

CS2310 Computer Programming

LT2: Operators, Basic I/O, Conditional Statements

Computer Science, City University of Hong Kong (Dongguan)

Semester A 2025-26

Outline

- Operators
- Conditional statements
- Basic I/O

Operators

- An operator specifies an operation to be performed on some values
 - These values are called the **operands** of the operator
- Common **Operators**:
 - Arithmetic Operators, e.g., +, -, * /, etc.
 - Assignment Operator
 - Comparison Operators
 - Logical Operators
 - ...
- Some of these have **meanings** that depend on the **context**

Increment & decrement operators

- Increment and decrement operators: **++** and **--**
 - `k++` and `++k` are equivalent to `k=k+1` `k += 1`
 - `k--` and `--k` are equivalent to `k=k-1` `k -= 1`
- **Post**-increment and **post**-decrement: `k++` and `k--`
 - `k`'s value is altered **AFTER** the expression is evaluated

```
int k=0, j;
j=k++;
(equal to: 1. j=k; 2. k=k+1;)
```
- **Pre**-increment and **pre**-decrement: `++k` and `--k`
 - `k`'s value is altered **BEFORE** evaluating the expression

```
int k=0, j;
j=++k;
(equal to: 1. k=k+1; 2. j=k;)
```



Post-increment and **post-decrement**: $k++$ and $k--$: $++k$ and $--k$

- k 's value is altered **AFTER** the expression is evaluated during the evaluation

```
int k=1, j;
```

```
j=k++; /* result: j is 1, k is 2 */
```

```
k=0;
```

```
i=1+ (k++) ;
```

0

Use original value of k

```
i=1+0  
=1
```

```
k=0;
```

```
i=1+ (++k) ;
```

1

Use updated value of k

```
i=1+1  
=2
```

Value of k will be 1 in both cases

What values are printed?

```
int k=0,i=0;  
cout << "i= " << i << endl;
```

```
k=0;  
i=1+(k++) ;  
cout << "i= " << i << endl;  
cout << "k= " << k << endl;
```

```
k=0;  
i=1+(++k) ;  
cout << "i= " << i << endl;  
cout << "k= " << k << endl;
```

Output

```
i=0  
i=1  
k=1  
i=2  
k=1
```

Division & modulus operators

- **Division** operator: $/$
 - Return the **quotient**, e.g. $5 / 2 = 2$
- **Modulus** operator: $\%$
 - Return the division **remainder**, e.g. $5 \% 2 = 1$
- Example: write a program that reads a **three**-digit integer number and prints the **sum** of each digit
 - E.g., Input: $N = 456$; Output: $15 (=4+5+6)$

	Hundreds	Tens	Ones
Bit weight:	10^2	10^1	10^0
	4	5	6

Hint 1: $456/100 = 4$

Hint 2: $456\%100 = 56$

Division & modulus operators

- Example: write a program that read a three-digit number and print the sum of the digits.

Code: lec02-03-sumdigit.cpp

Enter a number of three digits: 456
Sum of digits is: 15

Hint-1: For example, input a number $N = 346$, the output should be $3+4+6 = 13$.

Hint-2: Use % and / operators.

```
4 void main() {  
5  
6     int Num, a, b, c, Sum;  
7     cout << "Enter a number of three digits: ";  
8     cin >> Num;  
9     a = Num / 100;  
10    b = Num % 100;  
11    c = b / 10;  
12    b = b % 10;  
13    Sum = a + b + c;  
14    cout << "Sum of digits is: " << Sum << '\n';  
15  
16 }
```

a: $456 / 100 = 4$
b: $456 \% 100 = 56$
c: $56 / 10 = 5$
d: $56 \% 10 = 6$

Precedence & associativity of operators

- An expression may have more than one operator and its precise meaning depends on the **precedence** and **associativity** of the involved operators
- What is the value of variables a, b and c after the execution of the following statements

`int a, b = 2, c = 1;`

`a = b+++c;`

- Which of the following interpretation is right?

`a = (b++) + c;`

or `a = b + (++c);`

Precedence & associativity of operators

Precedence: **order** of evaluation for **different** operators.

- **Precedence** determines how an expression like $x \mathbf{R} y \mathbf{S} z$ should be evaluated (now \mathbf{R} and \mathbf{S} are **different** operators, e.g., $x + y / z$).

Associativity: **order** of evaluation for operators with the **same** precedence.

- **Associativity** means whether an expression like $x \mathbf{R} y \mathbf{R} z$ (where \mathbf{R} is a operator, e.g., $x + y + z$) should be evaluated '**left-to-right**' i.e. as $(x \mathbf{R} y) \mathbf{R} z$ or

'**right-to-left**' i.e. as $x \mathbf{R} (y \mathbf{R} z)$;

Precedence & associativity of operators

Operator Precedence (high to low)				Associativity
::				None
.	->	[]		Left to right
()	++(postfix)	--(postfix)		Left to right
+	-	++ (prefix)	-- (prefix)	Right to left
*	/	%		Left to right
+	-			Left to right
=	+=	-=	*= /= etc.	Right to left

Example: `a=b+++c`

`a=(b++)+c; or`

`a=b+(++c);`

Example: `int a, b=1;`

`a=b=3+1;`

Swapping the values

- We want to swap the content of two variables, a and b.
- What's wrong with the following program?

```
void main() {  
    int a=3, b=4;  
    a=b;  
    b=a;  
}
```

$$a=3$$

$$b=4$$

$$c=3$$



Swapping the values

- We want to swap the content of two variables, a and b.
- What's wrong with the following program?

```
void main() {  
    int a=3, b=4;  
    a=b;  
    b=a;  
}
```

- We need to make use of a temporary variable

c = a; /*save the old value of a*/

a = b; /*put the value of b into a*/

b = c; /*put old value of a (which is contained in c) to b*/

Expressions

- An ***expression*** is a combination of constants, variables, and function calls that evaluate to a **result**

- Example:

`x = 3.0*4.0;`

constants

`y = 2.0 + x;`

variables

`z = 5.0 + x/y - sqrt(x*3.0) ;`

function call

Logical Expressions

- *Boolean operation*: an **operation** that is applied to one or more **true (=1)** / **false (=0)** values
 - Specific operators: AND, OR, NOT, XOR (exclusive or), etc.
 - E.g., (Arithmetic) $C = A + B$; (Boolean) $C = A \text{ AND } B$

Comparative Operators

- Binary operators which accept **two** operands and compare them

Relational operators	Syntax	Example
Less than	<	x<y
Greater than	>	z>1
Less than or equal to	<=	b<=1
Greater than or equal to	>=	c>=2

Equality operators	Syntax	Example
Equal to	==	a==b
Not equal to	!=	b!=3

Logical Operators

- Used for combining **logical values** and create **new logical values**
- Logical AND (&&)
 - return **true** if **both** operands are **true**, false otherwise (e.g., **a>1&&b<1**)
- Logical OR (||)
 - return **false** if **both** operands are **false**, true otherwise
- Logical NOT (!)
 - invert the Boolean value of the operand

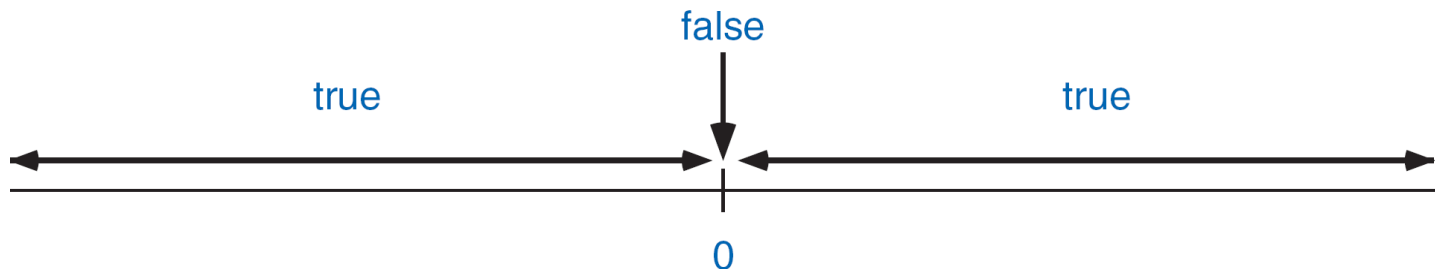
x	y	x&&y
true	true	true
true	false	false
false	true	false
false	false	false

x	y	x y
true	true	true
true	false	true
false	true	true
false	false	false

x	!x
true	false
false	true

Logical expression and operators

- Logical expressions can be **true** or **false** only
 - $x==3$
 - $y==x$
 - $x>10$
- In C++, any **non-zero** expression will be treated as logical **true**
 - $3-2$
 - $1-1$
 - $x=0$
 - $x=1$



Relational, equality & logical operators (Summary)

□ Relational operators

- less than <
- greater than >
- less than **or equal to** <=
- greater than **or equal to** >=

□ Equality operators

- equal to ==
- not equal to !=

□ Logical operators

- logical not !
- logical and &&
- logical or ||

PS: Expressions with above operators have a *true* or *false* value.

Precedence & associativity of popular operators

Operator precedence (high to low)	Associativity
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
	Left to right
?:	Right to left

Condition ? Expression2 : Expression3

Outline

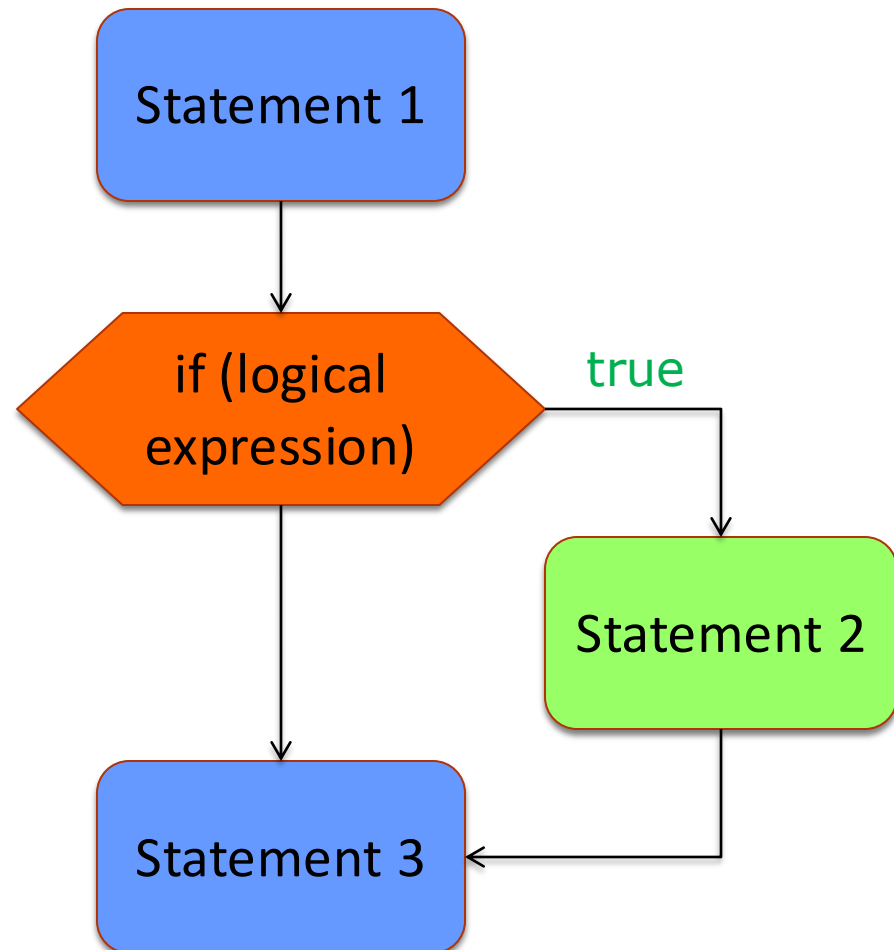
- Operators
- Conditional statements
- Basic I/O

if-statement: One-Way Conditional (1)

Execute **a statement** (or **a block of statements**) if **a specified condition** is **true**

```
statement1;  
if (condition)  
    statement2;  
statement3;
```

```
statement1;  
if (condition){  
    statement2;  
    statement22;  
    ...  
}  
statement3;
```



if-else: Two-Way Conditional (1)

Execute a statement (or a block of statements) if a specified condition is **true**. Otherwise, **another statement** (or **a block of statements**) will be executed

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

```
if (condition){
    statement1;
    statement2;
    ...
} else {
    statement3;
    statement4;
    ...
}
```

Multiple **else if** statements

- You can have many **nested** “else if” statements.

```
if (logical expression 1) {  
    //action for true  
    statement a;  
} else if (logical expression 2) {  
    //action for true  
    statement b;  
} else {  
    //action for false  
    statement;  
}
```

An example: input a **score**,
display **grade** according to:

A: 100 ~ 90, B: 89 ~ 75

C: 74 ~ 55, D: 54 ~ 0

```
if (score >= 90)  
    grade = 'A';  
else if (score >= 75)  
    grade = 'B';  
else if (score >= 55)  
    grade = 'C';  
else  
    grade = 'D';
```

Beware of empty statements!

```
int x=5;  
if (x!=5) ;  
    x=3 ;  
cout << x;  
/*output is 3*/
```

```
int x=5;  
if (x!=5)  
    x=3 ;  
cout << x;  
/*output is 5*/
```

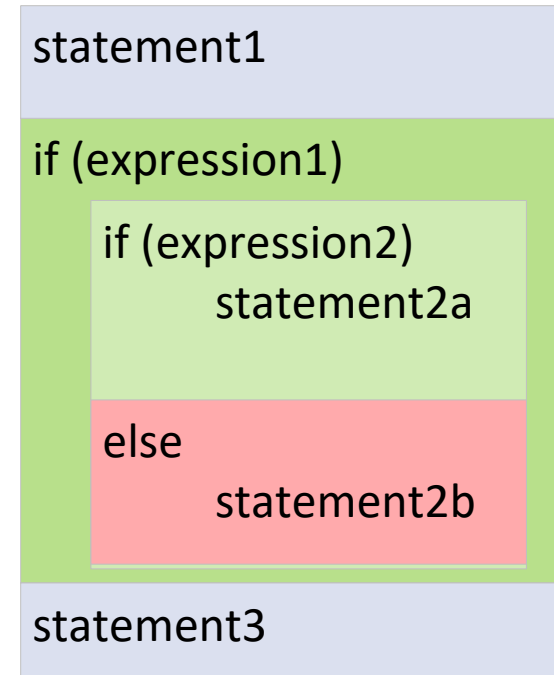
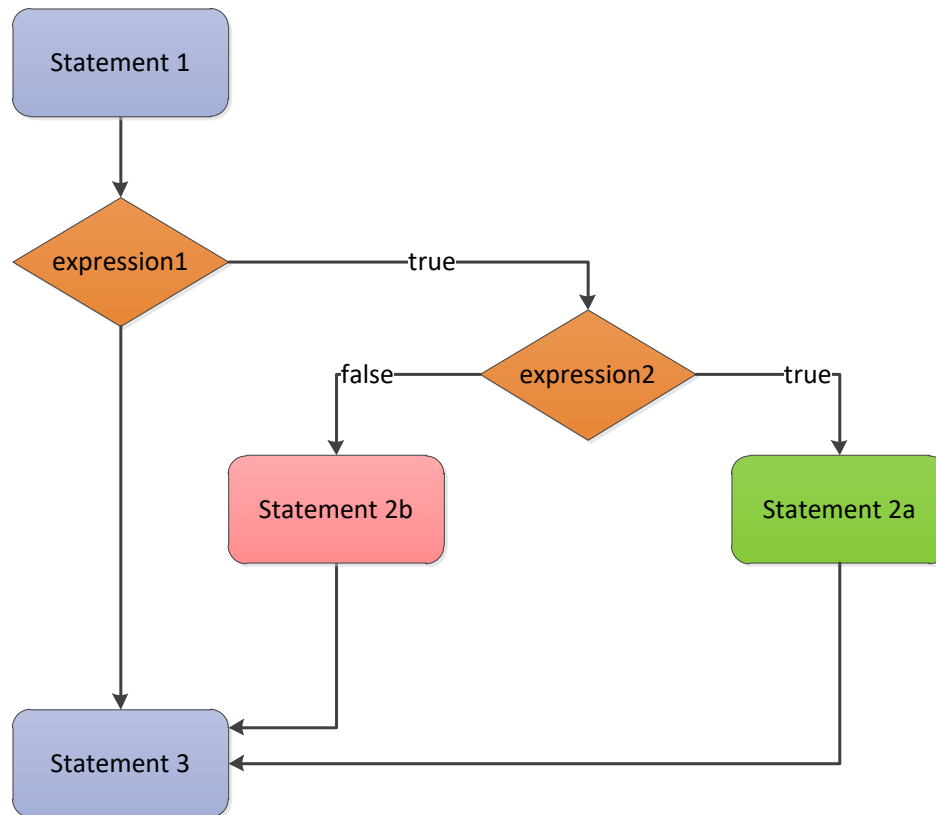
An empty statement can be specified by a semi-colon ';'. Empty statement specifies that **no action should be performed**.

For program on the right, **x=3 statement will NOT be executed** if x is equals to 5.

For the program on the left, **x=3 statement will be always executed**.

Nested if statement

An **if-else** statement is included with another **if** or **else** statement



Nested if statement

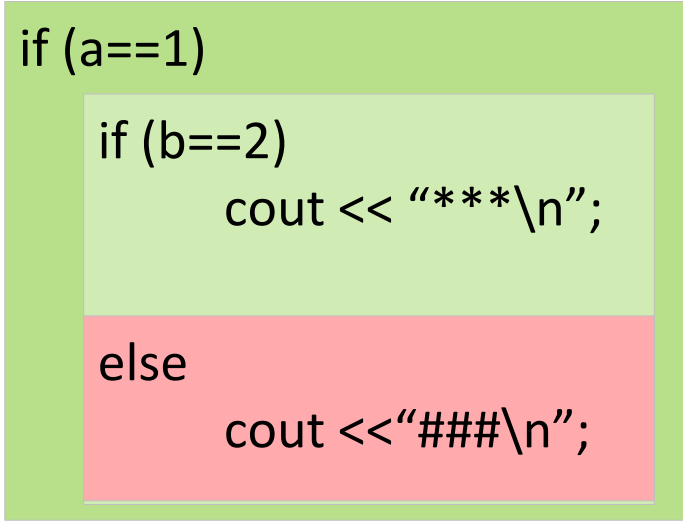
- With which “if” the “else” part is associated?

```
if (a==1)
    if (b==2)
        cout << "***\n";
else
    cout << "###\n";
```

```
if (a==1)
    if (b==2)
        cout << "***\n";
else
    cout << "###\n";
```

- An else attached to the **nearest** if.

```
if (a==1)
    if (b==2)
        cout << "***\n";
else
    cout << "###\n";
```



The diagram illustrates the execution of the code. It consists of two nested if-else blocks. The outer block is 'if (a==1)' and is highlighted with a light green background. Inside it is an inner block 'if (b==2)' with a light green background, and an 'else' block with a pink background. The 'else' block contains the statement 'cout << "###\n";'. This visualizes that the 'else' is attached to the nearest 'if' (the inner one), not the outer one.

```
if (a==1)
    if (b==2)
        cout << "***\n";
    else
        cout << "###\n";
```

Do not mix == and =

```
x=0;
y=1;
if (x = y) {
    cout << "x and y are equal";
}
else
    cout << "unequal";
```

Output: "x and y are equal"

The expression **x = y**

- Assign the value of y to x: **x becomes 1**
- The value of this expression is the value of **x** (which represents 1/TRUE)
 - False is represented by 0
 - Non-zero represents TRUE

C++ syntax is different from the math syntax

```
if (mark>=70 && mark<=100)
```

.....

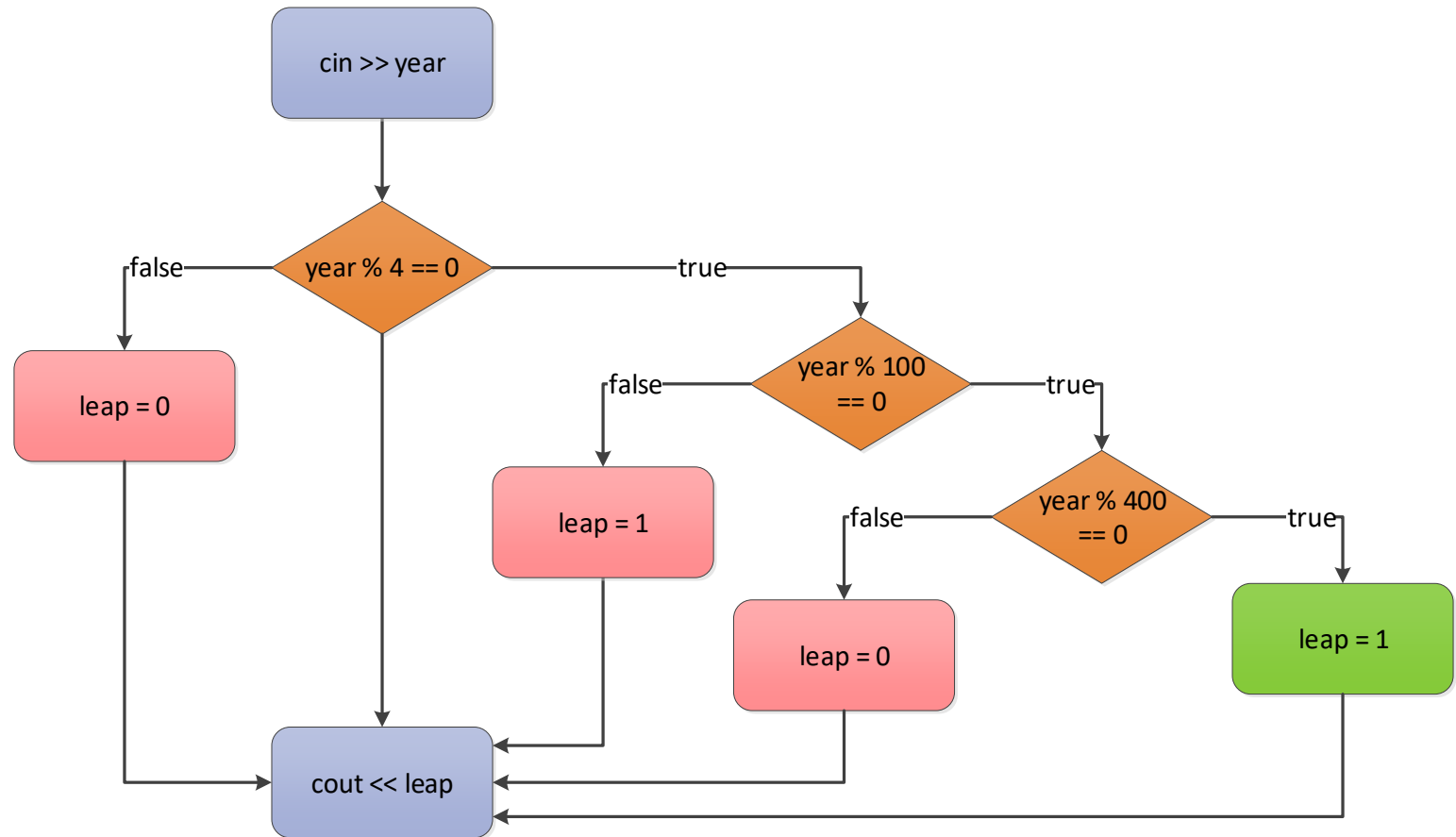
Can we express the above condition as follows?

```
if (70<=mark<=100)
```

.....

Ans: **No**

Example: check if a year is a leap year



Example: check if a year is a leap year

```
4 void main() {  
5  
6     int year;  
7     cout << "Please input year: ";  
8     cin >> year;  
9     if (year % 4 == 0) {  
10         if (year % 100 == 0) {  
11             if (year % 400 == 0)  
12                 cout << "It is a leap year" << endl;  
13             else  
14                 cout << "It is not a leap year" << endl;  
15         }  
16         else  
17             cout << "It is a leap year" << endl;  
18     }  
19     else  
20         cout << "It is not a leap year" << endl;  
21  
22 }
```

Short-circuit evaluation

- Evaluation of expressions containing `&&` and `||` stops as soon as the outcome *true* or *false* is known and this is called *short-circuit evaluation*

x	y	x&& y
true	true	true
true	false	false
false	true	false
false	false	false

x	y	x y
true	true	true
true	false	true
false	true	true
false	false	false

Short-circuit evaluation

- Evaluation of expressions containing `&&` and `||` stops as soon as the outcome *true* or *false* is known and this is called *short-circuit evaluation*,
- E.g., if(**1<0** `&&` **y==2**) is the same as if(**false** `&&` **true**), and thus **y==2** is not executed!

x	y	x&& y
true	true	true
true	false	false
false		false
false		false

x	y	x y
true		true
true		true
false	true	true
false	false	false

Short-circuit evaluation

- Evaluation of expressions containing `&&` and `||` stops as soon as the outcome *true* or *false* is known and this is called *short-circuit evaluation*,
- E.g., if(`1<0 && y==2`) is the same as if (`false && true/false`), and thus
- Short-circuit evaluation can improve program **efficiency**
- Short-circuit evaluation exists in some other programming languages, e.g., C and Java

Short-circuit evaluation

- Given **integer variables** `i`, `j` and `k`, what are the outputs when running the program below?

```

    x  &&  y
k = (i=2) && (j=2);
cout << i << j << endl;
/* 2  2 */
k = (i=0) && (j=3);
cout << i << j << endl; /* 0  2 */
k = i || (j=4);
cout << i << j << endl; /* 0  4 */
k = (i=2) || (j=5);
cout << i << j << endl; /* 2  4 */
```

x	y	x&& y
true	true	true
true	false	false
false		false
false		false

x	y	x y
true		true
true		true
false	true	true
false	false	false

switch statement

□ Syntax (*selection* statement)

```
switch (expression) {  
    case constant-expr1: statement1  
    case constant-expr2: statement2  
    ...  
    ...  
    case constant-exprN: statementN  
    default: statement  
}
```

switch statement: Syntax

```
switch(expression) //e.g., switch(x)
```

```
{
```

```
    case constant-expression://case 1:
```

```
        statement(s);
```

```
        break; //optional
```

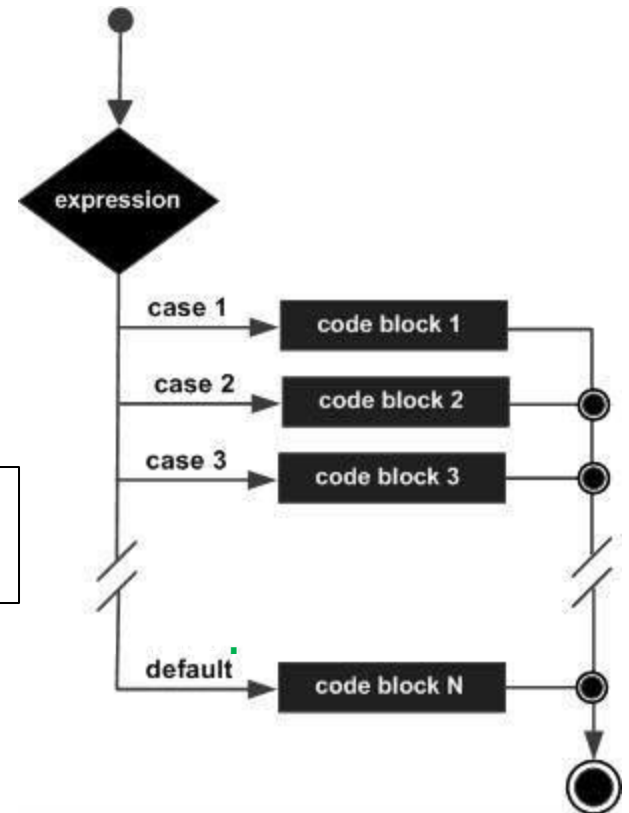
*Terminate the switch when a
break statement is encountered*

```
    default : //Optional
```

```
        statement(s);
```

```
}
```

*Go to the case label having a constant value that matches the value of **the switch expression**;
if a match is not found, go to the default label; if default label does not exist, terminate the switch*



Example

```
#include <iostream>
using namespace std;
void main(){
    int x;
    cin >> x;
    switch (x){
        case 0:
            cout << "Zero";
            break; /* no braces is needed */
        case 1:
            cout << "One";
            break;
        case 2:
            cout << "Two";
            break;
        default:
            cout << "Greater than two";
    } //end switch
}
```

Example

```
#include <iostream>
using namespace std;
void main(){
    int x=0;
    switch (x){
        case 0:
            cout << "Zero";
            x=1;
            break; /* no braces is needed */
        case 1:
            cout << "One";
            break;
        case 2:
            cout << "Two";
            break;
        default:
            cout << "Greater than two";
    } //end switch
}
```

switch statement

□ Semantics

- Evaluate the *switch expression* which results in an **integer** type (`int`, `long`, `short`, `char`)
- Go to the **case** label having a **constant** value that matches the value of the switch expression; if a match is not found, go to the *default* label; if *default* label does not exist, terminate the switch
- Terminate the switch when a **break** statement is encountered
- If there is no break statement, execution “*falls through*” to the next statement in the successful case

conditional (?:) operator

□ Syntax of ?: operator is

- `expr1 ? expr2 : expr3`
- E.g., `(a>b) ? a=1 : a=0`

□ Semantics

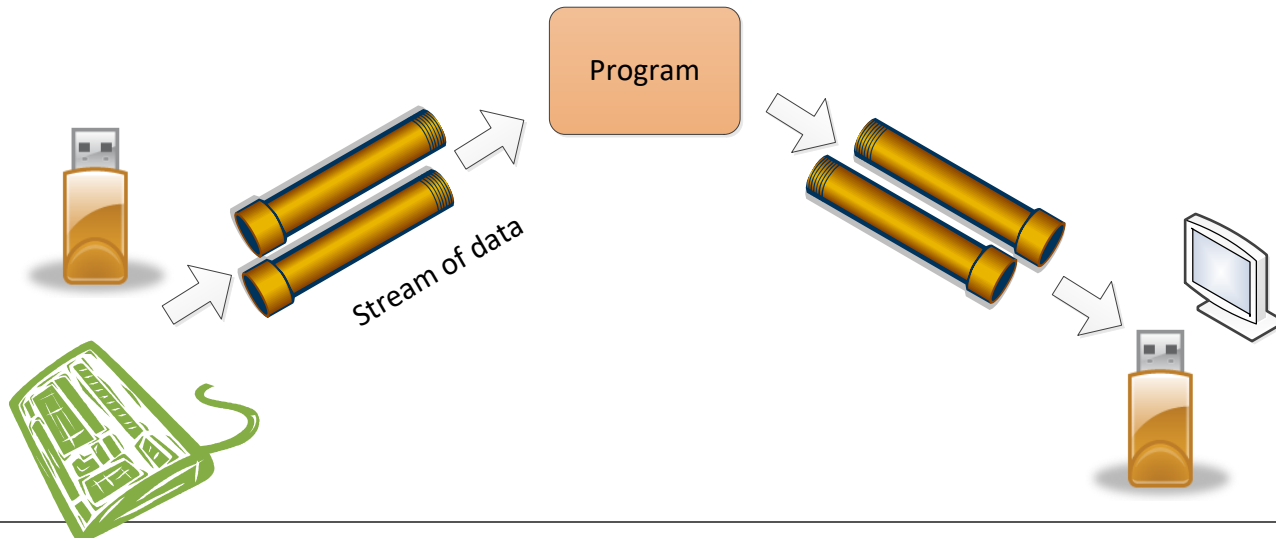
- `expr1` is evaluated
- If the result is non-zero/true, then execute `expr2`;
- else `expr3` is executed
- The value of the whole ?: expression is the value of expression evaluated at the end
- E.g., **finds the maximum value of x and y**

Outline

- Operators
- Conditional statements
- Basic I/O

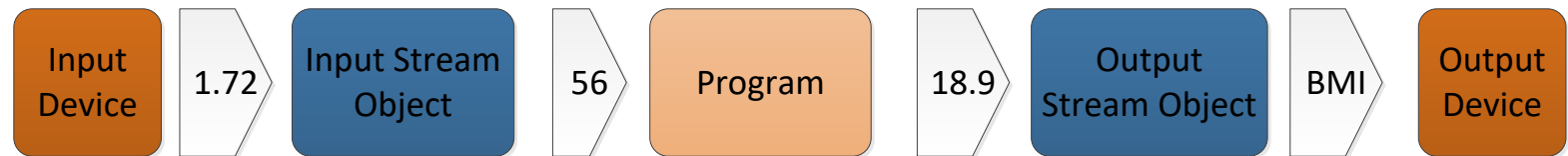
Basic I/O – Keyboard and Screen

- A program can do little if it cannot take **input** and produce **output**
- Most programs read user input from **keyboard** and **secondary storage**
- After process the input data, result is commonly display on **screen** or **write to storage** (disk)



Basic I/O – cin and cout

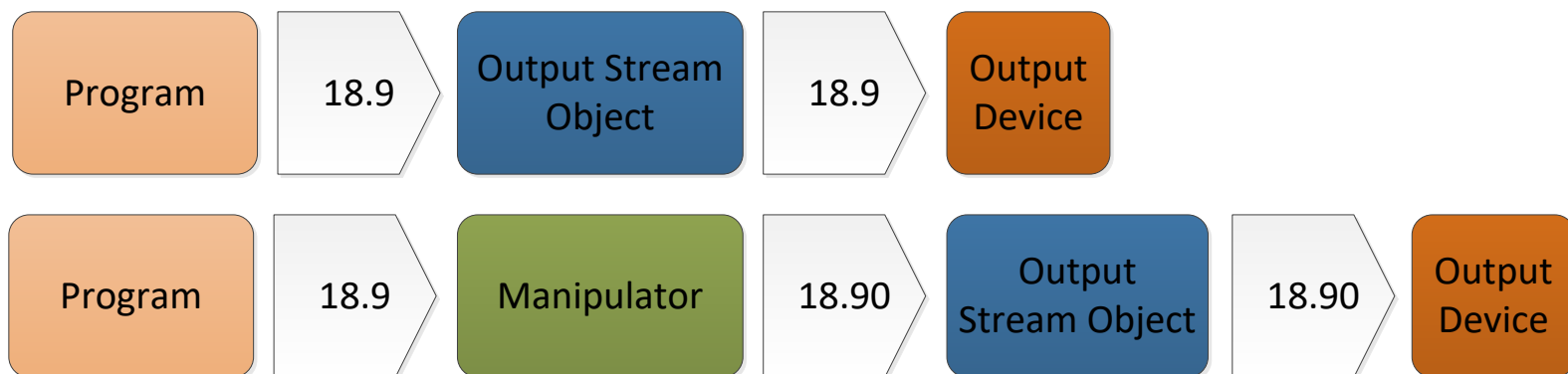
- C++ comes with an **iostream** package (library) for basic I/O.
- **cin** and **cout** are objects defined in **iostream** for *keyboard input* and *screen display*, respectively
- To read data from **cin** and write data to **cout**, we need to use **input operator (>>)** and **output operator (<<)**



```
cin >> weight
cin >> height
cout << "bmi"
cout << weight/height/height
```

cout: Output Operator (<<)

- Preprogrammed for **all** standard C++ **data types**
- It sends **bytes** to an output stream object, e.g. cout
- Predefined “**manipulators**” can be used to change the default **format** of arguments



cout: Output Operator <<

Type	Expression	Output
Integer	<code>cout << 21</code>	21
Float	<code>cout << 14.5</code>	14.5
Character	<code>cout << 'a';</code> <code>cout << 'H' << 'i'</code>	a Hi
Bool	<code>cout << true</code> <code>cout << false</code>	1 0
String	<code>cout << "hello"</code>	hello
New line (endl)	<code>cout << 'a' << endl << 'b';</code>	a b
Tab	<code>cout << 'a' << '\t' << 'b';</code>	a b
Special characters	<code>cout << \"\" << "Hello" << \"\" << endl;</code>	"Hello"
Expression	<code>int x=1;</code> <code>cout << 3+4 +x;</code>	8

cout – Change the **Width** of Output

- Change the width of output
 - Calling member function **width**(*width*) or using **setw** manipulator (which requires **iomanip** library: **#include <iomanip>**)
 - **Leading blanks** are added to any value **fewer than** the width
 - Effect last for **one field** only
 - If formatted output exceeds the width, the entire value prints

Approach	Example	Output (♦: space key)
1. cout.width (<i>width</i>) 2. setw (<i>width</i>)	cout.width (5); //or cout<<setw (5); cout << 123 << endl; cout << 123 << endl; cout.width (5); //or cout<<setw (5); cout <<1234567 << endl;	♦♦123 123 1234567

cout – Set the **format** and **precision** of floating-point Output

- Floating-point precision is **six** by default, i.e. **5 decimal points**
- Use **fixed** and **setprecision** manipulators to change the **printing format** and **precision value**.
- Must **#include <iomanip>**
- Effect is **permanent**

Default behavior

Example	Output
<code>cout << 1.34 << endl;</code>	1.34
<code>cout << 1.340 << endl;</code>	1.34
<code>cout << 1.3401234 << endl;</code>	1.34012
<code>cout << 0.0000000134 << endl;</code>	1.34e-008

fixed manipulator

- `cout<<fixed`: always uses the fixed point notation (**6** significant digits **after the decimal point**)
- It changes the meaning of precision (see the example)

Example	Output
<code>cout << fixed;</code>	
<code>cout << 1.34 << endl;</code>	1.340000
<code>cout << 1.340 << endl;</code>	1.340000
<code>cout << 0.0000000134 << endl;</code>	0.000000

setprecision manipulator

- Normally, **setprecision(n)** means output **n** significant digits
- But with “**fixed**”, **setprecision(n)** means output **n** significant digits **after the decimal point**

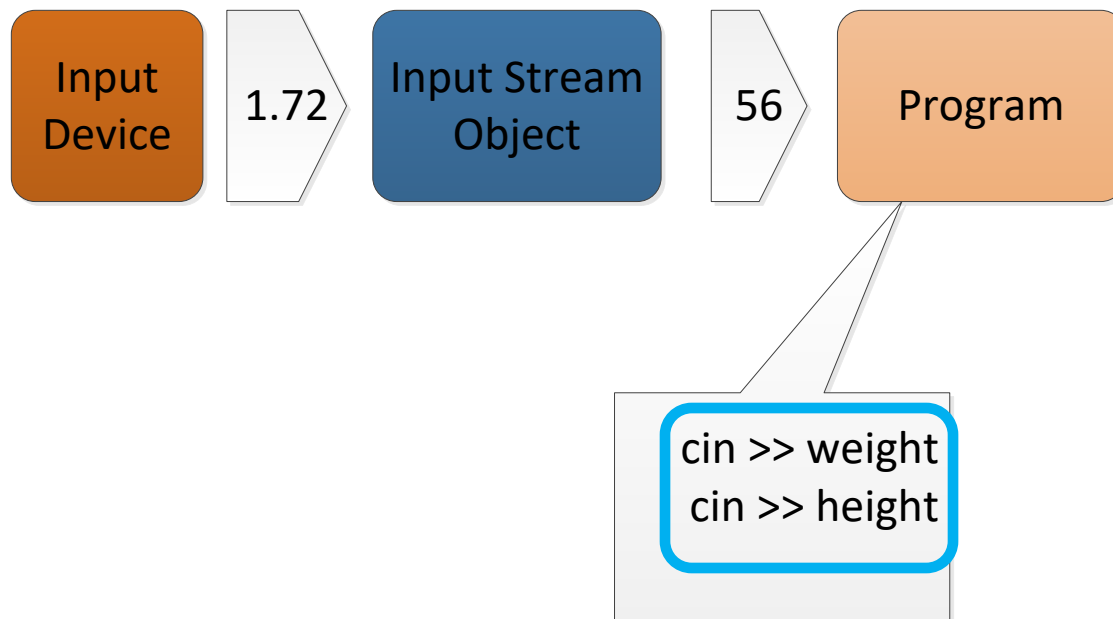
Example	Output
<pre>cout << setprecision(2); cout << 1.34 << endl;</pre>	1.3
<pre>cout << fixed; cout << 1.34 << endl; cout << 0.0000000134 << endl;</pre>	1.34 0.00

cout – Other Manipulator

Manipulators	Example	Output
fill	<pre>cout << setfill('*'); cout << setw(10); cout << 5.6 << endl; cout << setw(10); cout << 57.68 << endl;</pre>	<pre>*****5.6 *****57.68</pre>
radix	<pre>cout << oct << 11 << endl; // octal cout << hex << 11 << endl; // hexadecimal cout << dec << 11 << endl;</pre>	<pre>13 b 11</pre>
alignment	<pre>cout << setiosflags(ios::left); cout << setw(10); cout << 5.6 << endl;</pre>	<pre>5.6</pre>

cin: Input Operator (>>)

- Preprogrammed for **all** standard C++ **data types**
- Get **bytes** from an input stream object
- Depend on **white space** to separate incoming data values



Input Operator

Type	Variable	Expression	Input	x	y
Integer	int x,y;	cin >> x;	21	21	
		cin >> x >> y;	5 3	5	3
Float	float x,y;	cin >> x;	14.5	14.5	
Character	char x,y;	cin >> x;	a	a	
		cin >> x >> y;	Hi	H	i
String	char x[20]; char y[20];	cin >> x;	hello	hello	
		cin >> x >> y	Hello World	Hello	World

Programming styles

- Programmers should write code that is understandable to other people as well
- **Meaningful** variable **names**
- Which is more meaningful
 - `tax=temp1*temp2;`
 - `tax=price*tax_rate;`
- **Meaningful** **comments**
 - Write comments as you're writing the program
- **Indentations**

Indentation styles

```
void main()  
{  
    int x, y;  
    x = y++;  
}
```

```
void main() {  
    int x, y;  
    x = y++;  
}
```

Both are good. Choose one and stick with it.

```
void main()  
{  
int x, y;  
    x= y++;}
```

BAD!! Avoid this!!

Use of comments

- **Top** of the program
 - Include information such as the name of organization, programmer's name, date and purpose of program
- What is achieved by the **function**, the meaning of the arguments and the return value of the function
- Short comments should occur to the right of the statements when the effect of the statement is not obvious and you want to illuminate what the program is doing
- Which one of the following **comment** is more meaningful?
 - `tax = price * rate; // sales tax formula`
 - `tax = price * rate; // multiply price by rate`

Backup Slides

Assignment operator =

- Generic form

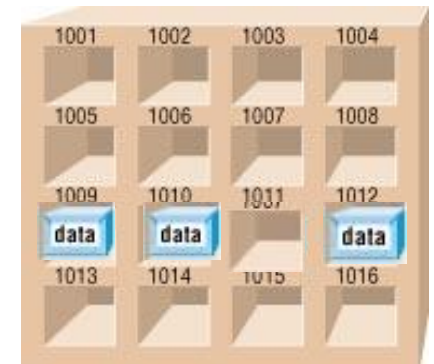
variable = expression;

- Variable

char x = 'a';

Type: char Name: X Address: 1009
97

Type: char Name: Y Address: 1010
98



Assignment operator =

- Generic form

variable = expression;

- = is an assignment operator that is different from the **mathematical equality** (which is == in C++)

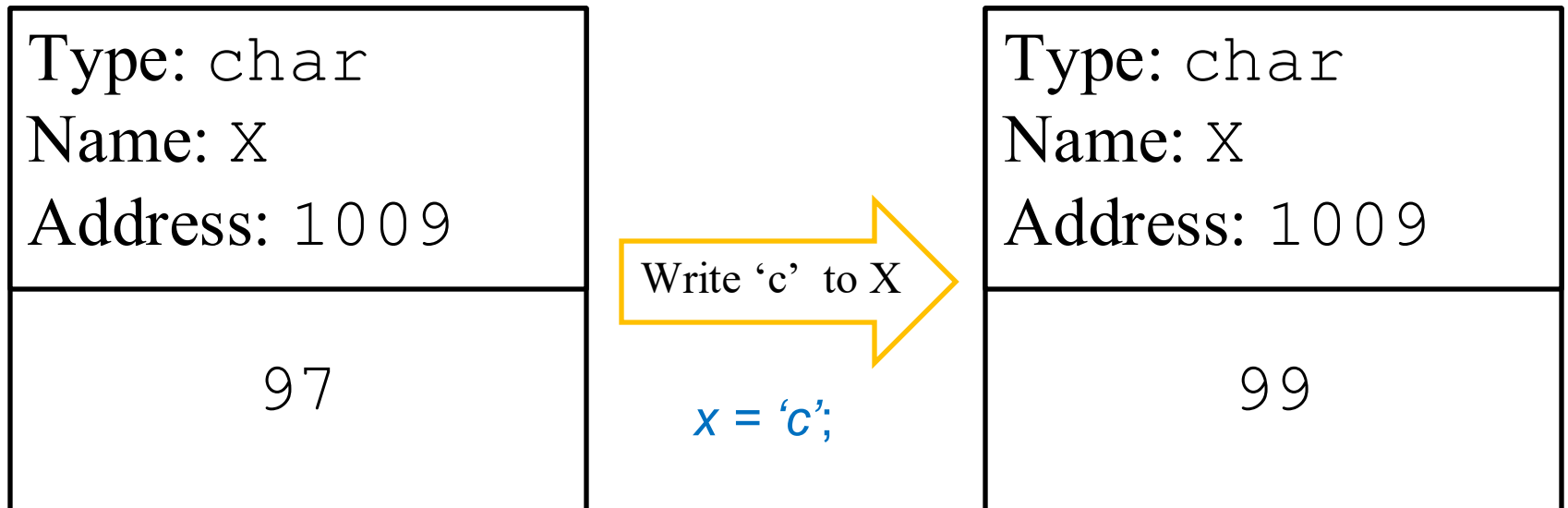
- An expression itself has a value, e.g.,

$a = (b = 2) + (c = 3);$  $a = 2 + 3;$

- An **assignment statement has a value** equal to the operand
- In the example, the value of **assignment statement** “ $b=2$ ” is 2 and “ $c = 3$ ” is 3
- Therefore, “ $a = \dots$ ” is 5

Assignment operator =

- **Write-to** a variable
- After the write, the previous stored value in the variable no longer exists, and is replaced by the new value



Efficient assignment operators

- Generic form of **efficient** assignment operators
*variable **op**= expression;*
where **op** is an operator. The meaning is
*variable = variable **op** (expression);*
- Popular efficient assignment operators include
+= -= *= /= %=
- Examples:

```
a = a + 5;  
a = a - 5;  
a = a + (b*c);  
a = a * (b+c);
```

Efficient assignment operators

- `i = i+1;`
 - `MOV A,D; //Move in A the content of the memory whose address is in D`
 - `ADD A, 1; //The addition of an constant`
 - `MOV D, A; //Move the result back to i (this is the '=' of the expression) ;`
- `i+=1;`
 - `ADD D,1; //Add an constant to a memory address stored value;`

switch statement: Syntax

```
switch(expression) //e.g., switch(x)
```

```
{
```

```
    case constant-expression://case 1:
```

```
        statement(s);
```

```
        break; //optional
```

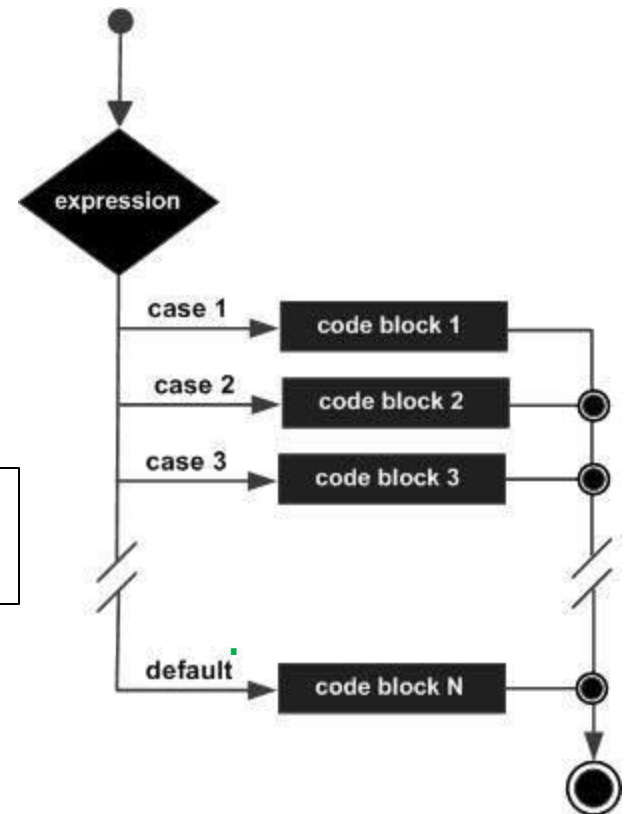
*Terminate the switch when a
break statement is encountered*

```
    default : //Optional
```

```
        statement(s);
```

```
}
```

*Go to the case label having a constant value that matches the value of **the switch expression**;
if a match is not found, go to the default label; if default label does not exist, terminate the switch*



switch statement: Syntax

switch(**expression**) //similar to if(expression)

{

case **constant**-expression://case 1:

statement(s);

break; //optional

case **constant**-expression://case 2:

statement(s);

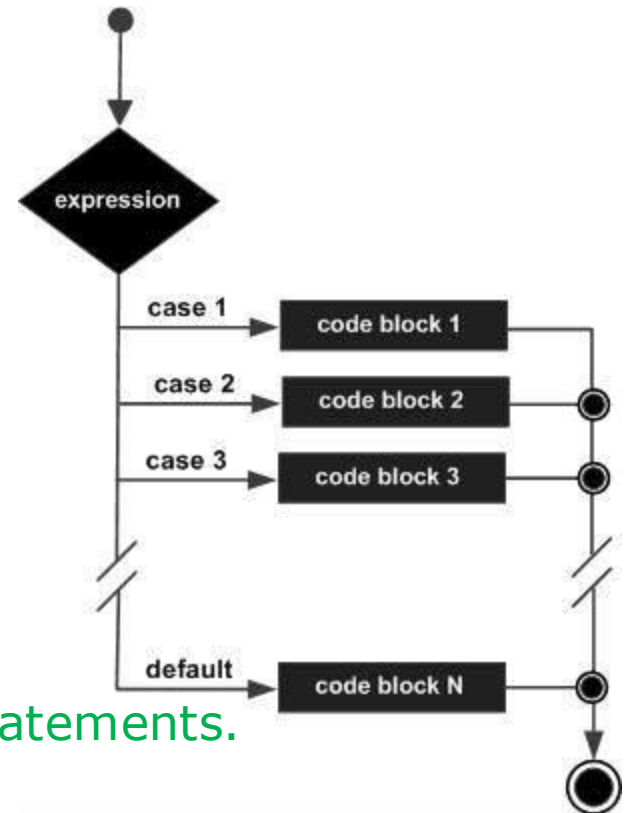
break; //optional

// you can have any number of case statements.

default : //Optional

statement(s);

}



If there is no break statement, execution “*falls through*” to the next statement in the succeeding case