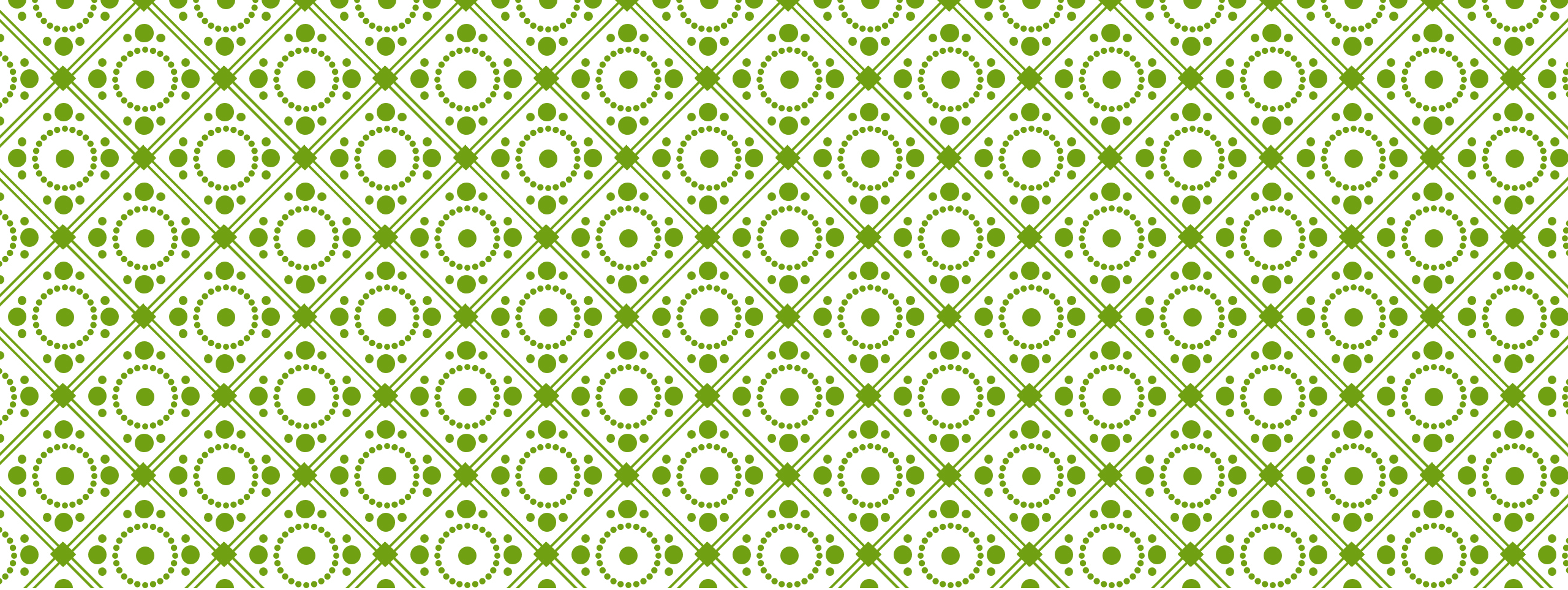


BASIC C++ PROGRAMMING

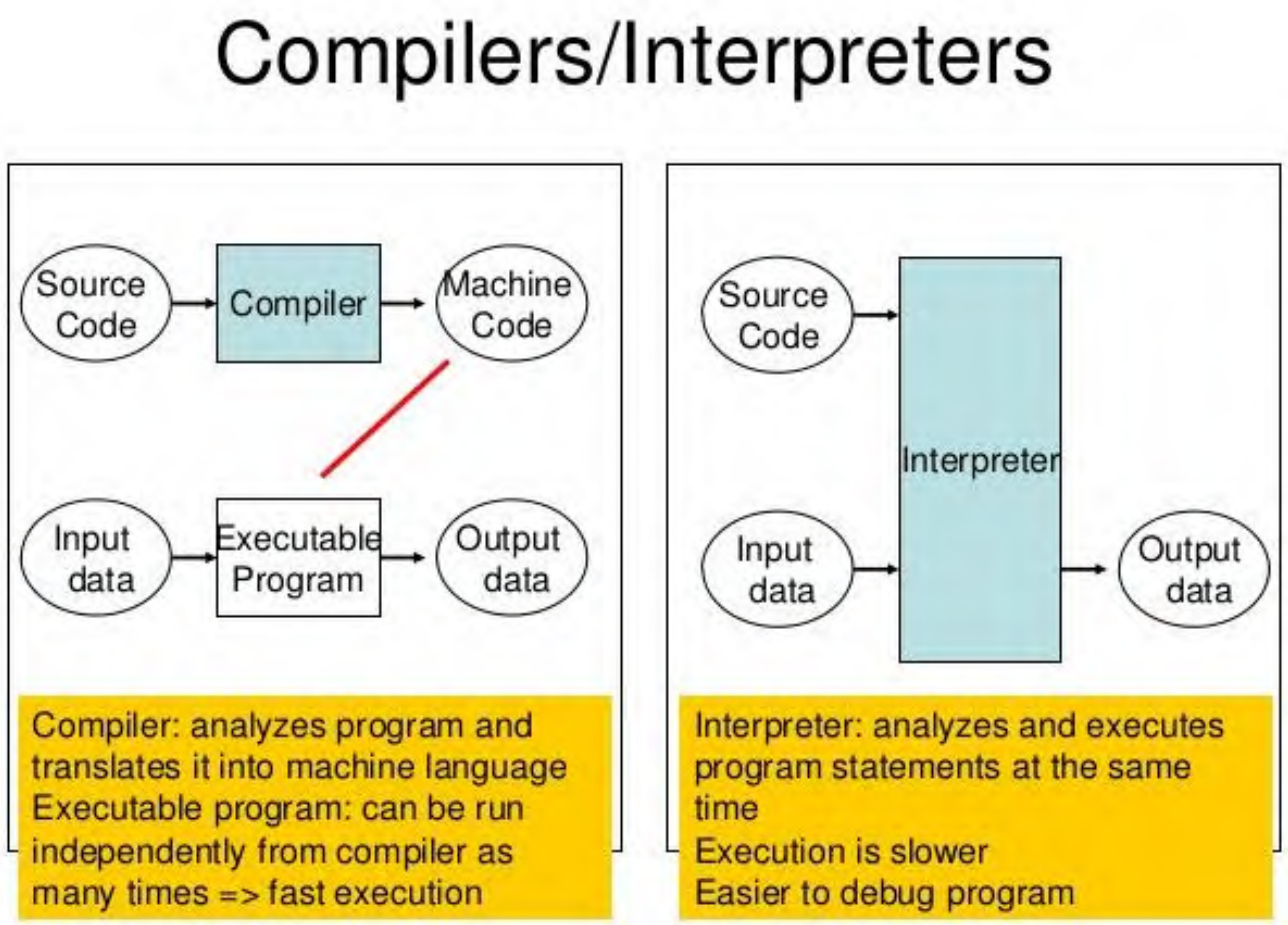
Elementary level

THE “HELLO, WORLD” PROGRAM

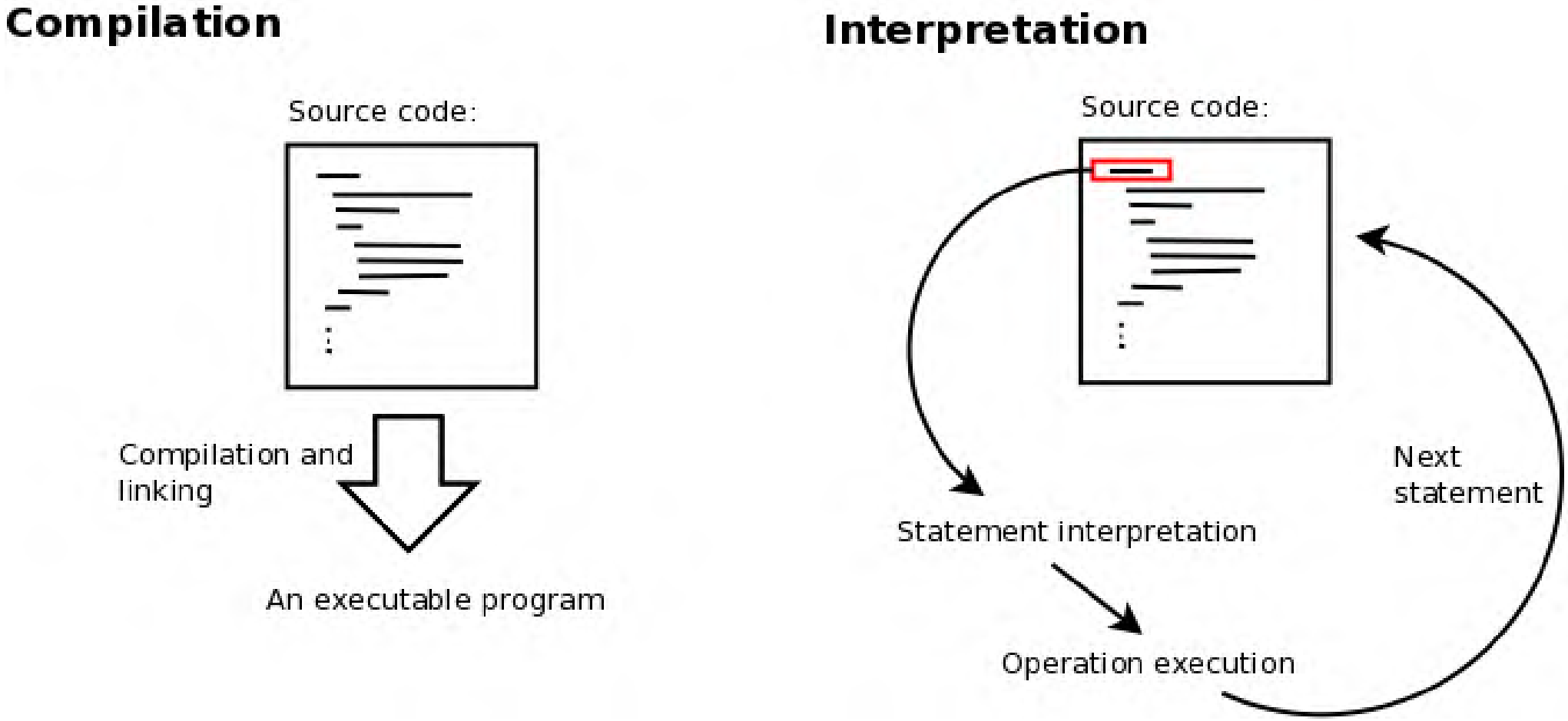
```
#include <iostream>
int main()
{
        std::cout <<
Hello,World!\n";
}
```

- A program -- a sequence of instructions that can be executed by a computer
- A compiler -- a software system that translates programs into the machine language (called binary code)
- IDEs (Integrated Development Environments) --- include their own specialized text editors and debuggers
- C++ is case-sensitive
- preprocessor directive -- #include
- identifier iostream -- the name of a file in the Standard C++ Library
- standard header -- iostream
- C++ program begins from the main function of the program

COMPILER VERSUS INTERPRETER



COMPILER VERSUS INTERPRETER



THE “HELLO, WORLD” PROGRAM

```
#include <iostream>
int main()
{
        std::cout <<
Hello,World!\n";
}
```

- Every C++ program must have one and only one main() function
- A program body is a sequence of program statements enclosed in braces { }
- standard output stream object (std::cout) ----- single symbol << represents the C++ output operator
- \n represent the newline character
- the preprocessor directive must precede the program on a separate line

MORE BASICS

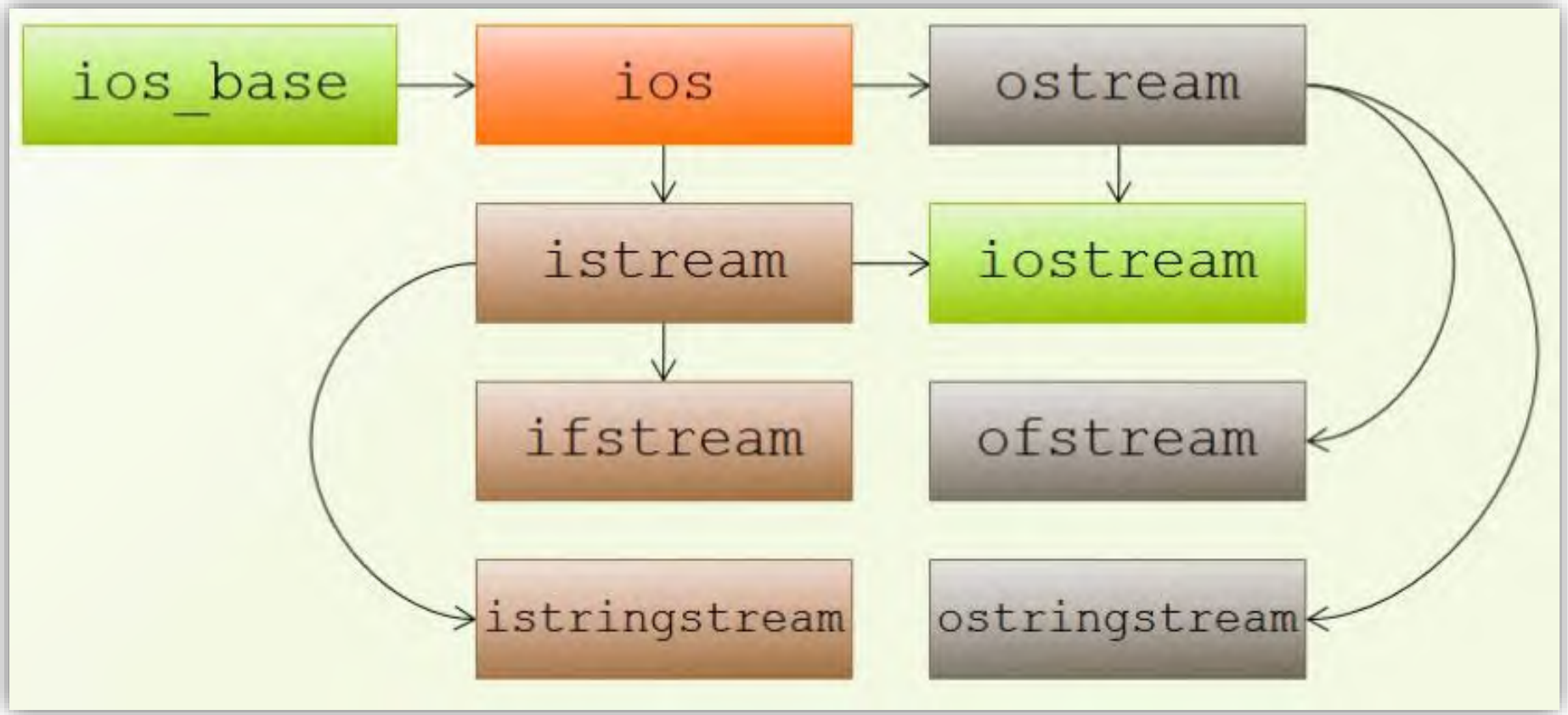
- A *comment* in a program --- a string of characters that the preprocessor removes before the compiler compiles the programs
- A *C style comment* --- any string of characters between the symbol /* and the symbol */
- A *namespace* -- a named group of definitions
- When objects that are defined within a namespace are used outside of that namespace,
  - either their names must be prefixed with the name of the namespace or
  - they must be in a block that is preceded by a using namespace statement

```
#include <iostream>
using namespace std;
/*C style

comment*/

int main()
{
        // prints "Hello, World!":
        cout << "Hello,World!\n";
        return 0;
}
```

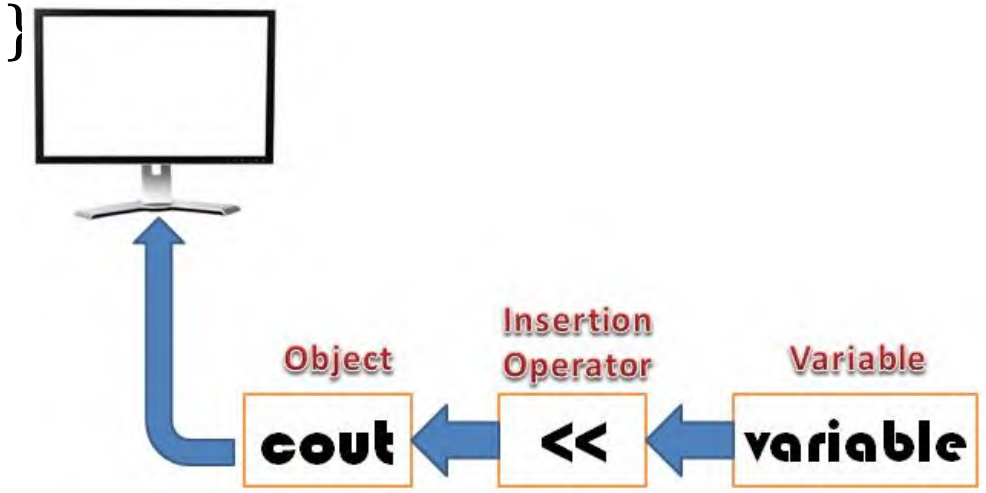
C++ BASIC INPUT/OUTPUT CLASSES



THE OUTPUT OPERATOR

- The symbol << is called the *output operator* in C++ (also called *put operator* or *stream insertion operator*)
- An *operator* is something that performs an action on one or more objects
- The output operator << performs the action of sending the value of the expression listed on its right to the output stream listed on its left
- The cout object is called a “stream” (output sent to it flows like a stream)

```
int main()
{
    // prints "Hello, World!":
    cout << "Hello,World!\n";
    return 0;
}
```



YET ANOTHER “HELLO, WORLD” PROGRAM

```
 int main()
 {
        // prints "Hello, World!":
        cout << "Hel" << "lo, Wo" << "rld!" << endl;
}
 int main()
 {
        // prints "Hello, World!":
        cout << "Hello, W" << 'o' << "rld" << '!' <<
'\n';
}
```

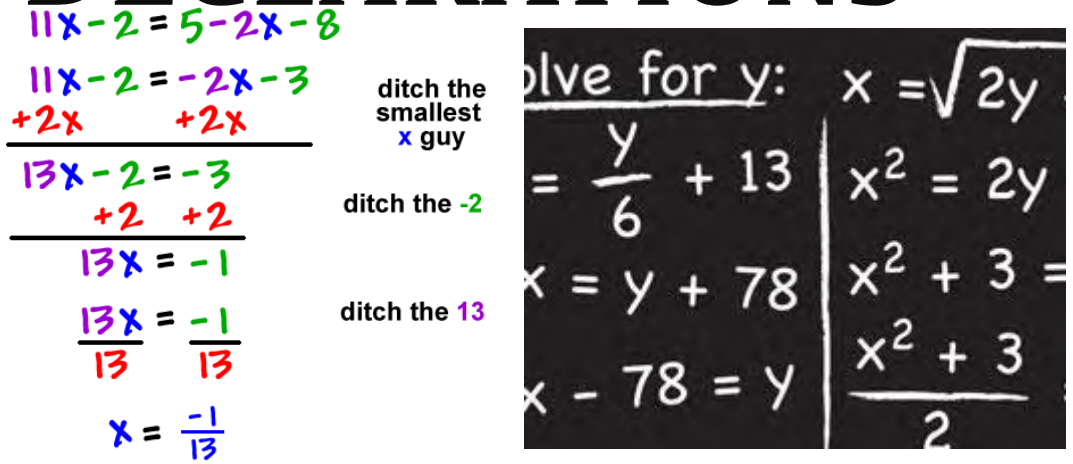
- The first three are strings that are concatenated together (*i.e.*, strung end-to-end) to form the single string "Hello,World!“
- *stream manipulator* object endl (meaning “end of line”)
- The three objects "Hel", "lo, Wo", and "rld!" are called *string literals*
- *horizontal tab* character '\t'
  alert character '\a'.
  Vertical tab character ‘\v’

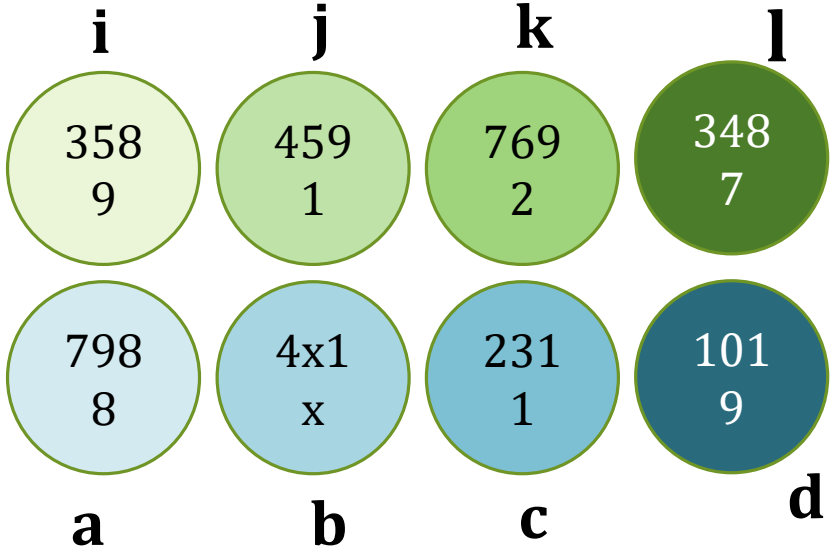
EXAMPLE 1.5 INSERTING NUMERIC LITERALS INTO THE STANDARD OUTPUT STREAM

```
int main()
{
        // prints "The Millennium ends Dec 31 2000.":
        cout << "The Millennium ends Dec " << 3 << 1 << ' ' << 2000 << endl;
}
```

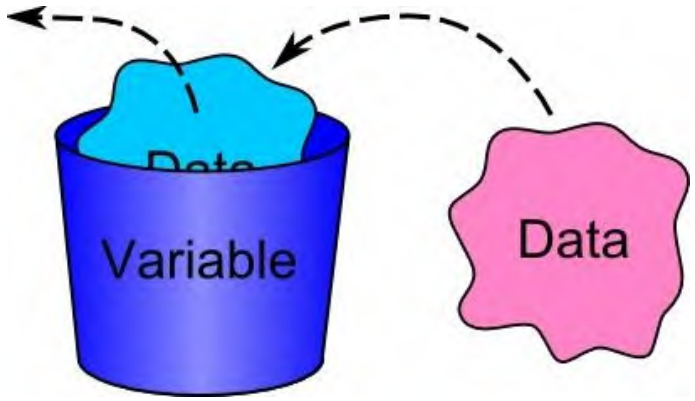
- ✓When numeric literals like 3 and 2000 are passed to the output stream, they are automatically
  - converted to string literals and
  - concatenated the same way as characters

1.5 VARIABLES AND THEIR DECLARATIONS





- A *variable* -- a symbol that represents a storage location in the computer's memory
- The information that is stored in that location is called the *value* of the variable.
- *variable* = *expression*;
  - First the *expression* is evaluated and
  - the resulting value is assigned to the *variable*



EXAMPLE 1.6 USING INTEGER VARIABLES

```
int main()
{
        // prints "m = 44 and n = 77":
        int m, n;
        m = 44; // assigns the value 44 to the variable m
        cout << "m = " << m;
        n = m + 33; // assigns the value 77 to the variable n
        cout << " and n = " << n << endl;
}
```

- Every variable in a C++ program must be declared before it is used
- The syntax: *specifier type name initializer*;
- The location of the declaration within the program determines the *scope* of the variable

1.6 PROGRAM TOKENS

- A computer program is a sequence of elements called *tokens*
- These tokens include keywords such as int, identifiers such as main, punctuation symbols such as {, and operators such as <<
- Token helps in finding errors, syntax errors and gives error messages after unsuccessful compilation

EXAMPLE 1.7 A PROGRAM'S TOKENS

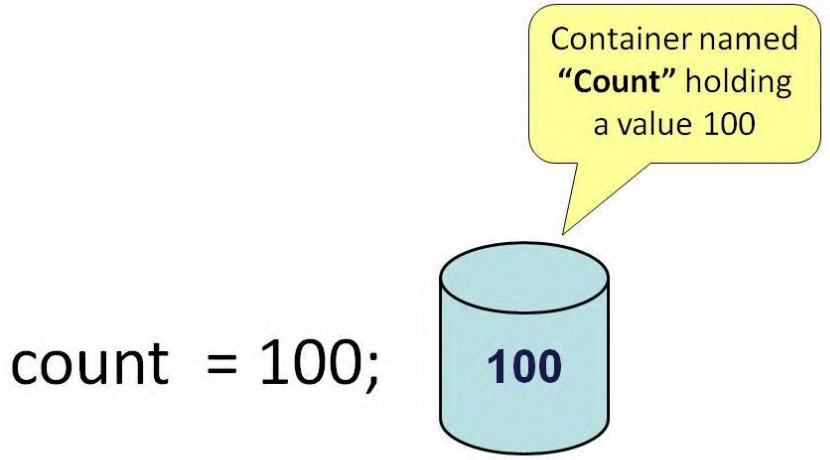
```
int main()
{
        // prints "n = 44":
        int n = 44;
        cout << "n = " << n << endl;
}
```

EXAMPLE 1.8 AN ERRONEOUS PROGRAM

```
int main()
{
        // THIS SOURCE CODE HAS AN ERROR:
        int n = 44
        cout << "n = " << n << endl;
}
```

1.7 INITIALIZING VARIABLES

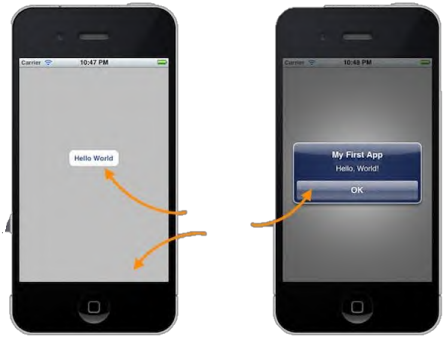
In most cases it is wise to initialize variables where they are declared.

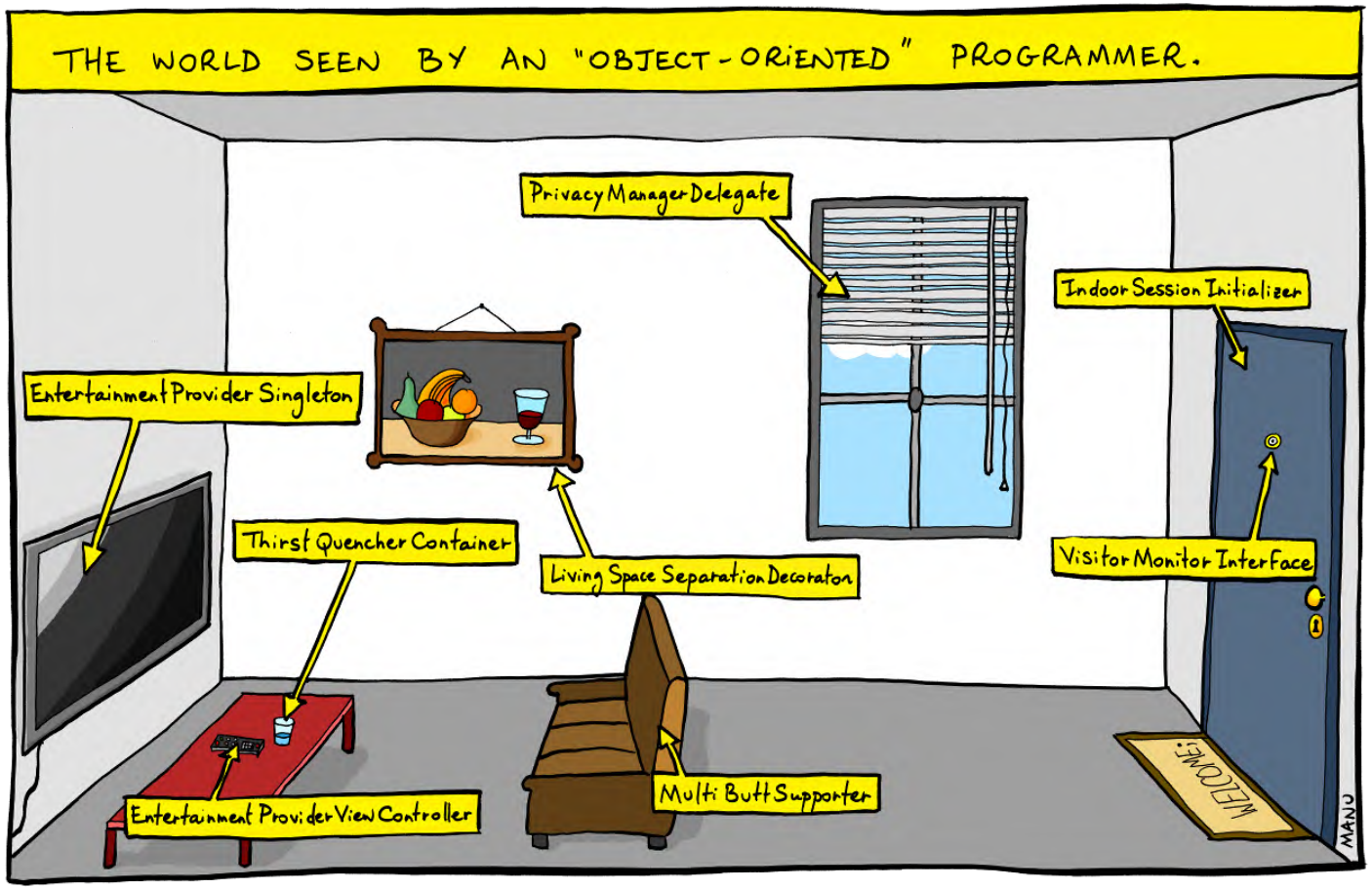


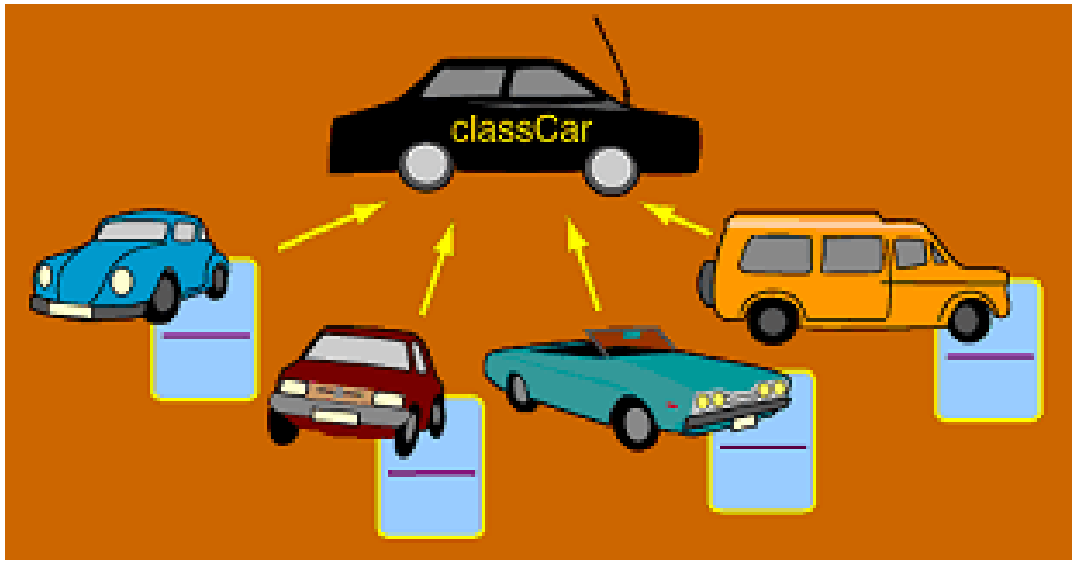
EXAMPLE 1.9 INITIALIZING VARIABLES

```
int main()
{
        // prints "m = ?? and n = 44":
        int m; // BAD: m is not initialized
        int n = 44;
        cout << "m = " << m << " and n = " << n << endl;
}
```

1.8 OBJECTS, VARIABLES, AND CONSTANTS







1.8 OBJECTS, VARIABLES, AND CONSTANTS

- An *object* is a contiguous region of memory that has an address, a size, a type, and a value
- The *address* of an object is the memory address of its first byte
- The *size* of an object is simply the number of bytes that it occupies in memory
- The *value* of an object is the constant determined by the actual bits stored in its memory location and by the object's type which prescribes how those bits are to be interpreted
- A *variable* is an object that has a name
- The word "variable" is used to suggest that the object's value can be changed
- An object whose value cannot be changed is called a *constant*
- Constants must be initialized when they are declared

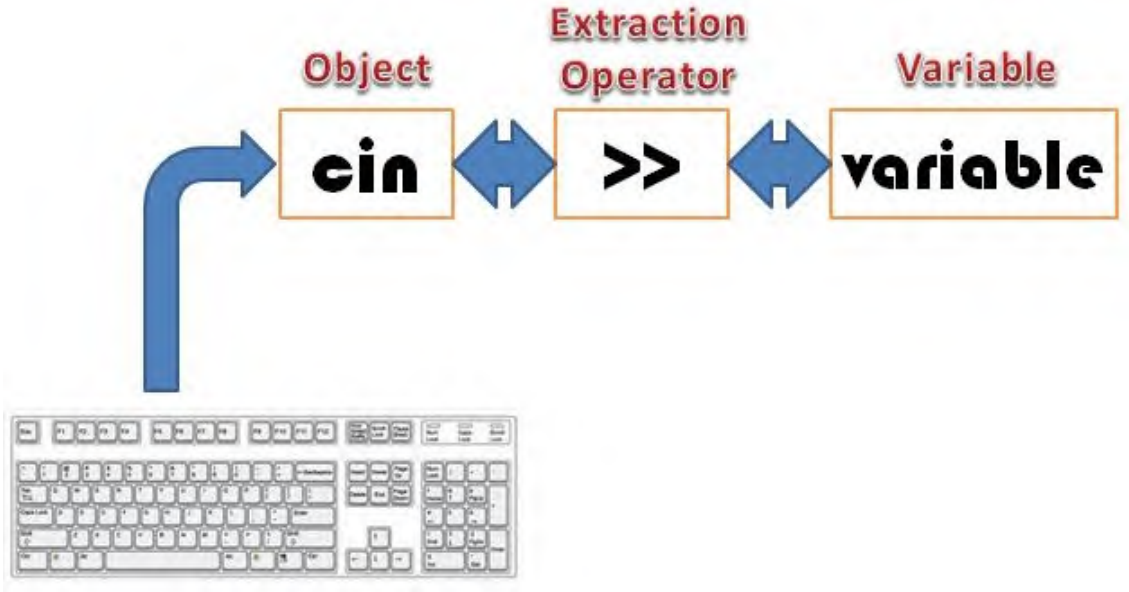
EXAMPLE 1.10 THE CONST SPECIFIER

```
int main()
{
        // defines constants; has no output:
        const char BEEP = '\b';
        const int MAXINT = 2147483647;
        const int N = MAXINT/2;
        const float KM_PER_MI = 1.60934;
        const double PI = 3.14159265358979323846;
}
```

It is customary to use all capital letters in constant identifiers to distinguish them from other kinds of identifiers

1.9 THE INPUT OPERATOR

- The *input operator* >>
- (also called the *get operator* or the *extraction operator*) works like the output operator <<

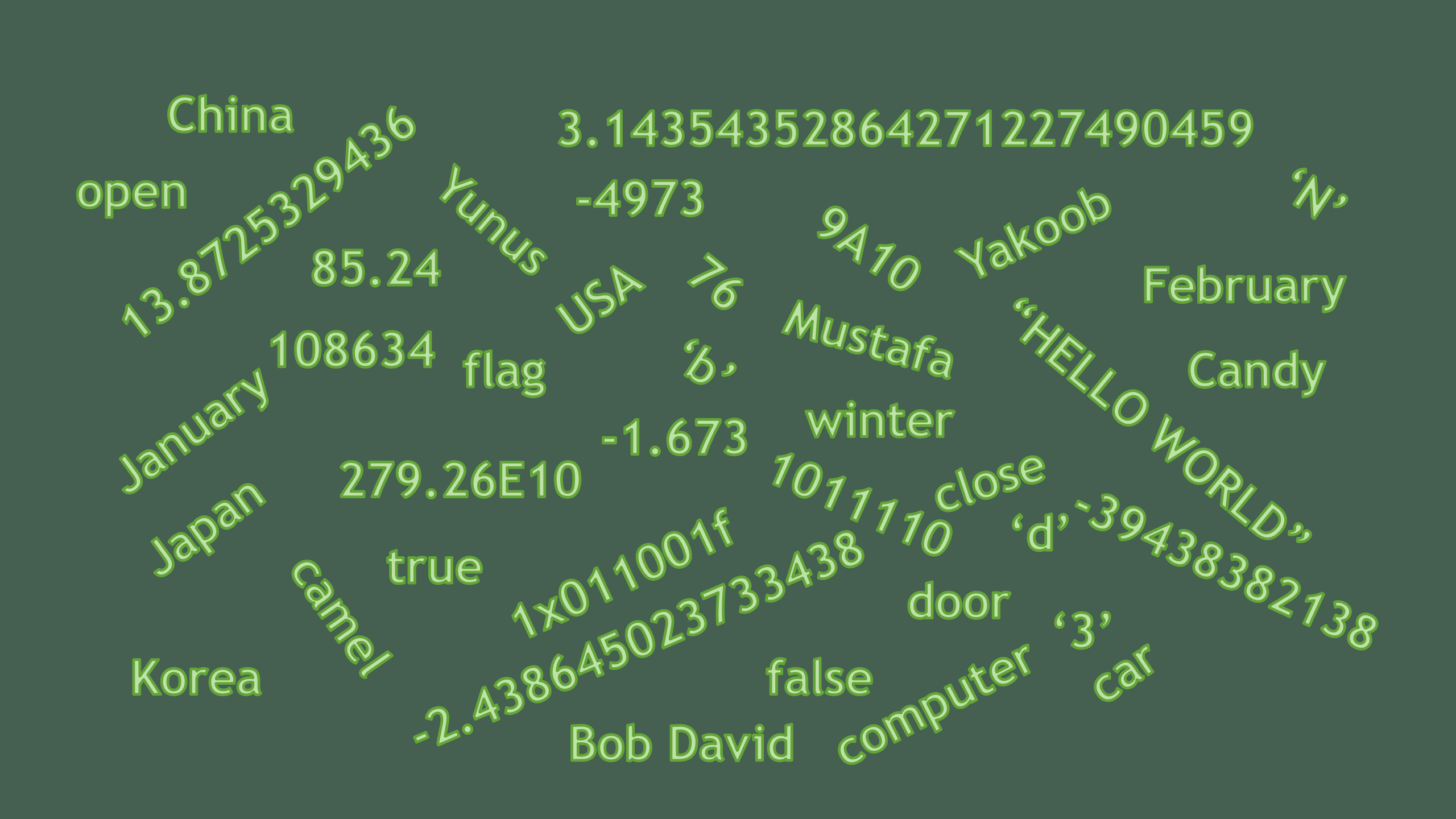


EXAMPLE 1.11 USING THE INPUT OPERATOR

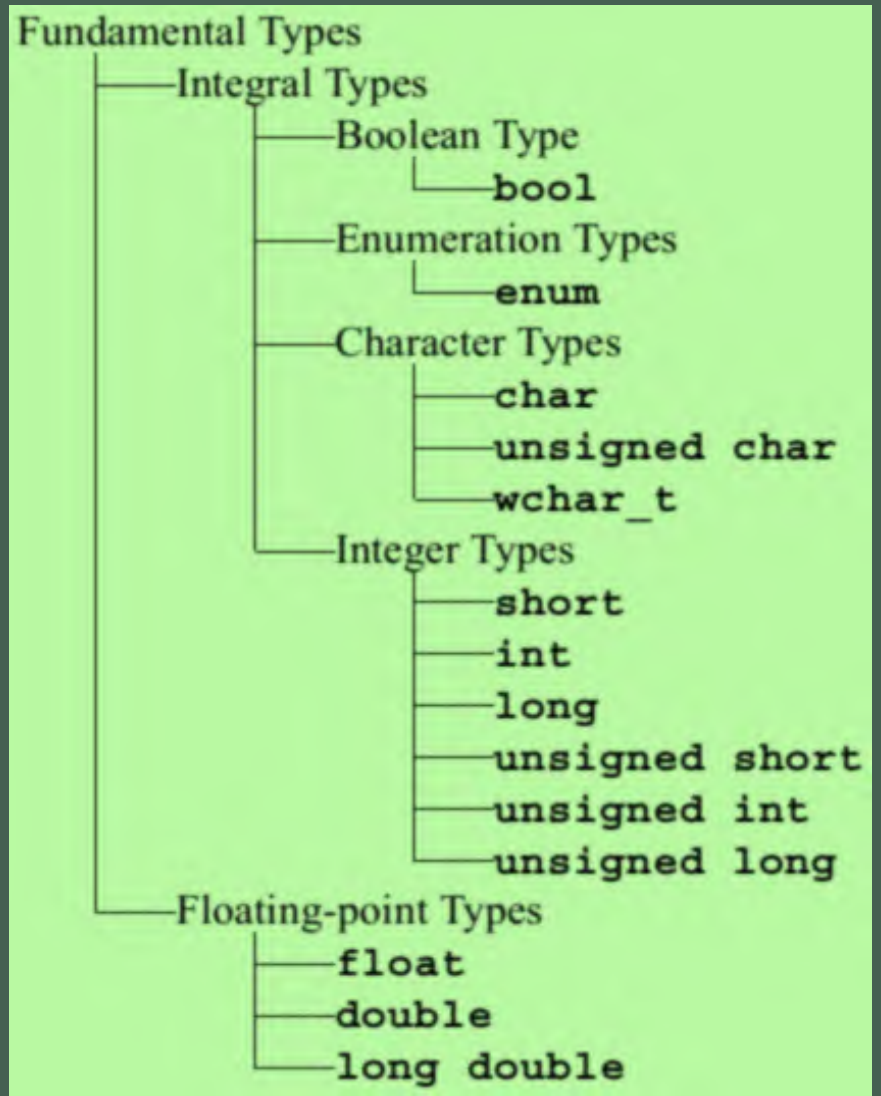
```
int main( )
{
        // tests the input of integers, floats, and characters:
        int m, n;
        cout << "Enter two integers: ";
        cin >> m >> n;
        cout << "m = " << m << ", n = " << n << endl;
        double x, y, z;
        cout << "Enter three decimal numbers: ";
        cin >> x >> y >> z;
        cout << "x = " << x << ", y = " << y << ", z = " << z << endl;
        char c1, c2, c3, c4;
        cout << "Enter four characters: ";
        cin >> c1 >> c2 >> c3 >> c4;
        cout << "c1 = " << c1 << ",c2 = " << c2 << ", c3 = " << c3 << ", c4 = " << c4 << endl;
}
```

FUNDAMENTAL TYPES

Unit 2


China
3.14354352864271227490459
open
13.87253294360
Yunus
-4973
'N'
85.24
9A10
Yakoob
February
USA
76
January
108634
flag
'6'
Mustafa
"HELLO WORLD"
Candy
-1.673
winter
Japan
279.26E10
1011110
close
1x0110001f
'd'
-394383821380
-2.4386450237334380
door
true
camel
'3'
Korea
false
computer
car
Bob David

NUMERIC DATA TYPES



- Standard C++ has 14 different *fundamental types*:
  - 11 integral types and 3 floating-point types
- The *integral types* include
  - boolean type **bool**,
  - enumeration types defined with the **enum** keyword,
  - three character types, and
  - six explicit integer types
- The three *floating-point types* are **float**, **double**, and **long double**

THE BOOLEAN TYPE

- A *boolean* type is an integral type whose variables can have only two values:
  - ✓**false** and
  - ✓**True**
- These values are stored as the integers 0 and 1

```
int main()
{ // prints the value of a boolean variable:
        bool flag=false;
        cout << "flag = " << flag << endl;
        flag = true;
        cout << "flag = " << flag << endl;
}
```

ENUMERATION TYPES

C++ allows you to define your own special data types

- An *enumeration type* is an integral type that is defined by the user
- Syntax:

enum *typename* { *enumerator-list* };

  - **Where enum** is a C++ keyword,
  - *typename* stands for an identifier that names the type being defined, and
  - *enumerator-list* stands for a list of names for integer constants

- The actual values defined in the enumerator-list are called *enumerators*
  - *(automatically assigned* ordinary integer constants like, 0, 1, 2, 3, ……)
  - These default values can be overridden in the *enumerator-list*

enum Semester { FALL, SPRING, SUMMER };

```
const int FALL = 0;
const int WINTER = 1;
const int SUMMER = 2;
```

enum Coin

{ PENNY=1, NICKEL=5, DIME=10, QUARTER=25 };

ENUMERATION TYPES

- If integer values are assigned to only some of the enumerators, then the ones that follow are given consecutive values
- it is legal to have several different enumerators with the same value
- Enumerations can also be anonymous in C++

```
enum Month { JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };

enum Answer { NO = 0, FALSE=0, YES = 1, TRUE=1, OK = 1 };

enum { I=1, V=5, X=10, L=50, C=100, D=500, M=1000 };
```

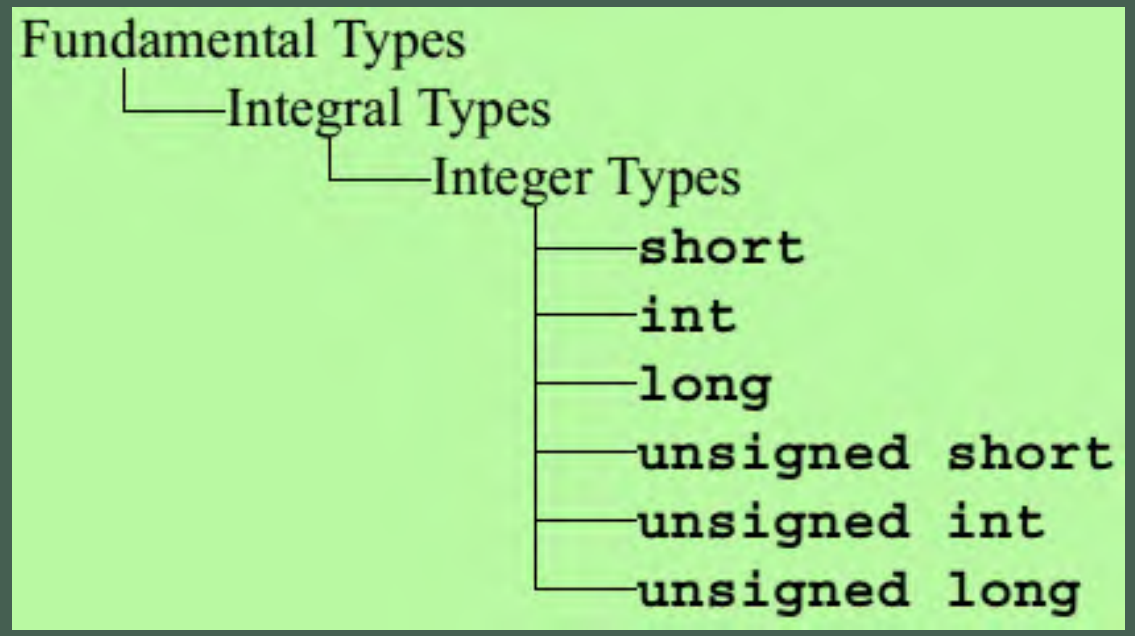
CHARACTER TYPES

- A *character type* is an integral type whose variables represent characters like the letter 'A' or the digit '8‘
- Character literals are delimited by the apostrophe (')
- Like all integral type values, character values are stored as integers

```
int main() {

//prints the character and its internally stored integer value:
        char c = 'A';
        cout << "c = " << c << ", int(c) = " << int(c) << endl;
        c = 't';
        cout << "c = " << c << ", int(c) = " << int(c) << endl;
        c = '\t'; // the tab character
        cout << "c = " << c << ", int(c) = " << int(c) << endl;
        c = '!';
        cout << "c = " << c << ", int(c) = " << int(c) << endl;
}
```

INTEGER TYPES



- There are 6 integer types in Standard C++
- These types have several names
- For example,
  - **short** is also named **short int**, and
  - **int** is also named **signed int**

INTEGER TYPE RANGES

This program prints the numeric ranges of the 6 integer types in C++:

```
#include <iostream>
#include <climits> // defines the constants SHRT_MIN,etc.
using namespace std;

int main() { // prints some of the constants stored in the <climits> header:
        cout << "minimum short = " << SHRT_MIN << endl;
        cout << "maximum short = " << SHRT_MAX << endl;
        cout << "maximum unsigned short = 0" << endl;
        cout << "maximum unsigned short = " << USHRT_MAX << endl;
        cout << "minimum int = " << INT_MIN << endl;
        cout << "maximum int = " << INT_MAX << endl;
        cout << "minimum unsigned int = 0" << endl;
        cout << "maximum unsigned int = " << UINT_MAX << endl;
        cout << "minimum long= " << LONG_MIN << endl;
        cout << "maximum long= " << LONG_MAX << endl;
        cout << "minimum unsigned long = 0" << endl;
        cout << "maximum unsigned long = " << ULONG_MAX << endl;
}
```

DATA TYPES, RANGES, MEMORY SPACES

`short`:	–32,768 to 32,767;	(2^8 values ⇒ 1 byte)
`int`:	–2,147,483,648 to 2,147,483,647;	(2^{32} values ⇒ 4 bytes)
`long`:	–2,147,483,648 to 2,147,483,647;	(2^{32} values ⇒ 4 bytes)
`unsigned short`:	0 to 65,535;	(2^8 values ⇒ 1 byte)
`unsigned int`:	0 to 4,294,967,295;	(2^{32} values ⇒ 4 bytes)
`unsigned long`:	0 to 4,294,967,295;	(2^{32} values ⇒ 4 bytes)

TYPES, MEMORY SPACES, RANGES

Variable Type	Memory Size	Signed Range	Unsigned Range
Integer	4 Bytes	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
Short Integer	2 Bytes	-32,768 to 32,767	0 to 65,535
Character	1 Byte	N/A	0 to 255
String	1 Byte * n	N/A	0 to n
Boolean	1 Byte	N/A	TRUE (1) or FALSE (0)
Float	4 Bytes	Accurate within 7 Significant Figures	
Double	8 Bytes	Accurate within 15 Significant Figures	

ARITHMETIC OPERATORS

C++ performs its numerical calculations by means of the five *arithmetic operators* +, -, *, /, and %

```
int main()
{       // tests operators +,-,*,/,and %:
        int m=54;
        int n=20;
        cout << "m = " << m << " and n = " << n << endl;
        cout << "m + n = " << m + n << endl; // 54+20 = 74
        cout << "m - n = " << m - n << endl; // 54-20 = 34
        cout << "m * n = " << m * n << endl; // 54*20 = 1080
        cout << "m / n = " << m / n << endl; // 54/20 = 2
        cout << "m % n = " << m % n << endl; // 54%20 = 14
}
```

THE INCREMENT AND DECREMENT OPERATORS

- The values of integral objects can be incremented and decremented with the ++ and -- operators, respectively
- Each of these operators has two versions: a “pre” version and a “post” version
  - The “pre” version performs the operation (either adding 1 or subtracting 1) on the object before the resulting value is used in its surrounding context
  - The “post” version performs the operation after the object’s current value has been used

```
int main() {

        //shows the difference between m++ and ++m:
        int m, n;
        m = 44;
        n = ++m;// the pre-increment operator is applied to m
        // now, m = m +1 and n = m + 1

        cout << "m = " << m << ", n = " << n << endl;
        m = 44;
        n = m++;// the post-increment operator is applied to m
        // now, m = m +1 and n = m

        cout << "m = " << m << ", n = " << n << endl;
}
```

COMPOSITE ASSIGNMENT OPERATORS

- The standard assignment operator in C++ is the equals sign =
- C++ also includes the following *composite assignment operators*: +=, -=, *=, /=, and %=
- By applying to a variable, the composite assignment operator carries out the indicated arithmetic operation to the variable along with the value on the right

```
int main() {

        // tests arithmetic assignment operators:
        int n = 22;
        cout << "n = " << n << endl;
        n += 9; // adds 9 to n : ( n = n +9 )
        cout << "After n += 9, n = " << n << endl;
        n -= 5; // subtracts 5 from n : ( n = n - 5 )
        cout << "After n -= 5, n = " << n << endl;
        n *= 2; // multiplies n by 2 : ( n = n * 2 )
        cout << "After n *= 2, n = " << n << endl;
        n / = 3; // divides n by 3 : ( n = n / 3 )
        cout << "After n /= 3, n = " << n << endl;
        n %= 7; // reduces n to the remainder from dividing by 7
        // ( n = n % 7 )

        cout << "After n %= 7, n = " << n << endl;
}
```

FLOATING-POINT TYPES

- C++ supports three real number types:
  - **float**,
  - **double**, and
  - **long double**

```
int main()
{
// tests the floating-point operators +, -, *, and /:
        double x = 54.0;
        double y = 20.0;
        cout << "x = " << x << " and y = " << y << endl;
        cout << "x + y = " << x + y << endl; // 54.0+20.0 = 74.0
        cout << "x - y = " << x - y << endl; // 54.0-20.0 = 34.0
        cout << "x * y = " << x * y << endl; // 54.0*20.0 = 1080.0
        cout << "x / y = " << x / y << endl; // 54.0/20.0 = 2.7
}
```

USING THE SIZEOF OPERATOR

This program tells you how much space each of the 12 fundamental types uses:

```
int main() { // prints the storage sizes of the fundamental types:
        cout << "Number of bytes used:\n";
        cout << "\t char: " << sizeof(char) << endl;
        cout << "\t short: " << sizeof(short) << endl;
        cout << "\t int: " << sizeof(int) << endl;
        cout << "\t long: " << sizeof(long) << endl;
        cout << "\t unsigned char: " << sizeof(unsigned char) << endl;
        cout << "\tunsigned short: " << sizeof(unsigned short) << endl;
        cout << "\t unsigned int: " << sizeof(unsigned int) << endl;
        cout << "\t unsigned long: " << sizeof(unsigned long) << endl;
        cout << "\t signed char: " << sizeof(signed char) << endl;
        cout << "\t float: " << sizeof(float) << endl;
        cout << "\t double: " << sizeof(double) << endl;
        cout << "\t long double: " << sizeof(long double) << endl;
}
```

READING FROM THE <cfloat> HEADER FILE

This program tells you the precision and magnitude range that the **float** type has on your system:

```
#include <cfloat> // defines the FLT constants
#include <iostream> // defines the FLT constants
using namespace std;


int main() { // prints the storage sizes of the fundamental types:
        int fbits = 8*sizeof(float); // each byte contains 8 bits
        cout << "float uses " << fbits << " bits:\n\t"
                << FLT_MANT_DIG - 1 << " bits for its mantissa,\n\t "
                << fbits - FLT_MANT_DIG << " bits for its exponent,\n\t "
                << 1 << " bit for its sign\n"
                << " to obtain: " << FLT_DIG << " sig. digits\n"
        << " with minimum value: " << FLT_MIN << endl
        << " and maximum value: " << FLT_MAX << endl;
}
```

TYPE CONVERSIONS

- type can be converted automatically to another
- if T is one type and v is a value of another type, then the expression:
  - ✓T(v) converts v to type T : (called as *type casting*)
  - ✓n = int(*expr*);

This program casts a double value into int value:

```
int main()
{
        // casts a double value as an int:
        double v = 1234.56789;
        int n = int(v);
        cout << "v = " << v << ", n = " << n << endl;
}
```

PROMOTION OF TYPES

This program promotes a char to a short to an int to a float to a double:

```
int main()
{
        // prints promoted vales of 65 from char to double:
        char c = 'A'; cout << " char c = " << c << endl;
        short k = c; cout << " short k = " << k << endl;
        int m = k; cout << " int m = " << m << endl;
        long n = m; cout << " long n = " << n << endl;
        float x = m; cout << " float x = " << x << endl;
        double y = x; cout << "double y = " << y << endl;
}
```

NUMERIC OVERFLOW

computers are manifestly prone to error when their numeric values become too large (the error is called *numeric overflow)*

This program repeatedly multiplies n by 1000 until it overflows.

```
int main() { // prints n until it overflows:
        int n=1000;
        cout << "n = " << n << endl;
        n *= 1000; // multiplies n by 1000
        cout << "n = " << n << endl;
        n *= 1000; // multiplies n by 1000
        cout << "n = " << n << endl;
        n *= 1000; // multiplies n by 1000
        cout << "n = " << n << endl;
}
```

This program repeatedly squares x until it overflows

```
int main() { // prints x until it overflows:
        float x=1000.0;
        cout << "x = " << x << endl;
        x *= x; // multiplies n by itself; i.e.,it squares x
        cout << "x = " << x << endl;
        x *= x; // multiplies n by itself; i.e.,it squares x
        cout << "x = " << x << endl;
        x *= x; // multiplies n by itself; i.e.,it squares x
        cout << "x = " << x << endl;
        x *= x; // multiplies n by itself; i.e.,it squares x
        cout << "x = " << x << endl;
}
```

ROUND-OFF ERROR

Round-off error is another kind of error that often occurs when computers do arithmetic on rational numbers

This program does some simple arithmetic to illustrate roundoff error:

```
int main()
{
// illustrates round-off error::
double x = 1000/3.0;
cout << "x = " << x << endl;// x = 1000/3
double y = x - 333.0;
cout << "y = " << y << endl; // y = 1/3
double z = 3*y - 1.0;
cout << "z = " << z << endl; // z = 3(1/3) - 1

if (z == 0) cout << "z == 0.\n";
else cout << "z does not equal 0.\n"; // z != 0
}
```

This program implements the *quadratic formula* to solve quadratic equations.

```
#include <cmath> // defines the sqrt() function
#include <iostream>
using namespace std;
int main() { // implements the quadratic formula
float a, b, c;
cout << "Enter the coefficients of a quadratic equation:" << endl;
cout << "\ta: ";
cin >> a;
cout << "\tb: ";
cin >> b;
cout << "\tc: ";
cin >> c;
cout << "The equation is: " << a << "*x*x + " << b << "*x + " << c << " = 0" << endl;
float d = b*b - 4*a*c; // discriminant
float sqrtd = sqrt(d);
float x1 = (-b + sqrtd)/(2*a);
float x2 = (-b - sqrtd)/(2*a);
cout << "The solutions are:" << endl;
cout << "\tx1 = " << x1 << endl;
cout << "\tx2 = " << x2 << endl;
cout << "Check:" << endl;
cout << "\ta*x1*x1 + b*x1 + c = " << a*x1*x1 + b*x1 + c << endl;
cout << "\ta*x2*x2 + b*x2 + c = " << a*x2*x2 + b*x2 + c << endl;
}
```

THE E-FORMAT FOR FLOATING-POINT VALUES

When input or output, floating-point values may be specified in either of two formats: *fixed-point* and *scientific*

This program shows how floating-point values may be input in scientific format:

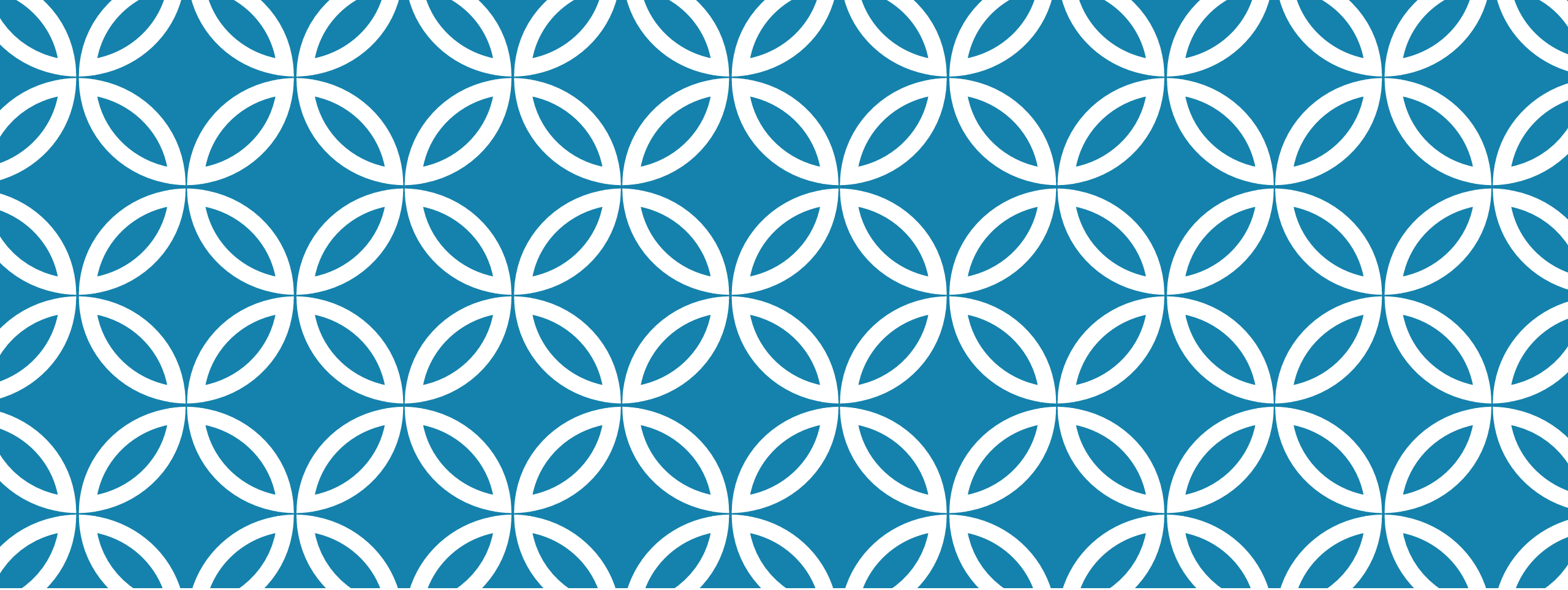
```
int main()
{       // prints double values in scientific e-format:
        double x;
        cout << "Enter float: "; cin >> x;
        cout << "Its reciprocal is: " << 1/x << endl;
}
```

SCOPE

- Scope of an identifier is that part of the program where it can be used
- scope resolution operator :: are used to access the global variables

```
int main()
{ // illustrates the scope of variables:
x = 11; // ERROR: this is not in the scope of x
int x;
  { x = 22; // OK: this is in the scope of x
  y = 33; // ERROR: this is not in the scope of y
  int y;
  x = 44; // OK: this is in the scope of x
  y = 55; // OK: this is in the scope of y
  }
x = 66; // OK: this is in the scope of x
y = 77; // ERROR: this is not in the scope of y
}
```

```
int x = 11; // this x is global
int main()
{ // illustrates the nested and parallel scopes:
int x = 22;
    { // begin scope of internal block
    int x = 33;
    cout << "In block inside main(): x = " << x << endl;
    } // end scope of internal block
cout << "In main(): x = " << x << endl;
cout << "In main(): ::x = " << ::x << endl;
} // end scope of main()
```



C++ PROGRAMMING

Chapter 3 (Selection)

3.1 THE IF STATEMENT

The **if** statement allows conditional execution.

if (*condition***)** *statement***;**

where *condition* is an integral expression and *statement* is any executable statement.

The statement will be executed only if the value of the integral expression is nonzero.

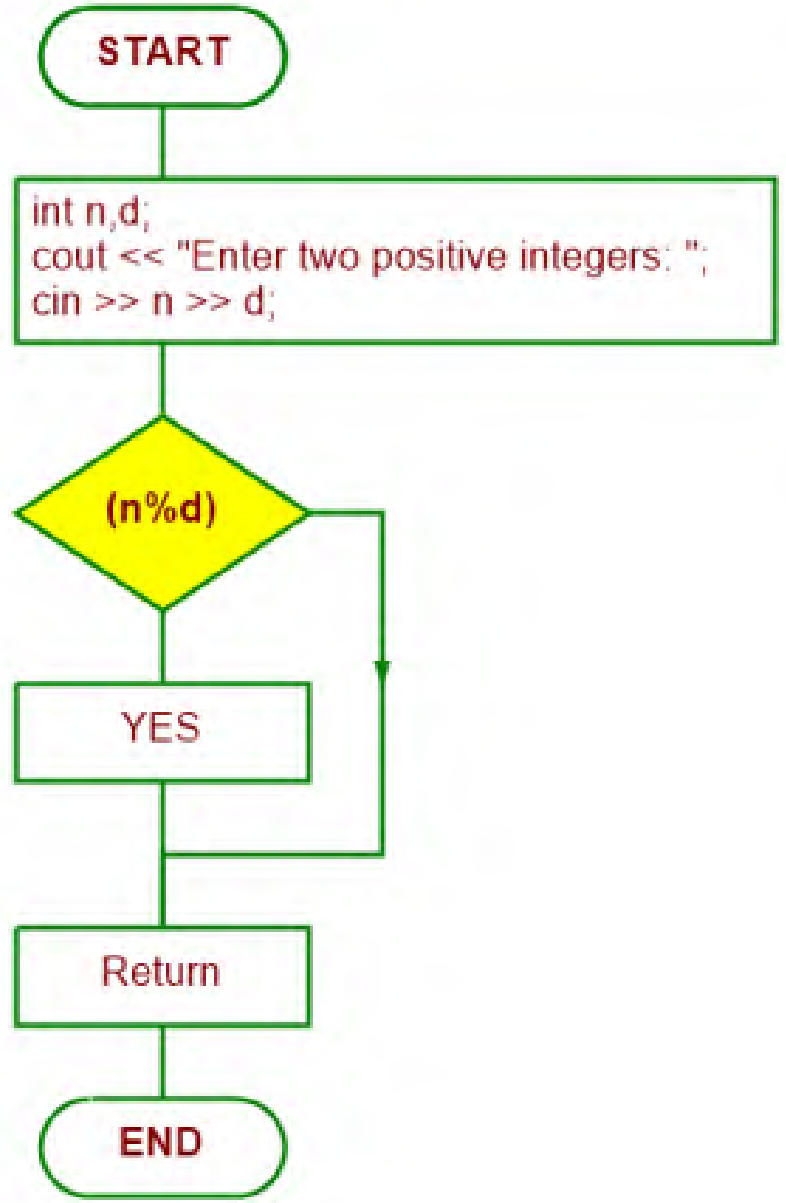
EXAMPLE 3.1 TESTING FOR DIVISIBILITY

This program tests if one positive integer is not divisible by another:

```
int main( )
{
        int n, d;
        cout << "Enter two positive integers: ";
        cin >> n >> d;
        if (n%d) cout << n << " is not divisible by " << d << endl;
}
```

In C++, whenever an integral expression is used as a condition, the value 0 (zero) means “false” and all other values mean “true”.

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n,d;
    cout << "Enter two positive integers: ";
    cin >> n >> d;
    if (n%d) cout << n << " is not divisible by " << d << endl;
    return 0;
}
```



3.2 THE IF..ELSE STATEMENT

The **if..else** statement causes one of two alternative statements to execute depending upon whether the condition is true.

if (*condition***)** *statement1***;**
else *statement2***;**

where *condition* is an integral expression and *statement1* and *statement2* are executable statements.

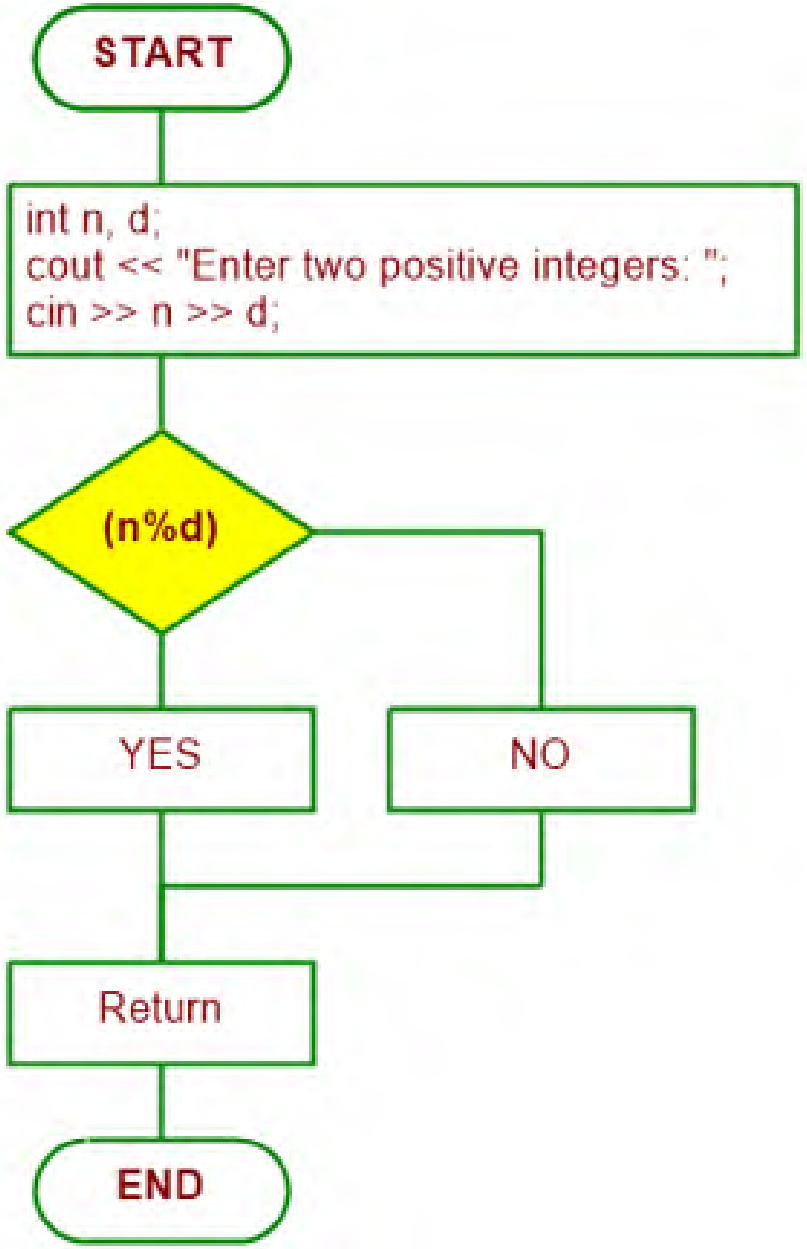
If the value of the condition is nonzero then *statement1* will execute; otherwise *statement2* will execute.

EXAMPLE 3.2 TESTING FOR DIVISIBILITY AGAIN

This program tests if one positive integer is not divisible by another (if..else statement):

```
int main( )
{
        int n, d;
        cout << "Enter two positive integers: ";
        cin >> n >> d;
        if (n%d) cout << n << " is not divisible by " << d << endl;
        else cout << n << " is divisible by " << d << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n, d;
    cout << "Enter two positive integers: ";
    cin >> n >> d;
    if (n%d) cout << n << " is not divisible by " << d << endl;
    else cout << n << " is divisible by " << d << endl;
    return 0;
}
```



3.3 KEYWORDS

A *keyword* in a programming language is a word that is already defined and is reserved for a unique purpose in programs written in that language.

Standard C++11 now has 74 keywords:

and	**and_eq**	**asm**	**auto**	**bitand**
bitor	**bool**	**break**	**case**	**catch**
char	**class**	**compl**	**const**	**const_cast**
continue	**default**	**delete**	**do**	**double**
dynamic_cast	**else**	**enum**	**explicit**	**export**

KEYWORDS

extern	dfalse	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	not	not_eq
operator	or	or_eq	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	using
union	unsigned	virtual	void	volatile
wchar_t	while	xor	xor_eq	

NOTE

There are two kinds of keywords: reserved words and standard identifiers.

A *reserved word* is a keyword that serves as a structure marker, used to define the syntax of the language. The keywords if and else are reserved words.	**A *standard identifier* is a keyword that names a specific element of the language. The keywords bool and int are standard identifiers because they are names of standard types in C++.**

3.4 COMPARISON OPERATORS

The six *comparison operators* are

1) **x < y** // x is less than y
2) **x > y** // x is greater than y
3) **x <= y** // x is less than or equal to y
4) **x >= y** // x is greater than or equal to y
5) **x == y** // x is equal to y
6) **x != y** // x is not equal to y

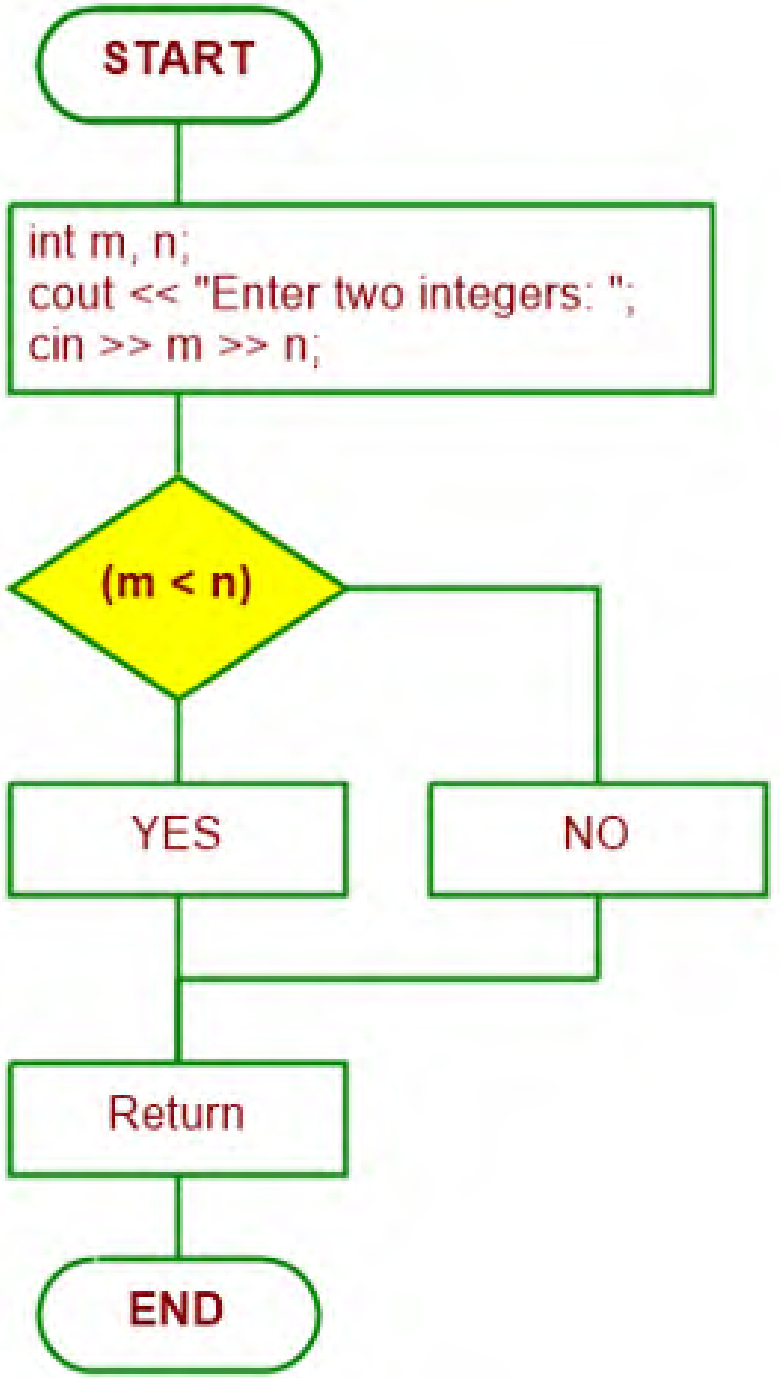
EXAMPLE 3.3 THE MINIMUM OF TWO INTEGERS

This program prints the minimum of the two integers entered:

```
int main( )
{
        int m, n;
        cout << "Enter two integers: ";
        cin >> m >> n;
        if (m < n) cout << m << " is the minimum." << endl;
        else cout << n << " is the minimum." << endl;
}
```

Note that in C++ the single equal sign “=” is the *assignment operator*, and the double equal sign “==” is the *equality operator*

```cpp
#include <iostream>
using namespace std;
int main()
{
    int m, n;
    cout << "Enter two integers: ";
    cin >> m >> n;
    if (m < n) cout << m << " is the minimum." << endl;
    else cout << n << " is the minimum." << endl;
    return 0;
}
```

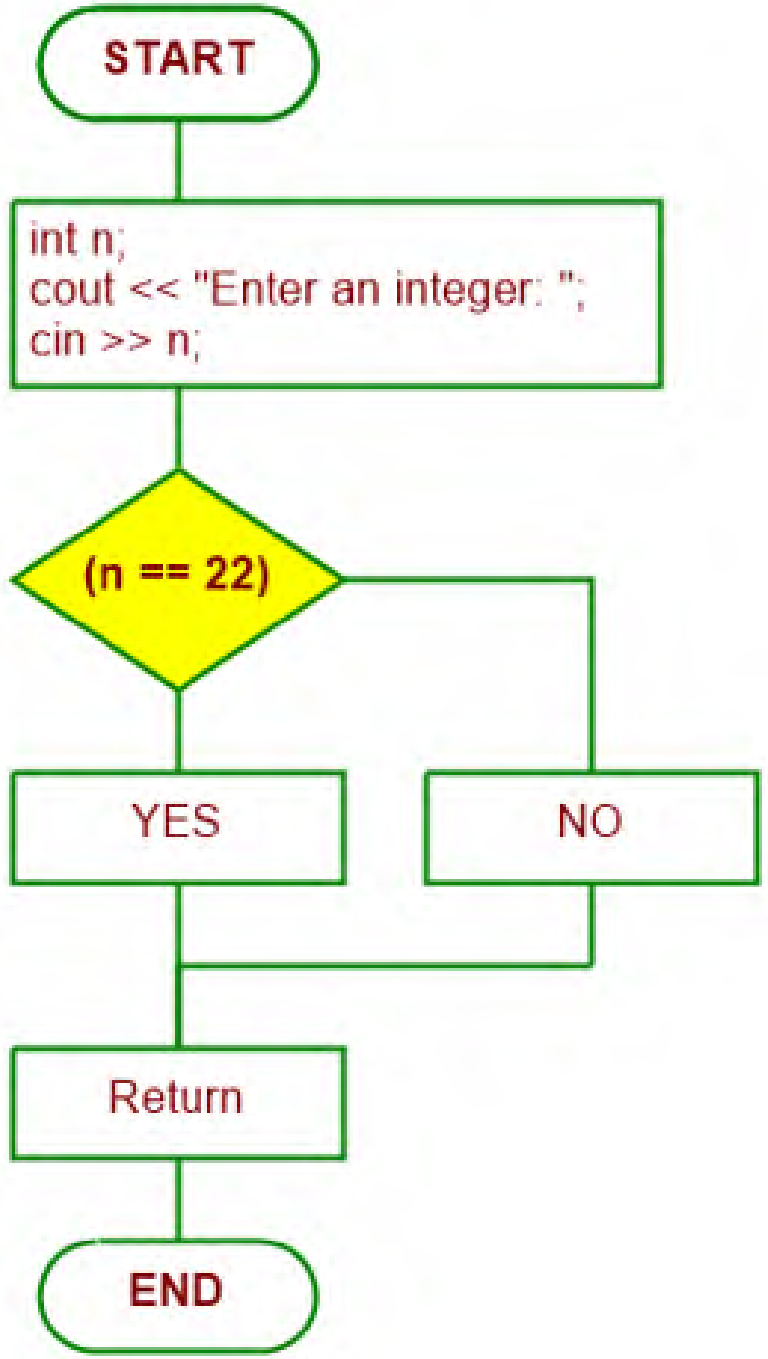


EXAMPLE 3.4 A COMMON PROGRAMMING ERROR

This program is erroneous:

```
int main( )
{
        int n;
        cout << "Enter an integer: ";
        cin >> n;
        if (n = 22) cout << n << " = 22" << endl;      // LOGICAL ERROR!
        else cout << n << " != 22" << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "Enter an integer: ";
    cin >> n;
    if (n == 22) cout << n << " = 22" << endl; // Boolean Expression
    else cout << n << " != 22" << endl;
    return 0;
}
```

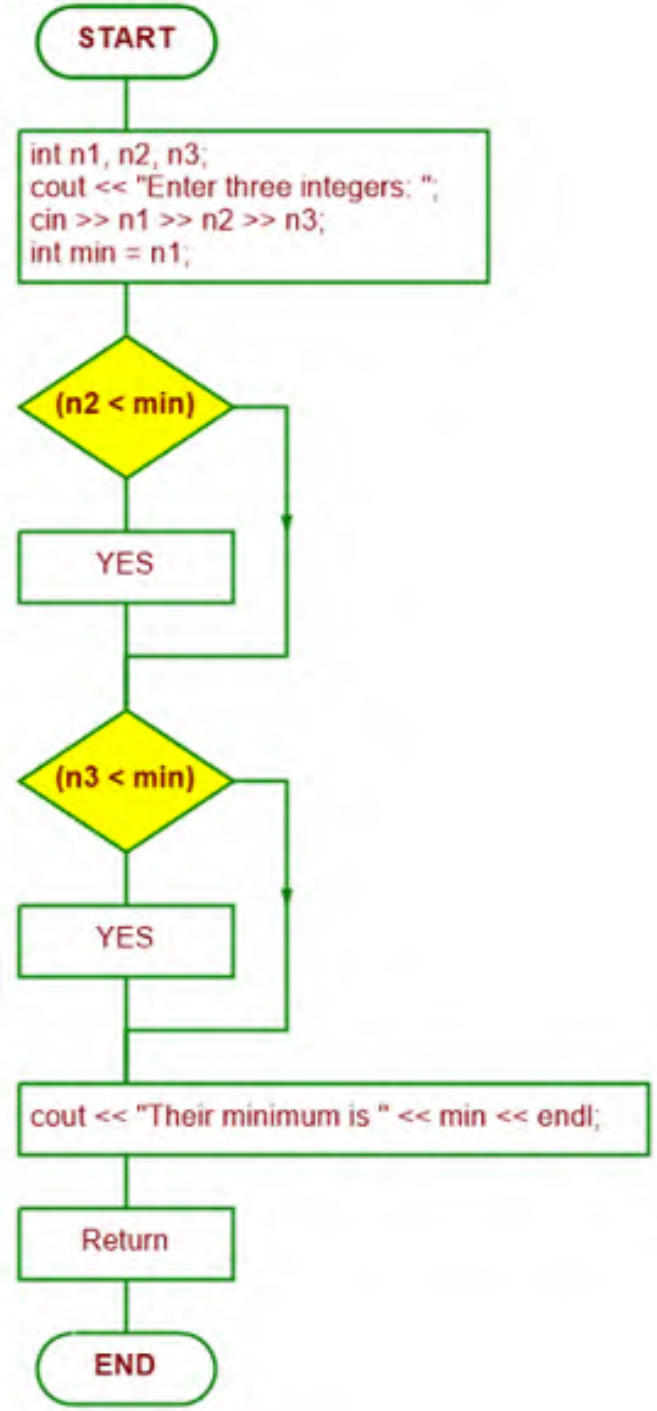


EXAMPLE 3.5 THE MINIMUM OF THREE INTEGERS

This program prints the minimum of the three integers entered:

```
int main( )
{
        int n1, n2, n3;
        cout << "Enter three integers: ";
        cin >> n1 >> n2 >> n3;
        int min=n1;                                  // now min <= n1
        if (n2 < min) min = n2;                      // now min <= n1 and min <= n2
        if (n3 < min) min = n3;                      // now min <= n1,min <= n2,and min <= n3
        cout << "Their minimum is " << min << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n1, n2, n3;
    cout << "Enter three integers: ";
    cin >> n1 >> n2 >> n3;
    int min = n1; // now min <= n1
    if (n2 < min) min = n2; // now min <= n1 and min <= n2
    if (n3 < min) min = n3; // now min <= n1,min <= n2,and min <= n3
    cout << "Their minimum is " << min << endl;
    return 0;
}
```



3.5 STATEMENT BLOCKS

A *statement block* is a sequence of statements enclosed by braces **{ }**, like these:

```
{ int temp=x; x = y; y = temp; }
```

```
{
int temp=x; x = y; y = temp;
}
```

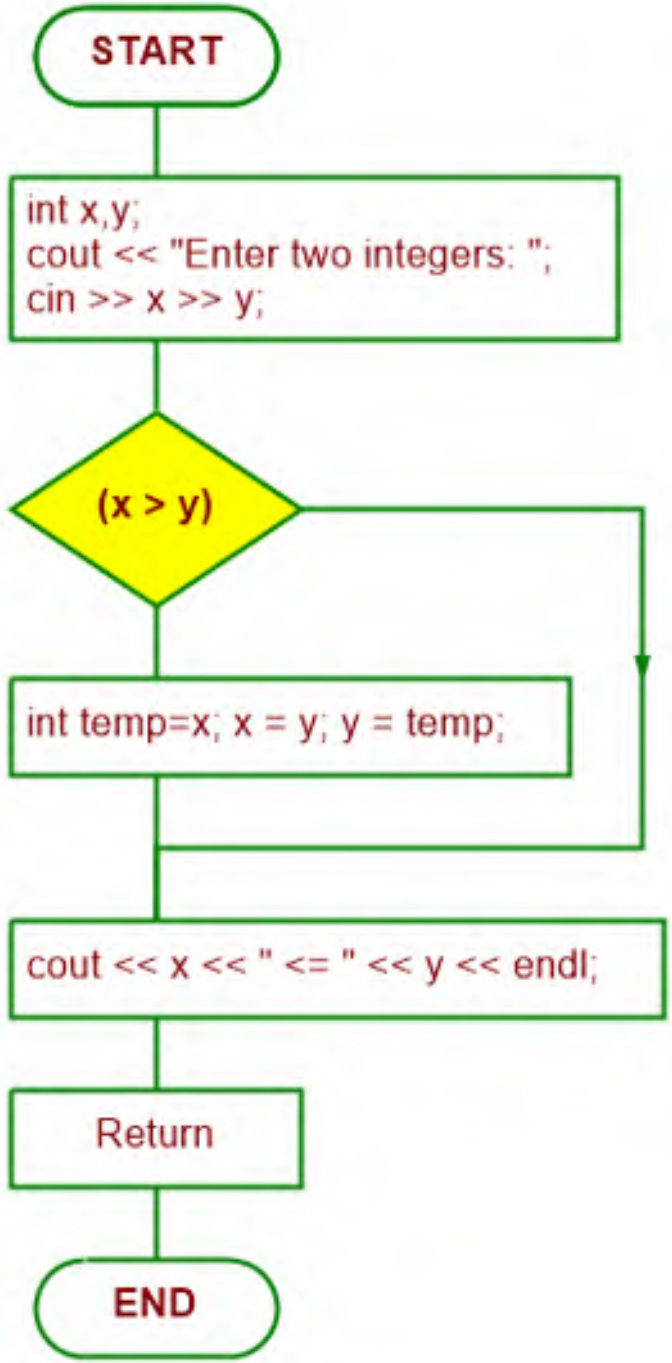
In C++ programs, a statement block can be used anywhere that a single statement can be used.

EXAMPLE 3.6 A STATEMENT BLOCK WITHIN AN IF STATEMENT

This program inputs two integers and then outputs them in increasing order:

```
int main( )
{
        int x, y;
        cout << "Enter two integers: ";
        cin >> x >> y;
        if (x > y) { int temp=x; x = y; y = temp; }                    // swap x and y
        cout << x << " <= " << y << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x,y;
    cout << "Enter two integers: ";
    cin >> x >> y;
    if (x > y) { int temp=x; x = y; y = temp; } // swap x and y
    cout << x << " <= " << y << endl;
    return 0;
}
```



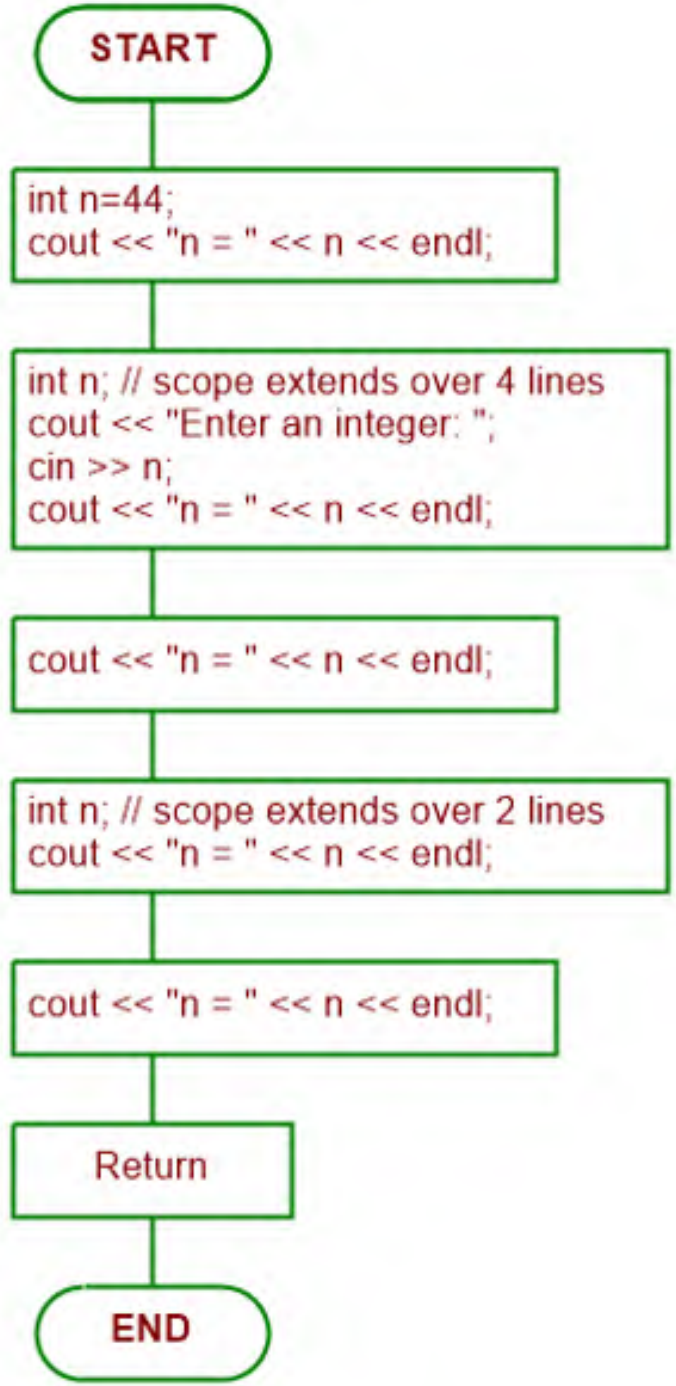
EXAMPLE 3.7 USING BLOCKS TO LIMIT SCOPE

This program uses the same name n for three different variables:

```
int main( )
{
        int n=44;
        cout << "n = " << n << endl;
        {       int n;                          // scope extends over 4 lines
                cout << "Enter an integer: ";
                cin >> n;
                cout << "n = " << n << endl;

        }
                {
                cout << "n = " << n << endl;    // the n that was declared first
                }
                        { int n;                // scope extends over 2 lines
                        cout << "n = " << n << endl;
                        }
        cout << "n = " << n << endl;            // the n that was declared first
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n=44;
    cout << "n = " << n << endl;
    { int n; // scope extends over 4 lines
    cout << "Enter an integer: ";
    cin >> n;
    cout << "n = " << n << endl;
    }
    { cout << "n = " << n << endl; // the n that was declared first
    }
    { int n; // scope extends over 2 lines
    cout << "n = " << n << endl;
    }
    cout << "n = " << n << endl; // the n that was declared first
    return 0;
}
```



3.6 COMPOUND CONDITIONS

Conditions such as n % d and x >= y can be combined to form compound conditions using the *logical operators* **&&** (and), || (or), and **!** (not).

p	q	p && q
T	T	T
T	F	F
F	T	F
F	F	F

p	q	p || q
T	T	T
T	F	T
F	T	T
F	F	F

p	!p
T	F
F	T

p && q evaluates to true if and only if both p and q evaluate to true.
p || q evaluates to false if and only if both p and q evaluate to false.
!p evaluates to true if and only if p evaluates to false.

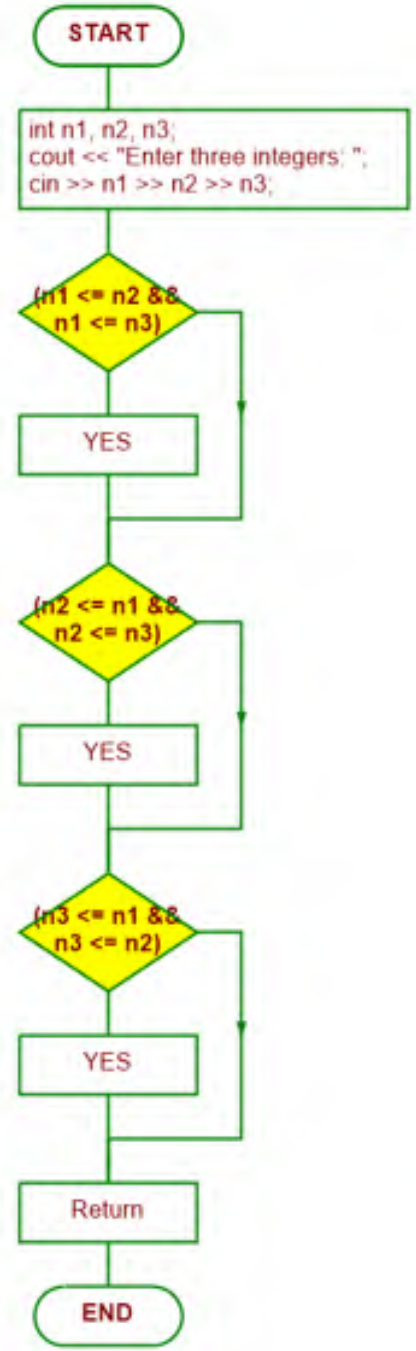
For example, (n % d || x >= y) will be false if and only if n % d is zero and x is less than y.

EXAMPLE 3.8 USING COMPOUND CONDITIONS

This version uses compound conditions to find the minimum of three integers:

```
int main( )
{
        int n1, n2, n3;
        cout << "Enter three integers: ";
        cin >> n1 >> n2 >> n3;
        if (n1 <= n2 && n1 <= n3) cout << "Their minimum is " << n1 <<endl;
        if (n2 <= n1 && n2 <= n3) cout << "Their minimum is " << n2 <<endl;
        if (n3 <= n1 && n3 <= n2) cout << "Their minimum is " << n3 <<endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n1, n2, n3;
    cout << "Enter three integers: ";
    cin >> n1 >> n2 >> n3;
    if (n1 <= n2 && n1 <= n3) cout << "Their minimum is " << n1 <<endl;
    if (n2 <= n1 && n2 <= n3) cout << "Their minimum is " << n2 <<endl;
    if (n3 <= n1 && n3 <= n2) cout << "Their minimum is " << n3 <<endl;
    return 0;
}
```

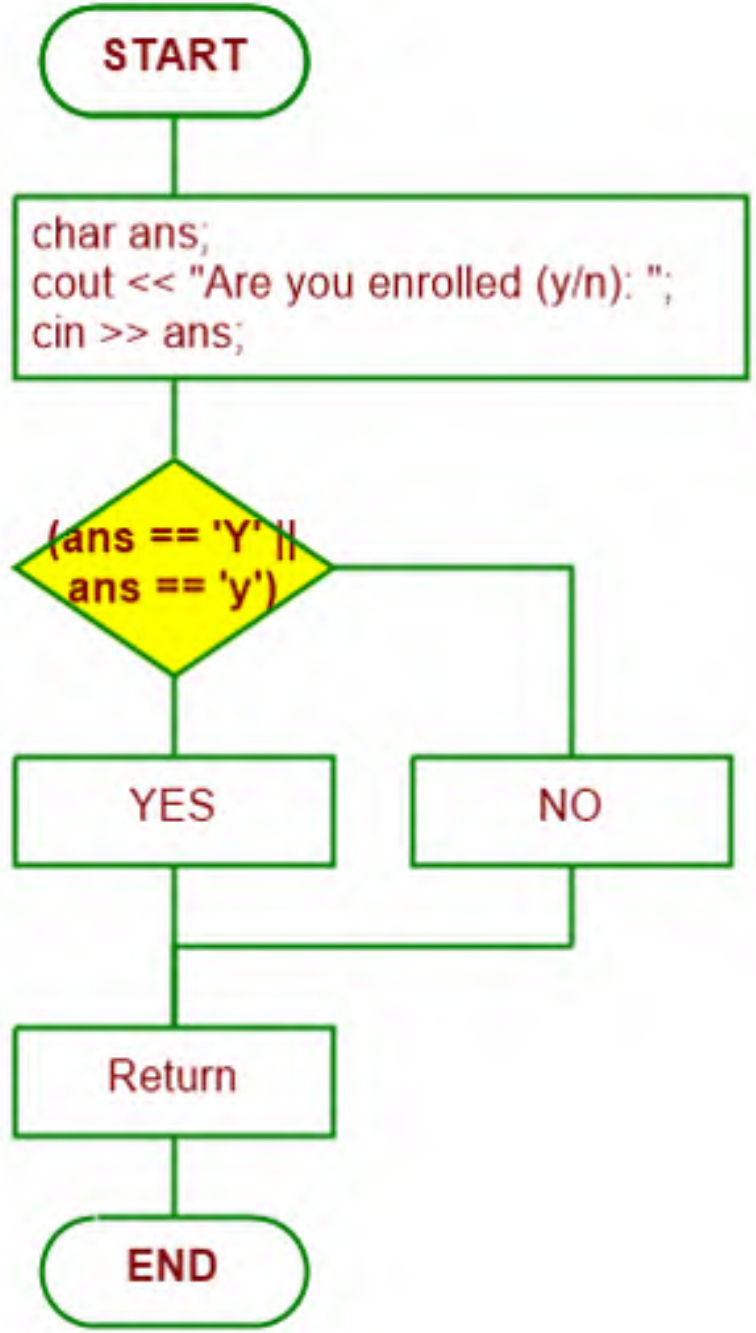


EXAMPLE 3.9 USER-FRIENDLY INPUT

This program allows the user to input either a “Y” or a “y” for “yes”:

```
int main( )
{
        char ans;
        cout << "Are you enrolled (y/n): ";
        cin >> ans;
        if (ans == 'Y' || ans == 'y') cout << "You are enrolled.\n";
        else cout << "You are not enrolled.\n";
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    char ans;
    cout << "Are you enrolled (y/n): ";
    cin >> ans;
    if (ans == 'Y' || ans == 'y') cout << "You are enrolled.\n";
    else cout << "You are not enrolled.\n";
    return 0;
}
```



3.7 SHORT-CIRCUITING

Compound conditions that use **&&** and || will not even evaluate the second operand of the condition unless necessary (Short Circuiting)

As the truth tables show,

- the condition p && q will be false if p is false (no need to evaluate q)
- the condition p || q will be true if p is true (no need to evaluate q)

p	q	p && q
T	T	T
T	F	F
F	T	F
F	F	F

p	q	p || q
T	T	T
T	F	T
F	T	T
F	F	F

EXAMPLE 3.10 SHORT-CIRCUITING

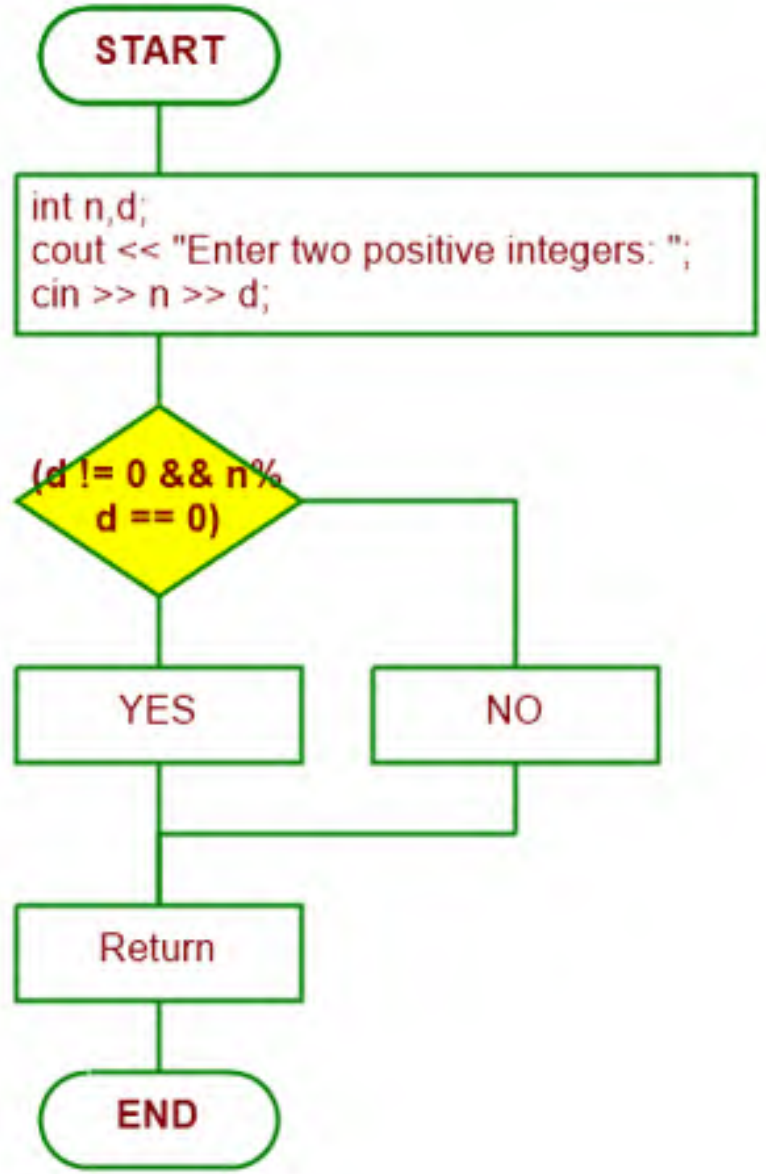
This program tests integer divisibility:

```
int main( )
{
        int n, d;
        cout << "Enter two positive integers: ";
        cin >> n >> d;

        if (d != 0 && n%d == 0)
        cout << d << " divides " << n << endl;

        else
        cout << d << " does not divide " << n << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n,d;
    cout << "Enter two positive integers: ";
    cin >> n >> d;
    if (d != 0 && n%d == 0) cout << d << " divides " << n << endl;
    else cout << d << " does not divide " << n << endl;
    return 0;
}
```



3.8 BOOLEAN EXPRESSIONS

A *Boolean expression* is a condition that is either true or false.

Example: the expressions

d > 0, n%d == 0, and (d > 0 && n%d == 0) are Boolean expressions.

Boolean expressions evaluate to integer values.

The value 0 (zero) means “false” and every nonzero value means “true.”

NOTE (Since all nonzero integer values are interpreted as meaning “true,” Boolean expressions are often masked.)

EXAMPLE 3.11 ANOTHER LOGICAL ERROR

This program is erroneous:

```
int main( )
{
        int n1, n2, n3;
        cout << "Enter three integers: ";
        cin >> n1 >> n2 >> n3;
        if (n1 >= n2 >= n3) cout << "max = x";       // LOGICAL ERROR!
}
```

3.9 NESTED SELECTION STATEMENTS

Like compound statements, selection statements can be used wherever any other statement can be used.

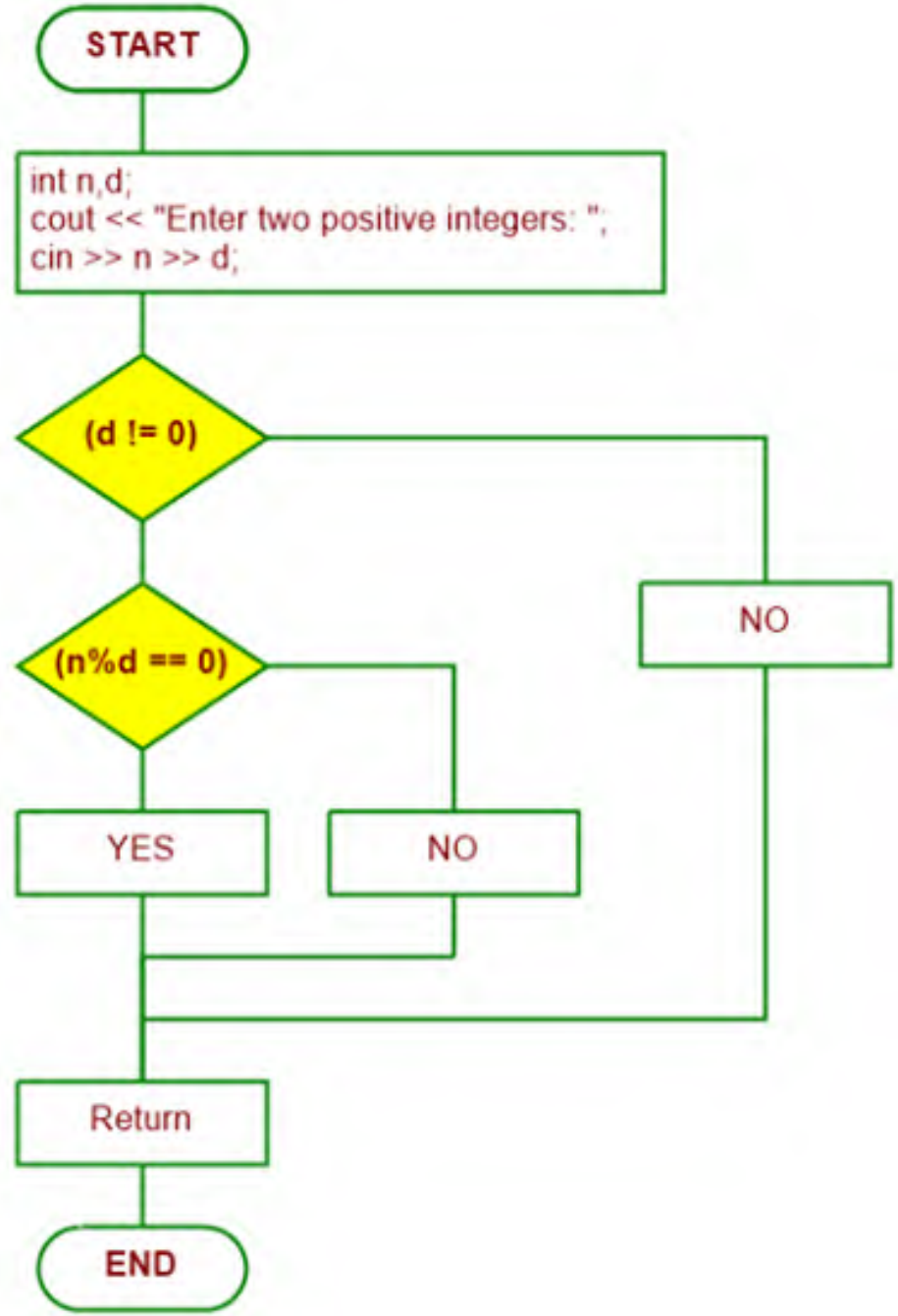
So a selection statement can be used within another selection statement. This is called *nesting* statements.

EXAMPLE 3.12 NESTING SELECTION STATEMENTS

Example of nesting selection statement:

```
int main( )
{
        int n, d;
        cout << "Enter two positive integers: ";
        cin >> n >> d;
                if (d != 0)
                        if (n%d == 0) cout << d << " divides " << n << endl;
                        else cout << d << " does not divide " << n << endl;
                else cout << d << " does not divide " << n << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n,d;
    cout << "Enter two positive integers: ";
    cin >> n >> d;
    if (d != 0)
        if (n%d == 0) cout << d << " divides " << n << endl;
        else cout << d << " does not divide " << n << endl;
    else cout << d << " does not divide " << n << endl;
    return 0;
}
```

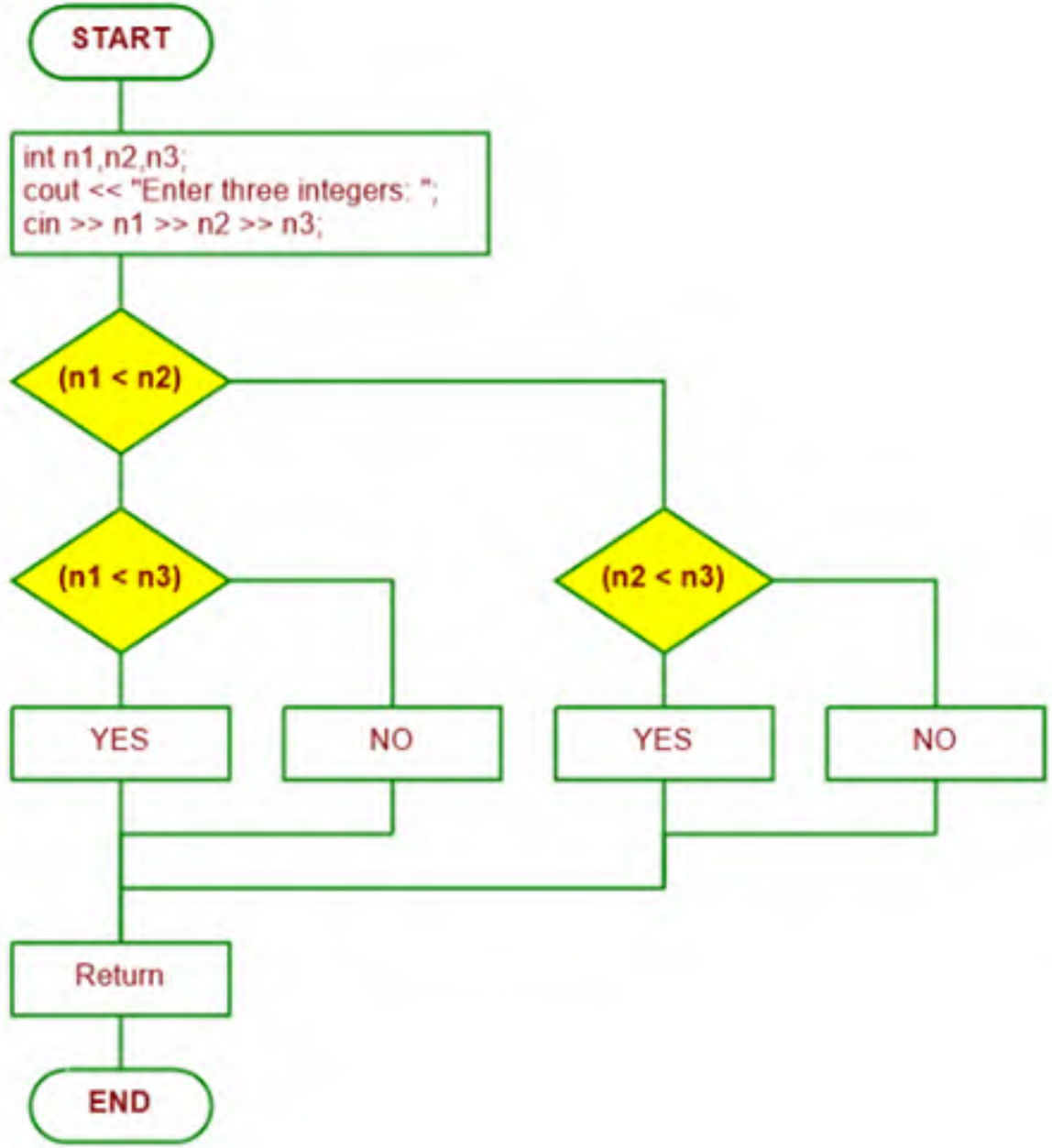


EXAMPLE 3.13 USING NESTED SELECTION STATEMENTS

This version uses nested if..else statements to find the minimum of three integers:

```
int main( )
{
        int n1, n2, n3;
        cout << "Enter three integers: ";
        cin >> n1 >> n2 >> n3;
                if (n1 < n2)
                        if (n1 < n3) cout << "Their minimum is " << n1 << endl;
                        else cout << "Their minimum is " << n3 << endl;
                else      // n1 >= n2
                        if (n2 < n3) cout << "Their minimum is " << n2 << endl;
                        else cout << "Their minimum is " << n3 << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int n1,n2,n3;
    cout << "Enter three integers: ";
    cin >> n1 >> n2 >> n3;
    if (n1 < n2)
        if (n1 < n3) cout << "Their minimum is " << n1 << endl;
        else cout << "Their minimum is " << n3 << endl;
    else // n1 >= n2
        if (n2 < n3) cout << "Their minimum is " << n2 << endl;
        else cout << "Their minimum is " << n3 << endl;
    return 0;
}
```



EXAMPLE 3.14 A GUESSING GAME

This program finds a number that the user selects from 1 to 8:

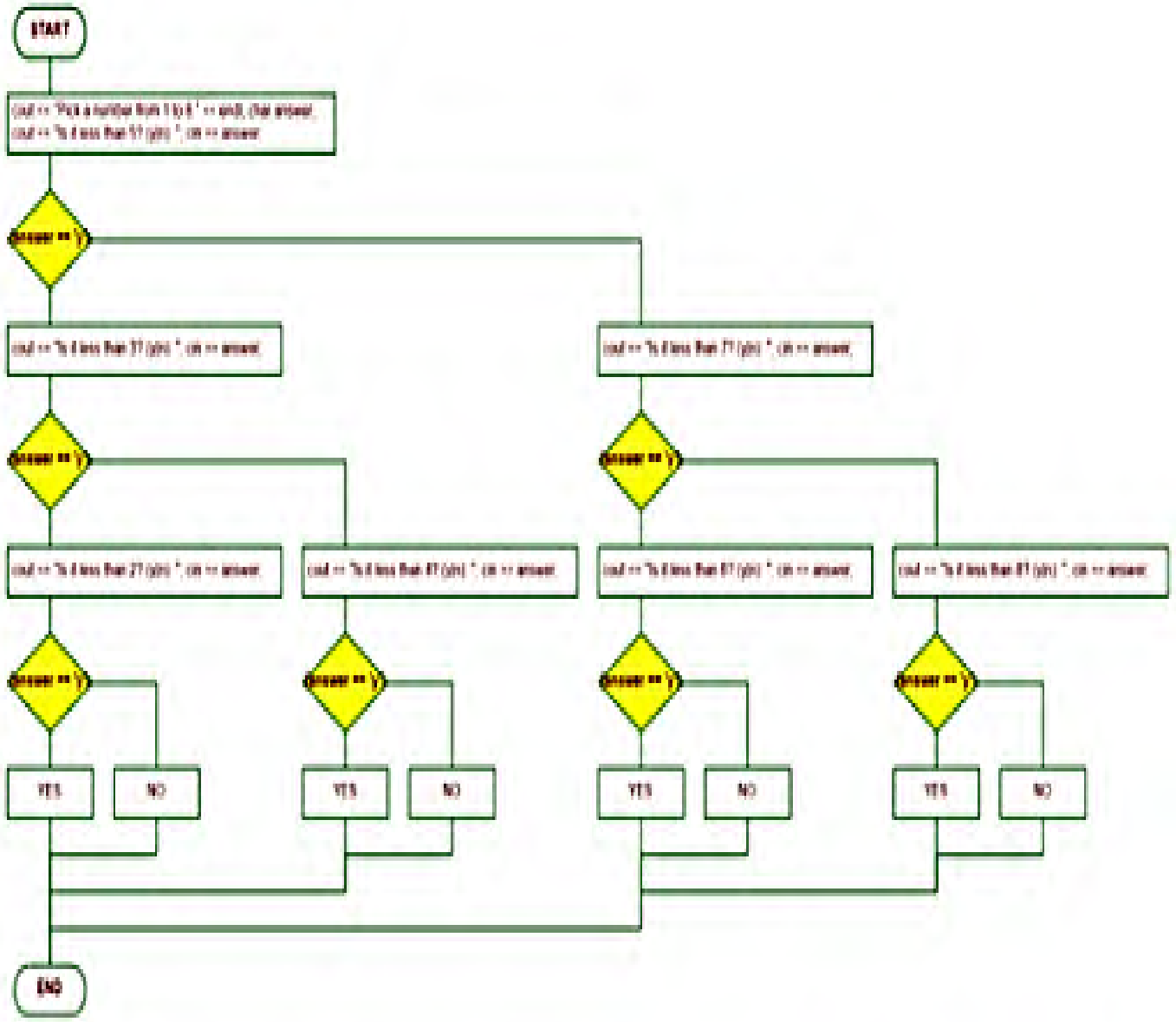
```
int main( )
{
  cout << "Pick a number from 1 to 8." << endl;
  char answer;
  cout << "Is it less than 5? (y|n): ";
  cin >> answer;
    if (answer == 'y')                                    // 1 <= n <= 4
    { cout << "Is it less than 3? (y|n): "; cin >> answer;
    if (answer == 'y')                                    // 1 <= n <= 2
    { cout << "Is it less than 2? (y|n): "; cin >> answer;
    if (answer == 'y') cout << "Your number is 1." << endl;
    else cout << "Your number is 2." << endl;
    }
    else                                                  // 3 <= n <= 4
    { cout << "Is it less than 4? (y|n): "; cin >> answer;
```

EXAMPLE 3.14 A GUESSING GAME

```
   if (answer == 'y') cout << "Your number is 3." << endl;
   else cout << "Your number is 4." << endl;
   }
   {
   }
   else                                                    // 5 <= n <= 8
   { cout << "Is it less than 7? (y|n): "; cin >> answer;
   if (answer == 'y')                                      // 5 <= n <= 6
   { cout << "Is it less than 6? (y|n): "; cin >> answer;
   if (answer == 'y') cout << "Your number is 5." << endl;
   else cout << "Your number is 6." << endl;
   }
   else                                                    // 7 <= n <= 8
   { cout << "Is it less than 8? (y|n): "; cin >> answer;
   if (answer == 'y') cout << "Your number is 7." << endl;
   else cout << "Your number is 8." << endl;
   }
   {
   }

}
```

```
int main() { cout << "Pick a number from 1 to 8." << endl; char answer;
    cout << "Is it less than 5? (y|n): "; cin >> answer;
    if (answer == 'y') // 1 <= n <= 4
    { cout << "Is it less than 3? (y|n): "; cin >> answer;
        if (answer == 'y') // 1 <= n <= 2
            { cout << "Is it less than 2? (y|n): "; cin >> answer;
            if (answer == 'y') cout << "Your number is 1." << endl;
            else cout << "Your number is 2." << endl; }
        else // 3 <= n <= 4
            { cout << "Is it less than 4? (y|n): "; cin >> answer;
            if (answer == 'y') cout << "Your number is 3." << endl;
            else cout << "Your number is 4." << endl; }
    } else // 5 <= n <= 8
    { cout << "Is it less than 7? (y|n): "; cin >> answer;
        if (answer == 'y') // 5 <= n <= 6
            { cout << "Is it less than 6? (y|n): "; cin >> answer;
            if (answer == 'y') cout << "Your number is 5." << endl;
            else cout << "Your number is 6." << endl; }
        else // 7 <= n <= 8
            { cout << "Is it less than 8? (y|n): "; cin >> answer;
            if (answer == 'y') cout << "Your number is 7." << endl;
            else cout << "Your number is 8." << endl; } }
}
```



3.10 THE ELSE IF CONSTRUCT

Nested **if..else** statements are often used to test a sequence of parallel alternatives, where only the **else** clauses contain further nesting.

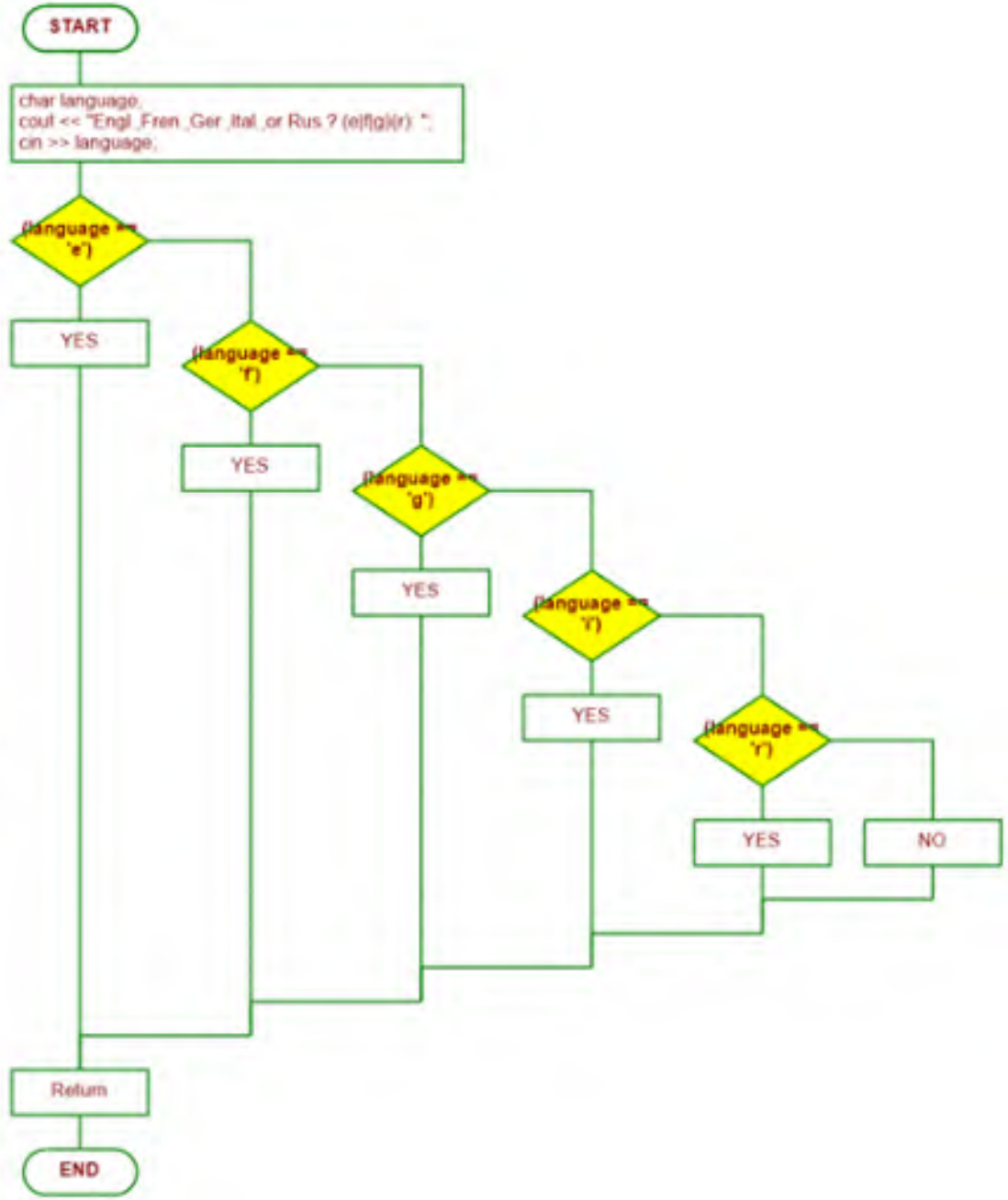
In that case, the resulting compound statement is usually formatted by lining up the **else if** phrases to emphasize the parallel nature of the logic.

EXAMPLE 3.15 USING THE ELSE IF CONSTRUCT FOR PARALLEL ALTERNATIVES

This program requests the user's language and then prints a greeting in that language:

```
int main( )
{
char language;
cout << "English, French, German, Italian, or Russian? (e|f|g|i|r): ";
cin >> language;
if (language == 'e') cout << "Welcome to ProjectEuclid.";
        else if (language == 'f') cout << "Bon jour,ProjectEuclid.";
        else if (language == 'g') cout << "Guten tag,ProjectEuclid.";
        else if (language == 'i') cout << "Bon giorno,ProjectEuclid.";
        else if (language == 'r') cout << "Dobre utre,ProjectEuclid.";
else cout << "Sorry; we don't speak your language.";
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    char language;
    cout << "Engl.,Fren.,Ger.,Ital.,or Rus.? (e|f|g|i|r): ";
    cin >> language;
    if (language == 'e') cout << "Welcome to ProjectEuclid.";
    else if (language == 'f') cout << "Bon jour,ProjectEuclid.";
        else if (language == 'g') cout << "Guten tag,ProjectEuclid.";
            else if (language == 'i') cout << "Bon giorno,ProjectEuclid.";
                else if (language == 'r') cout << "Dobre utre,ProjectEuclid.";
                    else cout << "Sorry; we don't speak your language.";
    return 0;
}
```

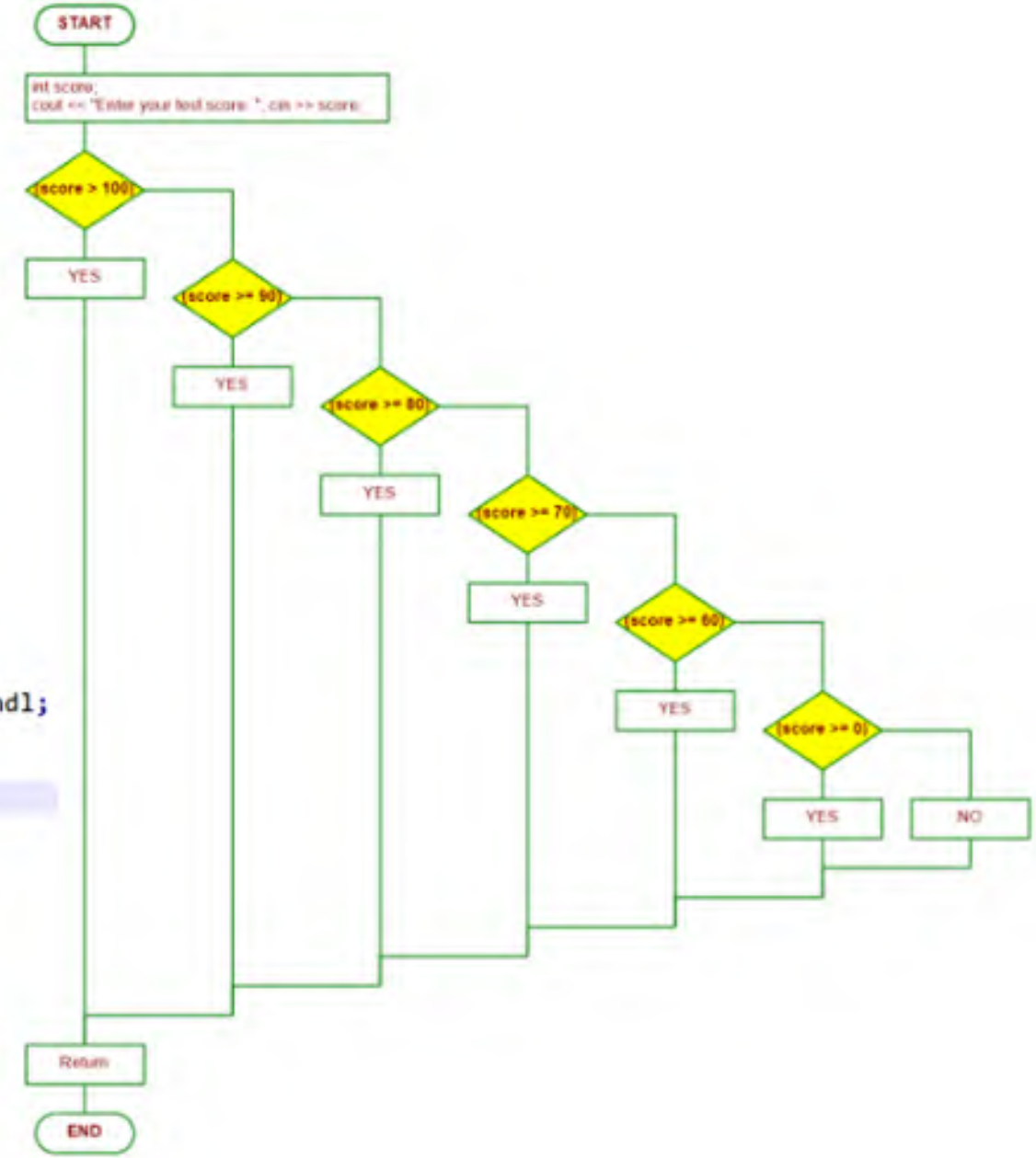


EXAMPLE 3.16 USING THE ELSE IF CONSTRUCT TO SELECT A RANGE OF SCORES

This program converts a test score into its equivalent letter grade:

```
int main( )
{
int score;
cout << "Enter your test score: "; cin >> score;
        if (score > 100) cout << "Error: that score is out of range.";
                else if (score >= 90) cout << "Your grade is an A." << endl;
                else if (score >= 80) cout << "Your grade is a B." << endl;
                else if (score >= 70) cout << "Your grade is a C." << endl;
                else if (score >= 60) cout << "Your grade is a D." << endl;
                else if (score >= 0) cout << "Your grade is an F." << endl;
        else cout << "Error: that score is out of range.";
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout << "Enter your test score: "; cin >> score;
    if (score > 100) cout << "Error: that score is out of range.";
    else if (score >= 90) cout << "Your grade is an A." << endl;
        else if (score >= 80) cout << "Your grade is a B." << endl;
            else if (score >= 70) cout << "Your grade is a C." << endl;
                else if (score >= 60) cout << "Your grade is a D." << endl;
                    else if (score >= 0) cout << "Your grade is an F." << endl;
                        else cout << "Error: that score is out of range.";
    return 0;
}
```



3.11 THE SWITCH STATEMENT

The switch statement can be used instead of the else if construct to implement a sequence of parallel alternatives.

switch (*expression*)
{

case *constant1***:** *statementList1***;**
case *constant2***:** *statementList2***;**
case *constant3***:** *statementList3***;**
: :
case *constantN***:** *statementListN***;**
default: *statementList0***;**

}

This evaluates the *expression* and then looks for its value among the **case** constants.

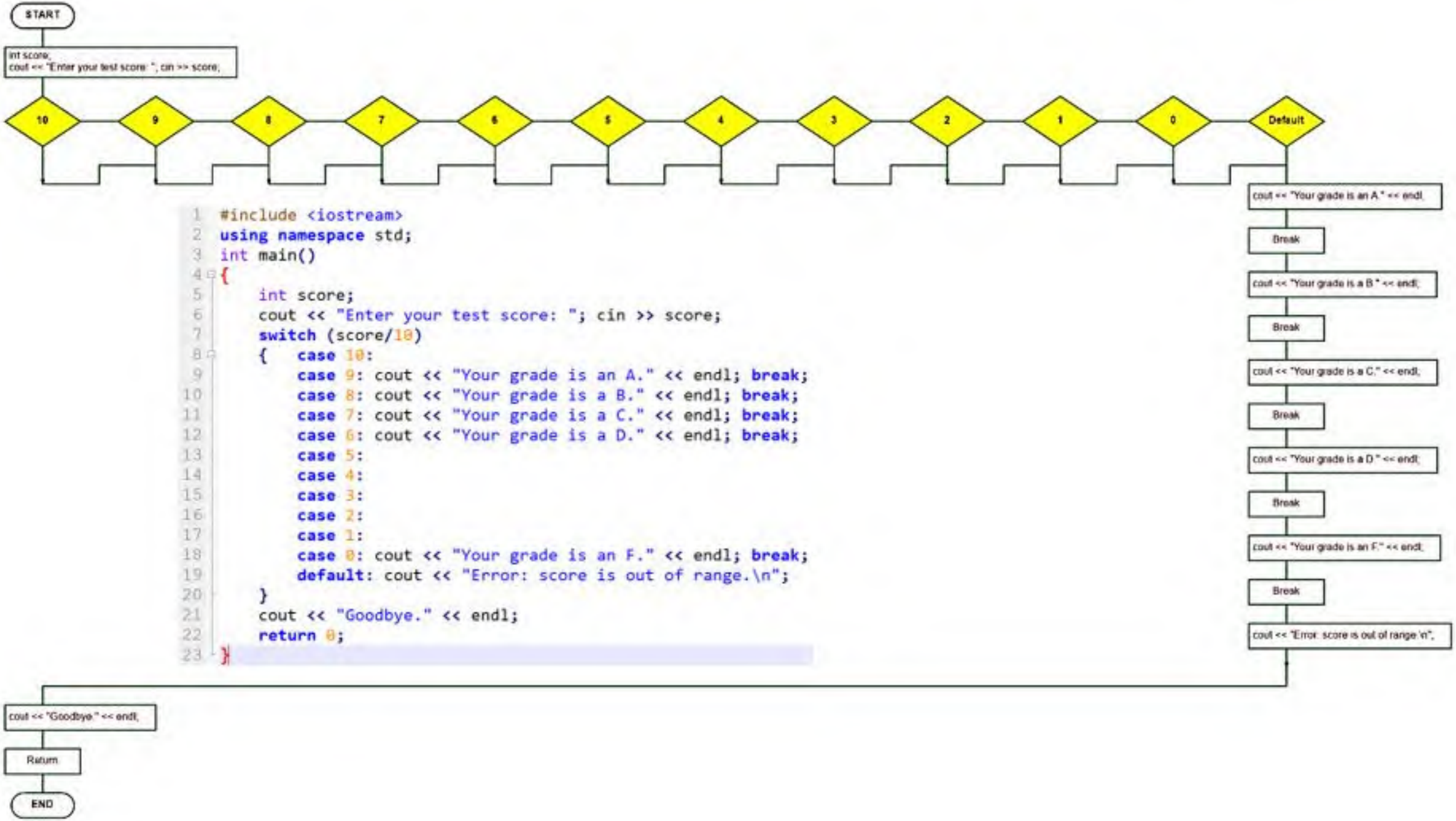
If the value is found among the constants listed, then the statements in the corresponding *statementList* are executed. Otherwise if there is a **default** (which is optional), then the program branches to its *statementList*.

The *expression* must evaluate to an integral type and the *constants* must be integral type as well.

EXAMPLE 3.17 USING A SWITCH STATEMENT TO SELECT A RANGE OF SCORES

This program converts a test score into its equivalent letter grade:

```
int main( )
{
        int score;
        cout << "Enter your test score: "; cin >> score;
        switch (score/10)
        {       case 10:
                case 9: cout << "Your grade is an A." << endl; break;
                case 8: cout << "Your grade is a B." << endl; break;
                case 7: cout << "Your grade is a C." << endl; break;
                case 6: cout << "Your grade is a D." << endl; break;
                case 5:
                case 4:
                case 3:
                case 2:
                case 1:
                case 0: cout << "Your grade is an F." << endl; break;
                default: cout << "Error: score is out of range.\n";
        }
cout << "Goodbye." << endl;
}
```




```cpp
#include <iostream>
using namespace std;
int main()
{
    int score;
    cout << "Enter your test score: "; cin >> score;
    switch (score/10)
    {   case 10:
        case 9: cout << "Your grade is an A." << endl; break;
        case 8: cout << "Your grade is a B." << endl; break;
        case 7: cout << "Your grade is a C." << endl; break;
        case 6: cout << "Your grade is a D." << endl; break;
        case 5:
        case 4:
        case 3:
        case 2:
        case 1:
        case 0: cout << "Your grade is an F." << endl; break;
        default: cout << "Error: score is out of range.\n";
    }
    cout << "Goodbye." << endl;
    return 0;
}
```

EXAMPLE 3.18 AN ERRONEOUS FALL-THROUGH IN A SWITCH STATEMENT

```
int main( )
{
int score;
cout << "Enter your test score: "; cin >> score;
        switch (score/10)
        {       case 10:
                case 9: cout << "Your grade is an A." << endl;       // LOGICAL ERROR
                case 8: cout << "Your grade is a B." << endl;       // LOGICAL ERROR
                case 7: cout << "Your grade is a C." << endl;       // LOGICAL ERROR
                case 6: cout << "Your grade is a D." << endl;       // LOGICAL ERROR
                case 5:
                case 4:
                case 3:
                case 2:
                case 1:
                case 0: cout << "Your grade is an F." << endl;      // LOGICAL ERROR
                default: cout << "Error: score is out of range.\n";
        }
cout << "Goodbye." << endl;
}
```

3.12 THE CONDITIONAL EXPRESSION OPERATOR

A special operator that often can be used in place of the **if…else** statement.

Syntax:
condition **?** *expression1* **:** *expression2*

It is a *ternary operator*; *i.e.*, it combines three operands to produce a value.

That resulting value is either the value of *expression1* or the value of *expression2*, depending upon the Boolean value of the *condition*.

Example: the assignment
min = **(x<y ? x : y)**;
would assign the minimum of x and y to min, because if the condition **x<y** is true, the expression **(x<y ? x : y)** evaluates to x; otherwise it evaluates to y.

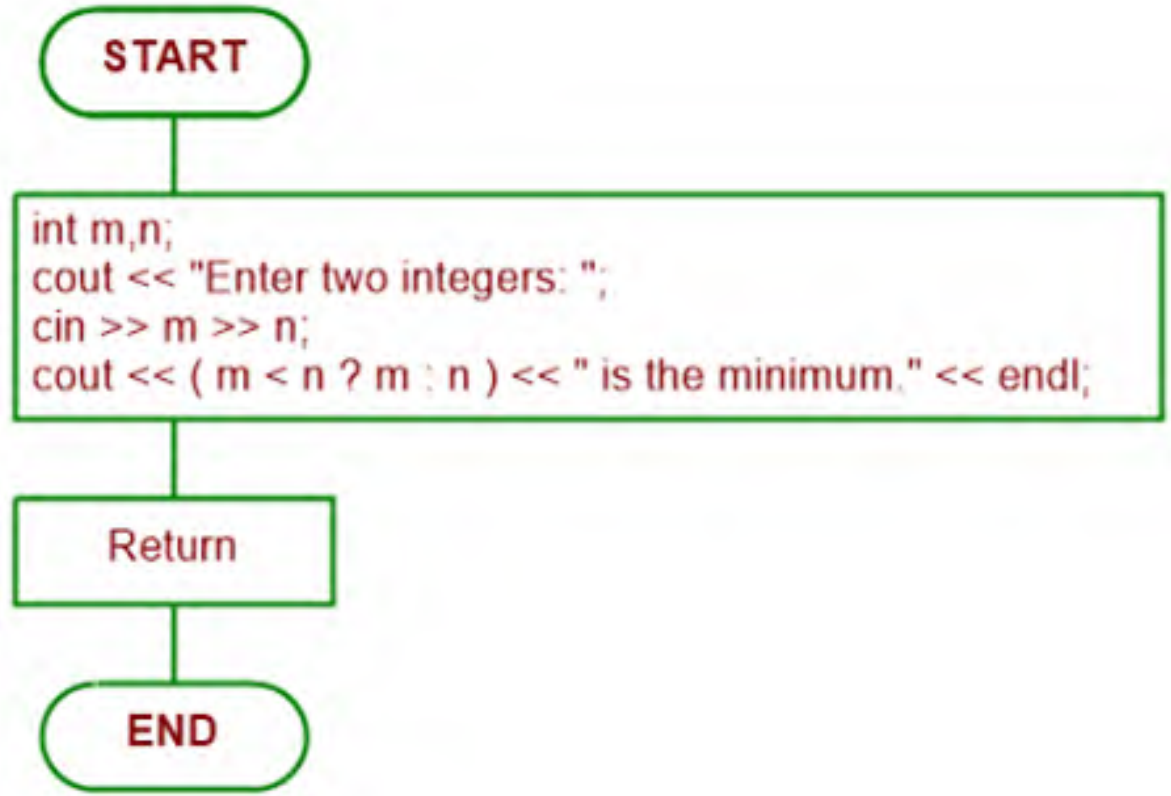
Conditional expression statements should be used sparingly: only when the condition and both expressions are very simple.

EXAMPLE 3.19 FINDING THE MINIMUM AGAIN

This program finds the minimum of two integers:

```
int main( )
{
        int m, n;
        cout << "Enter two integers: ";
        cin >> m >> n;
        cout << ( m<n ? m : n ) << " is the minimum." << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int m,n;
    cout << "Enter two integers: ";
    cin >> m >> n;
    cout << ( m < n ? m : n ) << " is the minimum." << endl;
    return 0;
}
```



UNIT 4

ITERATION (LOOP)

The repetition of a statement or block of statements in a program

4.1 THE WHILE STATEMENT

- The syntax for the while statement is
  - while (condition) statement;

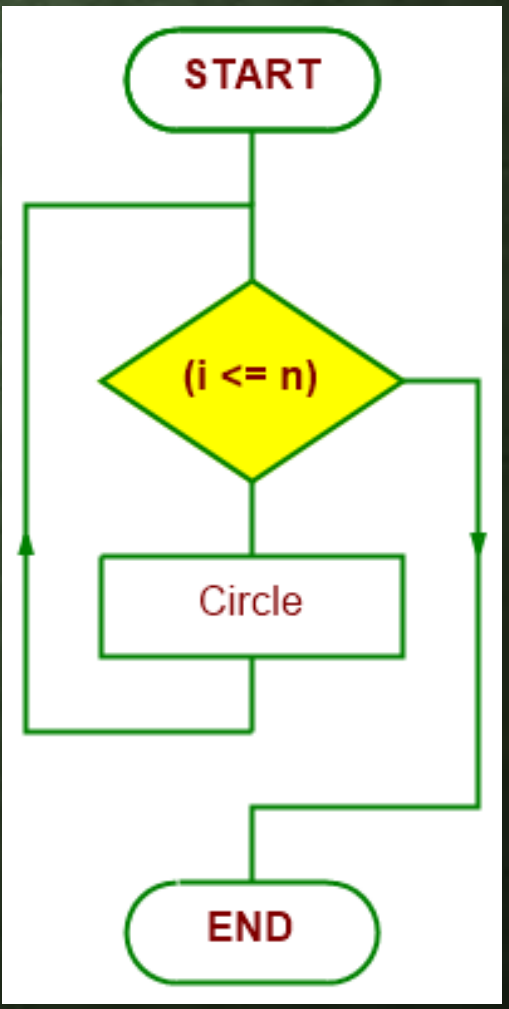
  where condition is an integral expression and statement is any executable statement.
- If the value of the expression is zero (meaning "false"), then the statement is ignored and program execution immediately jumps to the next statement that follows the while statement.
- If the value of the expression is nonzero (meaning "true"), then the statement is executed repeatedly until the expression evaluates to zero.

EXAMPLE 4.1 USING A WHILE LOOP TO COMPUTE A SUM OF CONSECUTIVE INTEGERS

This program computes the sum $1 + 2 + 3 + \cdots + n$, for an input integer n:

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (i <= n)
    sum += i++;
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

4.1.cpp



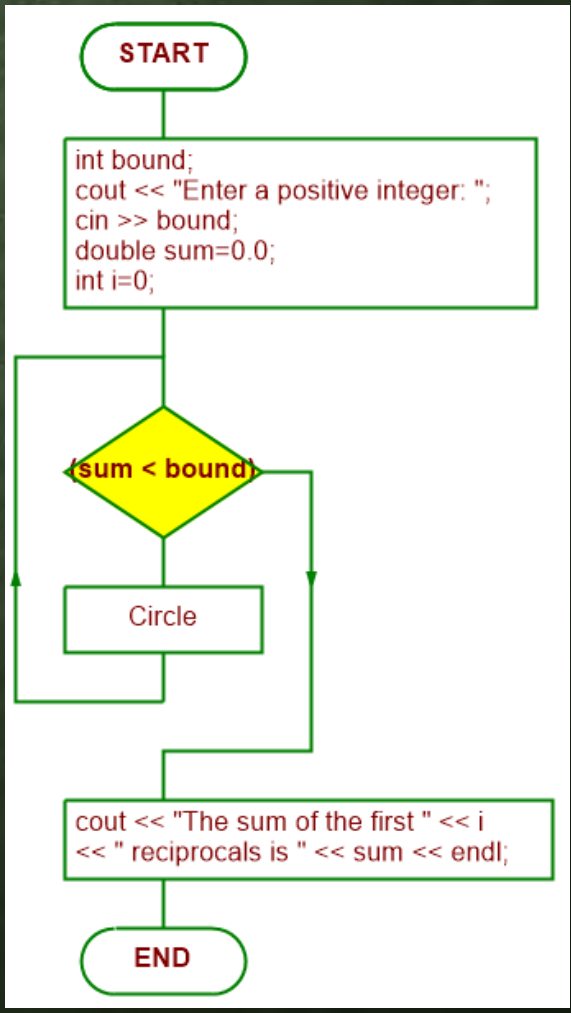
EXAMPLE 4.2 USING A WHILE LOOP TO COMPUTE A SUM OF RECIPROCALS

This program computes the sum of reciprocals $s = 1 + 1/2 + 1/3 + \cdots + 1/n$, where n is the smallest integer for which $n \geq s$:

```
int main()
{ int bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  double sum=0.0;
  int i=0;
  while (sum < bound)
    sum += 1.0/++i;
  cout << "The sum of the first " << i
       << " reciprocals is " << sum << endl;
}
```

i	sum
0	0.00000
1	1.00000
2	1.50000
3	1.83333
4	2.08333
5	2.28333
6	2.45000

4.2.cpp

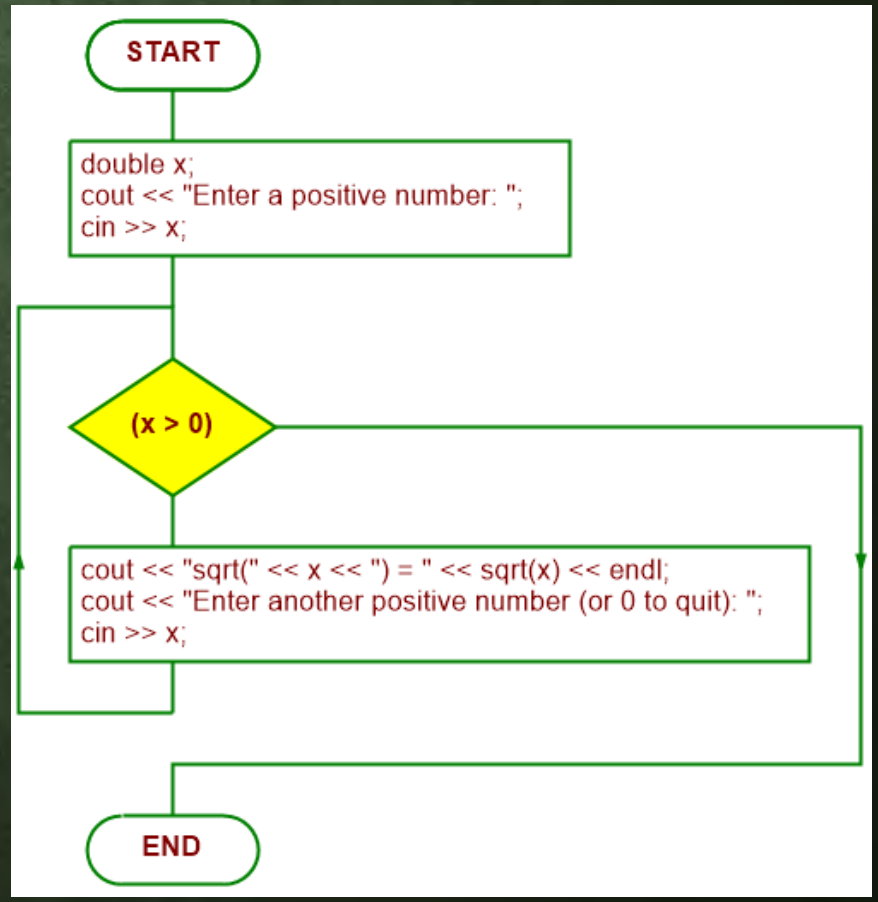


EXAMPLE 4.3 USING A WHILE LOOP TO REPEAT A COMPUTATION

This program prints the square root of each number input by the user. It uses a `while` loop to allow any number of computations in a single run of the program:

```
int main()
{ double x;
  cout << "Enter a positive number: ";
  cin >> x;
  while (x > 0)
  { cout << "sqrt(" << x << ") = " << sqrt(x) << endl;
    cout << "Enter another positive number (or 0 to quit): ";
    cin >> x;
  }
}
```

4.3.cpp



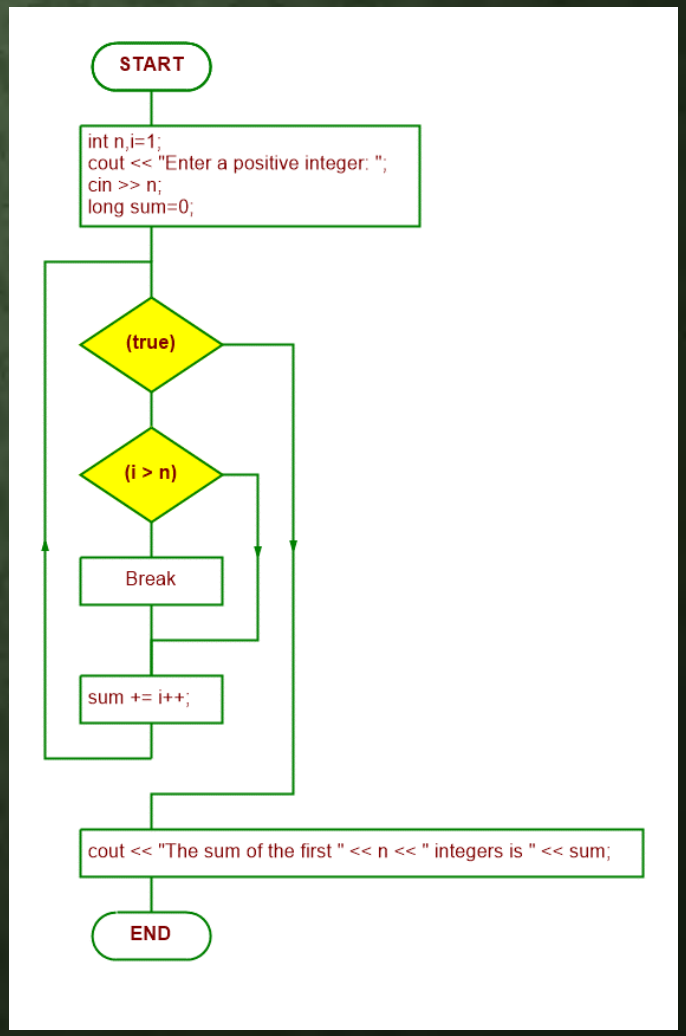
4.2 TERMINATING A LOOP

- The break statement is used to control loops.

EXAMPLE 4.4 USING A BREAK STATEMENT TO TERMINATE A LOOP

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;  // terminates the loop immediately
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

4.4.cpp

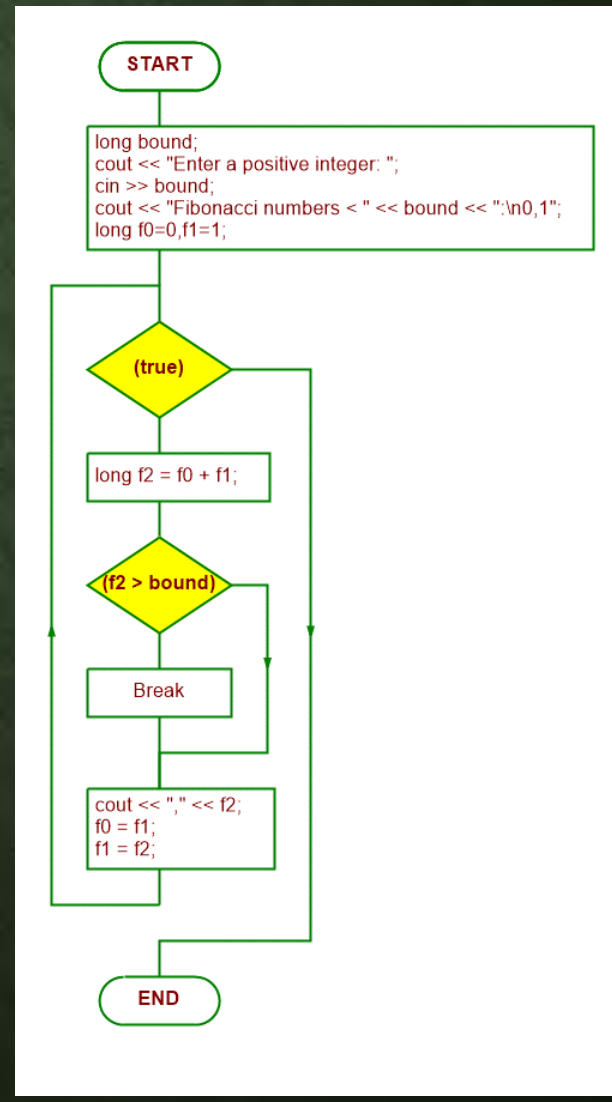


EXAMPLE 4.5 THE FIBONACCI NUMBERS

n	F_n
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	35

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) break;  // terminates the loop immediately
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

4.5.cpp

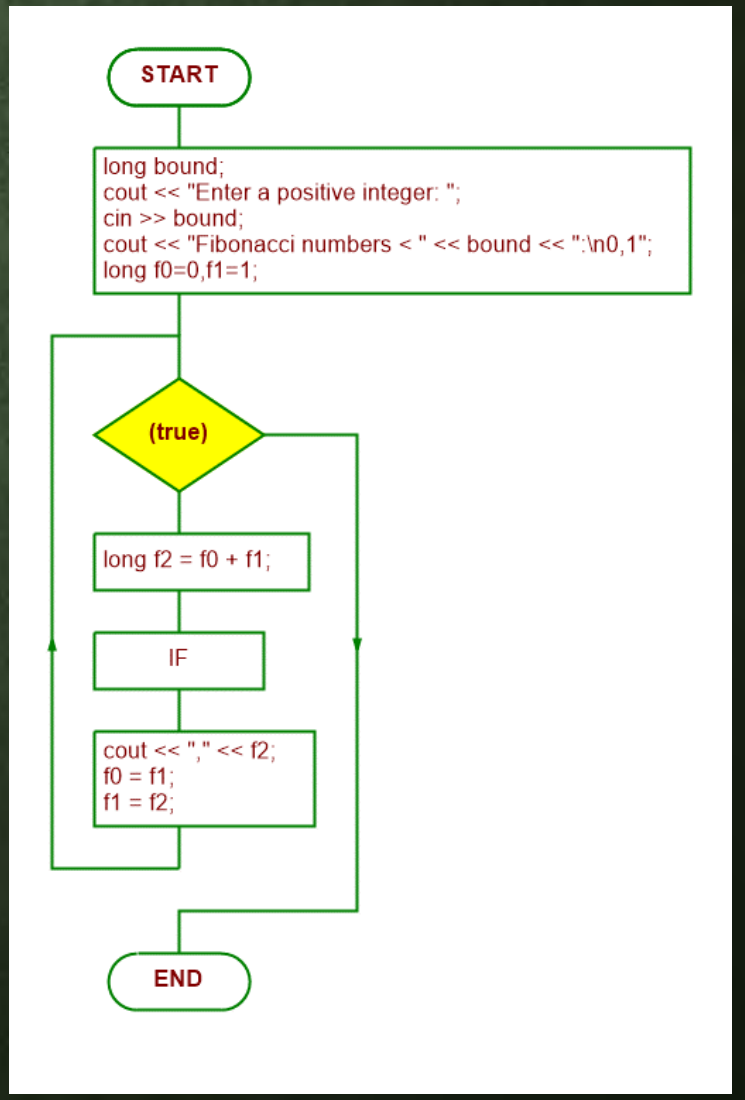


EXAMPLE 4.6 USING THE EXIT(0) FUNCTION

The `exit()` function provides another way to terminate a loop. When it executes, it terminates the program itself:

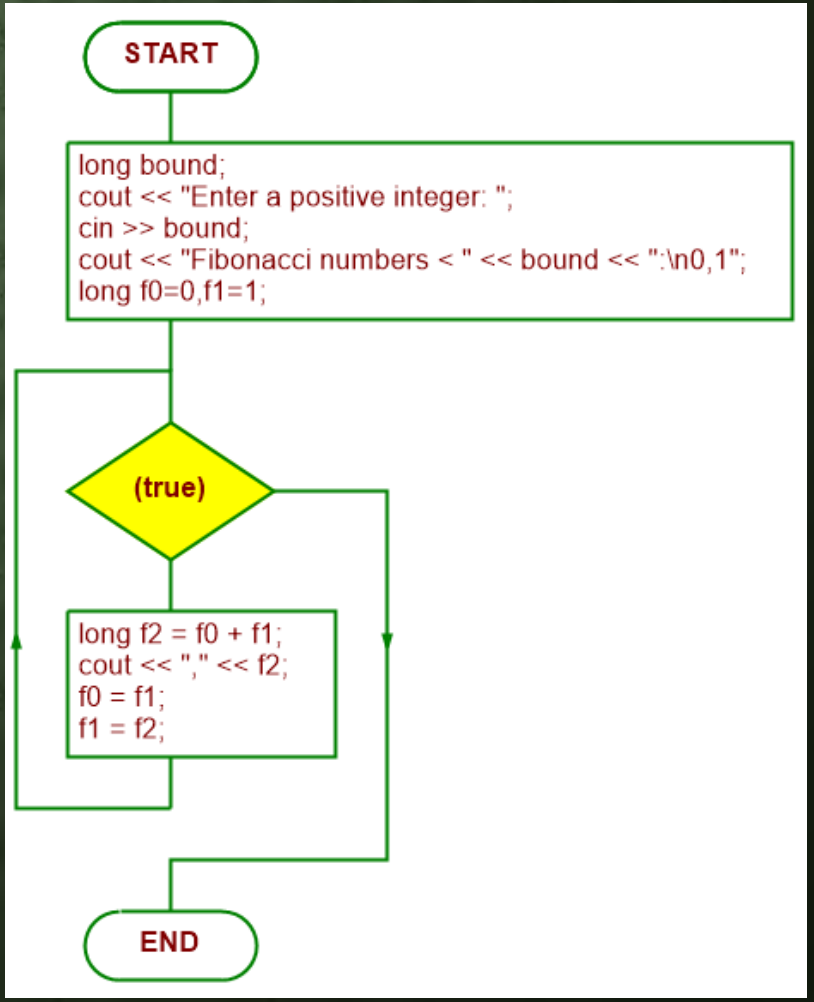
```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)
  { long f2 = f0 + f1;
    if (f2 > bound) exit(0);  // terminates the program immediately
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```

4.6.cpp



EXAMPLE 4.7 ABORTING INFINITE LOOP

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Fibonacci numbers < " << bound << ":\n0, 1";
  long f0=0, f1=1;
  while (true)          // ERROR: INFINITE LOOP!     (Press <Ctrl>+C.)
  { long f2 = f0 + f1;
    cout << ", " << f2;
    f0 = f1;
    f1 = f2;
  }
}
```



4.3 THE DO…WHILE STATEMENT

- The syntax for the do…while statement is

  do statement while (condition);

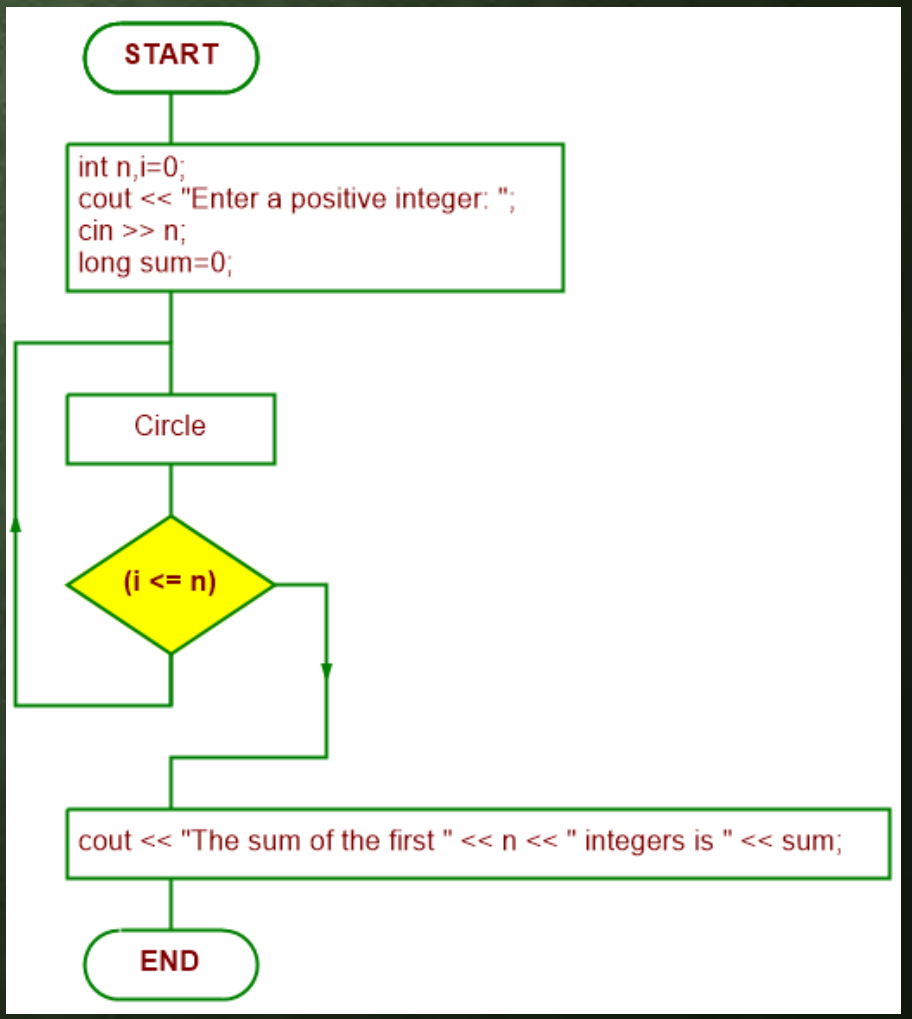
  where condition is an integral expression and statement is any executable statement.

- repeatedly executes the statement and then evaluates the condition until that condition evaluates to false.
- In do…while statement, its condition is evaluated at the end of the loop instead of at the beginning
- do…while loop will always iterate at least once, regardless of the value of its control condition.

EXAMPLE 4.8 USING A DO..WHILE LOOP TO COMPUTE A SUM OF CONSECUTIVE INTEGERS

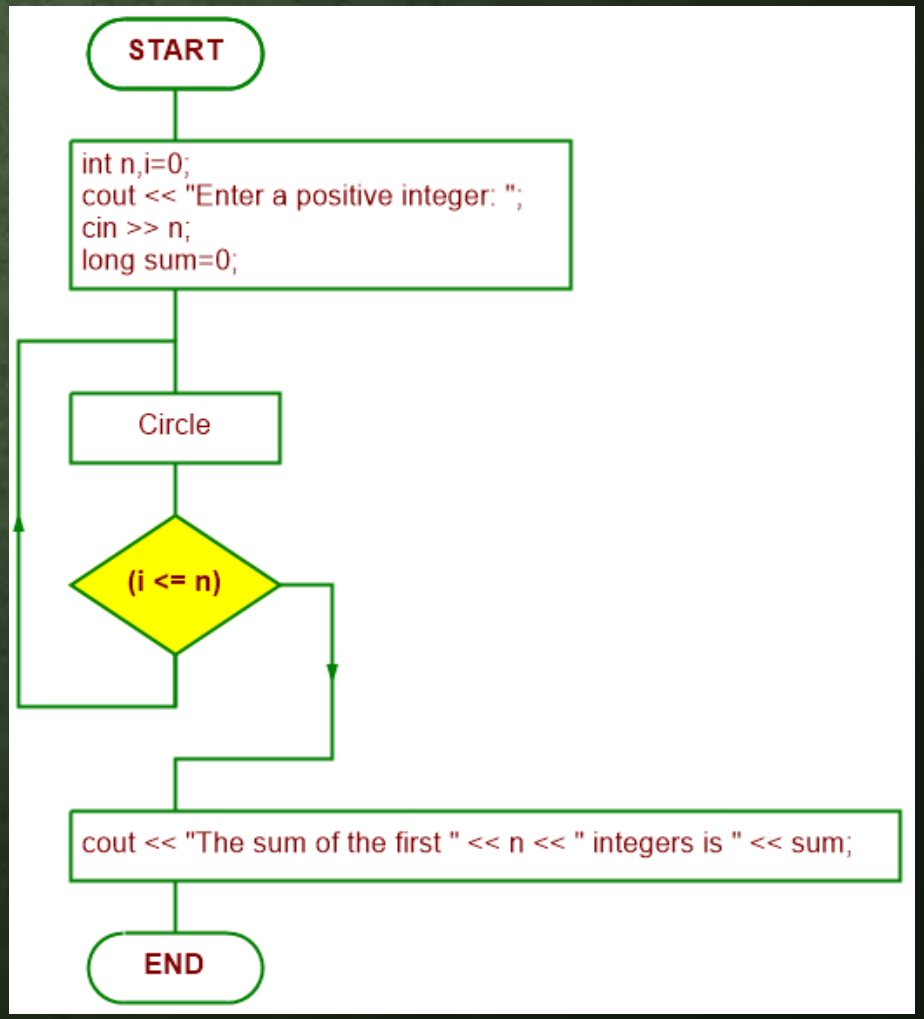
```
int main()
{ int n, i=0;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  do
    sum += i++;
  while (i <= n);
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

4.8.cpp



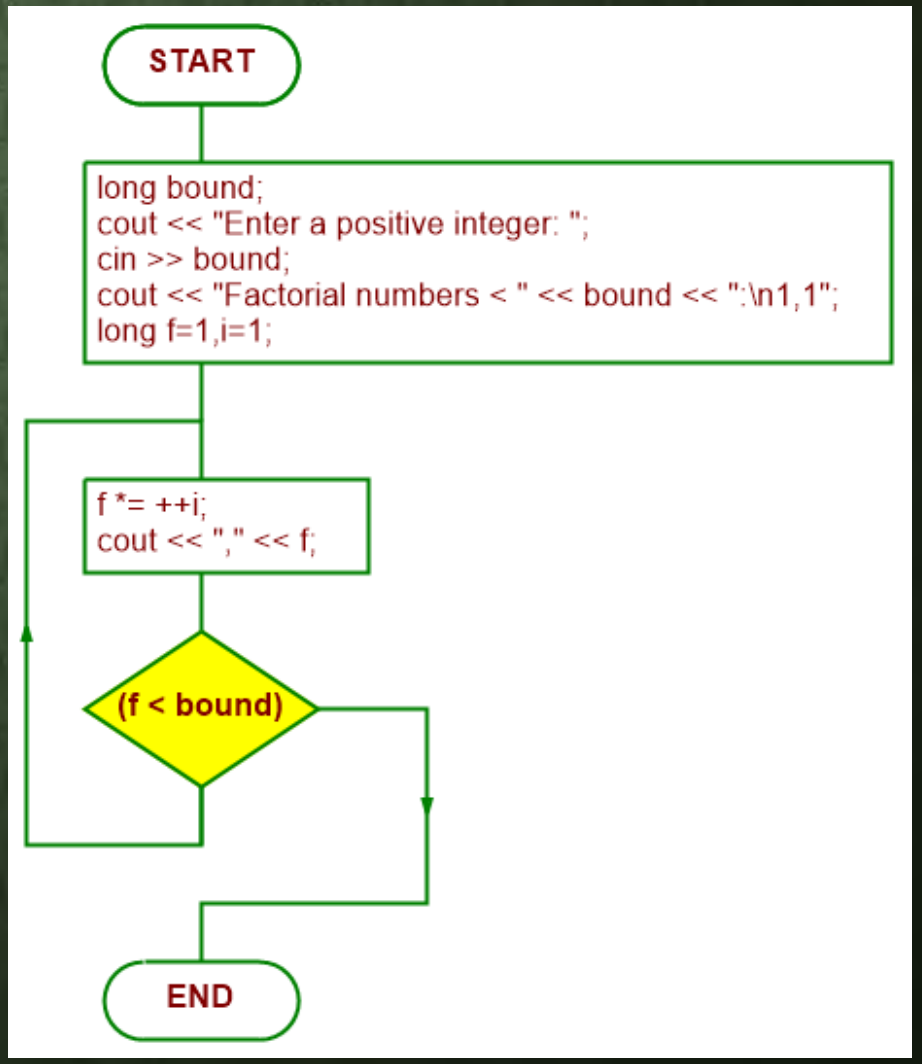
EXAMPLE 4.8 USING A DO..WHILE LOOP TO COMPUTE A SUM OF CONSECUTIVE INTEGERS

```
int main()
{ int n, i=0;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  do
    sum += i++;
  while (i <= n);
  cout << "The sum of the first " << n << " integers is " << sum;
}
```



EXAMPLE 4.9 THE FACTORIAL NUMBERS

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Factorial numbers < " << bound << ":\n1, 1";
  long f=1, i=1;
  do
  { f *= ++i;
    cout << ", " << f;
  }
  while (f < bound);
}
```



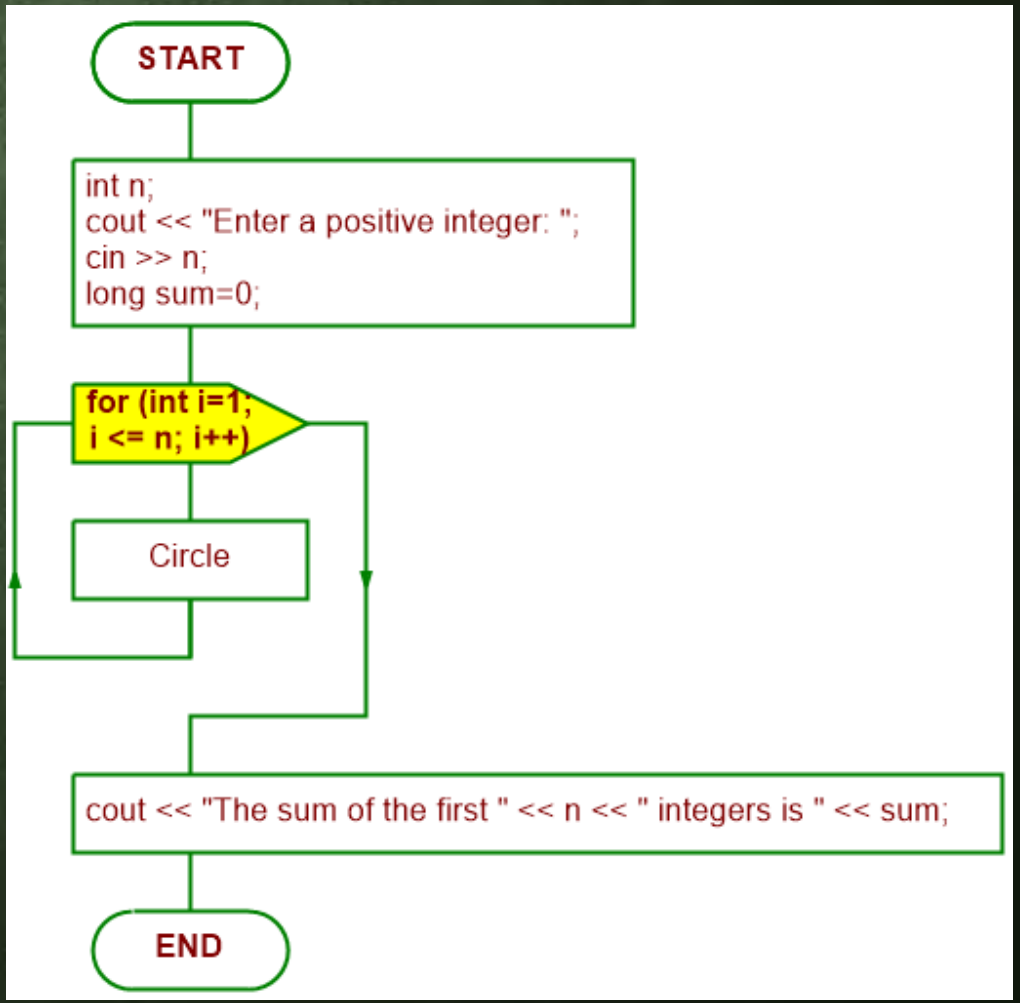
4.4 THE FOR STATEMENT

- The syntax for the for statement is
  for (*initialization***;** *condition***;** *update***)** *statement***;**
  where *initialization*, *condition*, and *update* are optional expressions, and *statement* is any executable statement.
- The *initialization* expression is used to declare and/or initialize control variable(s) for the loop; it is evaluated first, before any iteration occurs.
- The *condition* expression is used to determine whether the loop should continue iterating; it is evaluated immediately after the initialization; if it is true, the statement is executed.
- The *update* expression is used to update the control variable(s); it is evaluated after the statement is executed. So the sequence of events that generate the iteration are:
  1. evaluate the *initialization* expression;
  2. if the value of the *condition* expression is false, terminate the loop;
  3. execute the *statement*;
  4. evaluate the *update* expression;
  5. repeat steps 2–4.

EXAMPLE 4.10 USING A FOR LOOP TO COMPUTE A SUM OF CONSECUTIVE INTEGERS

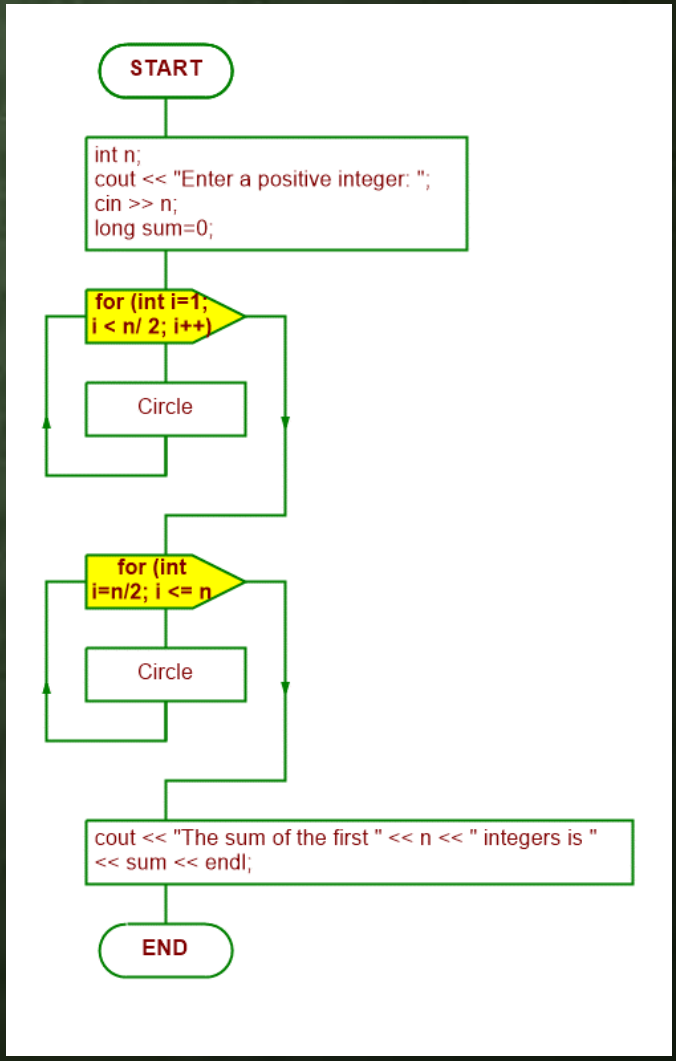
```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  for (int i=1; i <= n; i++)
    sum += i;
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

4.10.cpp



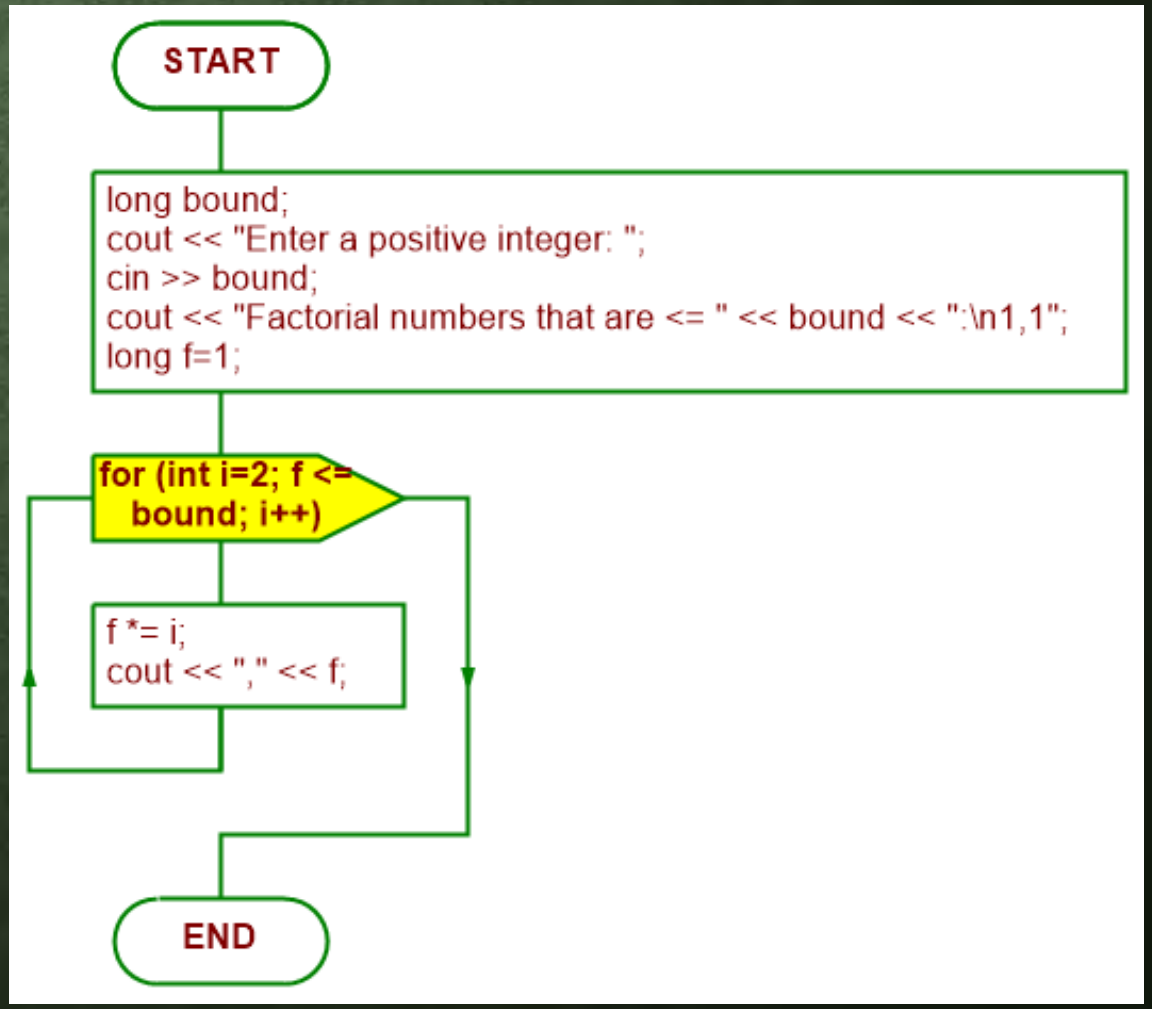
EXAMPLE 4.11 REUSING FOR LOOP CONTROL VARIABLE NAMES

```
int main()
{ int n;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  for (int i=1; i < n/2; i++)  // the scope of this i is this loop
    sum += i;
  for (int i=n/2; i <= n; i++)  // the scope of this i is this loop
    sum += i;
  cout << "The sum of the first " << n << " integers is "
       << sum << endl;
}
```



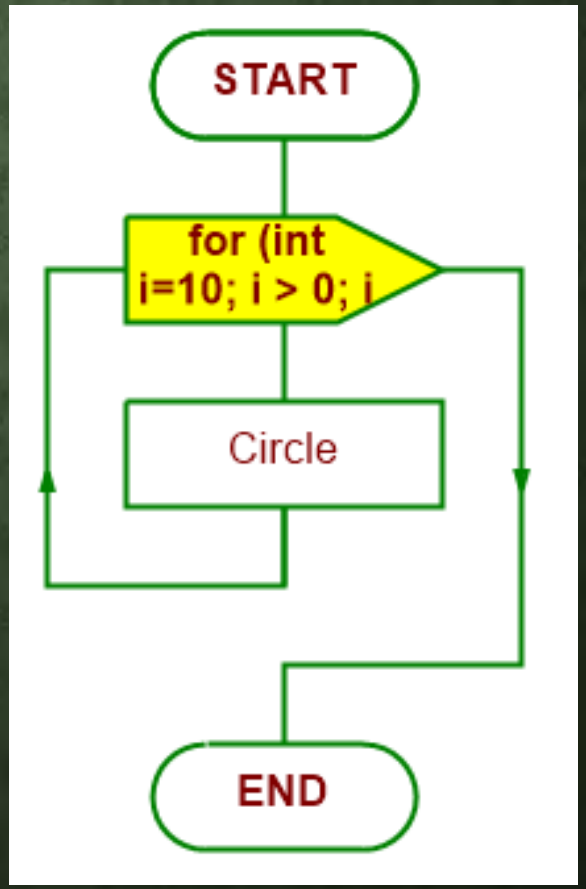
EXAMPLE 4.12 THE FACTORIAL NUMBERS AGAIN

```
int main()
{ long bound;
  cout << "Enter a positive integer: ";
  cin >> bound;
  cout << "Factorial numbers that are <= " << bound << ":\n1, 1";
  long f=1;
  for (int i=2; f <= bound; i++)
  { f *= i;
    cout << ", " << f;
  }
}
```



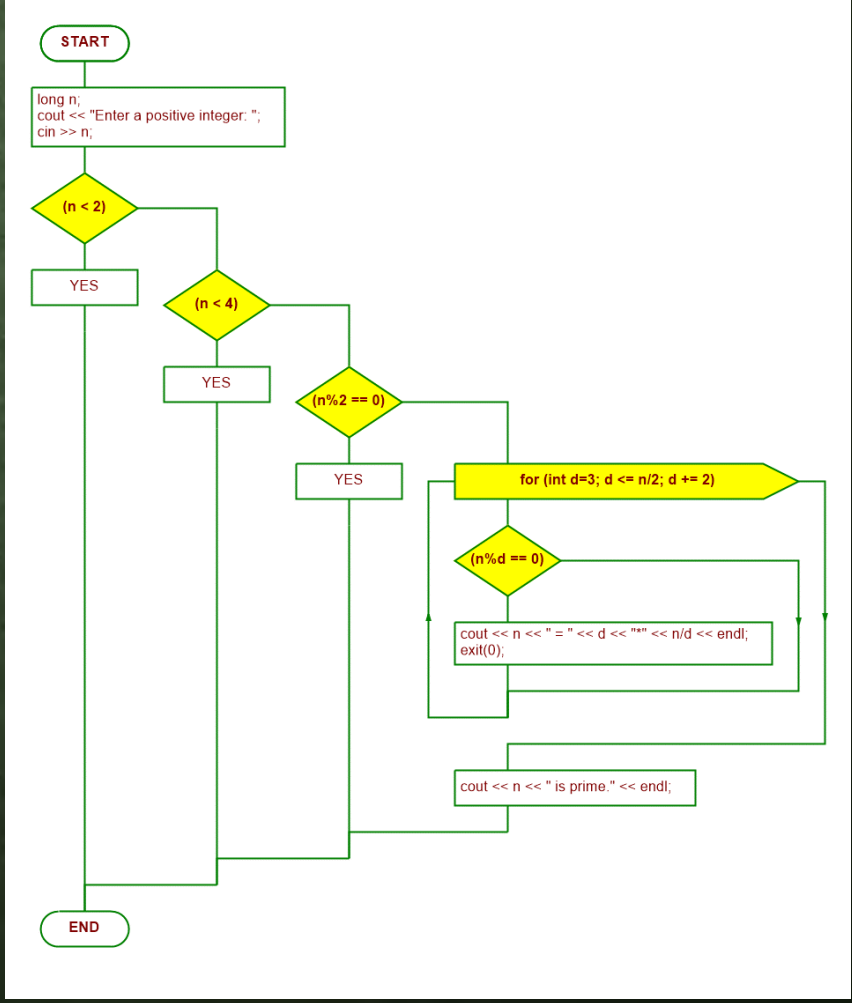
EXAMPLE 4.13 USING A DESCENDING FOR LOOP

```
int main()
{ for (int i=10; i > 0; i--)
    cout << " " << i;
}
```



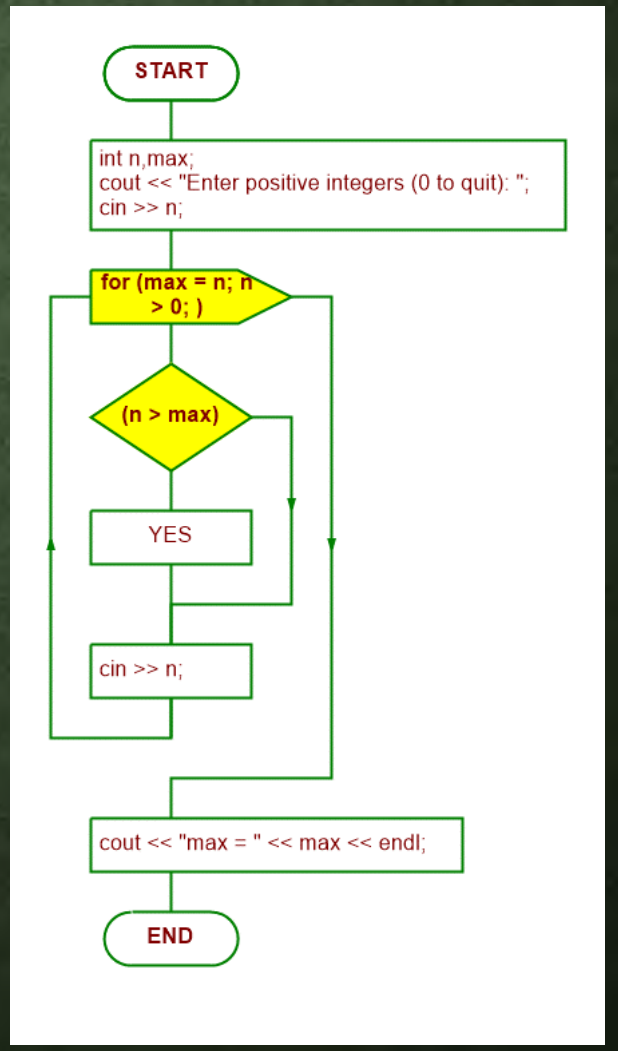
EXAMPLE 4.14 USING A FOR LOOP WITH A STEP GREATER THAN ONE

```
int main()
{ long n;
  cout << "Enter a positive integer: ";
  cin >> n;
  if (n < 2) cout << n << " is not prime." << endl;
  else if (n < 4) cout << n << " is prime." << endl;
  else if (n%2 == 0) cout << n << " = 2*" << n/2 << endl;
  else
  { for (int d=3; d <= n/2; d += 2)
      if (n%d == 0)
      { cout << n << " = " << d << "*" << n/d << endl;
        exit(0);
      }
    cout << n << " is prime." << endl;
  };
}
```



EXAMPLE 4.15 USING A SENTINEL TO CONTROL A FOR LOOP

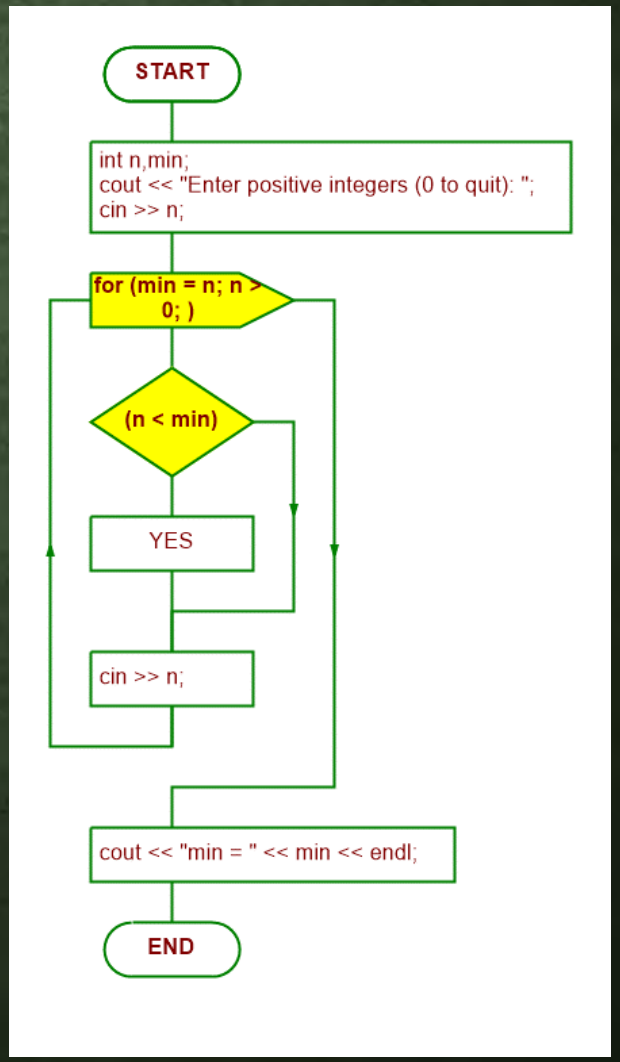
```
int main()
{ int n, max;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (max = n; n > 0; )
  { if (n > max) max = n;
    cin >> n;
  }
  cout << "max = " << max << endl;
}
```



EXAMPLE 4.16 USING A LOOP INVARIANT TO PROVE THAT A FOR LOOP IS CORRECT

```
int main()
{ int n, min;
  cout << "Enter positive integers (0 to quit): ";
  cin >> n;
  for (min = n; n > 0; )
  { if (n < min) min = n;
    // INVARIANT: min <= n for all n, and min equals one of the n
    cin >> n;
  }
  cout << "min = " << min << endl;
}
```

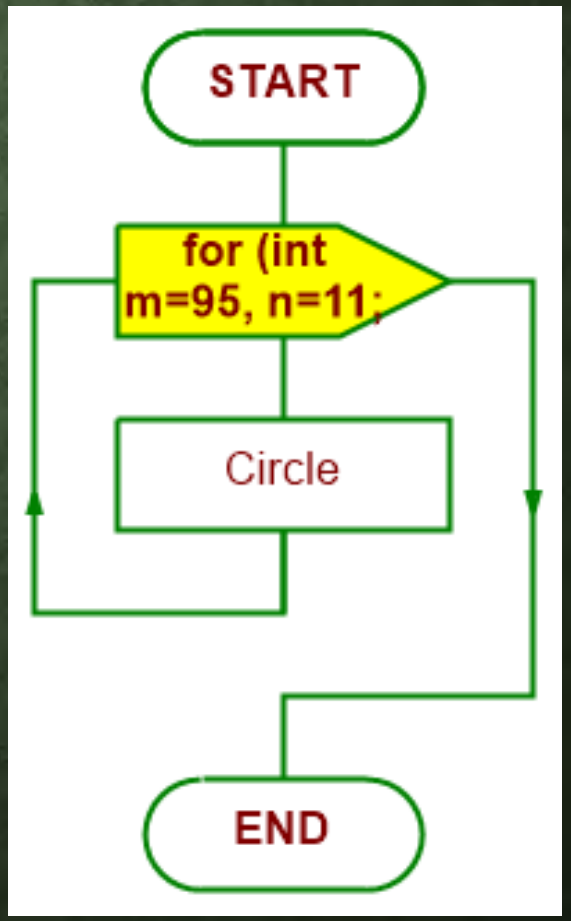
4.16.cpp



EXAMPLE 4.17 MORE THAN ONE CONTROL VARIABLE IN A FOR LOOP

```
int main()
{ for (int m=95, n=11; m%n > 0; m -= 3, n++)
    cout << m << "%" << n << " = " << m%n << endl;
}
```

4.17.cpp

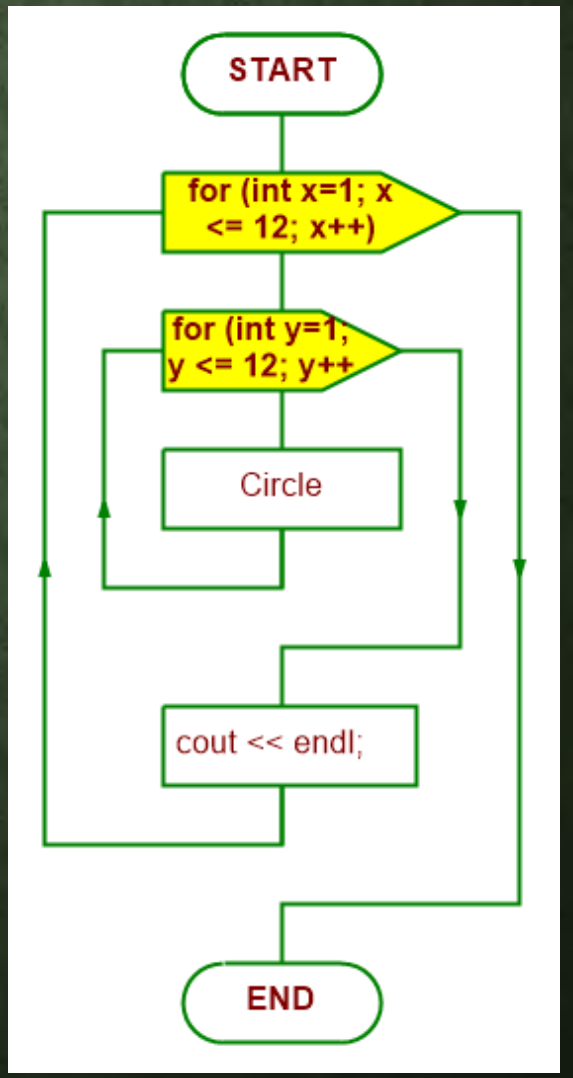


EXAMPLE 4.18 NESTING FOR LOOPS

This program prints a multiplication table:

```
#include <iomanip>    // defines setw()
#include <iostream>   // defines cout
using namespace std;
int main()
{ for (int x=1; x <= 12; x++)
  { for (int y=1; y <= 12; y++)
      cout << setw(4) << x*y;
    cout << endl;
  }
}
```

4.18.cpp



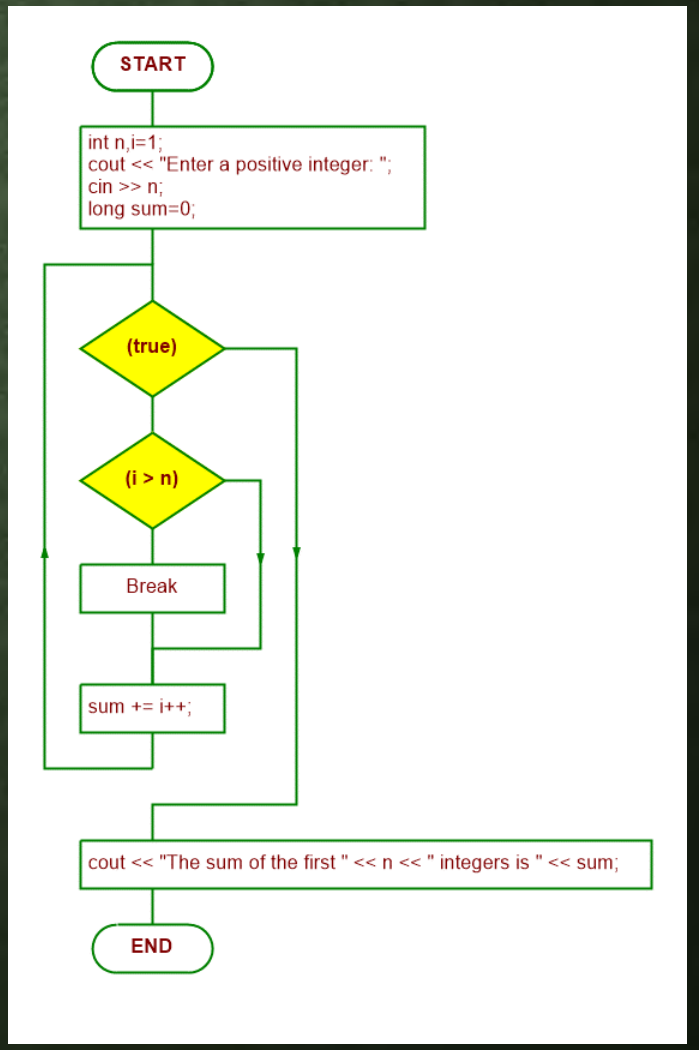
4.5 THE BREAK STATEMENT

- When the break statement executes, it terminates the loop, "breaking out" of the iteration at that point.

EXAMPLE 4.20 USING A BREAK STATEMENT TO TERMINATE A LOOP

```
int main()
{ int n, i=1;
  cout << "Enter a positive integer: ";
  cin >> n;
  long sum=0;
  while (true)
  { if (i > n) break;
    sum += i++;
  }
  cout << "The sum of the first " << n << " integers is " << sum;
}
```

4.20.cpp



4.6 THE CONTINUE STATEMENT

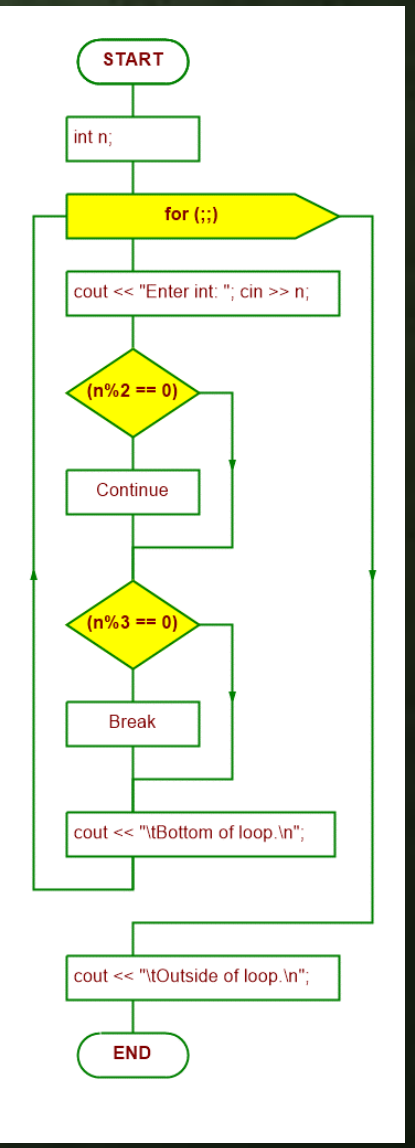
- skips the rest of the statements in the loop's block,
- transfers execution to the next iteration of the loop
- continues the loop after skipping the remaining statements in its current iteration

EXAMPLE 4.23 USING CONTINUE AND BREAK STATEMENTS

This little program illustrates the **continue** and **break** statements:

```
int main()
{ int n;
  for (;;)
  { cout << "Enter int: ";  cin >> n;
    if (n%2 == 0) continue;
    if (n%3 == 0) break;
    cout << "\tBottom of loop.\n";
  }
  cout << "\tOutside of loop.\n";
}
```

4.23.cpp



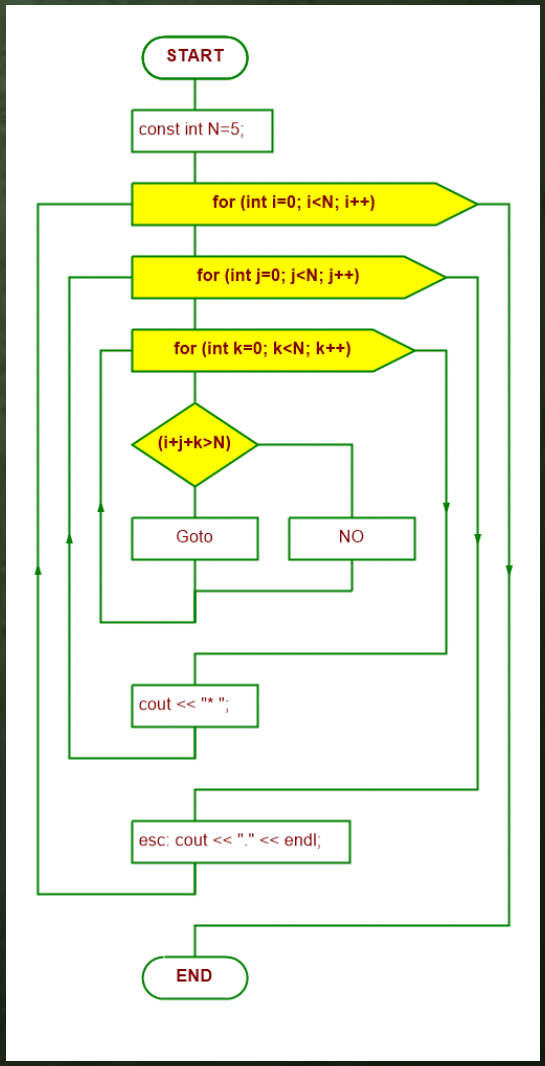
4.7 THE GOTO STATEMENT

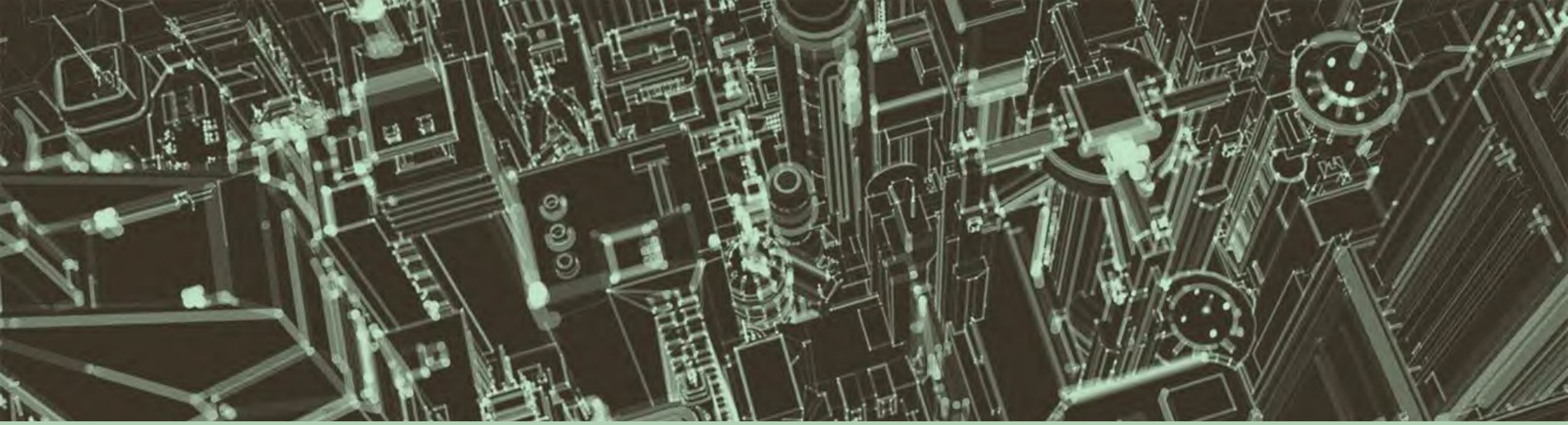
- another kind of jump statement
- Its destination is specified by a label within the statement.
- A label is simply an identifier followed by a colon placed in front of a statement
- Labels work like the case statements inside a switch statement: they specify the destination of the jump

EXAMPLE 4.24 USING A GOTO STATEMENT TO BREAK OUT OF A NEST OF LOOPS

```
int main()
{ const int N=5;
  for (int i=0; i<N; i++)
  { for (int j=0; j<N; j++)
    { for (int k=0; k<N; k++)
        if (i+j+k>N) goto esc;
        else cout << i+j+k << " ";
      cout << "* ";
    }
  esc: cout << "." << endl;  // inside the i loop, outside the j loop
  }
}
```


4.24.cpp





Function (Modularized Subprogram)

Unit 5 (Part 1)

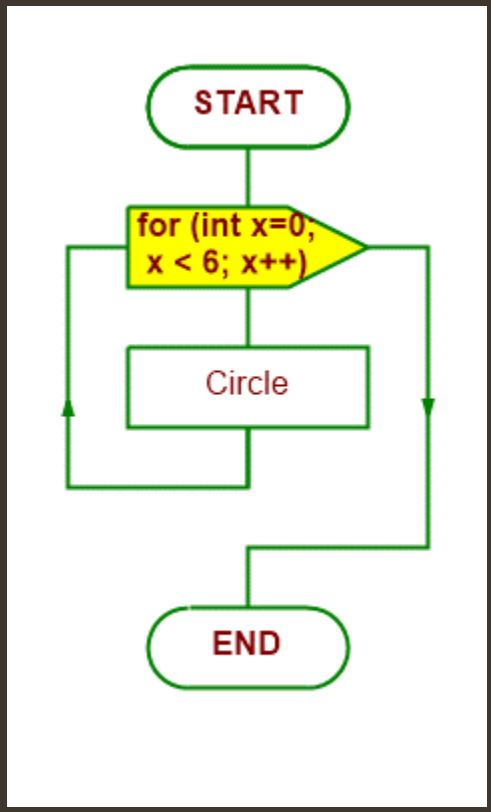
5.2 STANDARD C++ LIBRARY FUNCTIONS

- a collection of pre-defined functions and other program elements which are accessed through header files
- EXAMPLE: Square Root Function sqrt()
- EXAMPLE: Random Function rand()
- EXAMPLE: Time Function time()

EXAMPLE 5.1 The Square Root Function sqrt()

```
#include <cmath>      // defines the sqrt() function
#include <iostream>   // defines the cout object
using namespace std;
int main()
{ // tests the sqrt() function:
  for (int x=0; x < 6; x++)
    cout << "\t" << x << "\t" << sqrt(x) << endl;
}
```

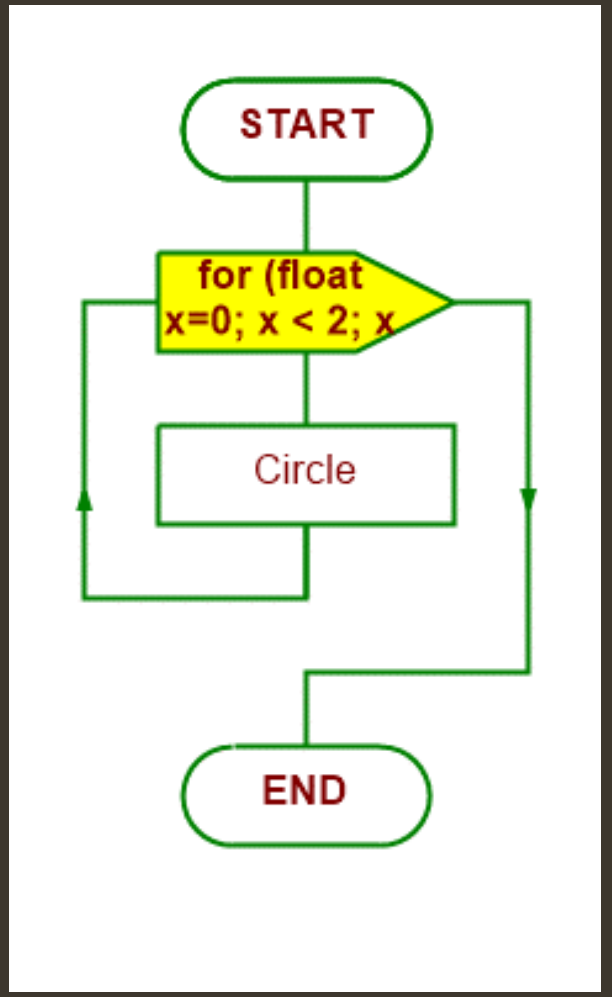
5.1.cpp



EXAMPLE 5.2 Testing a Trigonometry Identity

```
int main()
{ // tests the identity sin 2x = 2 sin x cos x:
  for (float x=0; x < 2; x += 0.2)
    cout << x << "\t\t" << sin(2*x) << "\t"
         << 2*sin(x)*cos(x) << endl;
}
```

5.2.cpp

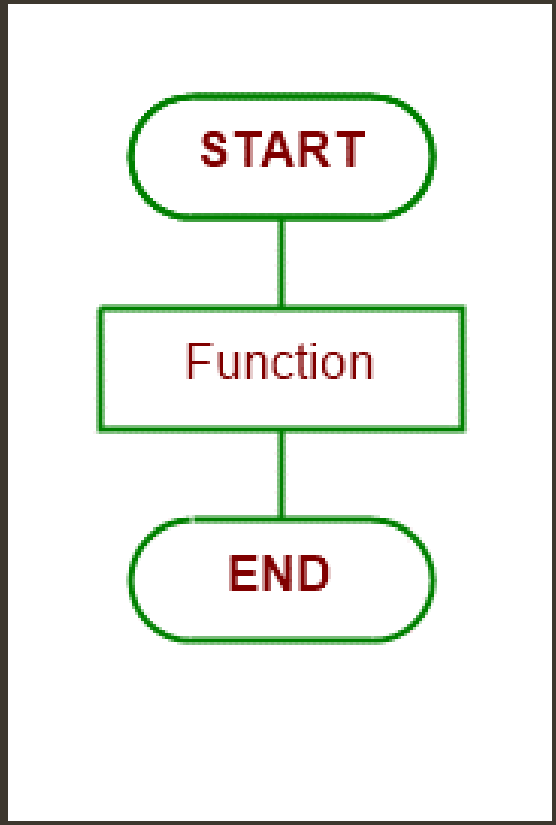


5.3 USER-DEFINED FUNCTION

- Function that are defined or generated by User/Programmer
- has two parts: its head and its body
- Syntax: return-type name(parameter-list)

EXAMPLE 5.3 A cube() Function

```
int cube(int x)
{ // returns cube of x:
  return x*x*x;
}
```



5.4 TEST DRIVER

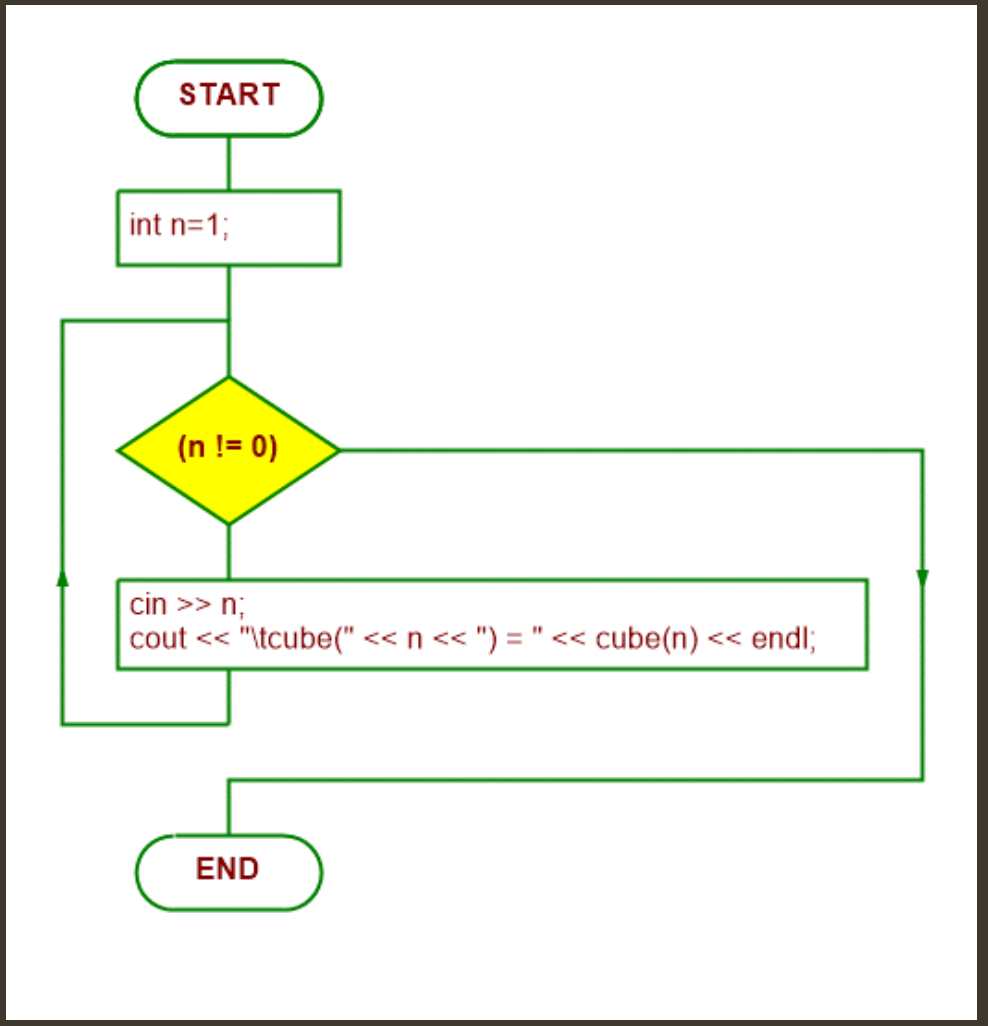
- The program to test the user-defined function
- temporary, ad hoc program, i.e. need not include all the usual niceties such as user prompts, output labels, and documentation
- Can be discarded after using it to test the function

EXAMPLE 5.4 A Test Driver for the cube() Function

```
int cube(int x)
{ // returns cube of x:
  return x*x*x;
}

int main()
{ // tests the cube() function:
  int n=1;
  while (n != 0)
  { cin >> n;
    cout << "\tcube(" << n << ") = " << cube(n) << endl;
  }
}
```

5.4.cpp

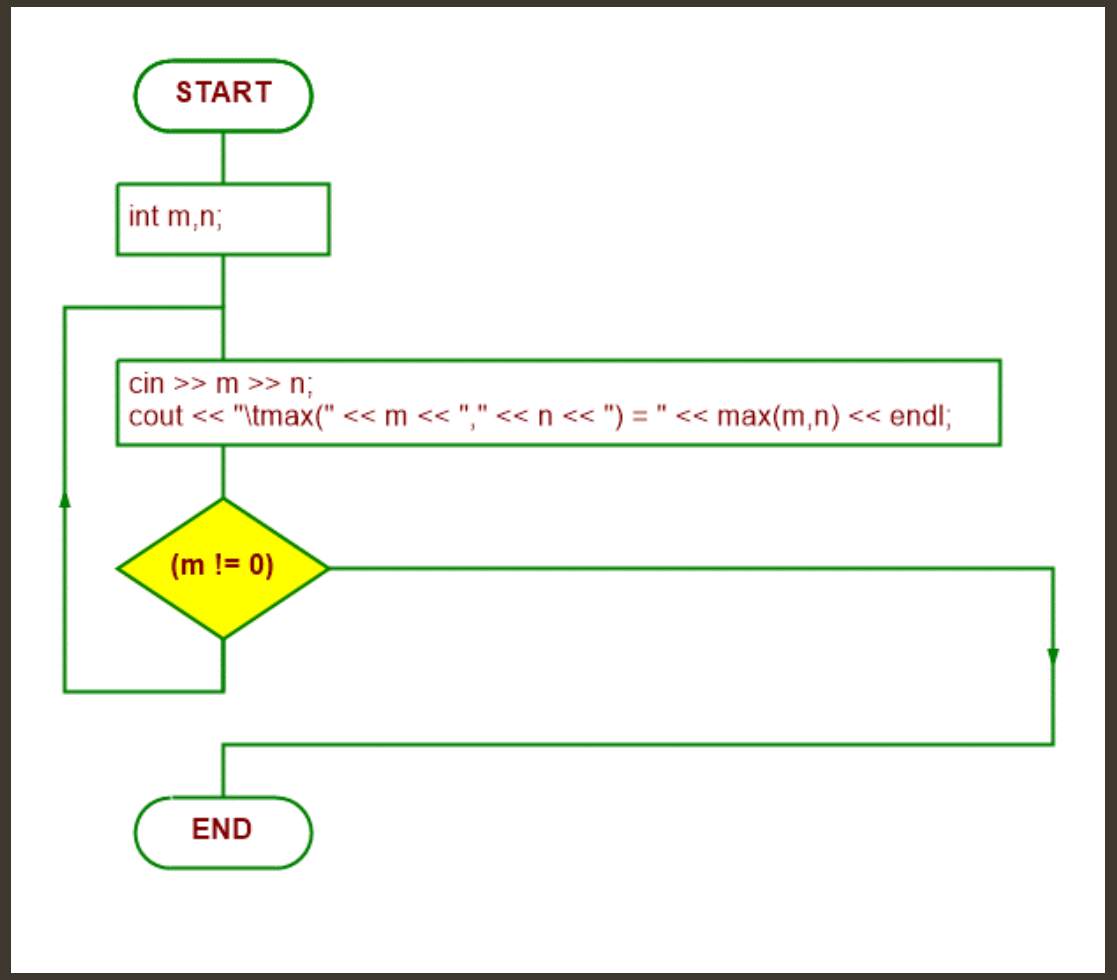


EXAMPLE 5.5 A Test Driver for the max() Function

```
int max(int x, int y)
{ // returns larger of the two given integers:
  if (x < y) return y;
  else return x;
}
```

```
int main()
{ // tests the max() function:
  int m, n;
  do
  { cin >> m >> n;
    cout << "\tmax(" << m << "," << n << ") = " << max(m,n) << endl;
  }
  while (m != 0);
}
```

5.5.cpp



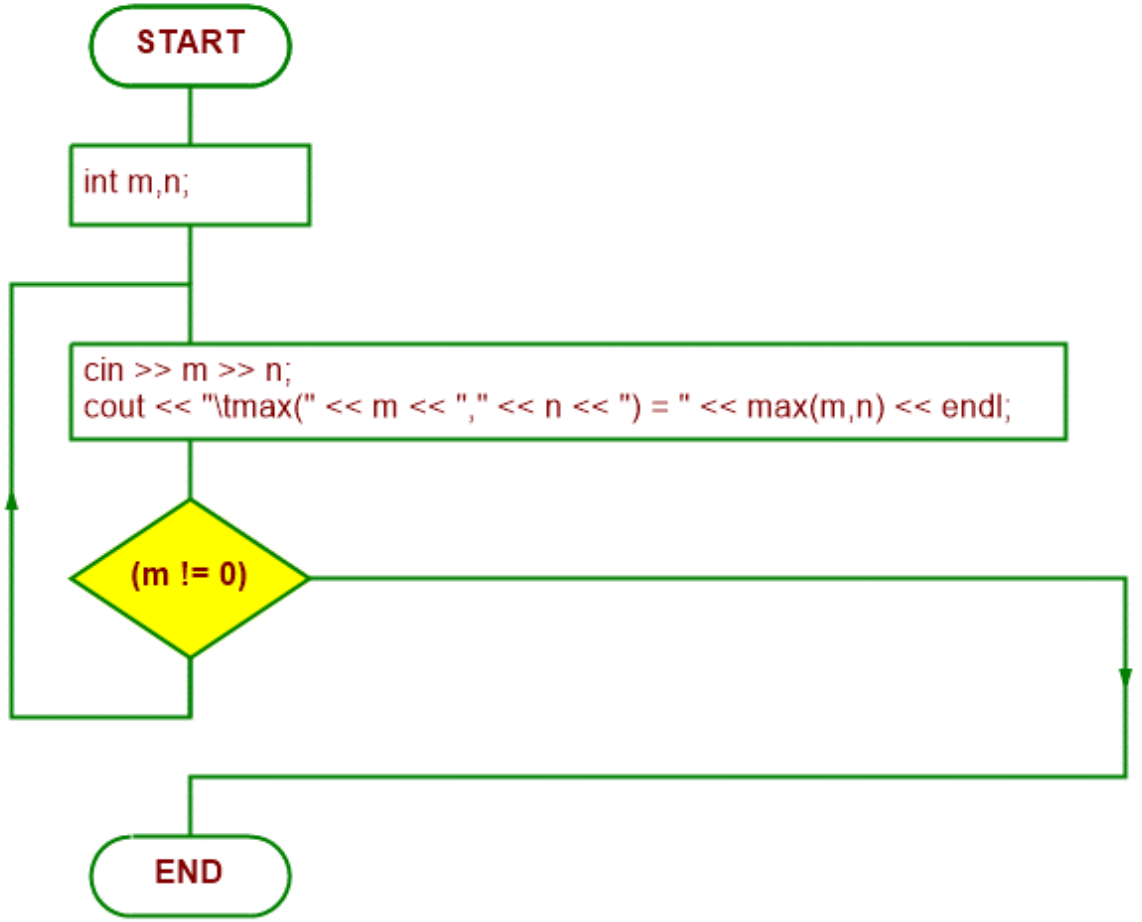
EXAMPLE 5.6 The max() Function with Declaration Separate from Definition

```
int max(int,int);
// returns larger of the two given integers:

int main()
{ // tests the max() function:
  int m, n;
  do
  { cin >> m >> n;
    cout << "\tmax(" << m << "," << n << ") = " << max(m,n) << endl;
  }
  while (m != 0);
}

int max(int x, int y)
{ if (x < y) return y;
  else return x;
}
```

5.6.cpp



ARRAY

UNIT 6

ARRAY

- a sequence of objects all of which have the same type
- Syntax: type array-name[array-size];
- The objects are called the elements of the array and are numbered consecutively 0, 1, 2, 3, ...
- These numbers are called index values or subscripts of the array
- The term “subscript” is used because as a mathematical sequence, an array would be written with subscripts: a0, a1, a2, ….
- The subscripts locate the element’s position within the array, thereby giving direct access into the array
- The method of numbering the i-th element with index i–1 is called zero-based indexing

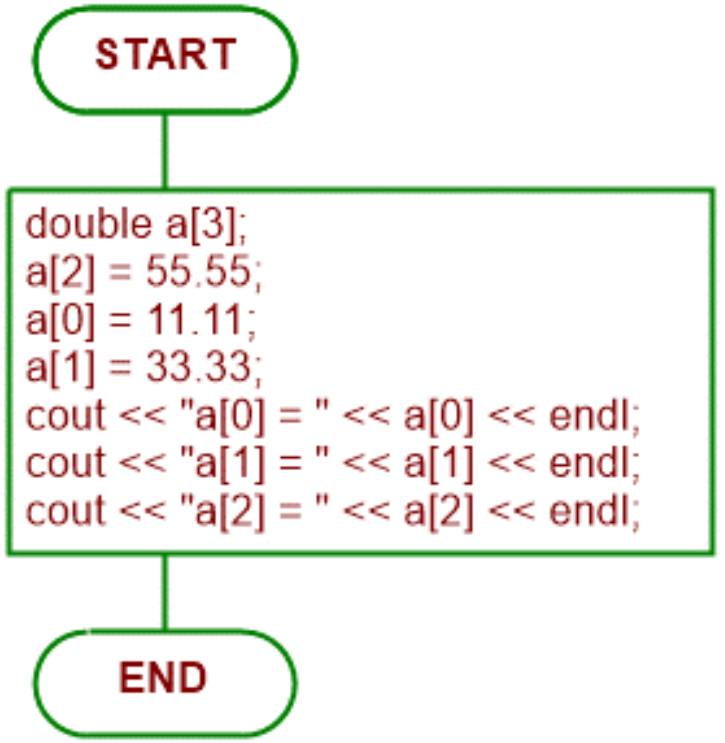
6.2 PROCESSING ARRAYS

- An array is a composite object
- it is composed of several elements with independent values
- In contrast, an ordinary variable of a primitive type is called a scalar object

EXAMPLE 6.1 USING DIRECT ACCESS ON ARRAYS

```
int main()
{ double a[3];
  a[2] = 55.55;
  a[0] = 11.11;
  a[1] = 33.33;
  cout << "a[0] = " << a[0] << endl;
  cout << "a[1] = " << a[1] << endl;
  cout << "a[2] = " << a[2] << endl;
}
```

6.1.cpp

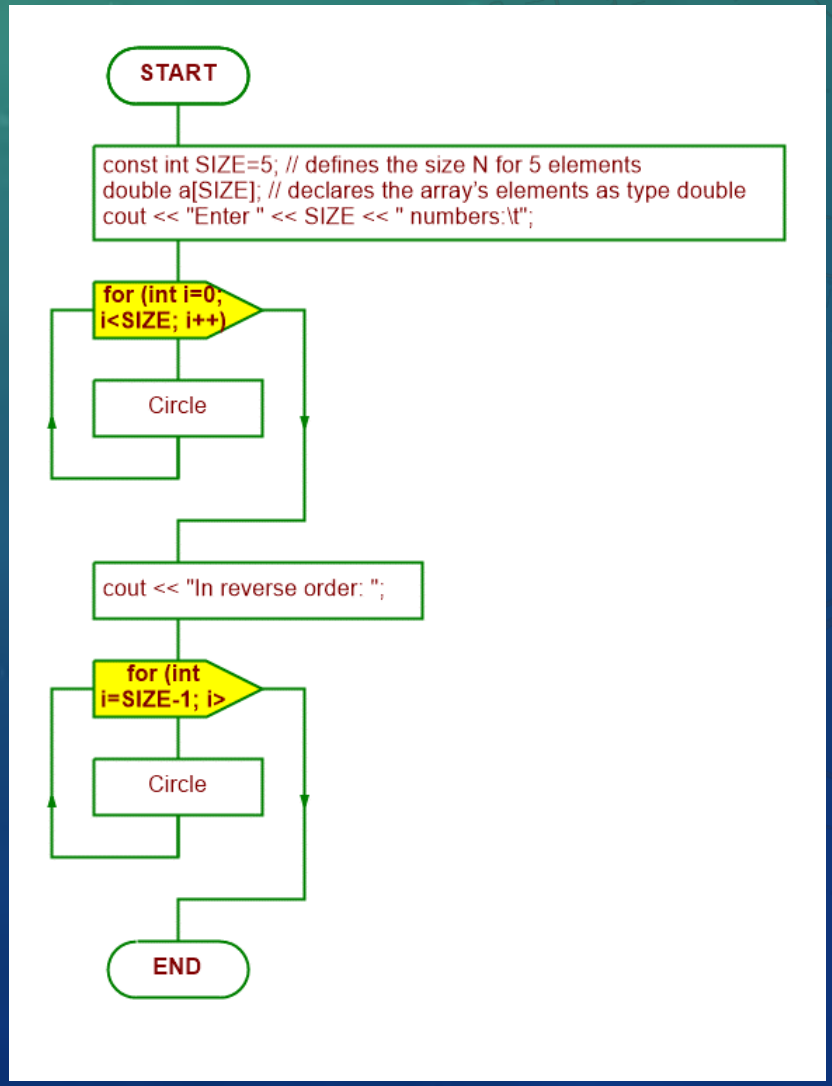


EXAMPLE 6.2 PRINTING A SEQUENCE IN ORDER

This program reads five numbers and then prints them in reverse order:

```
int main()
{ const int SIZE=5;  // defines the size N for 5 elements
  double a[SIZE];    // declares the array's elements as type double
  cout << "Enter " << SIZE << " numbers:\t";
  for (int i=0; i<SIZE; i++)
    cin >> a[i];
  cout << "In reverse order: ";
  for (int i=SIZE-1; i>=0; i--)
    cout << "\t" << a[i];
}
```

6.2.cpp

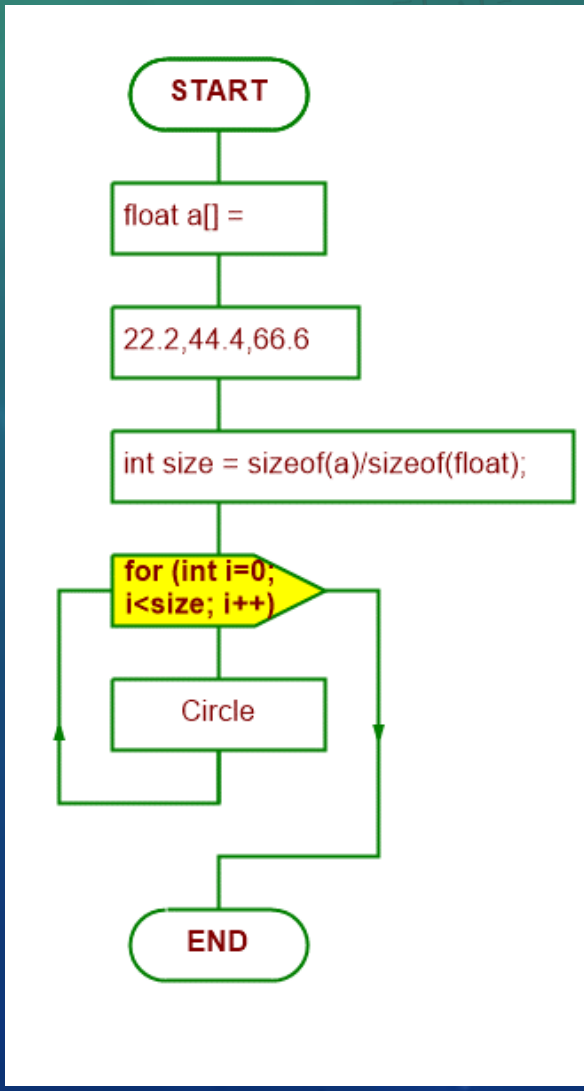


EXAMPLE 6.3 INITIALIZING AN ARRAY (VIA INITIALIZER LIST)

This program initializes the array a and then prints its values:

```
int main()
{ float a[] = { 22.2, 44.4, 66.6 };
  int size = sizeof(a)/sizeof(float);
  for (int i=0; i<size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

6.3.cpp

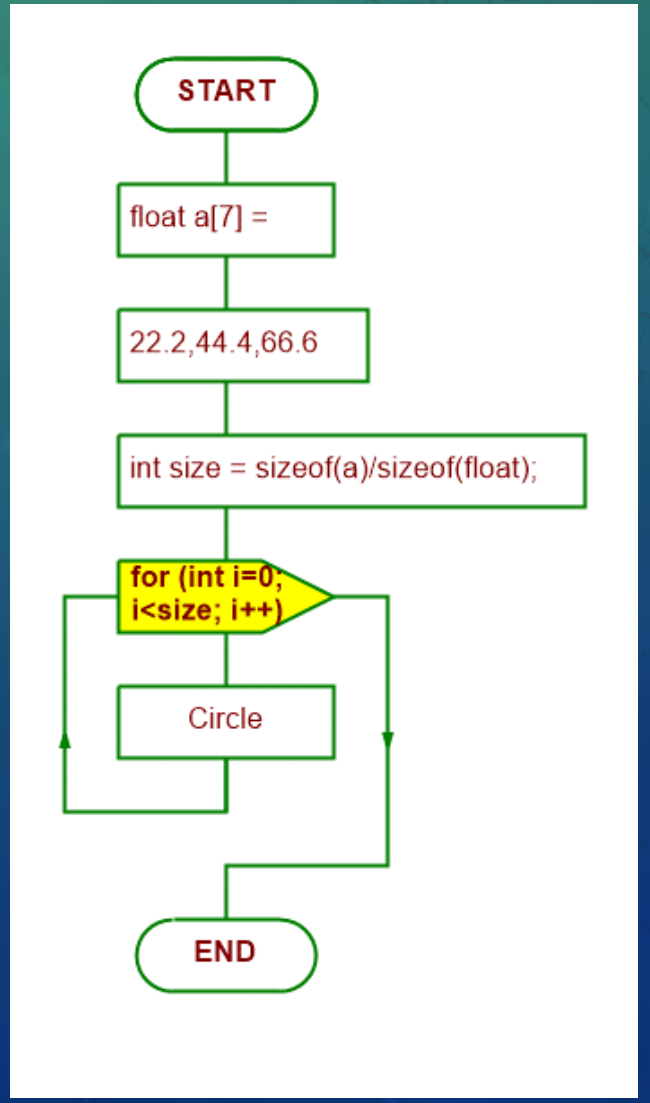


EXAMPLE 6.4 INITIALIZING AN ARRAY WITH TRAILING ZEROS

This program initializes the array a and then prints its values:

```
int main()
{ float a[7] = { 22.2, 44.4, 66.6 };
  int size = sizeof(a)/sizeof(float);
  for (int i=0; i<size; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

6.4.cpp



EXAMPLE 6.5 AN UNINITIALIZED ARRAY

This program initializes the array a and then prints its values:

```
int main()
{ const int SIZE=4;   // defines the size N for 4 elements
  float a[SIZE];      // declares the array's elements as type float
  for (int i=0; i<SIZE; i++)
    cout << "\ta[" << i << "] = " << a[i] << endl;
}
```

6.5.cpp