required element. For example to output the second element of this array:

```
cout<< mark[1]<<endl;
```

and the last element:

```
cout<< mark[4];
```



Arrays *

```
#include <iostream.h>
main(){
    int number[5];

    for (int i=0; i<5; i++){
        number[i]=i;    //initialize the array
        cout<<i;
        cout<<"\t"<<(number[i] * number[i])<<endl;
    }
    return 0;
}
```

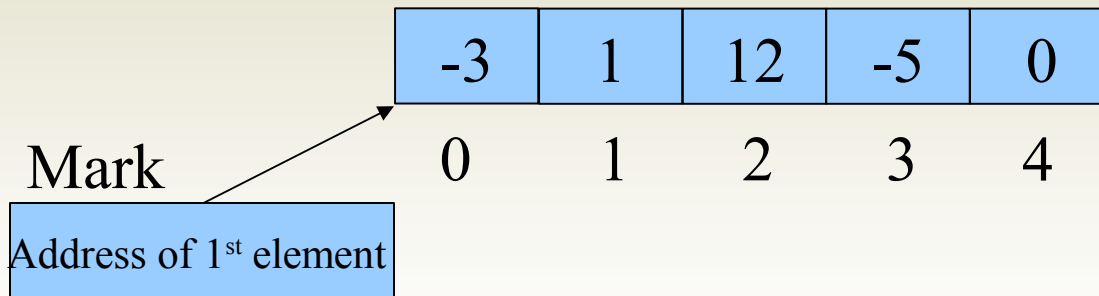


Arrays *

Array indexed variables are stored in memory in the same way as ordinary variables are stored. But with arrays, the locations of the Array indexed variables are always placed next to one another in the computer's memory.

For example consider the following array declaration:

```
int mark[5];
```





Arrays *

The most common programming error made when using arrays is attempting to reference a non-existent array index. For example consider the following array declaration:

```
int mark[5];
```

For this array, every index must evaluate to one of the integers between 0 and 4. if you write:

```
cout<<mark[i]<<endl;
```

Then *i* must evaluate to one of: 0, 1, 2, 3, 4. if it evaluates to anything else it is an error. This is an out of range error.

Be especially careful when using for-loops to manipulate arrays.



Strings

C++ can manipulate text strings in two ways: using `cstring` values used in C language and with the new `string` type that was recently added to the language. We will cover both methods in these slides.

A `cstring` variable is exactly the same thing as an array of characters. For example, the following array of characters is capable of storing a `cstring` value with 9 or fewer characters:

```
char name[10];
```

This is because a `cstring` variable is used in a different way than an ordinary array of characters. A `cstring` variable places the special symbol `'\0'` in the array immediately after the last character of the `cstring`.



Strings *

For example, consider the following cstring variable:

```
char name[10] = "MR Nice";  
name[0]           name[9]
```

The character `'\0'` is used to mark the end of the cstring. Since the character `'\0'` always occupies one element of the array, the length of the longest cstring that the array can hold is one less than the size of the array.

The character `'\0'` is called the null character. You can do input and output with cstring variables as you do with other types of variables.



Strings *

You cannot use a cstring variable in an assignment statement using `=`. Also, if you use `==` to test cstrings for equality, you will not get the result you expect. The reason is that cstrings and cstring variables are arrays rather than simple variables and values.

So you **CANNOT** do the following:

```
char name[10];  
name="Mr Nice";
```

This is illegal in the homeland of C++!. You can only do this when you declare the cstring variable as in:

```
char name[10]="Mr Nice";
```

Technically, this is an initialization and not an assignment and is legal.



Strings *

There are a number of ways to assign a value to a cstring variable. The easiest way is to use the predefined function `strcpy` as in:

```
strcpy(name, "Mr Nice");
```

This will assign (assignment) the value "Mr Nice" to character array `name`.

Also, you **CANNOT** use the operator `==` in an expression to test if two cstrings are equal. You can use the function `strcmp` as in:

```
char name1[]="Mr A", name2[]="Mr B";
```

```
if (strcmp(name1, name2)
```

```
    cout<<"The two names are NOT the same"<<endl;
```

```
else
```

```
    cout<<"The two names are the same"<<endl;
```



Strings *

Note that the function `strcmp` works differently than you might expect. if the function returns a zero it means that the two cstrings are the same (equal).

The functions `strcpy` and `strcmp` are defined in the library file `string.h` so you must include this library in your C++ programs to be able to use the functions.

Another useful function in this library is called `strlen` which returns the length of the specified cstring. ('\0' is not counted) For example:

```
int length=0;
char name[]="Software Engineering";
length=strlen(name);
cout<<length<<endl;    //will output ?
```



Strings Input and Output

As we have already seen, `cstring` output is easy; we just use the insertion operator `<<` to do output.

Also you can use the input operator `>>` to fill a `cstring` variable, but there is one thing to be aware of. All spaces are skipped when `cstrings` are read this way. For eg.

```
char a_cstring[20];  
cout<<"Enter some input:\n";  
cin>>a_cstring;  
cout<<a_string<<endl;
```

The output:

```
Enter some input:  
Mr Nice  
Mr
```



Strings Input and Output *

So there is a problem with `cstring` input when input contains spaces or tabs. The solution to this problem is to use the predefined function `getline` which is also defined in the `string.h` library.

Let's see an example:

```
char a_cstring[30];  
cout<<"Enter some text:\n";  
cin.getline(a_cstring, 30);  
cout<<a_cstring<<endl;
```

Here, the inputted text or string is copied into the `cstring` variable `a_cstring`. The number 30 specifies the number of characters to be copied into the variable `a_cstring`. (we will discuss functions after finishing this topic)



Functions and Procedural Abstraction

A natural way to solve large problems is to break them down into a series of smaller sub-problems, which can be solved more-or-less independently and then combined to arrive at a complete solution.

In programming too, you can follow a similar approach and divide large programs into smaller sub-programs; in C++ sub-programs are called functions.

Most programs consist of 3 main sections: input, some calculations and output. We can perform each of these sections separately and combine the results to produce the final complete program.

In larger programs, it's almost impossible or very difficult to do anything without dividing the program using functions. This follows the old Roman philosophy of divide-and-conquer.



Functions *

We have already seen functions such as `sqrt(...)` to get the square root of a number. Or the `strcpy(s1, s2)` which copies one string to another.

These are pre-defined functions that we can use in our programs. Somebody else has defined them; we just use them and we even don't need to know how they are defined as long as we know how to use them.

The function `sqrt(...)` is defined in a file that we can access via the C++ library '`math.h`'. And the function `strcpy(s1, s2)` is defined in a file which we can access via the library '`string.h`'.

You can have user-defined functions too. You can define your own functions to do specific tasks.



Functions *

In the next few slides we will try to write our own functions. At first we will put these functions in the same file as "main". Later we will see how to put them in separate files. Example:

```
#include <iostream.h>
int area(int length, int width);    //function declaration

main()
{
    int this_length, this_width, rectangle_area;
    cout<<"Enter the length followed by width:";
    cin>>this_length>>this_width;
    rectangle_area=area(this_length, this_width);    //function call
    cout<<"The rectangle area is "<<rectangle_area"<<endl;
    return 0;
}

int area(int length, int width)        //start of function definition
{
    int number;
    number= length * width;
    return number;                    //function returning a value
}
```




Functions *

We will now look at this program closely to see how functions work:

The structure of a function is similar to the structure of `"main"` with its own list of variable declarations and statements

A function may have a list of zero or more parameters inside its brackets, each of which has a separate type.

A function must be declared before it can be used or called.
Functions are declared just before the 'main' function begins.

Function declarations are a bit like variable declarations; they specify which type the function will return.

You can define as many functions as you require provided you declare them first.



Functions *

The function `'area(..., ...)'` returns a value of type `int`. And it takes two parameters. A parameter is variable; during a function call this parameter is replaced with a value.

In the this function, we have two parameters, both of type `int`. When we call the function `area(..., ...)` we pass two variables, `this-length` and `this-width`, to the function. The function `area(..., ...)` then does some action on these variables and at the end, returns some value of type `int`.

The parameters in the above function are called **value parameters**. When the function is called in the main function, it is passed the current values of the variables `this_length` and `this_width`. The function then stores these values in its own local variables and uses its own local copies in its subsequent computation.



Type Casting

Remember that $9/2$ is integer division, and evaluates to 4, not 4.5. If you want division to produce an answer of type double, then at least one of the two numbers must be of type double. Ex, $9/2.0$ returns 4.5.

We can do this because we had constants and we added a decimal point and a zero to one or both numbers. BUT if both the operands in the division are variables, not constants, then we would have a problem.

In C++ you can tell the computer to convert a value of one type to a value of another type:

```
double(9) / 2
```

produces 4.5 because the type double can also be used as a pre-defined function. Another ex, `double(2)` evaluates to 2.0.

This is called **type casting**.



Local Variables

As we have already seen in some of the functions that we have defined, you can have variables declared inside those functions.

These variables exist only when you call the function to which they belong. Variables declared inside functions are called **local variables**.

The **scope** of a local variable is the function inside which that variable is declared. It doesn't exist outside that function.

If you have a local variable in a function, you can have another variable with the same name that is declared in the main function or in another function and these will be different variables.

Remember that main is also a function; but a special one. Every C++ program must have the main function.



Global Constants and Variables

In general, constants are declared outside any functions, even outside the main function. This is good programming as it is usually the case that more than one function uses the same constant.

Constants are therefore usually declared as a group and just after any `#include` directives. Hence the name **global constant**.

Also, you can declare variables outside any function definitions. These are called **global variables**. The scope of global variables is the entire program, unlike local variables whose scope is limited to a particular function.

However, there is seldom any need to use global variables. Also, global variables make a program harder to understand and maintain. So we will not use global variables unless in exceptional cases.



Void Function

The functions that we have seen so far all returned a single value. In C++ a function must either return a single value or return no values at all. A function that returns no value is called a **void function**. Ex.

```
void myFunction(int num);  
{  
    num++;  
    cout<<" one plus your number is "<<num<<endl;  
}
```

As you can see, this function returns no values; it has no return statements. You call void functions like other C++ statements as in:

```
main()  
{  
    int x=0;  
    myFunction(x);    //function call  
    return 0;  
}
```



Call by Reference Parameters

Will the function in the following program swap values of x1 and x2?

```
main()
{
    int x1=5, x2=10;
    swap(x1, x2);
    cout<<"now x1 is "<<x1<<" and x2 is "<<x2<<endl;
    return 0;
}

void swap(int num1, int num2)
{
    int temp=num1;
    num1=num2;
    num2=temp;
}
```

No, it will not. Because here, x1 and x2 have been passed to swap by value. Copies of x1 and x2 are made and passed to swap and any changes to their values, inside the function, occur on the copies of the two variables.



Call by Reference Parameters *

The parameters x1 and x2 are call-by-value parameters. The value of the parameters is passed to the function not the variables themselves.

To ensure that the actual variables are passed to the function C++ supports call-by-reference parameters. This way, the address or the actual variables are passed to the function. To correct the `swap` function we need to use reference parameters as in:

```
void swap(int& num1, int& num2)           //call-by-reference parameters
{
    int temp=num1;
    num1=num2;
    num2=temp;
}
```

Notice that you need to append the ampersand sign & to the name.



Sorting Arrays

One of the most common programming tasks is sorting a list of values from highest to lowest or vice versa or a list of words into alphabetical order.

There are many sorting algorithms; some are easy to understand but not so efficient while some are efficient but hard to understand. One of the easiest sorting algorithms is called **selection sort**.

The selection sort algorithm works as follows:

```
for( int index=0; index<ARRAY_SIZE; index++)  
    Place the indexth smallest element in a[index]
```

Where *a* is an array and array-size is the declared size of the array. algorithms are usually expressed in pseudo-language.



Sorting Arrays *

The algorithm iterates through all the elements of the array one by one and at each iteration it places the smallest number in the array in the next suitable position.

Now we will implement this algorithm description in C++. We need functions to do the following:

- +To find the smallest number in the array
- +To swap two values
- +To sort the array

We will now implement each of these functions separately and then write a `main` function to test the functions.

(We have already implemented the `swap` function, see slide 112)



Sorting Arrays *

The sort function can be implemented as follows:

```
void sort(int array[])
{
    int index_of_next_smallest = 0;
    for(int index=0; index<ARRAY_SIZE-1; index++)
    {
        index_of_next_smallest=index_of_smallest(array, index);
        swap_values(array[index], array[index_of_next_smallest]);
    }
}
```

Notice that the last iteration is redundant ($ARRAY_SIZE - 1$ iterations)

We call another function to find the index of the next smallest value, and then call yet another function to swap that value with the value of the current position in the array.



Sorting Arrays *

Now we will implement the function which will find the index of the next smallest element of the array:

```
int index_of_smallest((int array[], int start_index)
{
    int min=a[start_index];
    int index_of_min=start_index;

    for (int index=start_index; index<ARRAY_SIZE; index++)
        if (array[index] < min)
        {
            min=array[index];
            index_of_min=index;
        }

    return index_of_min;    //return index of smallest number
}
```



Multidimensional Array

In C++, the elements of an array can be of any type. In particular, the elements of an array can themselves be arrays. Arrays of arrays are called multidimensional arrays.

The most common form of a multidimensional array is a two-dimensional array which is similar to a rectangular structure divided into rows and columns. This type of two-dimensional array is called a matrix.

You can have arrays of 3 or more dimensions. They are less common.

Consider the two-dimensional array matrix:

```
int matrix[2][3]={{2,2,2}, //two rows, three columns  
                 {2,2,2}};
```

This is an array (size 2) of two arrays (size 3).



Multidimensional Array

We will now look at an example involving two-dimensional arrays; in this example we will write a function to add to matrixes.

```
main()
{
    int array1[4][2]={ 0,1,
                       0,1,
                       0,1,
                       0,1};
    int array2[4][2]={ 1,1,
                       1,1,
                       2,2,
                       2,2};
    addMatrixes(array1, array2);

    return 0;
}
```



Multidimensional Array *

The function takes two two-dimensional arrays as parameters. It adds the two matrixes and puts the result into the first array. It then prints the result matrix on the screen.

```
void addMatrixes(int a[][2], int b[][2])
{
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<2; j++)
            a[i][j]=a[i][j]+ b[i][j]; //adding
    }
    // to display the result on the screen
    for(i=0; i<4; i++)
    {
        for(j=0; j<2; j++)
            cout<<array1[i][j]<<" ";
        cout<<endl;
    }
}
```



Multidimensional Array *

For multi-dimensional array parameters, all the dimension sizes except the first must be given. This makes sense if you think of a multi-dimensional array as an array of arrays. In this example we have an array each element of which is an `int` array of size 4. Remember that if you have an array parameter, you do not have to specify the the array size in the square brackets.

Multi-dimensional arrays are mainly used to perform matrix operations and numerical analysis calculations. The base type of a multi-dimensional array can be any type; but for numerical calculations the type is usually `int` or `double`.

In the example on the previous page, we saw how to add two matrixes. You can also do other matrix operations like matrix multiplication, matrix transformations using two-dimensional arrays.