Introduction to Object Oriented Programming (OOP)

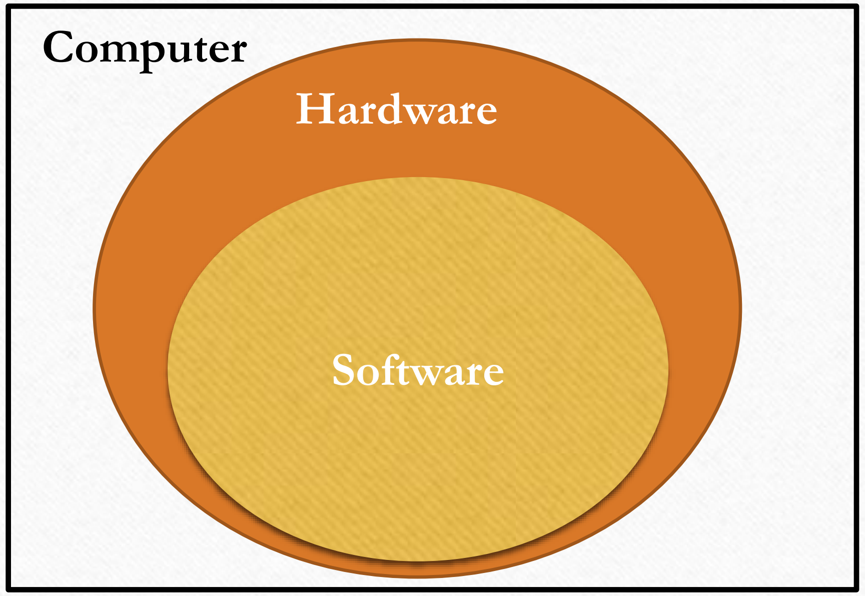
Module 1

OOP (IT 2005)

4th Semester ECSc

Introduction

- A programming language is a language specifically designed to express computations that can be performed by the computers.



Programming languages are the primary tools for creating software

First Generation: Machine Language

- This is the lowest level of programming language.
- Computers understand only this language.
- All the commands and data values are expressed using 0s and 1s.
- Advantages
  - Code can be directly executed by the computer.
  - Execution is fast.
- Disadvantages
  - Code is difficult to wrtte.
  - Code is difficult to understand by other people.

Second Generation: Assembly Language

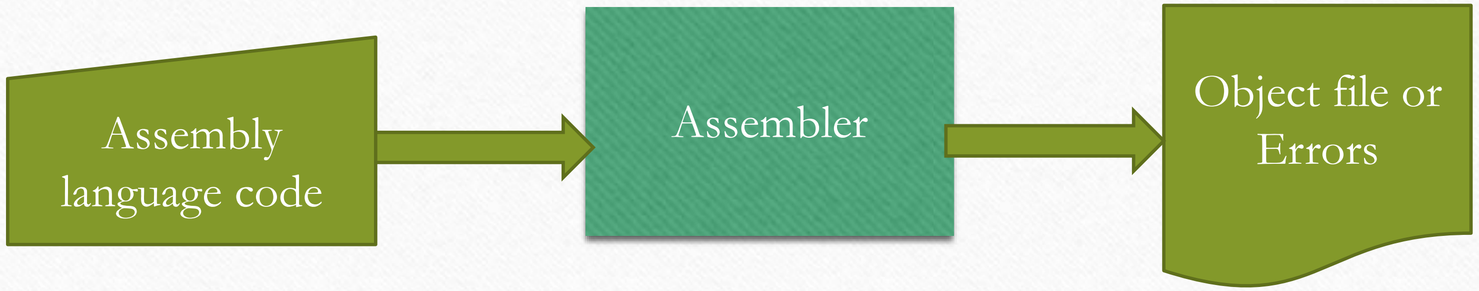
- Assembly languages developed in the mid-1950s.
- Assembly languages are symbolic programming languages.
- The language uses symbolic codes, called **mnemonic code.**
- Examples of mnemonic codes: ADD for addition, CMP for compare, MOV for copying data from one register to another register, etc.
- 8085, 8086 microprocessor codes are assembly level programming.

Second Generation: Assembly Language

- Advantages
  - It is easy to understand.
  - It is easier to write programs in assembly language than in machine language.
- Disadvantages
  - Code is machine dependent and thus non-portable.
  - Programmers must have the knowledge of processor's internal architecture.
- How does computer understand assembly level program? **Assembler**

Assembler

- An assembler is a special program that converts an assembly level program into a machine language.
- The result is an object file that can be executed.
- The assembler itself does not execute the object file.

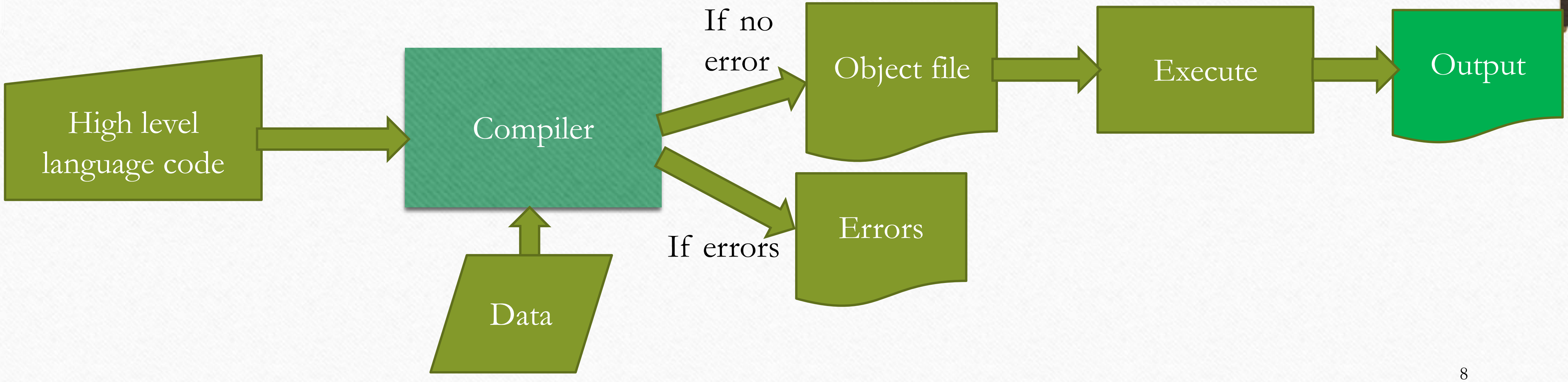


Third Generation: High Level Language

- The third generation languages are programmer friendly.
- FORTRAN, COBOL, C, C++, Java, etc. are high level programming languages.
- Advantages
    - The code is machine independent.
    - It is easy to learn and use the language.
- Disadvantages
    - Code may not be optimized.
    - The code is less efficient
- How does computer understand a high level program? **Compiler**

Compiler

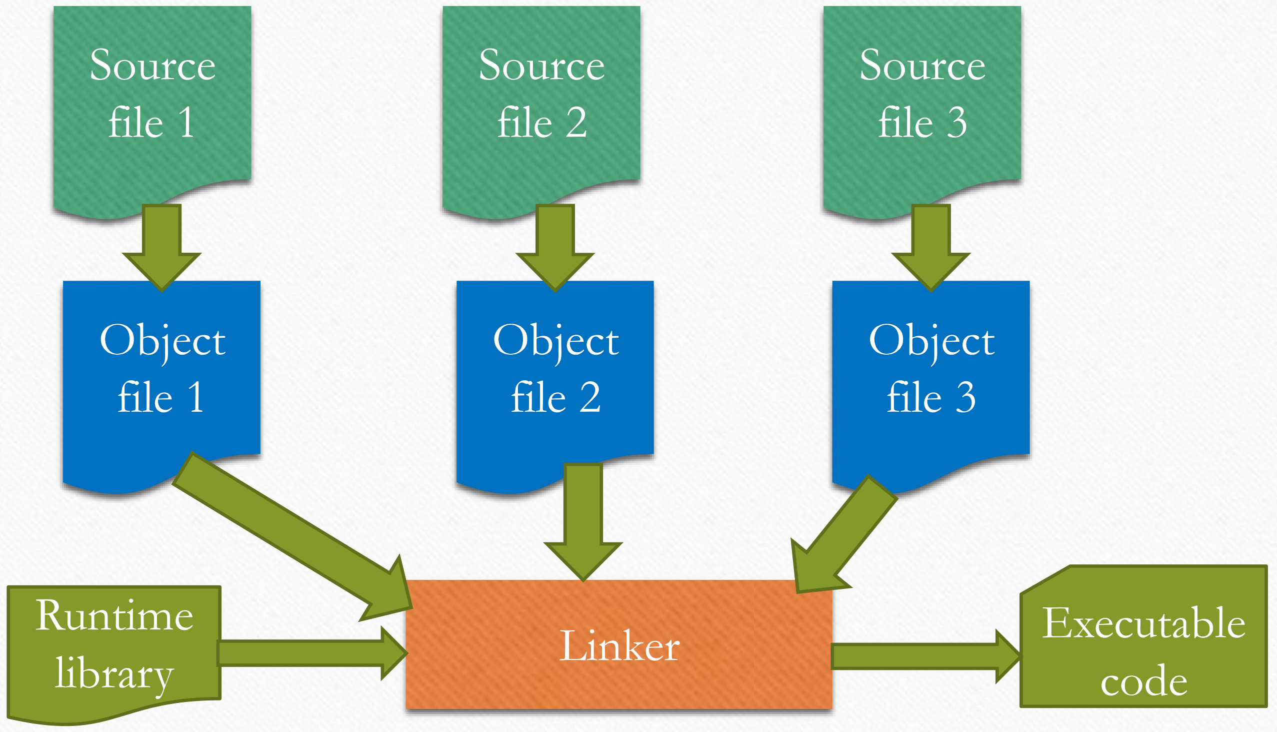
- A compiler is a special type of program that transforms the source code written in high level language into machine language.



Linker

- Software development in the real world usually follows a modular approach.
- In this approach, a program is divided into several modules to code, edit, debug, test and maintenance them.
- A module written for one program can also be used for another program.
- When a module is compiled, an object file of that module is generated.
- A linker is a program that binds the object files of all the modules together to form an executable program.
- A compiler automatically invokes the linker as the final step in compiling a code.

Linker


Source file 1
Source file 2
Source file 3
Object file 1
Object file 2
Object file 3
Runtime library
Linker
Executable code

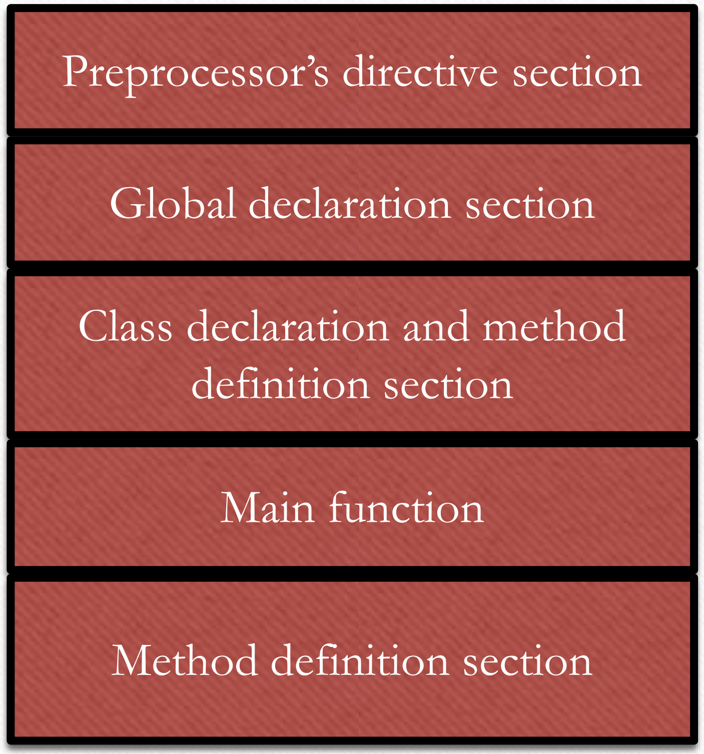
C++ Compilers

- **Clang** is a C++ compiler developed by Apple
- **MinGW-w64** provides the library files, header files and runtime support needed for the GNU C++ compilers to run on a Windows system. The w64 means that the compiler supports 64-bit version along side 32-bit version.
- **Microsoft Visual Studio Express** is a package that has an integrated development environment (IDE), compilers for C++, C# and Visual Basic.

History of C++

- C++ is a general purpose programming language developed by Bjarne Stroustrup in 1979 at Bell Labs.
- Stroustrup started working on “C with Classes”.
- In 1983, the name of the language was changed to C++, where ++ refers to adding new features in the C language.
- C++98 was released in 1998, C++03 was published in 2003, C++11 was released in 2011 and the latest version C++17 was published in 2017.

Structure of C++ Program


Preprocessor's directive section
Global declaration section
Class declaration and method definition section
Main function
Method definition section

Structure of C++ Program

Preprocessor’s directive section

Global declaration section

Class declaration and method definition section

Main function

Method definition section

```
// My first program
#include <iostream>
using namespace std;
int main()
{
   cout << “Hello C++” << endl;
   return 0;
}
```

Comment Lines in C++ Program

- **Single line comment**: // My first program
- **Multiple line comment**
- /* Author: Arighna Deb

  My first program */

Tokens in C++

- Keywords
- Variables
- Constants
- Strings
- Operators

Keywords

auto	break	case	char	const	continue	default	do	**Common to C and C++**
double	else	enum	extern	float	for	goto	if	
int	long	register	return	short	signed	sizeof	static	
struct	switch	typedef	union	unsigned	void	volatile	while	
asm	new	template	catch	operator	this	class	private	**Specific to C++**
throw	delete	protected	try	friend	public	virtual	inline	

Identifier

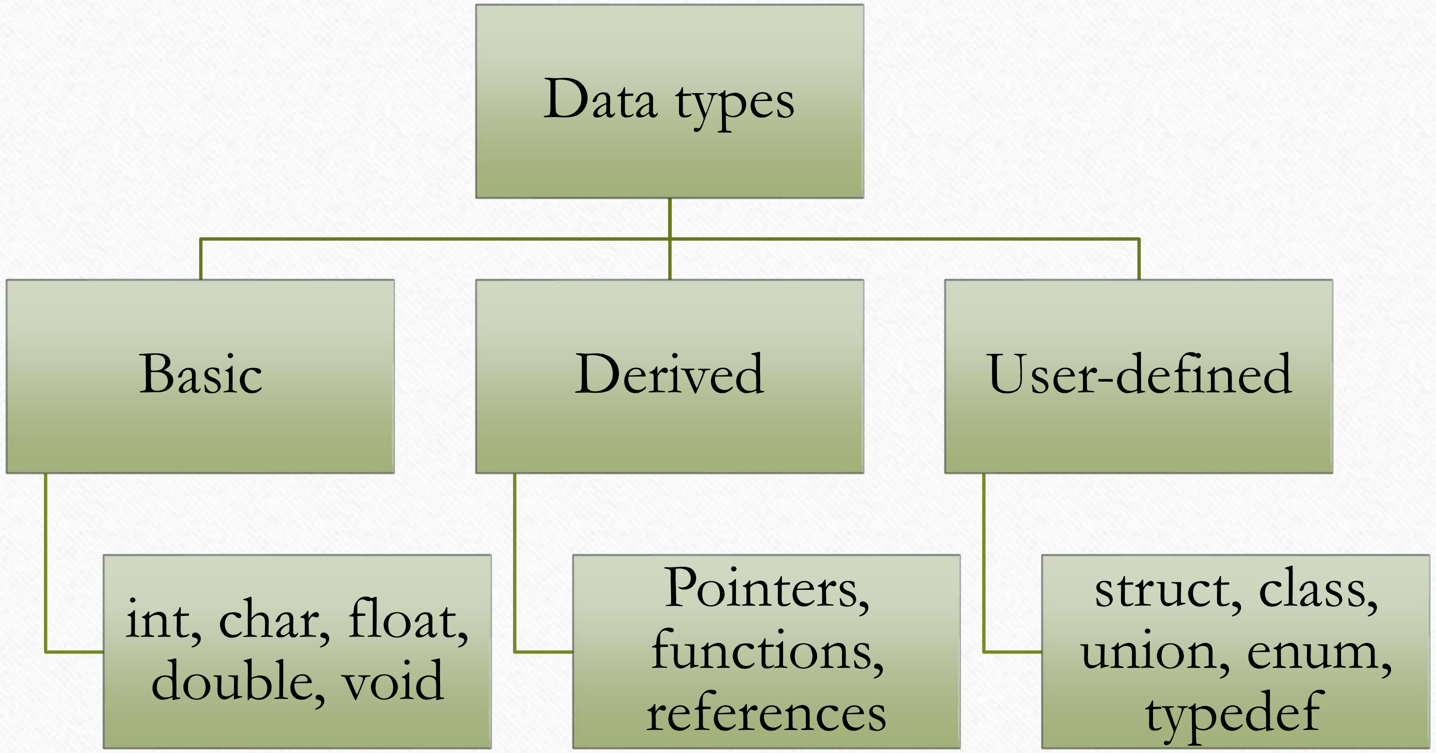
- Identifiers are basically the names given to program elements such as variables, arrays and functions.
- An identifier may consist of an alphabet, digit or an underscore.
- Rules for forming identifier name
  - The name cannot include any special characters or punctuation marks, except the underscore.
  - There cannot be two successive underscores.
  - Keywords cannot be used as identifiers.
  - Identifiers should not contain more than 31 characters.

Identifier

- **Examples of valid identifiers**
- roll_number, marks, DA, Roll_No
- **Examples of invalid identifiers**
- 2004238_student, @rollnumber, roll-number, roll.number

C++ is a case sensitive language.

Data Types in C++


Data types
Basic
Derived
User-defined
int, char, float, double, void
Pointers, functions, references
struct, class, union, enum, typedef

Program Example 1

```
// Program to print values of variables of different data types
#include <iostream>
using namespace std;
main()
{
   // Declare and initialize variables
   int num = 7;
   float amt = 123.45;
   char code = ‘A’;
   double pi = 3.1415926536;

// Print the values of the variables
   cout << “\n NUM = ” << num << endl;
   cout << “Amount = ” << amt << endl;
   cout << “Code = ” << code << endl;
   cout << “PI = ” << pi << endl;
   return 0;
}
```

Constants

- Constants are identifiers whose value does not change.
- The value of the constant is known to the compiler at the compile time.
- Constant names are usually written in UPPERCASE to visually distinguish them from other variable names written in lowercase.
- A constant can be defined either using **const** keyword or **#define** preprocessor directive.

Program Example 2

```
#include <iostream>
using namespace std;
main()
{
   // Declare a constant
    const double PI = 3.1415926536;
   cout << “PI = ” << PI << endl;

    return 0;
}
```

Constant declaration syntax:

const data_type var_name = value;

Program Example 3

```
#include <iostream>
using namespace std;
main()
{
  // Declare a constant
   const double PI = 3.1415926536;
  cout << “PI = ” << PI << endl;

   PI = 3.142;
   cout << “PI = ” << PI << endl;
   return 0;
}
```

Constant declaration syntax:

const data_type var_name = value;

Incorrect assignment

Program Example 4

```
#include <iostream>
#define PI 3.1415926536
using namespace std;
main()
{
   cout << “PI = ” << PI << endl;

   return 0;
}
```

Alternative constant declaration syntax:
#define Identifier_Name Value

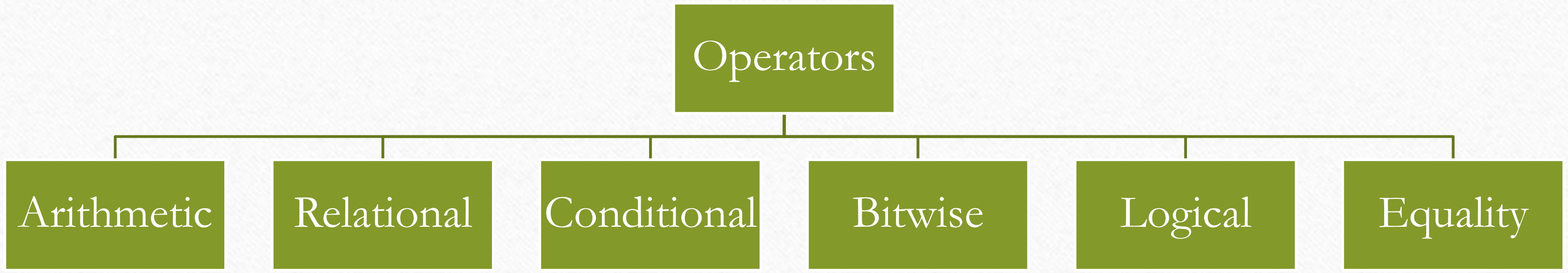
Program Example 5

```
#include <iostream>
#define PI 3.1415926536
using namespace std;
main()
{
   cout << "PI = " << PI << endl;

   PI = 3.142;
   cout << "PI = " << PI << endl;
   return 0;
}
```

Alternative constant declaration syntax:
#define Identifier_Name Value

Incorrect assignment

Operators in C++



Program Example 4

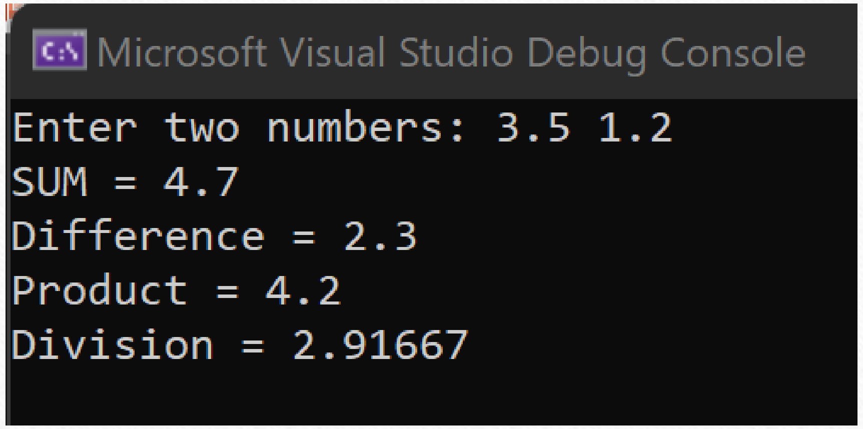
- A C++ program to perform addition, subtraction, multiplication, division, integer division, and modulo division of two integer numbers.

```
#include <iostream>
 using namespace std;
 int main()
 {
   int num1, num2;
   int sum, diff, prod, idiv, mdiv;
   float fdiv;
   cout << “Enter the two numbers: ”;
   cin >> num1 >> num2;
   sum = num1 + num2;
   diff  = num1 – num2;
   prod = num1 * num2;
   idiv = num1/num2;
   mdiv = num1%num2;   // Modulo division
   fdiv = (float)num1/num2;  // Type conversion
   /* insert print statements to print
      all the results of arithmetic operations */
   return 0;
 }
```

Program Example 5

- A C++ program to perform addition, subtraction, division, and multiplication of two floating point numbers.

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

 int main()
 {
   float num1, num2;
   cout << "\n Enter two numbers: ";
   cin >> num1 >> num2;
   cout << "\n Sum = " << num1 + num2 << endl;
   cout << "Difference = " << num1 – num2 << endl;
   cout << "Product = " << num1 * num2 << endl;
   cout << "Division = " << num1/num2 << endl;
   return 0;
}
```



```
Enter two numbers: 3.5 1.2
SUM = 4.7
Difference = 2.3
Product = 4.2
Division = 2.91667
```

Modulus Operator (%)

- The modulus operator (%) finds the remainder of an integer division.
- This operator can be applied only to integer operands and cannot be used on float or double operands.
- Example:

```
float num = 20.0;
cout << “\n Result = ” << num % 5;
```

Modulus Operator (%)

- The modulus operator (%) finds the remainder of an integer division.
- This operator can be applied only to integer operands and cannot be used on float or double operands.
- Example:

```
float nu      20.0;
cout <      Result = ” << num % 5;
```

Compilation error

Modulus Operator (%)

- Few examples

16 % 3 = 1   -16 % 3 = -1   16 % -3 = 1   -16 % -3 = -1

Modulus Operator (%)

- Few examples

16 % 3 = 1 -16 % 3 = -1 16 % -3 = -1 -16 % -3 = -1

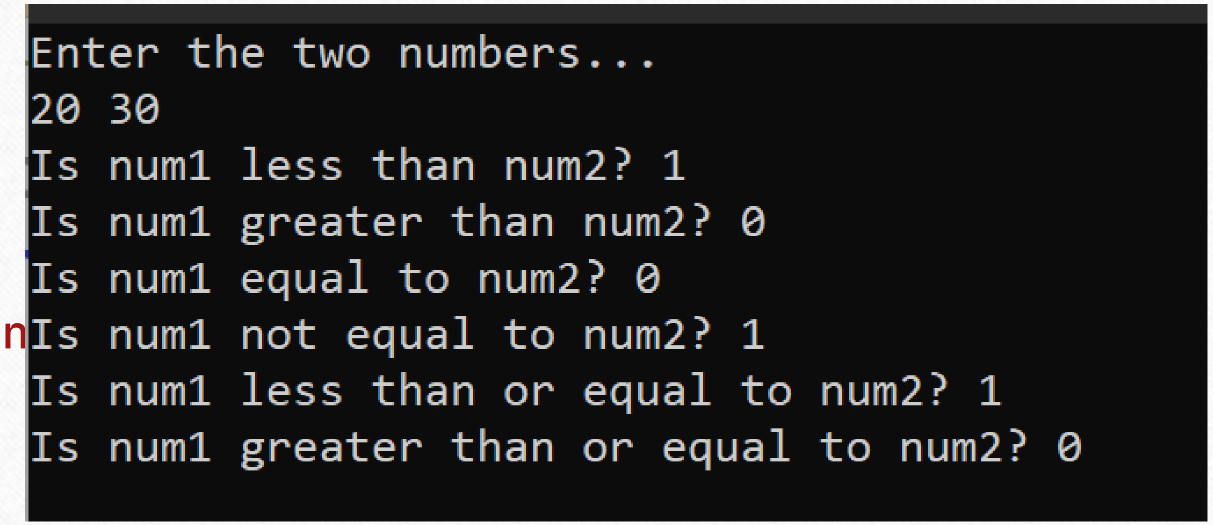
- **While performing modulo division, the sign of the result is always the sign of the first operand (the dividend).**

Relational Operators

- A relational operator is an operator that compares two values.
- These operators are also known as the comparison operators.
- A C++ program to show the use of relational operators (Program Example 6).

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter the two numbers..." << endl;
int num1, num2;
cin >> num1 >> num2;

cout << "Is num1 less than num2? " << (num1 < num2) << endl;
cout << "Is num1 greater than num2? " << (num1 > num2) << endl;
cout << "Is num1 equal to num2? " << (num1 == num2) << endl;
cout << "Is num1 not equal to num2? " << (num1 != num2) << endl;
cout << "Is num1 less than or equal to num2? " << (num1 <= num2) << endl;
cout << "Is num1 greater than or equal to num2? " << (num1 >= num2) << endl;

return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter the two n
int num1, num2;
cin >> num1 >> num2;

cout << "Is num1 less than num2? " << (num1 < num2) << endl;
cout << "Is num1 greater than num2? " << (num1 > num2) << endl;
cout << "Is num1 equal to num2? " << (num1 == num2) << endl;
cout << "Is num1 not equal to num2? " << (num1 != num2) << endl;
cout << "Is num1 less than or equal to num2? " << (num1 <= num2) << endl;
cout << "Is num1 greater than or equal to num2? " << (num1 >= num2) << endl;

return 0;
}
```



```
Enter the two numbers...
20 30
Is num1 less than num2? 1
Is num1 greater than num2? 0
Is num1 equal to num2? 0
Is num1 not equal to num2? 1
Is num1 less than or equal to num2? 1
Is num1 greater than or equal to num2? 0
```

Unary Operators

- Increment operator (++)
- Decrement operator (--)
- The increment/decrement operators have two variants- prefix or postfix.
- Prefix expression (++x or --x)
- Postfix expression (x++ or x--)
- **++x is not same as x++ and --x is not same as x--.** Important

Program Example 7

- A C++ program to illustrate the use of unary increment operator.

```cpp
#include <iostream>
using namespace std;
int main()
{
int x = 10;
int y, z;
y = x++;
cout << "Y = " << y << endl;
cout << "X = " << x << endl;
cout << endl;
cout << endl;
z = ++x;
cout << "Z = " << z << endl;
cout << "X = " << x << endl;
return 0;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
int x = 10;
int y, z;
y = x++;
cout << "Y = " << y << endl;
cout << "X = " << x << endl;
cout << endl;
cout << endl;
z = ++x;
cout << "Z = " << z << endl;
cout << "X = " << x << endl;
return 0;
}
```

Equivalent to
y = x;
x = x + 1;

Equivalent to
x = x + 1;
z = x;

Program Example 8

- Write a C++ program to illustrate the use of unary decrement operator.

[Hint: Please see the Program Example 7 and replace increment operator by decrement operator. Observe the output of the program]

Conditional Operator

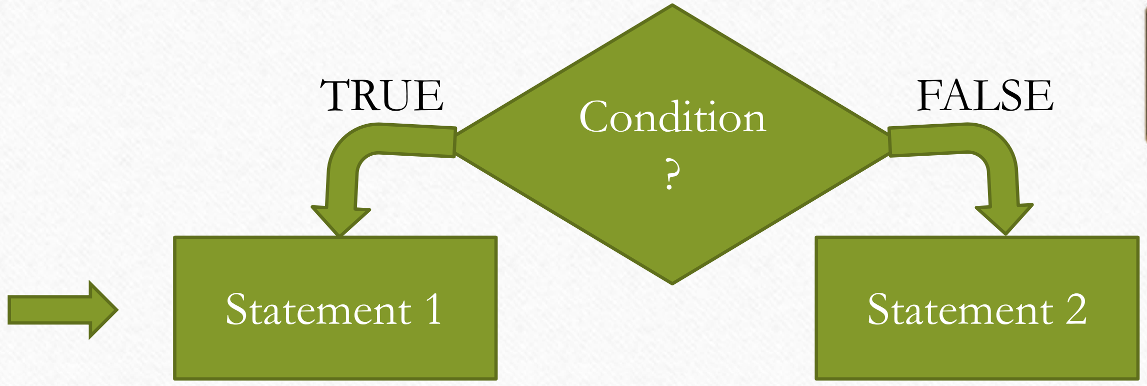
- The conditional operator or ternary operator is similar to an *if…else* statement.
- Syntax of the conditional operator:

```
Condition ? Statement 1 : Statement 2
```

Conditional Operator

- The conditional operator or ternary operator is similar to an *if…else* statement.
- Syntax of the conditional operator:

Condition ? Statement 1 : Statement 2



Program Example 8

- A C++ program to find the larger number between two integer numbers using conditional operator.

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter the two numbers..." << endl;
int num1, num2;
cin >> num1 >> num2;

int larger = (num1 > num2) ? num1 : num2;

cout << "Number 1 = " << num1 << endl;
cout << "Number 2 = " << num2 << endl;
cout << "Larger = " << larger << endl;

return 0;
}
```

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

int main()
{
cout << "Enter the two numbers..." << endl;
int num1, num2;
cin >> num1 >> num2;

int larger = (num1 > num2) ? num1 : num2;

cout << "Number 1 = " << num1 << endl;
cout << "Number 2 = " << num2 << endl;
cout << "Larger = " << larger << endl;

return 0;
}
```

Equivalent to $\Rightarrow$

```
if num1 > num2
   larger = num1
 else
   larger = num2
```

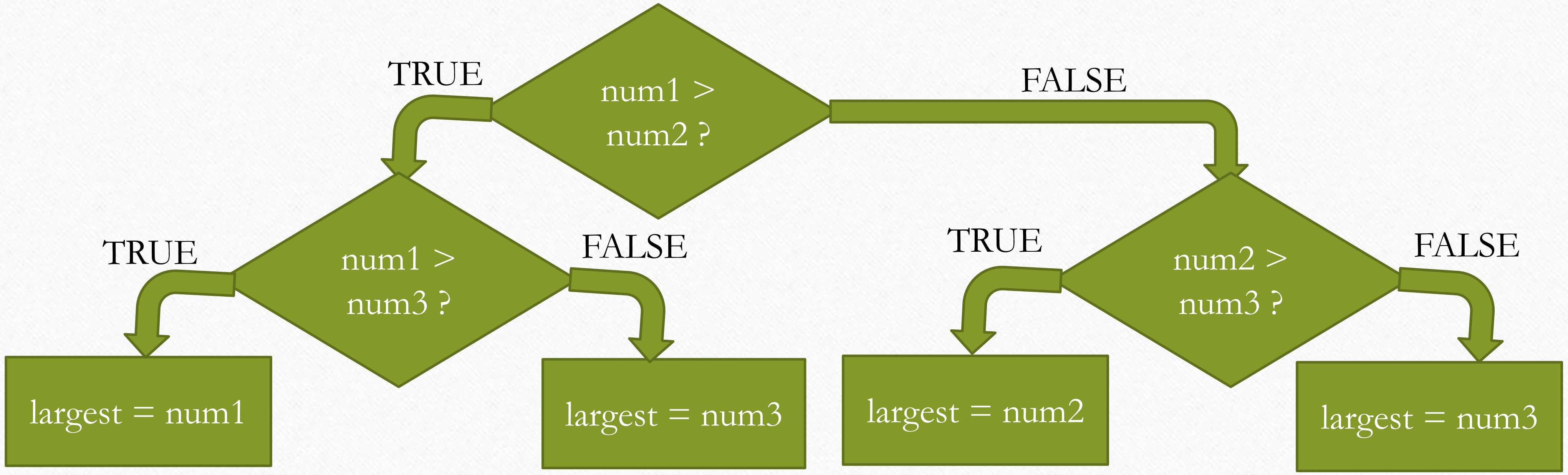
Program Example 9

- A C++ program to find the largest among three integer numbers using conditional operator.
- Here, we use nested conditional operator
- Syntax of nested conditional operator

Condition 1 ? (Condition 2 ? Statement 1 : Statement 2) : (Condition 3 ? Statement 4 : Statement 5)

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter the three numbers..." << endl;
int num1, num2, num3;
cin >> num1 >> num2 >> num3;

int largest = (num1 > num2) ? (num1 > num3 ? num1 : num3) : (num2 > num3 ?
num2 : num3);

cout << "Number 1 = " << num1 << endl;
cout << "Number 2 = " << num2 << endl;
cout << "Number 3 = " << num3 << endl;
cout << "Largest = " << largest << endl;

return 0;
}
```

```
int largest = (num1 > num2) ? (num1 > num3 ? num1 : num3) : (num2 > num3 ?
num2 : num3);
```

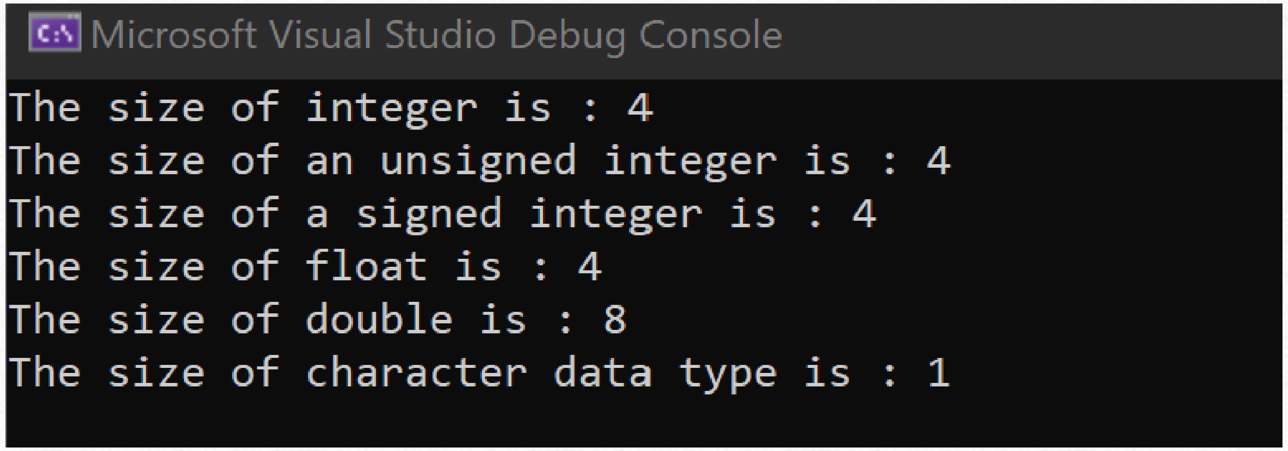


sizeof Operator

- The sizeof is a unary operator.
- It is used to calculate the sizes of data types.
- It returns the space required by a data type in bytes.
- A C++ program to show the use of **sizeof** operator (Program Example 10)

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "The size of integer is : " << sizeof(int) << endl;
cout << "The size of an unsigned integer is : " << sizeof(unsigned int) <<
endl;
cout << "The size of a signed integer is : " << sizeof(signed int) << endl;
cout << "The size of float is : " << sizeof(float) << endl;
cout << "The size of double is : " << sizeof(double) << endl;
cout << "The size of character data type is : " << sizeof(char) << endl;

return 0;
}
```

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

int main()
{
cout << "The size of integer is : " << sizeof(int) << endl;
cout << "The size of an unsigned integer is : " << sizeof(unsigned int) <<
endl;
cout << "The size of a signed integer is : " << sizeof(signed int) << endl;
cout << "The size of float is : " << sizeof(float) << endl;
cout << "The size of double is : " << sizeof(double) << endl;
cout << "The size of character data type is : " << sizeof(char) << endl;

return 0;
}
```



```
The size of integer is : 4
The size of an unsigned integer is : 4
The size of a signed integer is : 4
The size of float is : 4
The size of double is : 8
The size of character data type is : 1
```

C++ Decision Control Statements

- if statement
- if-else statement
- if-else if-else statement
- switch case
- while
- do-while
- for

C++ Decision Control Statements

- if statement
- if-else statement
- if-else if-else statement
- switch case
- **while**
- **do-while**
- **for**

$$\left.\begin{array}{l}\textbf{while}\\ \textbf{do-while}\\ \textbf{for}\end{array}\right\}\text{Iterative statements}$$

Program Example 11

- A C++ program to determine whether a person is eligible to vote.

```cpp
#include <iostream>
using namespace std;

int main()
{
int age;
cout << "Enter the age: ";
cin >> age;

if (age >= 18)
  cout << "You are eligible to vote." << endl;

return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int main()
{
int age;
cout << "Enter the age: ";
cin >> age;

if (age >= 18)
  cout << "You are eligible to vote." << endl;

return 0;
}
```

For a single statement within *if*, brackets are not necessary.
For multiple statements within *if*, curly brackets are necessary.

Program Example 12

- A C++ program to find the larger number between two numbers.

Program Example 12

- A C++ program to find the larger number between two numbers.

```cpp
#include <iostream>
using namespace std;

int main()
{
int a, b, larger;
cout << "Enter two numbers: ";
cin >> a >> b;
if (a > b)
    larger = a;
else
    larger = b;

cout << "Large = " << larger << endl;

return 0;
}
```

Program Example 13

- A C++ program to find the largest among three numbers.

```cpp
#include <iostream>
using namespace std;
int main()
{
int a, b, c, largest;
cout << "Enter three numbers: ";
cin >> a >> b >> c;
if (a > b)
{
    if (a > c)
        largest = a;
    else
        largest = c;
}
else
{
    if (b > c)
        largest = b;
    else
        largest = c;
}
cout << "Large = " << largest << endl;
return 0;
}
```

Program Example 14

- A C++ program to take input from the user and then check whether it is a number or a character. If it is a character, determine whether it is in upper case or lower case.
- This code shows the usage of **if-else if-else**

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

int main()
{
char ch;
cout << "Enter any character: ";
cin >> ch;

if (ch >= 'A' && ch <= 'Z’)
    cout << "Upper case letter." << endl;
else if (ch >= 'a' && ch <= 'z’)
    cout << "Lower case letter." << endl;
else if (ch >= 0 && ch <= 9)
    cout << "Number." << endl;
else
    cout << "Invalid entry!" << endl;

return 0;
}
```

switch case Statement

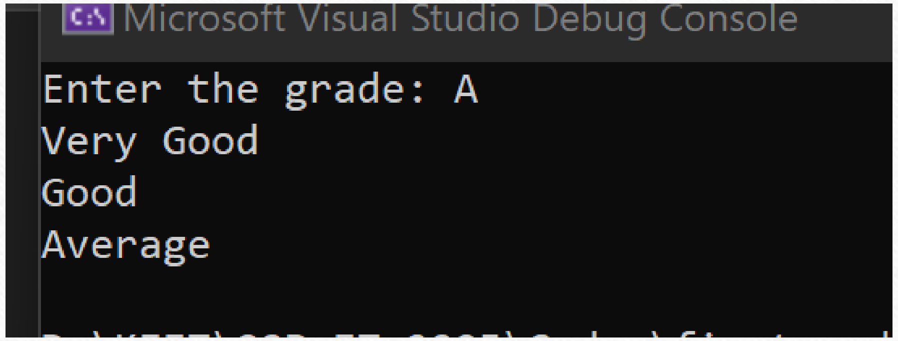
- When there are many conditions to test, usage of if-else becomes complex.
- **switch case** statements are often used in such situations.
- A C++ program to show the usage of switch case statement (Program Example 15)

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

int main()
{
char grade;
cout << "Enter the grade: ";
cin >> grade;

switch (grade)
{
case 'O' : cout << "Outstanding" << endl;
break;
case 'E' : cout << "Excellent" << endl;
break;
case 'A' : cout << "Very Good" << endl;
break;
case 'B' : cout << "Good" << endl;
break;
case 'C' : cout << "Average" << endl;
break;
case 'D' : cout << "Poor" << endl;
break;
case 'F' : cout << "Fail" << endl;
break;
default: cout << "Invalid" << endl;
break;
}
return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int main()
{
char grade;
cout << "Enter the grade: ";
cin >> grade;

switch (grade)
{
case 'O' : cout << "Outstanding" << endl;
break;
case 'E' : cout << "Excellent" << endl;
break;
case 'A' : cout << "Very Good" << endl;
break;
case 'B' : cout << "Good" << endl;
break;
case 'C' : cout << "Average" << endl;
break;
case 'D' : cout << "Poor" << endl;
break;
case 'F' : cout << "Fail" << endl;
break;
default: cout << "Invalid" << endl;
break;
}
return 0;
}
```

If we remove the break, what will happen?

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

int main()
{
char grade;
cout << "Enter the grade: ";
cin >> grade;

switch (grade)
{
case 'O' : cout << "Outstanding" << endl;
break;
case 'E' : cout << "Excellent" << endl;
break;
case 'A' : cout << "Very Good" << endl;
case 'B' : cout << "Good" << endl;
case 'C' : cout << "Average" << endl;
break;
case 'D' : cout << "Poor" << endl;
break;
case 'F' : cout << "Fail" << endl;
break;
default: cout << "Invalid" << endl;
break;
}
return 0;
}
```



```
Enter the grade: A
Very Good
Good
Average
```

Iterative Statements

- while loop
- do-while loop
- for loop

while loop

- Syntax

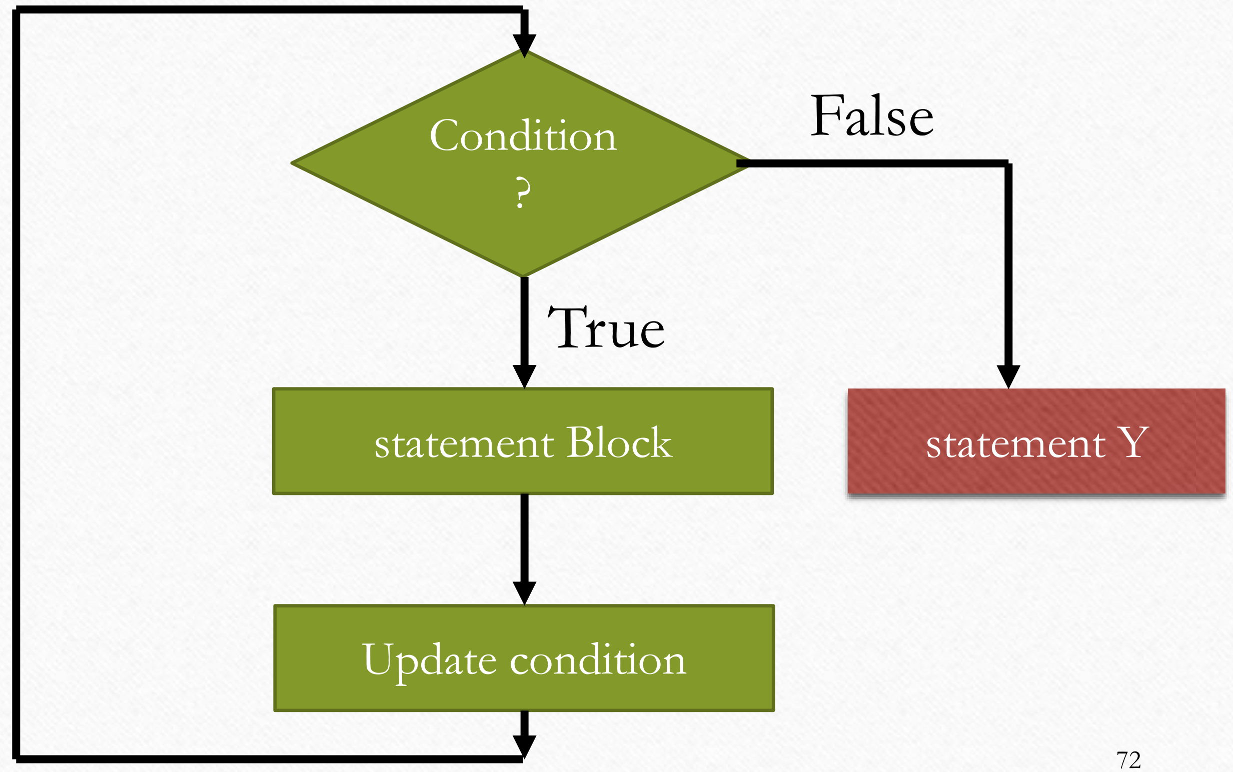
```
while(condition)
{
   statement Block;
 }
statement Y;
```

while loop

- Syntax

```
while(condition)
{
    statement Block;
}
statement Y;
```

Execution flow



while loop

- A C++ program to calculate the sum of consecutive numbers from 1 to 11 (Program Example 16)

```cpp
#include <iostream>
using namespace std;

int main()
{
int i = 1;
int sum = 0;

while (i <= 11)
{
    sum += i;
    i++;
}

cout << "Sum of 1 to 11 is " << sum << endl;
return 0;
}
```

Output: 66

do-while loop

- Syntax

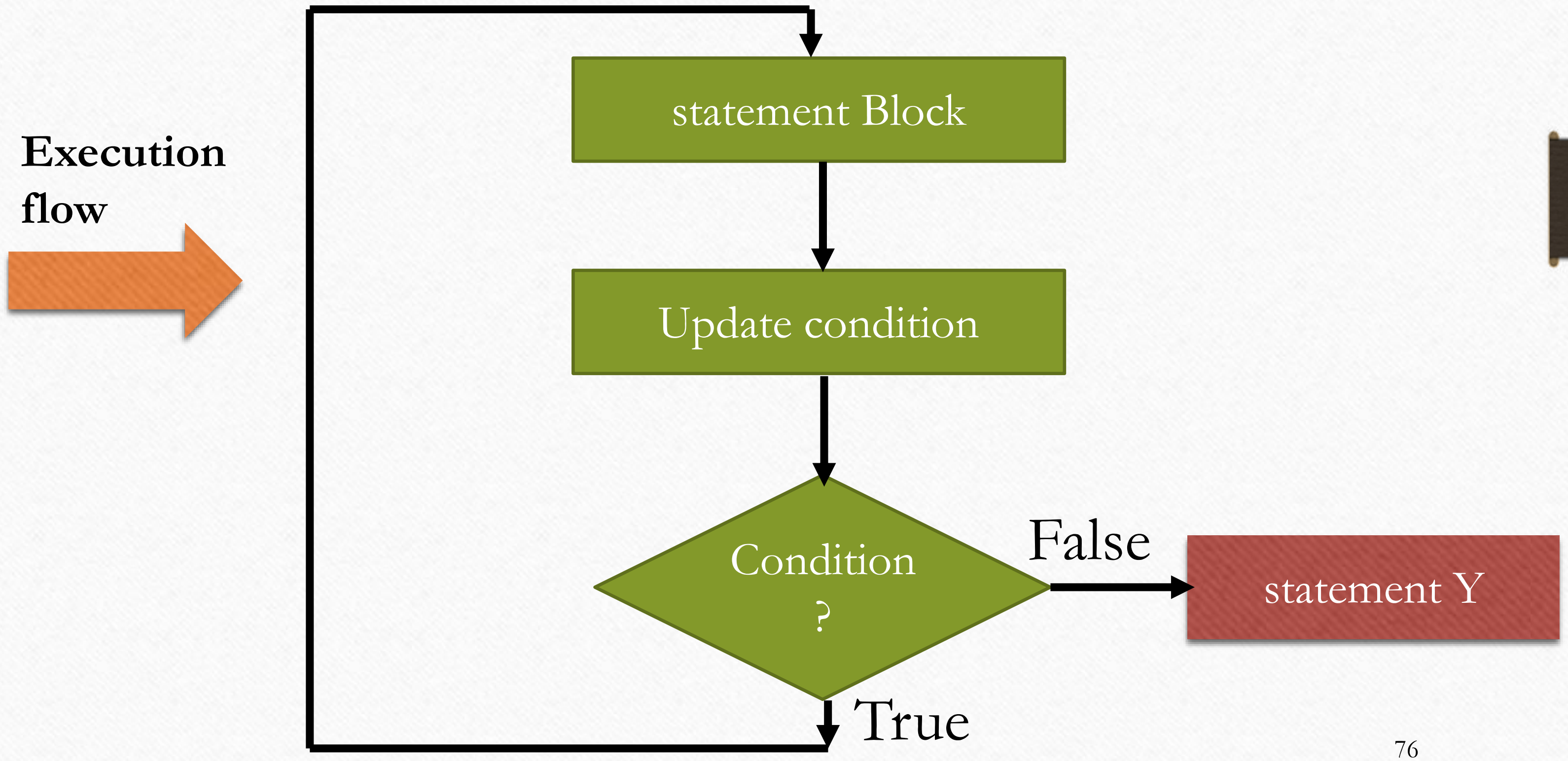
```
do
{
    statement Block;
} while(condition);
statement Y;
```

do-while loop

- Syntax

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

Execution flow



do-while loop

- The major disadvantage is that it always executes at least once.
- Loop will be executed even if we enter invalid data.
- A C++ program to calculate the sum of consecutive numbers from 1 to 11 (Program Example 17)

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

int main()
{
int i = 1;
int sum = 0;

do
{
sum += i;
i++;
} while (i <= 11);

cout << "Sum of 1 to 11 is " << sum << endl;
return 0;
}
```

Output: 66

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

int main()
{
int i = 12;
int sum = 0;

do
{
sum += i;
i++;
} while (i <= 11);

cout << "Sum of 1 to 11 is " << sum << endl;
return 0;
}
```

Mistakenly typed i = 12

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

int main()
{
int i = 12;
int sum = 0;

do
{
sum += i;
i++;
} while (i <= 11);

cout << "Sum of 1 to 11 is " << sum << endl;
return 0;
}
```

Mistakenly typed i = 12

Output: 12 (incorrect)

for loop

- Syntax

```
for(loop variable initialization; condition; increment/decrement loop variable)
```

- A C++ program to calculate the sum of consecutive numbers from 1 to 11 (Program Example 18)

```cpp
#include <iostream>
using namespace std;

int main()
{
int sum = 0;

for (int i = 1; i <= 11; i++)
sum += i;

cout << "Sum of 1 to 11 is " << sum << endl;
return 0;
}
```

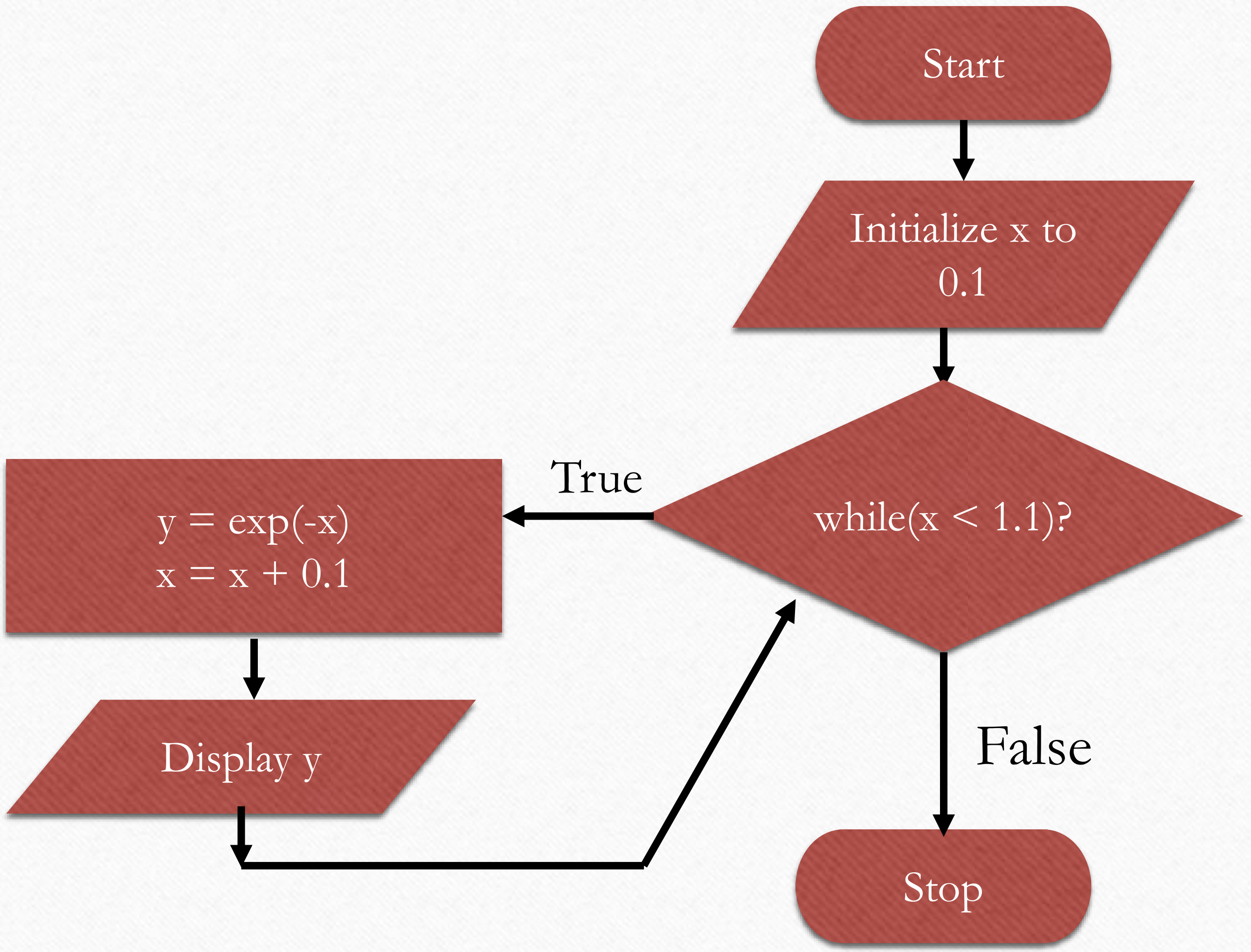
Output: 66

Practice Problem 1

- Write a C++ program to print a table of values of the function $y = e^{-x}$ for x varying from 0 to 1 in steps of 0.1. The table should appear as follows.

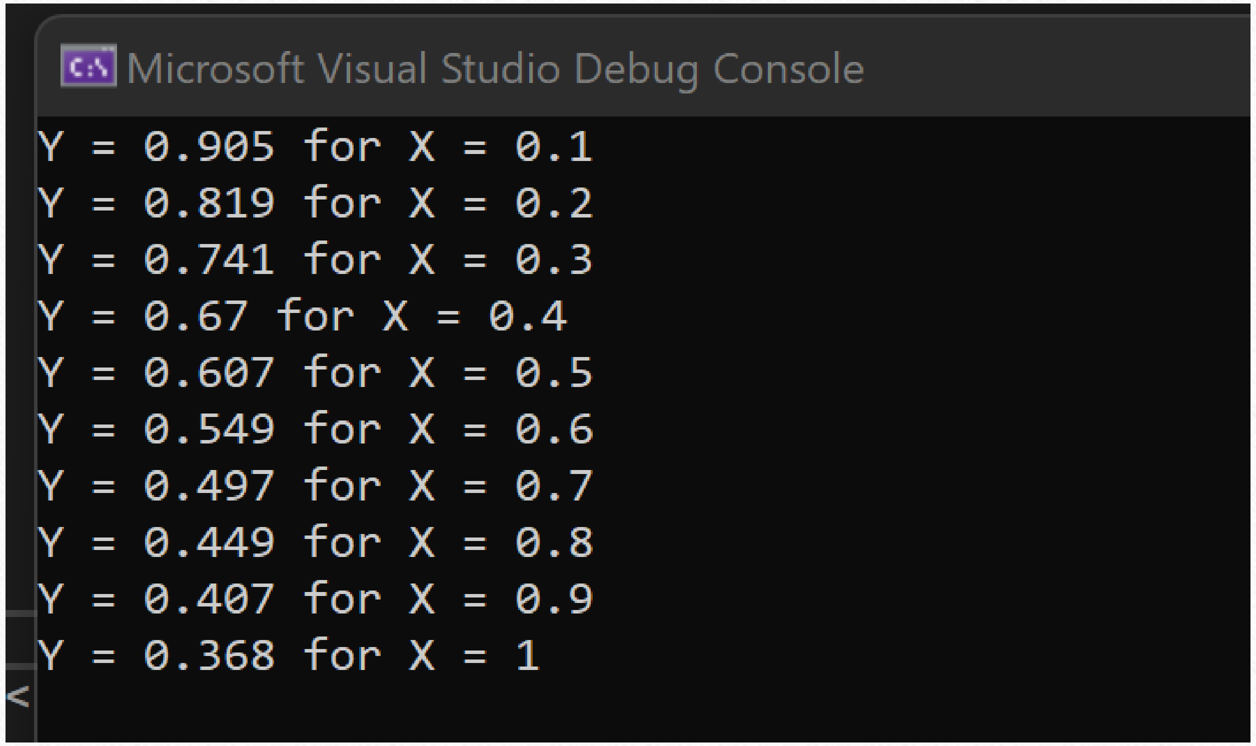
x	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
y=exp(-x)										

- Flowchart
- Program


Start
Initialize x to 0.1
while(x < 1.1)?
True
y = exp(-x)
x = x + 0.1
Display y
False
Stop

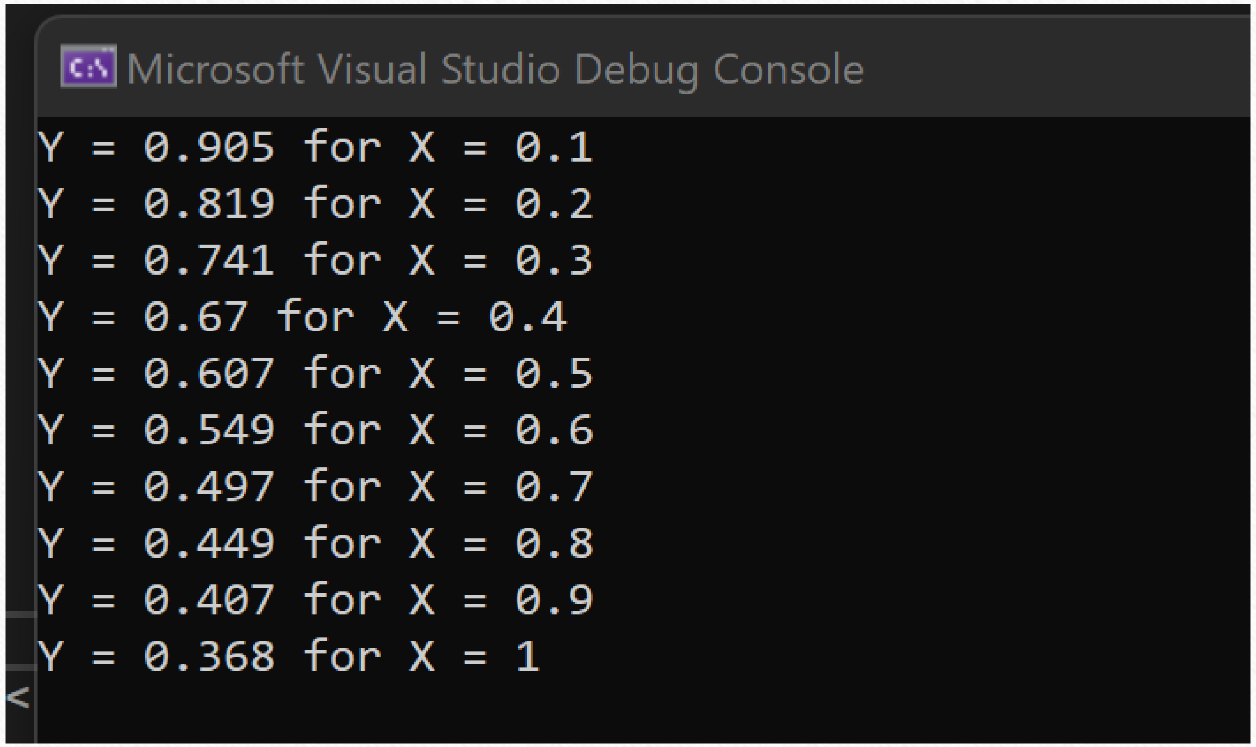
```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x = 0.1, y;

while (x < 1.1)
{
    y = exp(-x);
    cout << "Y = " << y << " for X = " << x << endl;
    x = x + 0.1;
}

return 0;
}
```

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x = 0.1, y;

while (x < 1.1)
{
    y = exp(-x);
    cout << "Y = " << y << " for X = " << x << endl;
    x = x + 0.1;
}

return 0;
}
```



```
Y = 0.905 for X = 0.1
Y = 0.819 for X = 0.2
Y = 0.741 for X = 0.3
Y = 0.67 for X = 0.4
Y = 0.607 for X = 0.5
Y = 0.549 for X = 0.6
Y = 0.497 for X = 0.7
Y = 0.449 for X = 0.8
Y = 0.407 for X = 0.9
Y = 0.368 for X = 1
```

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x = 0.1, y;

while (x < 1.1)
{
    y = exp(-x);
    cout << "Y = " << y << " for X = " << x << endl;
    x = x + 0.1;
}

return 0;
}
```



Microsoft Visual Studio Debug Console

```
Y = 0.905 for X = 0.1
Y = 0.819 for X = 0.2
Y = 0.741 for X = 0.3
Y = 0.67 for X = 0.4
Y = 0.607 for X = 0.5
Y = 0.549 for X = 0.6
Y = 0.497 for X = 0.7
Y = 0.449 for X = 0.8
Y = 0.407 for X = 0.9
Y = 0.368 for X = 1
```

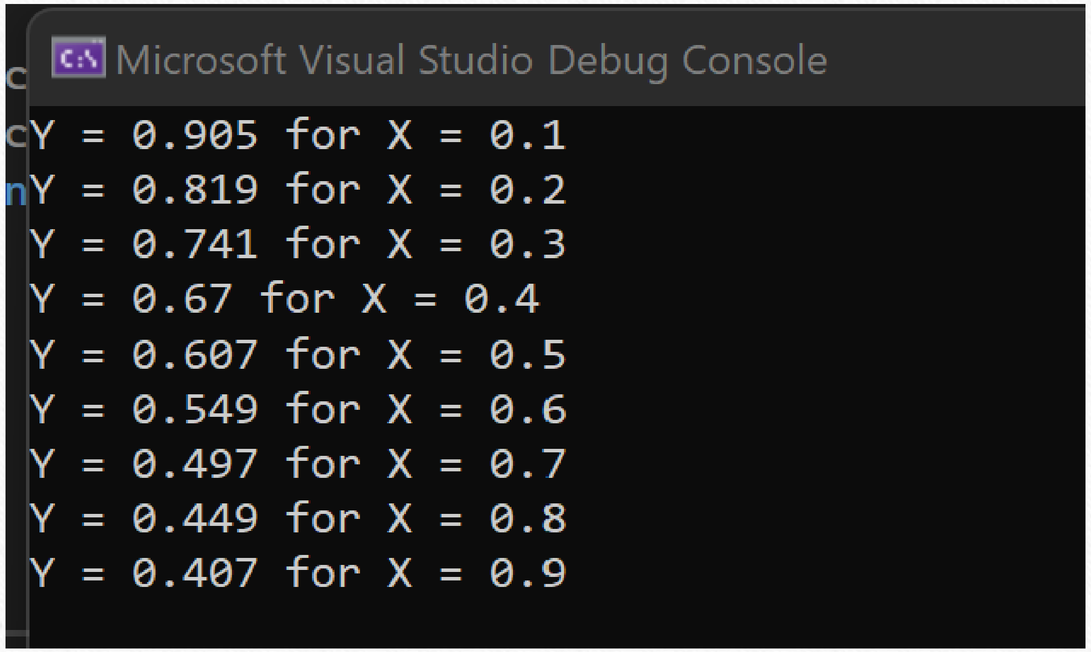
Replacing it by
while(x <= 1.0)

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x = 0.1, y;

while (x <= 1.0)
{
    y = exp(-x);
    cout << "Y = " << y << " for X = " << x << endl;
    x = x + 0.1;
}

return 0;
}
```

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x = 0.1, y;

while (x <= 1.0)
{
    y = exp(-x);
    cout << "Y = " << y << " for X = " << x << endl;
    x = x + 0.1;
}

return 0;
}
```

X = 1.0

is missing



```
Y = 0.905 for X = 0.1
Y = 0.819 for X = 0.2
Y = 0.741 for X = 0.3
Y = 0.67 for X = 0.4
Y = 0.607 for X = 0.5
Y = 0.549 for X = 0.6
Y = 0.497 for X = 0.7
Y = 0.449 for X = 0.8
Y = 0.407 for X = 0.9
```

Loop runs 9 times instead of 10 times

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x, y;
int count = 1;

while (count <= 10)
{
x = count * 0.1;
y = exp(-x);
cout << "Y = " << y << " for X = " << x << endl;
count++;
}
return 0;
}
```

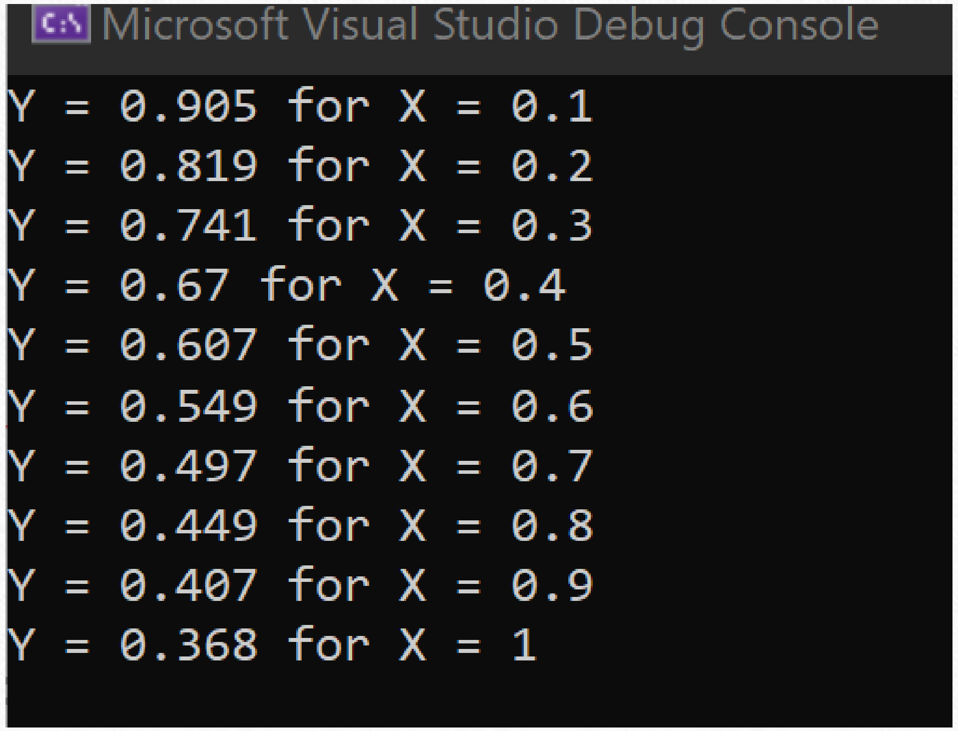
Modified loop

Modified loop

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout.precision(3);

float x, y;
int count = 1;

while (count <= 10)
{
x = count * 0.1;
y = exp(-x);
cout << "Y = " << y << " for X = " << x << endl;
count++;
}
return 0;
}
```

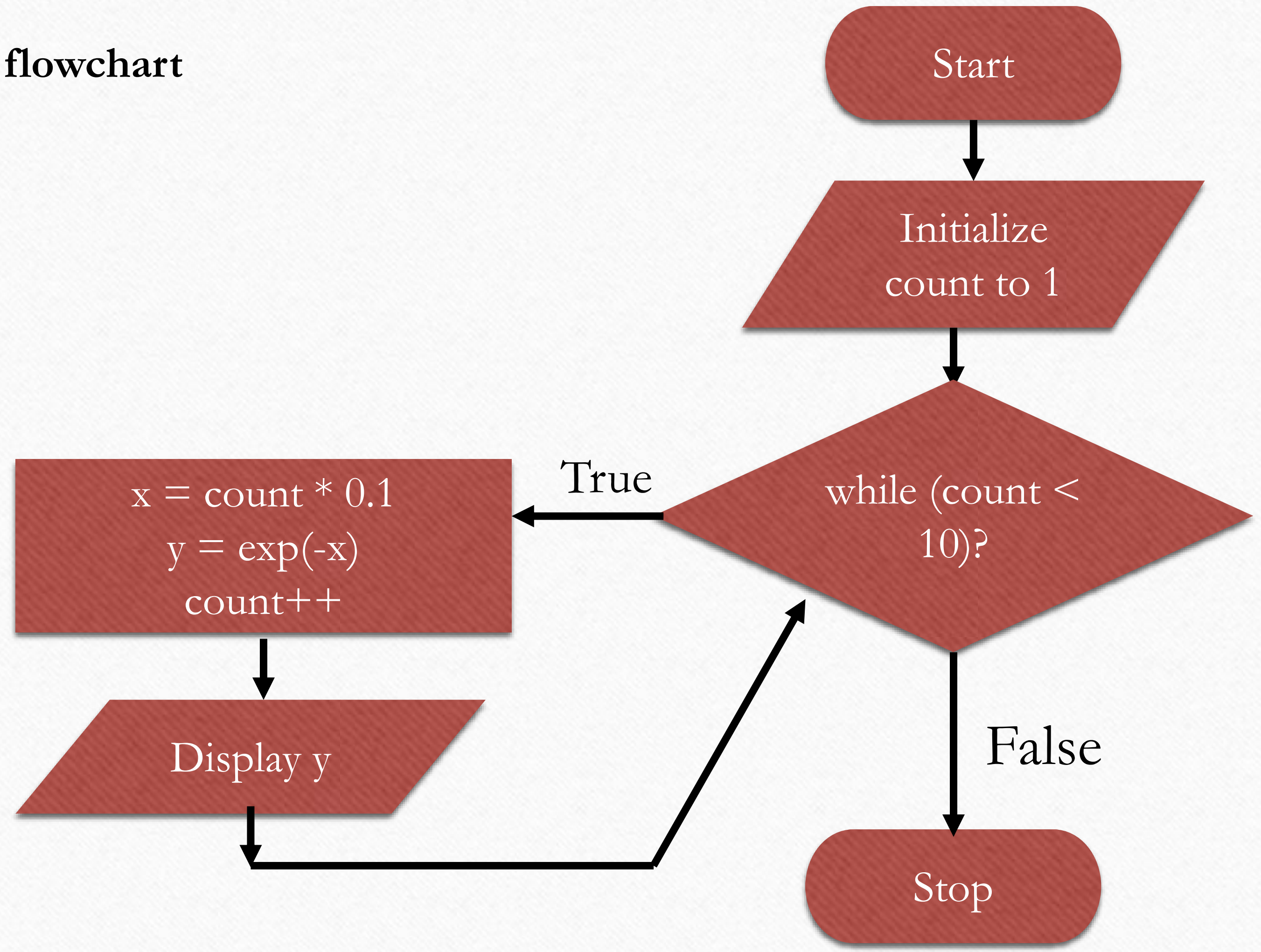
Modified loop

Modified loop


Microsoft Visual Studio Debug Console

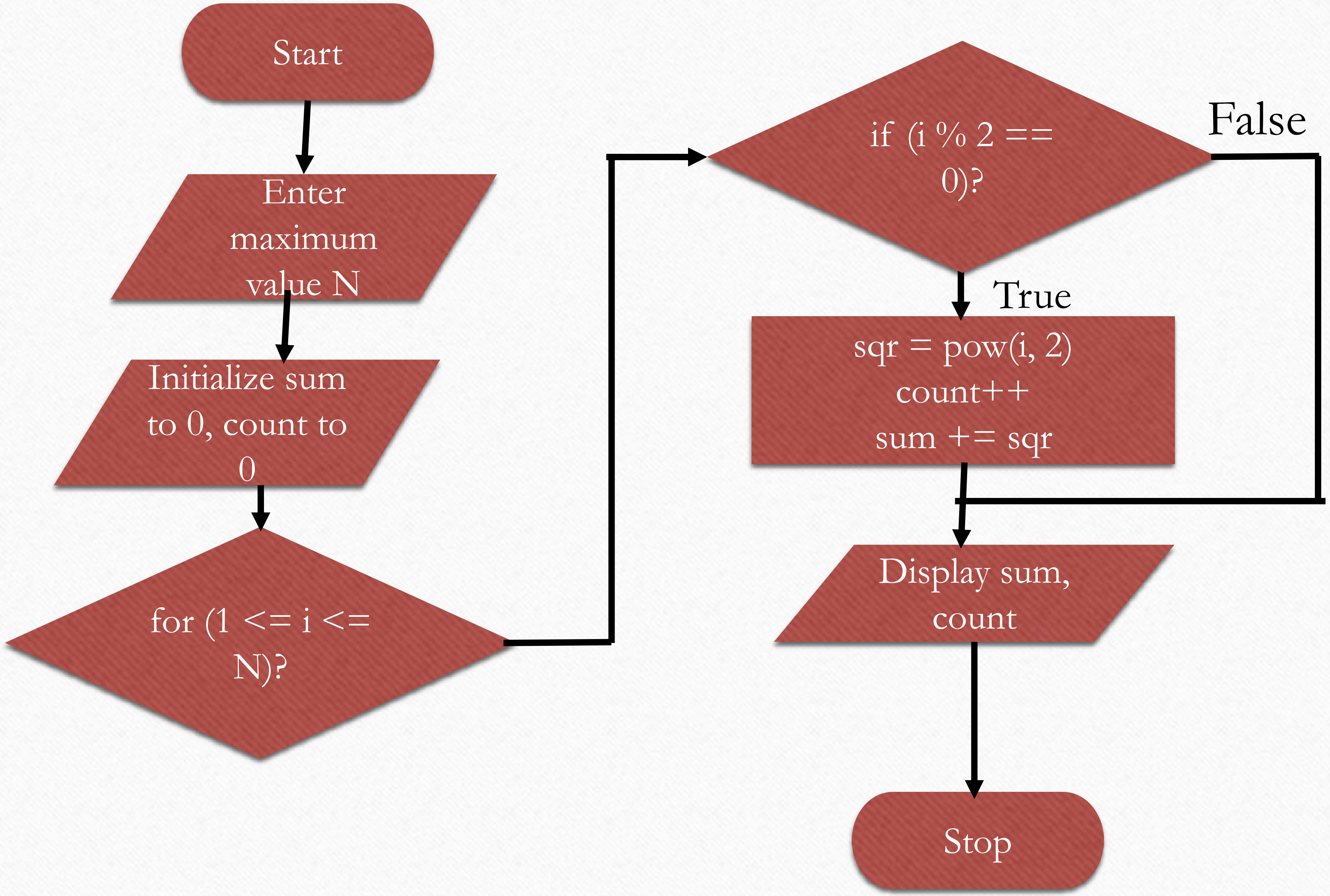
```
Y = 0.905 for X = 0.1
Y = 0.819 for X = 0.2
Y = 0.741 for X = 0.3
Y = 0.67 for X = 0.4
Y = 0.607 for X = 0.5
Y = 0.549 for X = 0.6
Y = 0.497 for X = 0.7
Y = 0.449 for X = 0.8
Y = 0.407 for X = 0.9
Y = 0.368 for X = 1
```

Alternative flowchart



Practice Problem 2

- Write a C++ program to sum of squares of even numbers up to the value of N. Take the value of N from the user.
- Flowchart
- Program

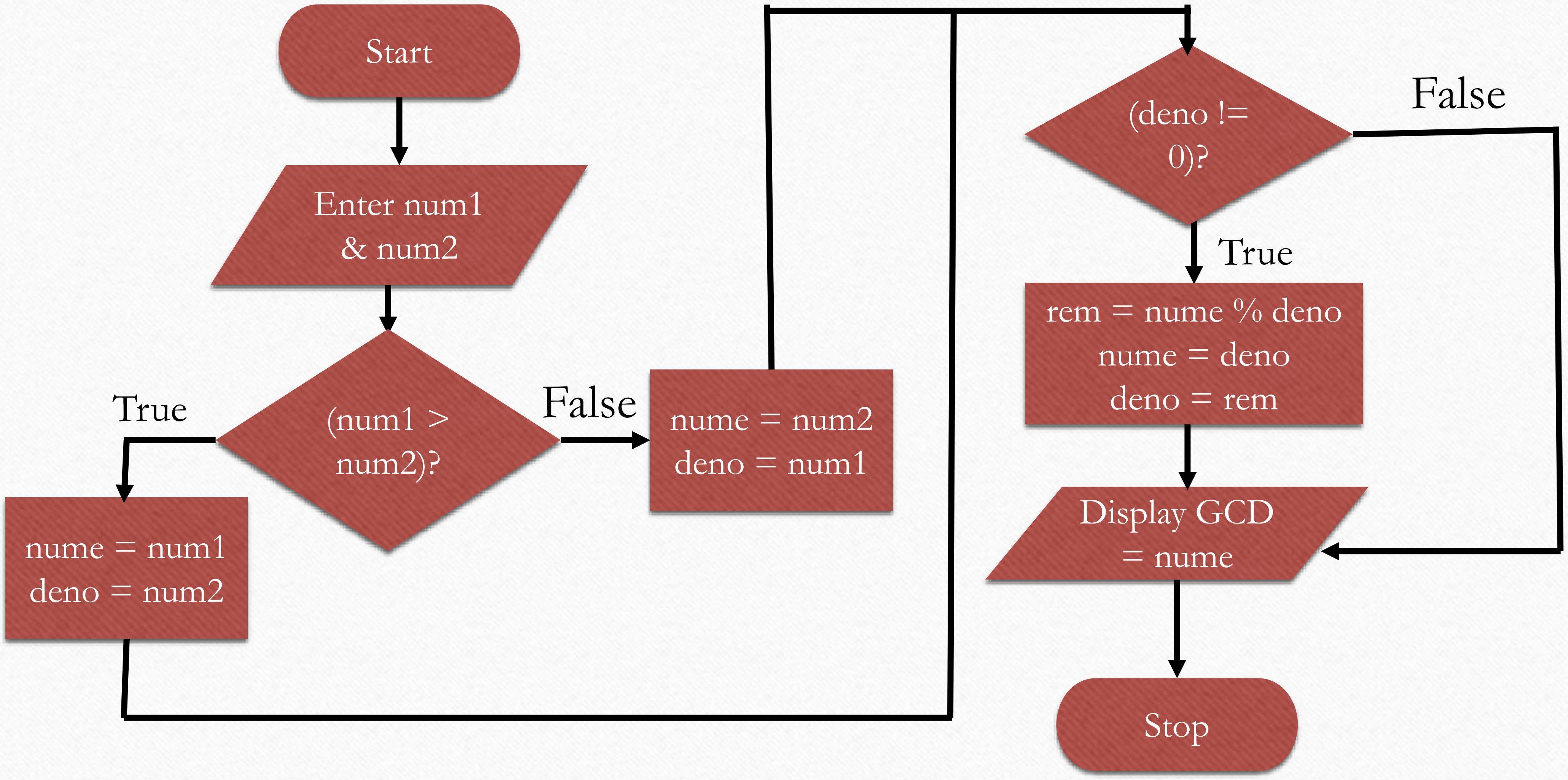


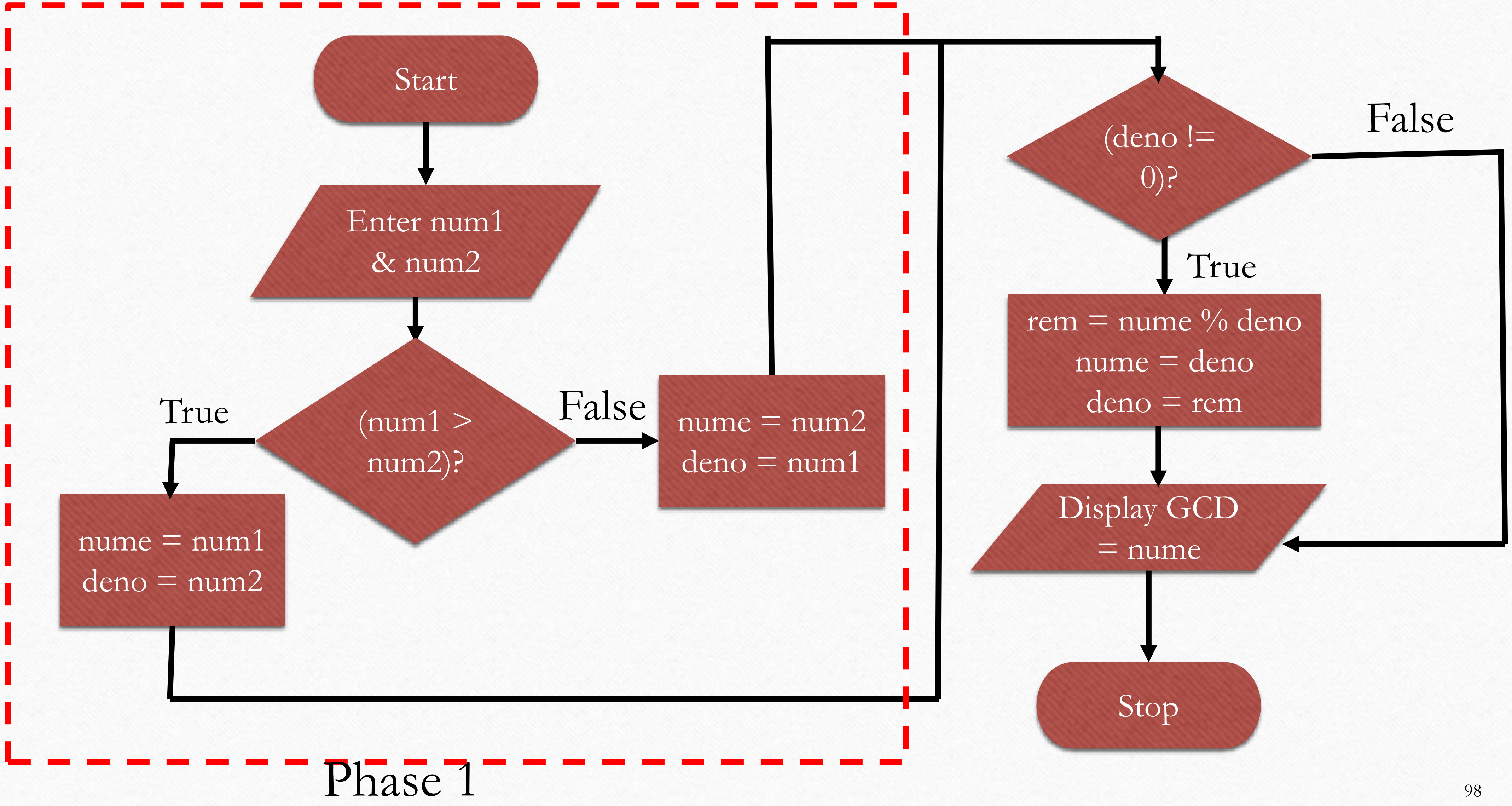
Start
Enter maximum value N
Initialize sum to 0, count to 0
for (1 <= i <= N)?
if (i % 2 == 0)?
False
True
sqr = pow(i, 2)
count++
sum += sqr
Display sum, count
Stop

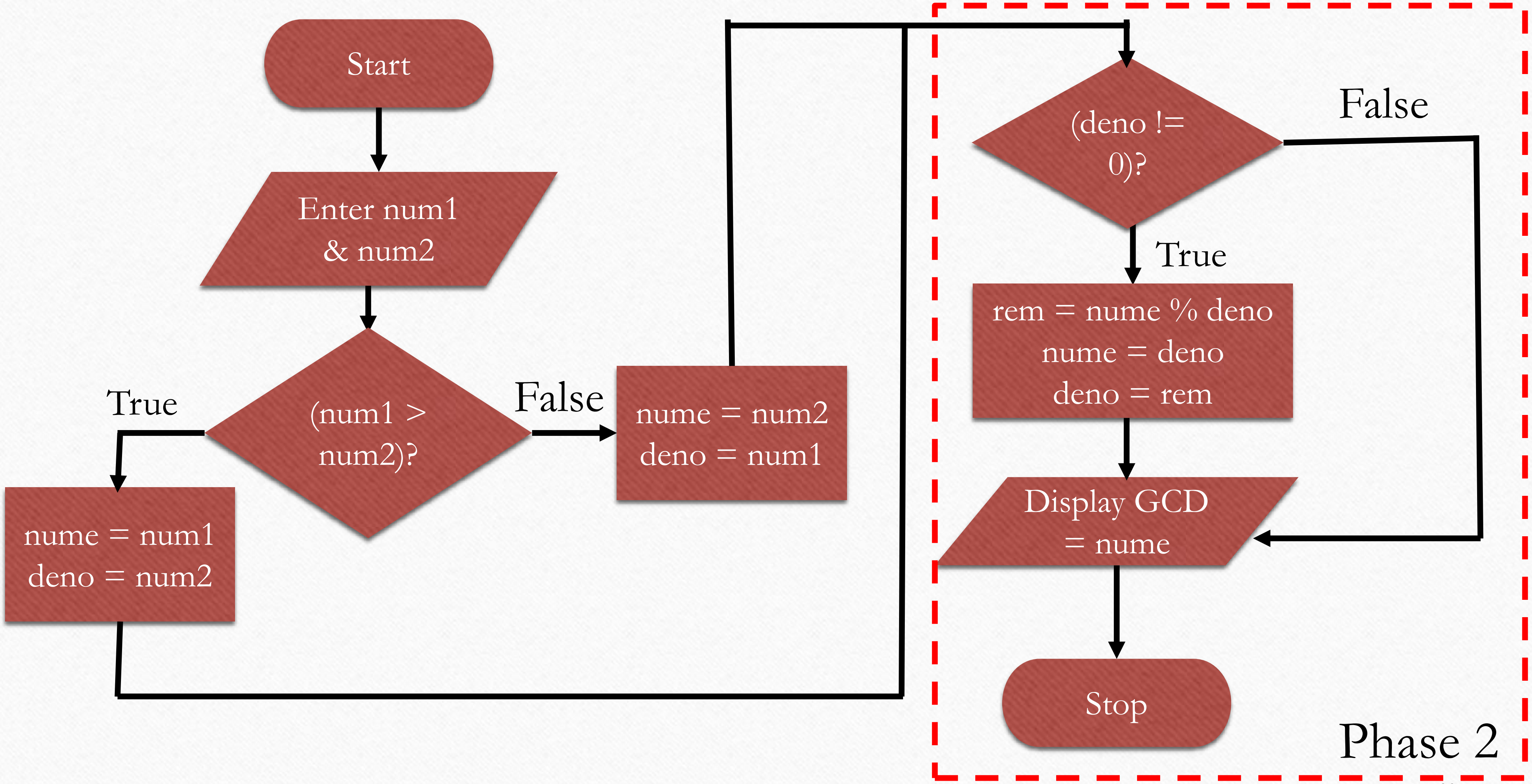
```cpp
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
cout << "Enter the value of N: ";
int N;
cin >> N;
int sum = 0, count = 0; int sqr;
for (int i = 1; i <= N; i++)
{
if (i % 2 == 0)
{
sqr = pow(i, 2);
count++;
sum += sqr;
}
else { }
}
cout << "Number of even numbers from 1 to "
<< N << " is " << count << endl;
cout << "Sum of squares of even numbers = "
<< sum << endl;

return 0;
}
```

Practice Problem 3

- Write a C++ program to calculate the GCD of two numbers
- Flowchart
- Program


Start
Enter num1 & num2
(num1 > num2)?
True
False
nume = num1
deno = num2
nume = num2
deno = num1
(deno != 0)?
False
True
rem = nume % deno
nume = deno
deno = rem
Display GCD = nume
Stop


Start
Enter num1 & num2
(num1 > num2)?
True
False
nume = num1
deno = num2
nume = num2
deno = num1
(deno != 0)?
False
True
rem = nume % deno
nume = deno
deno = rem
Display GCD = nume
Stop
Phase 1


Start
Enter num1 & num2
(num1 > num2)?
True
False
nume = num1
deno = num2
nume = num2
deno = num1
(deno != 0)?
False
True
rem = nume % deno
nume = deno
deno = rem
Display GCD = nume
Stop
Phase 2

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter the two numbers: ";
int num1, num2;
cin >> num1 >> num2;

int nume, deno, rem;

if (num1 > num2)
{
    nume = num1;  deno = num2;
}
else
{
    nume = num2; deno = num1;
}
```

```cpp
while (deno != 0)
{
    rem = nume % deno;
    nume = deno;
    deno = rem;
}

cout << "GCD of " << num1 << " and " << num2
<< " is = " << nume << endl;

return 0;
}
```

Practice Problem 4

- A class has 3 students. Each student must read 3 subjects. Write a C++ program to read the marks obtained by each student in each subject. Also calculate the total marks obtained by each student and average marks obtained in each subject.

Student ID	**Subject 1 (100)**	**Subject 2 (100)**	**Subject 3 (100)**
Student 1	75	80	55
Student 2	80	75	63
Student 3	70	64	42

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

#define ROWS 3
#define COL 3
int main()
{
int student[ROWS][COL];
int total;
float avg;

for (int i = 0; i < ROWS; i++)
{
cout << "Student " << (i + 1) << endl;
for (int j = 0; j < COL; j++)
{
cout << "Enter the marks of subject " << (j + 1) << endl;
cin >> student[i][j];
}
}
// calculate total marks obtained by each student
for (int i = 0; i < ROWS; i++)
{
total = 0;
for (int j = 0; j < COL; j++)
total += student[i][j];
cout << "Total marks of student " << (i + 1) << " is " << total << endl;
}
// calculate average marks obtained in each subject
for (int i = 0; i < ROWS; i++)
{
total = 0;
for (int j = 0; j < COL; j++)
total += student[j][i];
avg = (float)total / ROWS;
cout << "Average marks in subject " << (i + 1) << " is " << avg << endl;
}

return 0;
}
```

Functions

- Three parts
- Function declaration
- Function call
- Function definition

Passing Parameters To Function

- There are three ways in which arguments can be passed to the called function.
  - Call-by value
  - Call-by reference
  - Call-by address

Call-by Value

- A C++ program to add two integer numbers (Program Example 19).

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

int sum(int x, int y); // Function declaration

int main()
{
cout << "Enter two numbers: ";
int num1, num2;
cin >> num1 >> num2;
int res = sum(num1, num2);  // Function call
cout << "The result is " << res << endl;
return 0;
}

// Function definition
int sum(int x, int y)
{
return (x + y);
}
```

Call-by Value

- Disadvantage
  - It takes time to copy the data whenever a function is called.
- Solution to this problem
  - Call-by reference

Call-by Reference

- A C++ program to add two integer numbers (Program Example 20).

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

int sum(int &x, int &y); // Function declaration

int main()
{
cout << "Enter two numbers: ";
int num1, num2;
cin >> num1 >> num2;
int res = sum(num1, num2);  // Function call
cout << "The result is " << res << endl;
return 0;
}

// Function definition
int sum(int &x, int &y)
{
return (x + y);
}
```

Call-by Reference

- A C++ program to add two integer numbers (Program Example 20).

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

int sum(int &x, int &y); // Function declaration

int main()
{
cout << "Enter two numbers: ";
int num1, num2;
cin >> num1 >> num2;
int res = sum(num1, num2);  // Function call
cout << "The result is " << res << endl;
```

```
return 0;
}

// Function definition
int sum(int &x, int &y)
{
   return (x + y);
}
```

Program Example 21

- A C++ program to find the biggest of three integers using functions (call-by value).

```cpp
#include <iostream>
using namespace std;

int max(int x, int y, int z); // Function
declaration

int main()
{
cout << "Enter three numbers: ";
int num1, num2, num3;
cin >> num1 >> num2 >> num3;

int res = max(num1, num2, num3);  // Function
call
cout << "The result is " << res << endl;

return 0;
}
```

```cpp
// Function definition
int max(int x, int y, int z)
{
int largest = 0;
largest = (x > y) ? (x > z ? x :
z) : (y > z ? y : z);
return largest;
}
```

Program Example 22

- A C++ program to find the biggest of three integers using functions (call-by reference).

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

int max(int &x, int &y, int &z); // Function
declaration

int main()
{
cout << "Enter three numbers: ";
int num1, num2, num3;
cin >> num1 >> num2 >> num3;

int res = max(num1, num2, num3);  // Function
call
cout << "The result is " << res << endl;

return 0;
}
```

```
// Function definition
int max(int &x, int &y, int &z)
{
int largest = 0;
largest = (x > y) ? (x > z ? x :
z) : (y > z ? y : z);
return largest;
}
```

Practice Program 5

- Write a C++ program to calculate C(n/r).
- Use function (call-by value)
- Formula: $n_{C_r} = \frac{n!}{(n-r)!r!}$
- Draw the flowchart
- Write the C++ code

```cpp
#include <iostream>
using namespace std;

int fact(int num); // Function declaration

int main()
{
cout << "Enter the values of n and r: ";
int n, r;
cin >> n >> r;

int d = n - r;
int res = fact(n)/(fact(d)*fact(r));  //
Function call
cout << "Result = " << res << endl;

return 0;
}
```

```cpp
// Function definition
int fact(int num)
{
int f = 1;
for (int i = num; i >= 1; i--)
    f = f * i;
return f;
}
```

Practice Program 6

- Write a C++ program to calculate P(n/r).
- Use function (call-by reference)
- Formula: $n_{P_r} = \frac{n!}{(n-r)!}$
- Draw the flowchart
- Write the C++ code

```cpp
#include <iostream>
using namespace std;

int fact(int &num); // Function declaration

int main()
{
cout << "Enter the values of n and r: ";
int n, r;
cin >> n >> r;

int d = n - r;
int res = fact(n) / fact(d);  // Function
call
cout << "Result = " << res << endl;

return 0;
}
```

```cpp
// Function definition
int fact(int &num)
{
int f = 1;
for (int i = num; i >= 1; i--)
    f = f * i;
return f;
}
```

Function Overloading

- It allows one function to perform different tasks.
- Example: int sum(int x, int y); double sum(double x, double y)
- Function overloading is called a static or compile-time polymorphism.
- In static or compile-time polymorphism, compiler compares the number and type of arguments used in function call to determine the appropriate function definition.

Function Overloading

```
int sum (int x, int y)
 { return (x + y); }
```

```
res = sum(5, 23); // function call
```

```
double sum (double x, double y)
{ return (x + y); }
```

Function Overloading

```
int sum (int x, int y)
 { return (x + y); }
```

?

```
res = sum(5, 23); // function call
```

Which function does the compiler execute?

```
double sum (double x, double y)
{ return (x + y); }
```

?

Function Overloading

```
int sum (int x, int y)
 { return (x + y); }
```

```
res = sum(5, 23); // function call
```

```
double sum (double x, double y)
 { return (x + y); }
```

Compiler compares number and type of arguments.
Both functions have same number of arguments but types are different.

Function Overloading

```
int sum (int x, int y)
  { return (x + y); }
```

Selected function for execution

```
res = sum(5, 23); // function call
```

```
double sum (double x, double y)
 { return (x + y); }
```

Compiler compares number and type of arguments.
Both functions have same number of arguments but types are different.

Caution ⚠

Functions that cannot be overloaded

```
int my_func() { return 1; }
char my_func() { return 'I'; }
```

Different return type ✗

```
int my_func(int *ptr);
char my_func(int ptr[]);
```

Pointer and array ✗

```
int my_func(int n);
char my_func(const int n);
```

const and variable parameters ✗

Caution ⚠

Functions that cannot be overloaded

```
int my_func(int n);
char my_func(int &n);
```

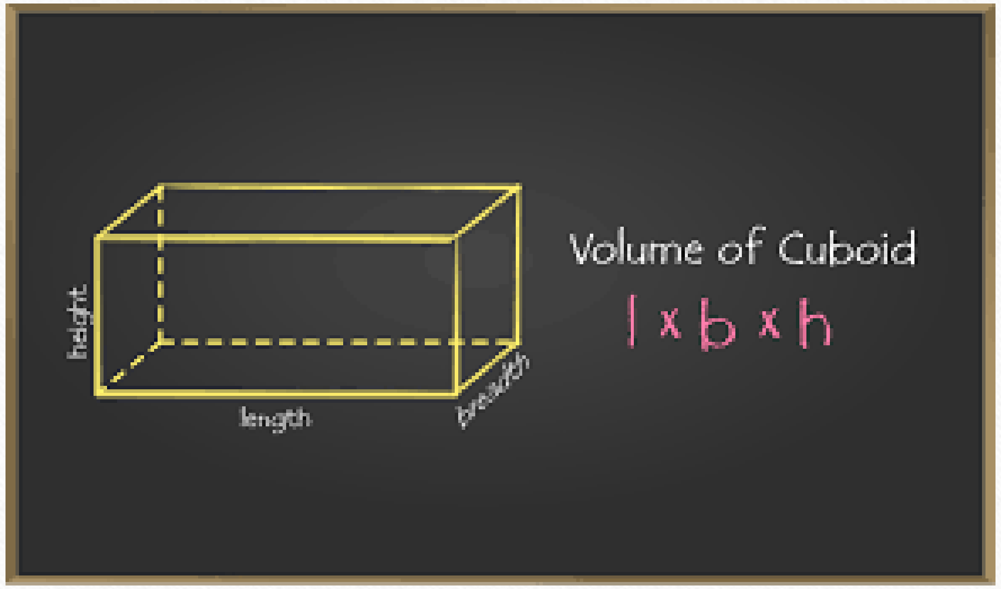
Normal and reference variable ✗

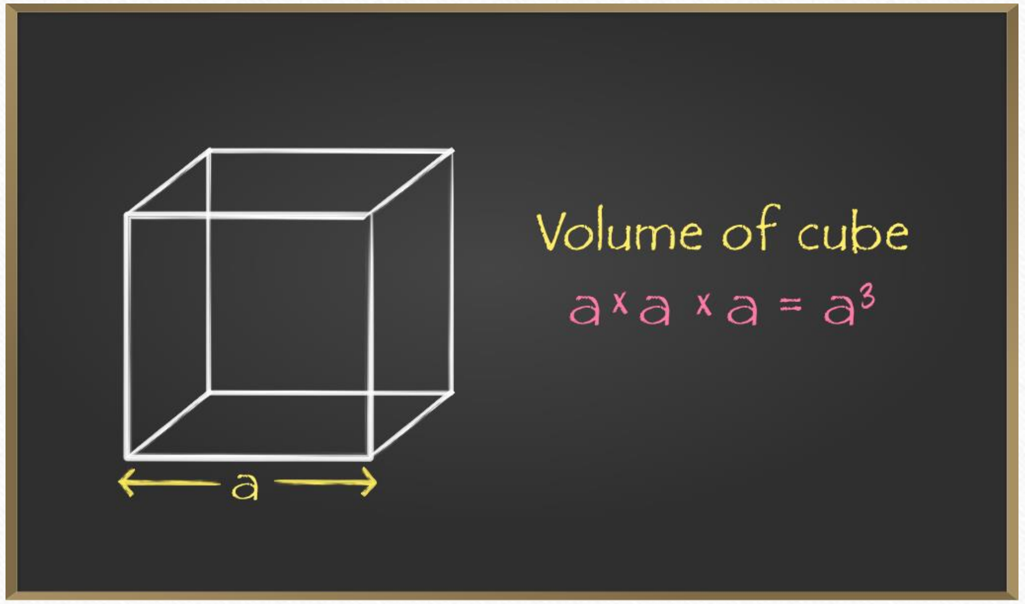
```
int my_func(int n);
char my_func(int n = 10);
```

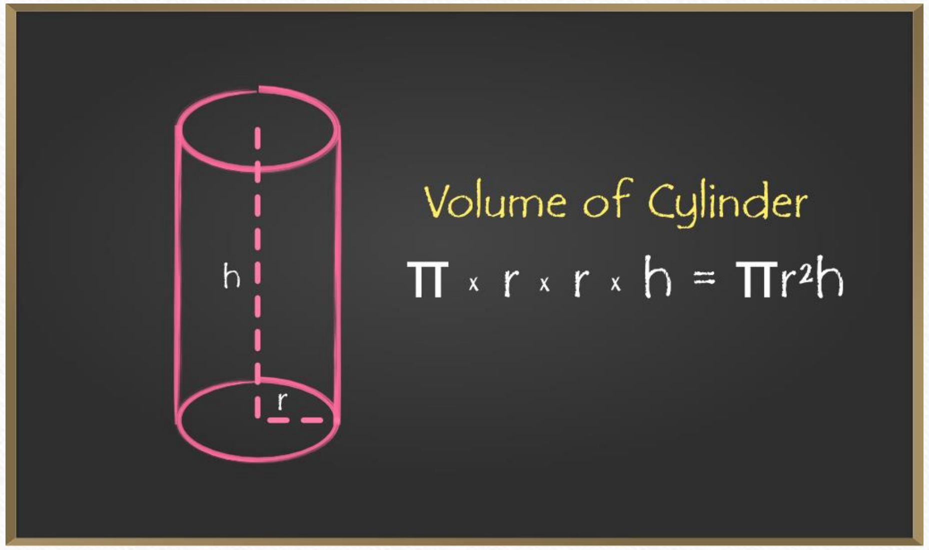
Variable and default value ✗

Program Example 23

- A C++ program to calculate the volume of a cube, a cylinder and a cuboid using function overloading.


height
length
breadth
Volume of Cuboid
l x b x h


a
Volume of cube
a x a x a = a³


h
r
Volume of Cylinder
π x r x r x h = πr²h

```cpp
#include <iostream>
using namespace std;

int volume(int side); // cube
float volume(float radius, float height); // cylinder
int volume(int length, int breadth, int height); // cuboid
 int main()
 {
 int s, l, b, h1;
 float r, h2;

 cout << "Enter the side of a cube: ";
 cin >> s;

 cout << "Enter the radius and height of a cylinder: ";
 cin >> r >> h2;

 cout << "Enter the length, breadth and height of a cuboid: ";
 cin >> l >> b >> h1;
```

```
  cout << "Volume of a cube: " << volume(s) << endl;
  cout << "Volume of a cylinder: " << volume(r, h2) << endl;
  cout << "Volume of a cuboid: " << volume(l, b, h1) << endl;

  return 0;
  }
// cube
int volume(int side)
{
    return (side * side * side);
}
// cylinder
float volume(float radius, float height)
{
    const float PI = 3.141;
    return (PI * radius * radius * height);
}
```

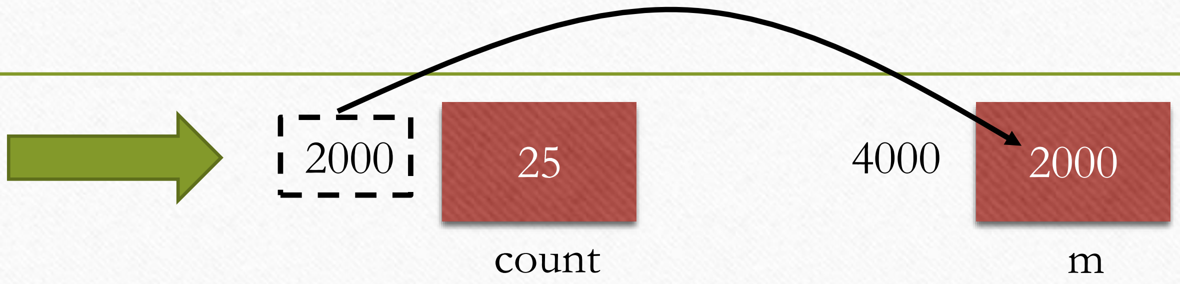
```
// cuboid
int volume(int length, int breadth, int height)
{
  return (length * breadth * height);
}
```

Pointers

- A pointer is a variable that holds a memory address of another variable.
- There are two special pointer operators: ***** and **&**
- **&** returns the memory address of its operands.
- ***** returns the value located at the address that follows.

Example

- m = &count;



variable m holds the address of the variable count.
m does not hold the content of count

- q = *m;

First content of m is accessed which is 2000.
Second content of location 2000 is accessed which is 25.
Finally 25 is stored in variable q

Find the Outputs

```
int num = 10;
int* p;
p = &num;

cout << *p << endl;
cout << p << endl;
```

Find the Outputs

```
int num = 10;
int* p;
p = &num;

cout << *p << endl;   →   10
cout << p << endl;
```

Find the Outputs

```
 int num = 10;
int* p;
p = &num;

cout << *p << endl;
cout << p << endl;
```

$\Rightarrow$ 10

$\Rightarrow$ **address of num**

Program Example 24

- A C++ program to increment a count value by 1. The range of the count should be 0 to 10. Use pointer operators.

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

int main()
{
int m = 0;
int* count = &m;

for (int i = 0; i <= 10; i++)
{
    cout << *count << endl;
    *count = *count + 1;
}

return 0;
}
```

Initializing a pointer variable

Incrementing the value of a pointer variable

Program Example 25

- A C++ program to add two integers using pointer operators.

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter two numbers: ";
int num1, num2;
cin >> num1 >> num2;

int* n1, * n2;
n1 = &num1;
n2 = &num2;

cout << "Sum = " << (*n1 + *n2) << endl;
return 0;
}
```

```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter two numbers: ";
int num1, num2, sum;
cin >> num1 >> num2;

int* n1, * n2, *psum;
n1 = &num1;
n2 = &num2;
psum = &sum;

*psum = (*n1 + *n2);
cout << "Sum = " << *psum << endl;
return 0;
}
```

Alternative code
Storing the result of addition in a third variable

Passing Pointers To Functions

- Call-by address

Program Example 26

- A C++ program to add two integers using function (pass function arguments as call-by address).

```cpp
#include <iostream>
using namespace std;

void add(int* x, int* y, int *t); // Function declaration

int main()
{
cout << "Enter two numbers: ";
int num1, num2, sum;
cin >> num1 >> num2;

add(&num1, &num2, &sum);
cout << "Sum = " << sum << endl;
return 0;
}
// Function definition
void add(int* x, int* y, int *t)
{
*t = *x + *y;
}
```

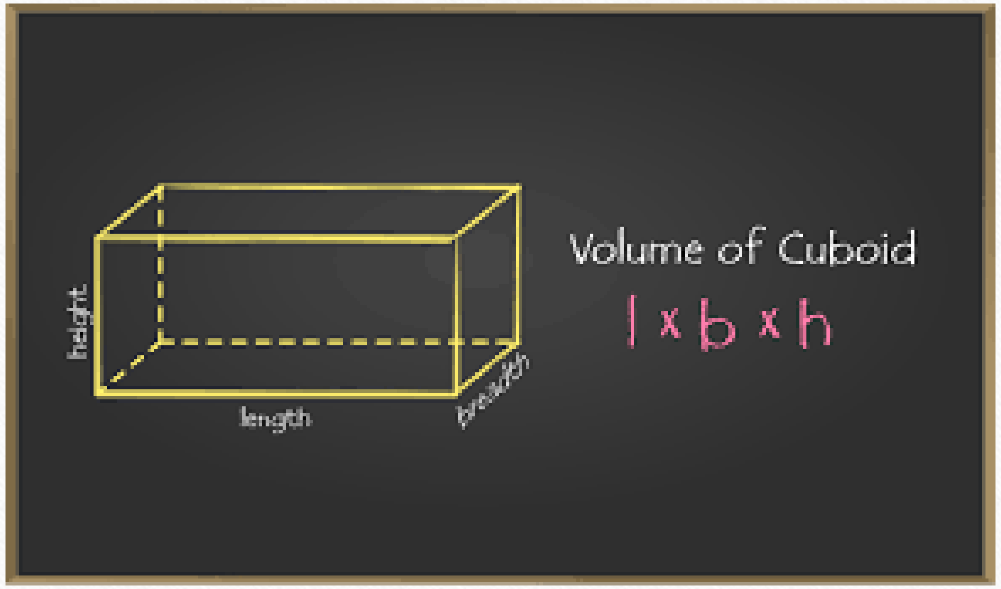
Program Example 27

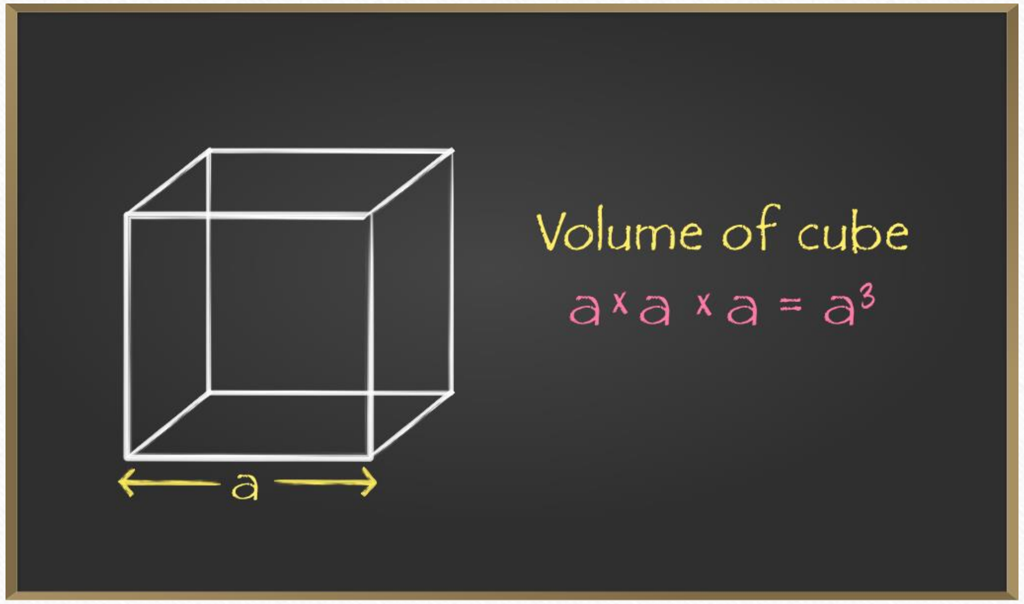
- A C++ program to find the biggest of three integers using functions (call-by address).

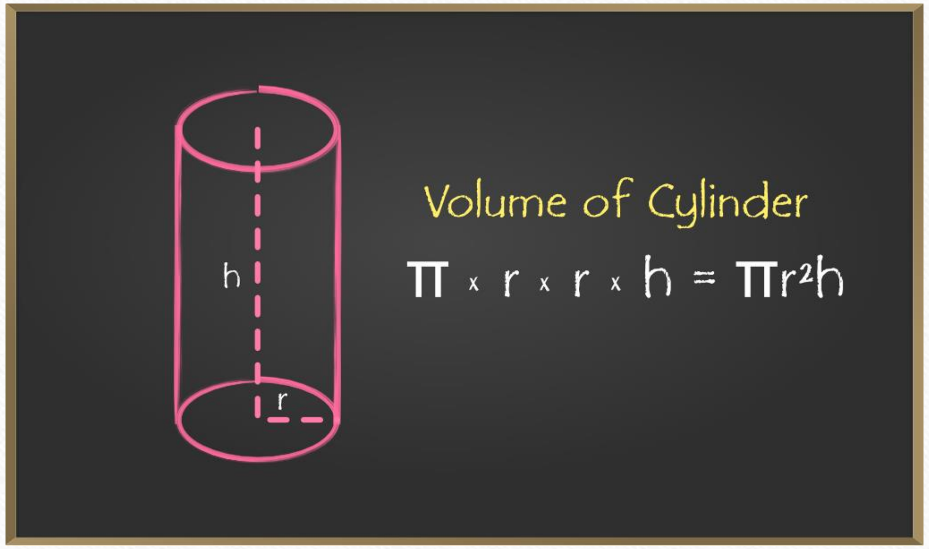
```cpp
#include <iostream>
using namespace std;

void largest(int* x, int* y, int* z, int *res); // Function declaration

int main()
{
cout << "Enter three numbers: ";
int num1, num2, num3, large;
cin >> num1 >> num2 >> num3;

largest(&num1, &num2, &num3, &large); // Function call
cout << "Largest = " << large << endl;
return 0;
}
// Function definition
void largest(int* x, int* y, int* z, int *res)
{
    *res = (*x > *y) ? (*x > *z ? *x : *z) : (*y > *z ? *y : *z);
}
```

Program Example 28

- A C++ program to calculate the volume of a cube, a cylinder and a cuboid using function overloading and pointers.


height
length
breadth
Volume of Cuboid
$l \times b \times h$


a
Volume of cube
$a \times a \times a = a^3$


h
r
Volume of Cylinder
$\pi \times r \times r \times h = \pi r^2 h$

```cpp
#include <iostream>
using namespace std;

int volume(int *side); // cube
float volume(float *radius, float *height); // cylinder
int volume(int *length, int *breadth, int *height); // cuboid
int main()
{
int s, l, b, h1;
float r, h2;

cout << "Enter the side of a cube: ";
cin >> s;

cout << "Enter the radius and height of a cylinder: ";
cin >> r >> h2;

cout << "Enter the length, breadth and height of a cuboid: ";
cin >> l >> b >> h1;
```

```cpp
  cout << "Volume of a cube: " << volume(&s) << endl;
  cout << "Volume of a cylinder: " << volume(&r, &h2) << endl;
  cout << "Volume of a cuboid: " << volume(&l, &b, &h1) << endl;

  return 0;
  }
// cube
int volume(int *side)
{
    return (*side * *side * *side);
}
// cylinder
float volume(float *radius, float *height)
{
    const float PI = 3.141;
    return (PI * *radius * *radius * *height);
}
```

```cpp
// cuboid
int volume(int *length, int *breadth, int
*height)
{
   return (*length * *breadth * *height);
}
```

Dynamic Memory Allocation in C++

- The process of allocating memory to the variables during execution of a program or at a run-time is known as dynamic memory allocation.
- We reserve space only at run-time for variables that are actually required.
- Dynamic memory allocation gives the best performance in situations when we do not know memory requirements in advance.
- For dynamic memory allocation, we need the pointers.

Memory Allocation Process

- In C++, the **new** operator is used for dynamic memory allocation.
- Syntax: **pointer_variable = new data_type**
- For example, if we want to allocate memory for an integer, then we can write

```
int *ptr;
ptr = new int;
```

Memory Allocation Process

- To allocate memory for an array, we write

```
int *arr = new int[n];
```

- A C++ program to allocate dynamic memory for an array (Program Example 29)

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

int main()
{
cout << "Enter the number of elements: ";
int n;
cin >> n;
int* arr = new int[n]; // dynamic array

cout << "Enter the array elements..." << endl;
for (int i = 0; i < n; i++)
    cin >> arr[i];

cout << "The array elements (after adding with 5) are ..." << endl;
for (int i = 0; i < n; i++)
    cout << (arr[i] + 5) << endl;

return 0;
}
```

During run-time, the size of the array will be allocated

Releasing the Used Memory Space

- When a variable is allocated memory space during compile time, the memory used by that variable is automatically released.
- When we dynamically allocate memory, it is our responsibility to release the space when it is not required.
- This can be done using **delete** operator.
- Syntax: **delete pointer_variable**; // for a single variable
- Syntax: **delete [] pointer_variable**; // for an array

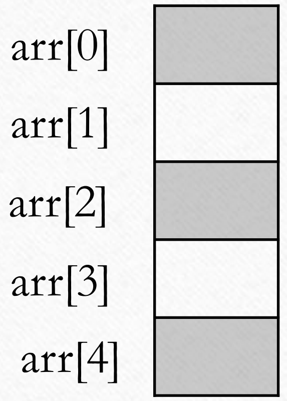
```cpp
#include <iostream>
using namespace std;

int main()
{
cout << "Enter the number of elements: ";
int n;
cin >> n;

int* arr = new int[n]; // dynamic array

cout << "Enter the array elements..." << endl;
for (int i = 0; i < n; i++)
    cin >> arr[i];

cout << "The array elements (after adding with 5) are ..." << endl;
for (int i = 0; i < n; i++)
    cout << (arr[i] + 5) << endl;
delete [] arr;  // de-allocating the space
arr = 0; // reset the pointer to null

if (arr == NULL)
    cout << "The array is deleted successfully!" << endl;

return 0;
}
```

← **Observe it carefully**

After deallocating the space, as a good programming practice, reset the pointer to null.

Dynamic Allocation of 2D Array

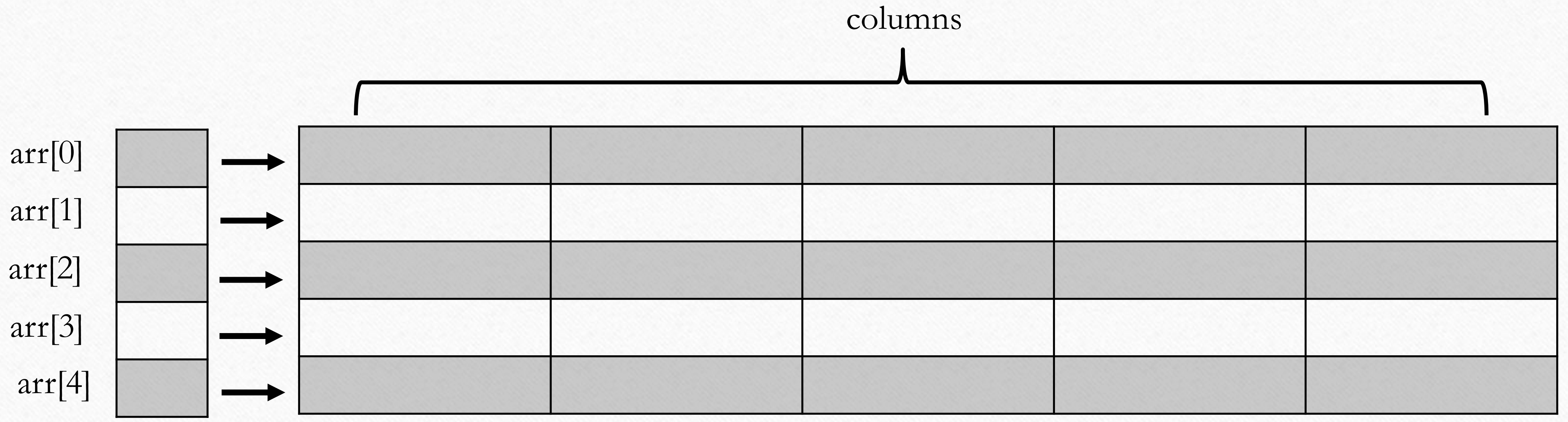
- A 2D array stored in memory in Row Major form.



Row major form

Dynamic Allocation of 2D Array

- A 2D array stored in memory in Row Major form.



Row major form

Program Example 30

- A C++ program to illustrate the dynamic allocation and deallocation of memory space for a 2D array

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

int main()
{
cout << "Enter the number of rows: ";
int row;
cin >> row;

cout << "Enter the number of columns: ";
int col;
cin >> col;

int **arr = new int *[row]; // creating rows of dynamic 2D array

if (arr == NULL)
{
    cout << "Memory could not be allocated:(";
    exit(1);
}
```

Observe it carefully

```
// dynamically allocates memory for columns in each row
for (int i = 0; i < row; i++)
    arr[i] = new int[col];
```

Observe it carefully

```
cout << "Enter the array elements..." << endl;
for (int i = 0; i < row; i++)
    for(int j = 0; j < col; j++)
        cin >> arr[i][j];

cout << "The array elements (after adding with 5) are ..." << endl;
for (int i = 0; i < row; i++)
    for(int j = 0; j < col; j++)
        cout << (arr[i][j] + 5) << endl;
```

```
// dynamically deallocates memory for columns in each row
for (int i = 0; i < row; i++)
    delete arr[i];                ←  Observe it carefully

delete[] arr;  // de-allocating the space
arr = 0; // reset the pointer to null      ←  Observe it carefully

if (arr == NULL)
    cout << "The array is deleted successfully!" << endl;

return 0;
}
```

Program Example 31

- A C++ program to create an array dynamically and find the maximum value among the elements. Release the memory space if the array is no longer required.

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

void maximum(int* arr, int size, int& large, int& pos); // function declaration

int main()
{
cout << "Enter the number of elements in an array: ";
int n;
cin >> n;

int* arr = new int[n]; // creating 1-d array dynamically

cout << "Enter the array elements..." << endl;
for (int i = 0; i < n; i++)
    cin >> arr[i];
int largest, index;
maximum(arr, n, largest, index);  // Function call
cout << "The largest number in the array is " << largest << " located at
index " << index << endl;

// dynamically deallocates memory for columns in each row
for (int i = 0; i < row; i++)
    delete arr[i];

delete[] arr;  // de-allocating the space
arr = 0; // reset the pointer to null

return 0;
}
```

```cpp
/ Function definition
void maximum(int* arr, int size, int& large, int& pos)
{
large = arr[0];
for(int i = 0; i < size; i++)
    if (arr[i] > large)
    {
        large = arr[i];
        pos = i;
    }
}
```

Practice Program 7

- Write a C++ program to calculate $XA + YB$, where A and B are two 2-d arrays that must be created dynamically. Also X and Y are two constants where, $X = 2$ and $Y = 3$.

Practice Program 8

- Write a C++ program to build an array of N random numbers in the range 1 to 100. Perform the following operations on the array:

a) Count the number of elements that are completely divisible by 3.

b) Count the number of even and odd elements.

c) Find the positions of the smallest and largest numbers in the array.

Hint: For the generation of random numbers within range 1 to 100, use int random = 1 + (rand() % 100);
Also use #include <cstdlib> header file

User-defined Data Types

- Structures (or struct)
- Enumerated (or enum)
- Class

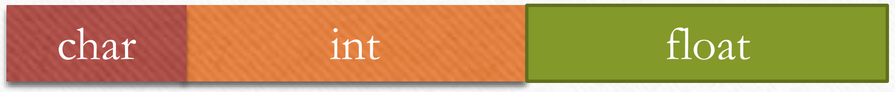
Structure

- One of the unique features of C language is structures.
- Structure allows us to pack together data of different types.

Structure

- One of the unique features of C language is structures.
- Structure allows us to pack together data of different types.
- Example:

```
struct student
{
  char name[20];
  int roll_number;
  float total_marks;
};
```



struct declares a new data type **student** that has three fields known as **structure members**.

Structure Variable Declaration

- We can create a variable of data type **student**

```
struct student stud1;  // C style declaration
```

- A is a variable of type **student** and has three member variables.
- C++ declaration of structure variable

```
student stud1;
```

Style 1

Style 2

```
struct student
{
   char name[20];
   int roll_number;
   float total_marks;
}stud1;
```

Accessing the Members

- Syntax of accessing a member of a structure:

struct_var.member_name;

- The member variables can be accessed using the dot operators

```
stud1.name = “John”;
stud1.roll_number = 999;
stud1.total_marks = 595.5;
```

```
cin.getline(stud1.name, 20);
cin >> stud1.roll_number;

cout << stud1.total_marks;
```

Initializing Structures

Method 1

```
struct student
{
   char name[20];
   int roll_number;
   float total_marks;
}stud1 = {“John”, 999, 595.5};
```

Method 2

```
struct student
{
   char name[20];
   int roll_number;
   float total_marks;
};
 student stud1 = {“John”, 999, 595.5};
```

Observation: All the member variables are initialized. This is called full initialization of structure variable.

Partial Initialization of Structures

Method 1

```
struct student
{
   char name[20];
   int roll_number;
   float total_marks;
}stud1 = {“John”, 999};
```

Method 2

```
struct student
{
   char name[20];
   int roll_number;
   float total_marks;
};
 student stud1 = {“John”, 999};
```

Observation: Member variable, *total_marks* remains un-initialized.

Limitations of C Structures

- The standard C does not allow the struct data type to be treated like built-in data types.
- For example,

```
struct complex
{
   float x; float y;
};
 struct complex c1, c2, c3;
```

- The complex numbers c1, c2, c3 can easily be assigned values using dot operators, but we cannot perform any operation on these variables directly.

Limitations of C Structures

- For example, c3 = c1 + c2; is illegal in C.
- Another important limitation of C structures is that they do not allow data hiding.
- Structure members can be accessed using structure variables by any functions.
- In other words, structure members are public members.

Extensions to Structures

- C++ supports all the features of structures as defined in C.
- But C++ has extended the capabilities further to suit its OOP philosophy.
- C++ structures or popularly known as **class** allows both variables and functions as class members, permits **data encapsulation** or **hiding** and provides a mechanism called **inheritance**.
- The only difference between a structure and a class is that, by default, the members of a class are private, whereas, by default, members of a structure are public.

Program Example 32

- Write a C++ program to enter two points and then calculate the distance between them. Use a structure called **point.**

```cpp
#include <iostream>
#include <cmath>
using namespace std;

struct point
{
    int x;
    int y;
};
float distance(int x1, int y1, int x2, int y2); // function declaration
int main()
{
    point p1, p2;
    cout << "Enter the coordinates of first point: ";
    cin >> p1.x >> p1.y;
    cout << "Enter the coordinates of second point: ";
    cin >> p2.x >> p2.y;
    cout << "Distance: " << distance(p1.x, p1.y, p2.x, p2.y) << endl;
    return 0;
}
```

```
// Function definition
float distance(int x1, int y1, int x2, int y2)
{
    return (sqrt(pow((x2 - x1), 2) + pow((y2 - y1), 2))) ;
}
```

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Program Example 33

- A class has 3 students. Each student must read 3 subjects. Write a C++ program to read the marks obtained by each student in each subject. Also calculate the total marks obtained by each student and average marks obtained in each subject. Apply the concept of structure.

Student Name	**Roll Number**	**Subject 1 (100)**	**Subject 2 (100)**	**Subject 3 (100)**
ABC	2001	75	80	55
XYZ	2002	80	75	63
LOC	2003	70	64	42

```cpp
#include <iostream>
#include <cmath>
using namespace std;

struct student
{
    char name[5];
    int roll;
    float sub1;
    float sub2;
    float sub3;
};

float total(float x, float y, float z); // function declaration
float avg(float x, float y, float z);  // function declaration
```

```
int main()
{
student stud1 = {"ABC", 2001};
student stud2 = { "XYZ", 2002 };
student stud3 = { "LOC", 2003 };

cout << "Enter the marks of three subjects (Student 1)...";
cin >> stud1.sub1 >> stud1.sub2 >> stud1.sub3;

cout << "Enter the marks of three subjects (Student 2)...";
cin >> stud2.sub1 >> stud2.sub2 >> stud2.sub3;

cout << "Enter the marks of three subjects (Student 3)...";
cin >> stud3.sub1 >> stud3.sub2 >> stud3.sub3;
cout << "Total marks of student 1: " << total(stud1.sub1, stud1.sub2,
stud1.sub3) << endl;
cout << "Total marks of student 2: " << total(stud2.sub1, stud2.sub2,
stud2.sub3) << endl;
cout << "Total marks of student 3: " << total(stud3.sub1, stud3.sub2,
stud3.sub3) << endl;

cout << "Average marks in subject 1: " << avg(stud1.sub1, stud2.sub1,
stud3.sub1) << endl;
cout << "Average marks in subject 2: " << avg(stud1.sub2, stud2.sub2,
stud3.sub2) << endl;
cout << "Average marks in subject 3: " << avg(stud1.sub3, stud2.sub3,
stud3.sub3) << endl;

return 0;
}
```

```
// Function definition
float total(float x, float y, float z)
{
    return (x + y + z);
}

float avg(float x, float y, float z)
{
    return ((x + y + z) / 3.0);
}
```

Enumerated Data Types

- The enumerated data type is a user-defined type based on the standard integer type.
- In enumerated data type, each integer value is assigned an identifier.
- This identifier is known as enumeration constant.

Enumerated Data Types

- To define enumerated data types, we use the keyword **enum.**
- Syntax:

```
enum enumeration_name {identifier1, identifier2, … , identifierN};
```

- Example:

```
enum COLORS {Violet, Indigo, Blue, Green, Yellow, Orange, Red};
```

Enumerated Data Types

```
enum COLORS { Violet, Indigo, Blue, Green, Yellow, Orange, Red};
```

- Note that no built-in data types are used in the above example.
- A new data type called COLORS is now created.
- Here, COLORS is the name given to the set of constants.
- The default values in our case are: Violet = 0, Indigo = 1, Blue = 2, Green = 3, Yellow = 4, Orange = 5, Red = 6

Enumerated Data Types

- Explicit values can be assigned to these integer constants.

```
enum COLORS {Violet = 6, Indigo = 1, Blue = 4, Green = 3, Yellow = 5,
                    Orange = 7, Red = 2};
```

- Example (in our case): Violet = 6, Indigo = 1, Blue = 4, Green = 3, Yellow = 5, Orange = 7, Red = 2
- The value of an enumerator constant is always of **int** type.

Enumerated Data Types

- The following rules apply to the members of an enumeration list:
    - An enumeration list may contain duplicate constant values. Therefore, two different identifiers may be assigned the same value, say, Violet = 7, Red = 7.
    - Every enumeration name must be different from the other enumeration name within the same scope and visibility.
    - The identifiers in the enumeration list must be different from other identifiers in the same list and same scope.

Enum Variables

- The syntax for declaring a variable of an enumerated data type is

```
enumeration_name variable_name;
```

- To create a variable of COLORS, we can write

```
COLORS background;
```

- Alternative way to declare a variable of COLORS

```
enum COLORS {Violet, Indigo, Blue, Green, Yellow, Orange, Red} background;
```

Program Example 34

```
#include <iostream>
using namespace std;
int main()
{
enum COLORS {Violet, Indigo, Blue, Green, Yellow, Orange, Red};
cout << "Violet: " << Violet << endl;
cout << "Indigo: " << Indigo << endl;
cout << "Blue: " << Blue << endl;
cout << "Green: " << Green << endl;
cout << "Yellow: " << Yellow << endl;
cout << "Orange: " << Orange << endl;
cout << "Red: " << Red << endl;
return 0;
}
```

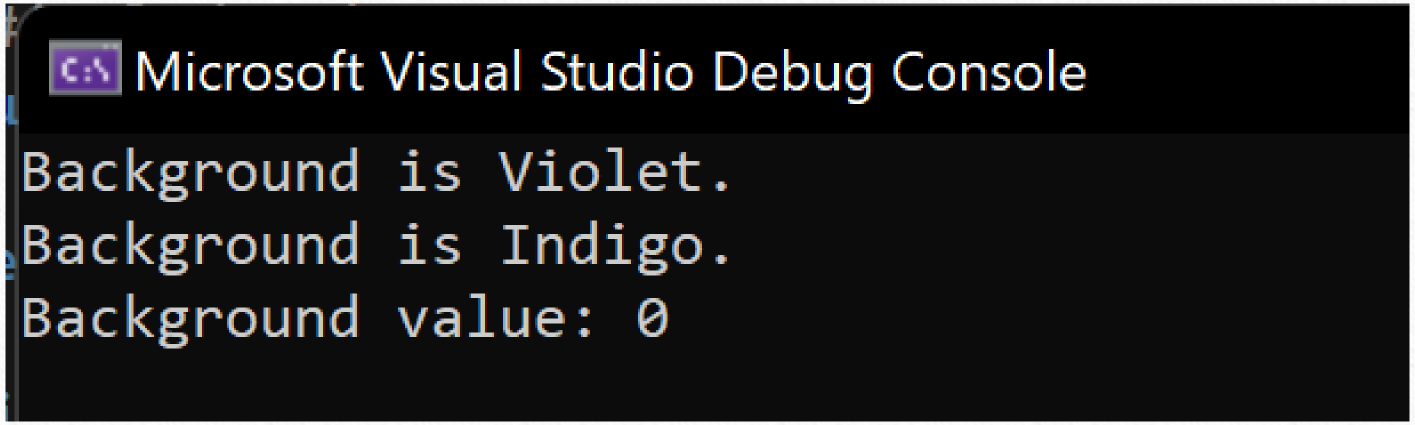
Program Example 35

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

enum COLORS { Violet, Indigo, Blue, Green, Yellow, Orange, Red };

int main()
{
COLORS background;
background = Red;
cout << background << endl;
return 0;
}
```

Assigning values to an enumerated data type

Program Example 36

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

enum COLORS { Violet, Indigo, Blue, Green, Yellow, Orange, Red } background;

int main()
{
    switch (background)
    {
        case Violet: cout << "Background is Violet." << endl;
        case Indigo: cout << "Background is Indigo." << endl; break;
        default: cout << "No background color!" << endl; break;
    }
cout << "Background value: " << background << endl;
return 0;
}
```

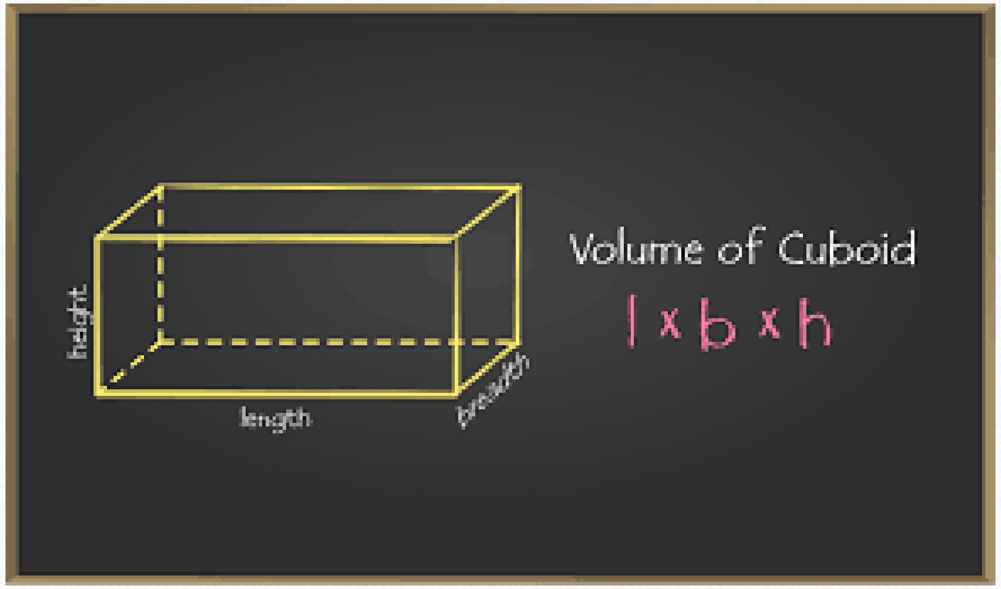


Microsoft Visual Studio Debug Console

```
Background is Violet.
Background is Indigo.
Background value: 0
```

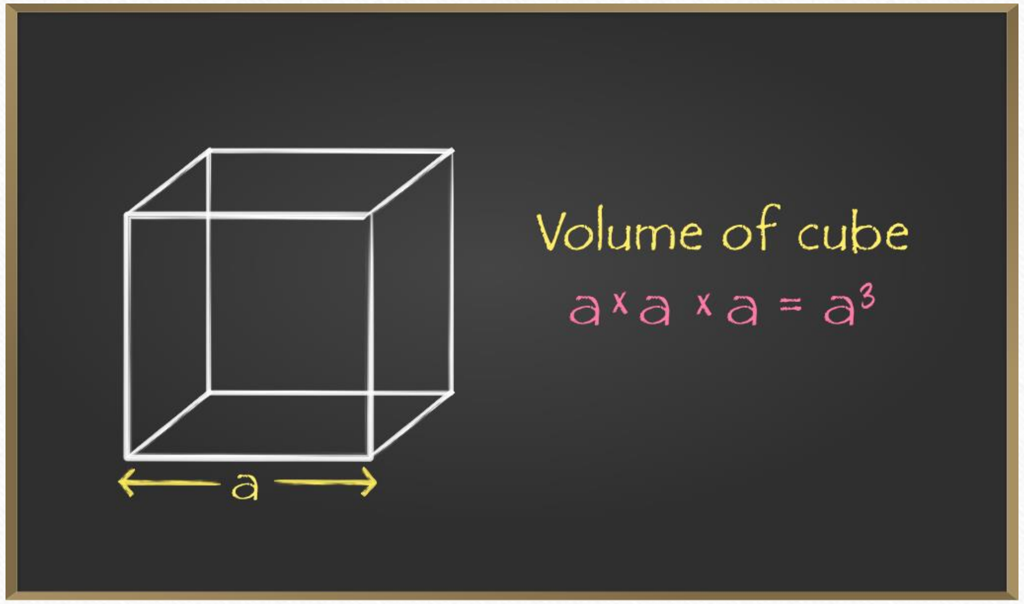
Program Example 37

- Define an enumerated data type called SHAPE which contains cube, cylinder and cuboid as enumerated items. Write a C++ program to calculate the volume of each enumerated item using suitable functions.



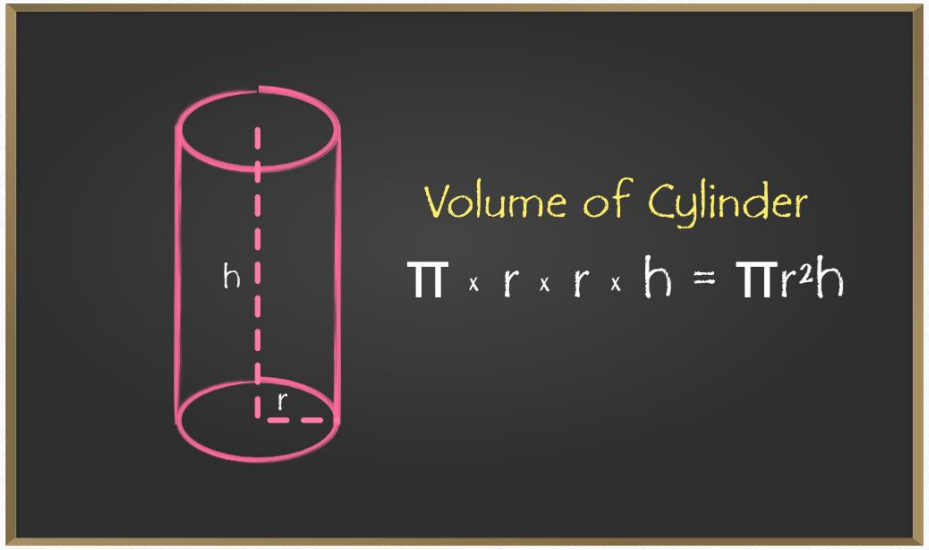
Volume of Cuboid

$l \times b \times h$



Volume of cube

$a \times a \times a = a^3$



Volume of Cylinder

$\pi \times r \times r \times h = \pi r^2 h$

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int volume(int side); // cube
float volume(float radius, float height); // cylinder
int volume(int length, int breadth, int height); // cuboid

enum SHAPE {cube, cylinder, cuboid} sh;

int main()
{
    switch (sh)
    {
        case cube: cout << "Enter the side of a cube: "; int s;
                   cin >> s;
                   cout << "Volume of a cube: " << volume(s) << endl;
    case cylinder: cout << "Enter the radius and height of a cylinder: ";
float r, h2; cin >> r >> h2;
         cout << "Volume of a cylinder: " << volume(r, h2) << endl;
case cuboid: cout << "Enter the length, breadth and height of a cuboid: ";
int l, b, h1;
cin >> l >> b >> h1;
cout << "Volume of a cuboid: " << volume(l, b, h1) << endl; break;
default: cout << "Something went wrong!" << endl; break;
}
return 0;
}
```

```
// cube
int volume(int side)
{
    return (pow(side, 3));
}
// cylinder
float volume(float radius, float height)
{
    const float PI = 3.141;
    return (PI * pow(radius, 2) * height);
}
```

```
// cuboid
int volume(int length, int breadth, int height)
{
    return (length * breadth * height);
}
```

Summary of Module 1

- C++ operators
- Conditional statements
- Three methods of function calls
- Function overloading
- Dynamic memory allocation
- Pointers and reference
- Structure and enumerated data types

Introduction to Object Oriented Programming (OOP)

Module 1

OOP (IT 2005)

4th Semester ECSc