Ch4: while, for, do…while, switch

while Iteration Statement

while Iteration Statements

while statements repeat an action (or group of actions) in their bodies while a condition remains true. When the condition is false, the iteration terminates, and the first statement after the body of while will execute. If the condition is initially false, the action (or group of actions) will not execute. Conditions are usually formed by using the relational and equality operators.

Body of while that can be a single statement or a **Block** of several statements in {}.

```
while (condition){
  statement;
  ...
  statement;
}
```

```
int x{3};
while (x <= 100) {
  x = 3 * x;
}
```

- Not providing in the body of a while statement an action that eventually causes the condition to become false results in a logic error called an **infinite loop** (the loop never terminates).

Counter-Controlled and Sentinel-Controlled Iterations

The iterations can be classified into two general categories:

- **Counter-Controlled Iteration** (**Definite Iteration**): The iteration/loop where we know the number of loop executions in advance.
  - A control variable called a **counter** is used to control the number of iterations.

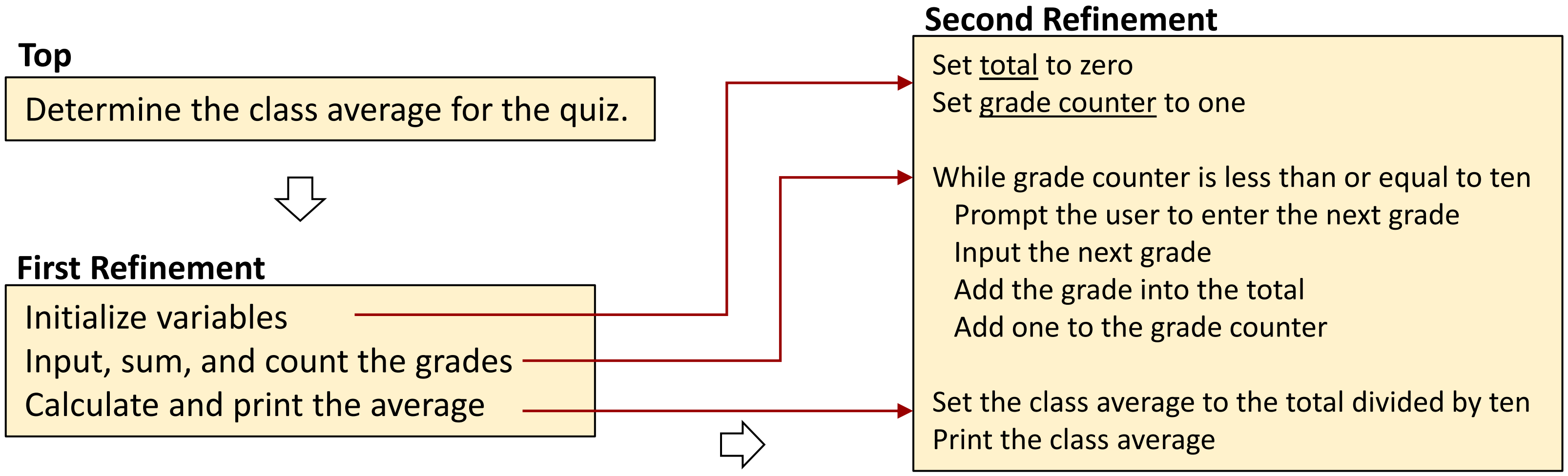
- **Sentinel-Controlled Iteration** (**Indefinite Iteration**): The iteration/loop where we do not know the number of loop executions in advance, and it depends on decisions made inside the loop.
  - A control variable called a **sentinel variable** (signal value, dummy value, or flag value) is used to indicate the end of iterations. The value of this variable changes inside the loop and the loop breaks when this value satisfies the loop condition.

Typically, for statements are used for counter-controlled iteration and while/do…while statements for sentinel-controlled iteration. However, for and while/do…while statements can each be used for either iteration type.

Sample Program: Averaging Student Grades Using Counter-Controlled Iteration

Problem: A class of 10 students took a quiz. The grades (integers in the range 0-100) for this quiz are available to you. Determine the class average on the quiz.

- Before starting to write your code for any problem, it is recommended to first develop an **algorithm** using a tool like **pseudocode** for the solution. This is essential to the development of well-structured programs. Once a correct algorithm has been specified, producing a working code from it is usually straightforward.
- A problem-solving approach using pseudocode is called **Top-Down, Stepwise Refinement**; Top is a single statement that conveys the overall function of the program. It is then divided into a series of smaller tasks. Each refinement is a complete specification of the algorithm, only the level of detail varies.



Top

```
Determine the class average for the quiz.
```

First Refinement

```
Initialize variables
Input, sum, and count the grades
Calculate and print the average
```

Second Refinement

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
   Prompt the user to enter the next grade
   Input the next grade
   Add the grade into the total
   Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Sample Program: Averaging Student Grades Using Counter-Controlled Iteration

```
// Solving the class-average problem using counter-controlled iteration.
#include <iostream>

int main() {
  // initialization phase
  int total{0}; // initialize sum of grades entered by the user
  unsigned int gradeCounter{1}; // initialize grade # to be entered next

  // processing phase uses counter-controlled iteration
  while (gradeCounter <= 10) { // loop 10 times
    std::cout << "Enter grade: "; // prompt
    int grade;
    std::cin >> grade; // input next grade
    total = total + grade; // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter by 1
  }

  // termination phase
  int average{total / 10}; // int division yields int result

  // display total and average of grades
  std::cout << "\nTotal of all 10 grades is " << total;
  std::cout << "\nClass average is " << average << std::endl;
}
```

Remarks

```
int total{0};
unsigned int gradeCounter{1};
```

- Variables used to store **totals** are normally initialized to 0.
- In general, **counters** that should store only nonnegative integer values should be declared with unsigned types, and are normally initialized to 0 or 1, depending on how they're used.
- Floating-point values are **approximate**. Thus, controlling counting loops with floating-point variables can result in imprecise counter values and inaccurate tests for termination.

```
int grade;
```

- Variable grade, declared in the body of the while loop, is a **local variable** of the block.
- It is a good practice to define a variable within a loop, since identifiers should be confined to the smallest possible scope. However, it is a bad practice to initialize a variable by a complex function within a loop if you could just as well do it once before the loop runs. In loops, avoid calculations for which the result never changes; those should typically be placed before the loop.

Thus, if compute() always returns the same value:

(Good Practice)

```
int value = compute();
while (something) {
   doSomething(value);
}
```

(Bad Practice)

```
while (something) {
   int value = compute();
   doSomething(value);
}
```

Sample Program: Averaging Student Grades Using Sentinel-Controlled Iteration and Stacked Control Structures

Problem: Develop a class-averaging program that processes grades for an arbitrary number of students each time it is run.

Using Top-Down, Stepwise Refinement Method:

Top

```
Determine the class average for the quiz.
```

First Refinement

```
Initialize variables
Input, sum, and count the grades
Calculate and print the average
```

Second Refinement

```
Initialize total to zero
Initialize counter to zero

Prompt the user to enter the first grade or sentinel
Input the first grade (possibly the sentinel)
While the user has not yet entered the sentinel
   Add this grade into the running total
   Add one to the grade counter
   Prompt the user to enter the next grade
   Input the next grade (possibly the sentinel)

If the counter is not equal to zero
   Set the average to the total divided by the counter
   Print the average
else
   Print "No grades were entered"
```

(if...else for preventing of division by zero)

The program tests for a **sentinel value** (here, -1 to prevent confusion with an acceptable input value) after each grade is input and terminates the loop when the user enters the sentinel value.

Sample Program: Averaging Student Grades Using Sentinel-Controlled Iteration and Stacked Control Structures

```cpp
// Solving the class-average problem using sentinel-controlled iteration.
#include <iostream>
#include <iomanip> // parameterized stream manipulators

int main() {
  // initialization phase
  int total{0}; // initialize sum of grades
  unsigned int gradeCounter{0}; // initialize # of grades entered so far

  // processing phase
  // prompt for input and read grade from user
  std::cout << "Enter grade or -1 to quit: ";
  int grade;
  std::cin >> grade;

  // loop until sentinel value read from user
  while (grade != -1)  {
    total = total + grade; // add grade to total
    gradeCounter = gradeCounter + 1; // increment counter

    // prompt for input and read next grade from user
    std::cout << "Enter grade or -1 to quit: ";
    std::cin >> grade;
  }
```

```cpp
  // termination phase
  // if user entered at least one grade...
  if (gradeCounter != 0) {
    // use number with decimal point to calculate average of grades
    double average{static_cast<double>(total) / gradeCounter};

    // display total and average (with two digits of precision)
    std::cout << "\nTotal of the " << gradeCounter
      << " grades entered is " << total;
    std::cout << std::setprecision(2) << std::fixed;
    std::cout << "\nClass average is " << average << std::endl;
  }
  else { // no grades were entered, so output appropriate message
    std::cout << "No grades were entered" << std::endl;
  }
}
```

- It is a good practice to use prompts to <u>remind the user</u> of the sentinel in a sentinel-controlled loop.

- If the sentinel value is input, the loop terminates, and the program does not add –1 to the total.

Remarks

This code keep reading values and store into x (using sentinel-controlled iteration) until it encounters a data type other than the data type assigned to x or the end-of-file (EOF) indicator, where the condition evaluates to false.

```
#include <iostream>
int main() {
  int x;
  while (std::cin >> x) {
    std::cout << x << std::endl;
  }
}
```

The end-of-file indicator is system-dependent:

- On UNIX/Linux/Mac OS X systems: <Ctrl> d
- On Windows systems: <Ctrl> z

Whenever x is used inside the loop, you know it has been read successfully.

Sample Program: Analysis of Examination Results Using Nested Control Structures

Problem: By having a list of these 10 students, write a program to analyze how well the students did on an exam:

- Input each test result (i.e., a 1 if the student passed or a 2 if the student failed).
- Count the number of test results of each type.
- Display the number of students who passed and the number who failed.
- If more than eight students passed the exam, print “Congratulations!”.

Top

```
Analyze exam results and decide whether it is satisfactory
```

First Refinement

```
Initialize variables
Input the 10 exam results, and count passes and failures
Print a summary of the exam results and decide whether it is satisfactory
```

Second Refinement

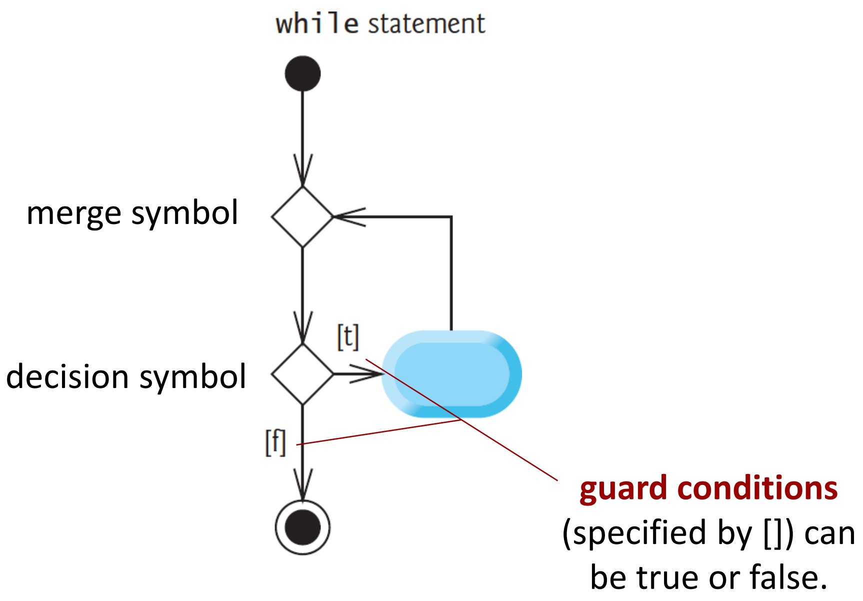
```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to 10
  Prompt the user to enter the next exam result
  Input the next exam result

  If the student passed
    Add one to passes
  Else
    Add one to failures

  Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
  Print “Congratulations!”
```

Sample Program: Analysis of Examination Results Using Nested Control Structures

```cpp
// Analysis of examination results using nested control statements.
#include <iostream>

int main() {
  // initializing variables in declarations
  unsigned int passes{0};
  unsigned int failures{0};
  unsigned int studentCounter{1};

  // process 10 students using counter-controlled loop
  while (studentCounter <= 10) {
    // prompt user for input and obtain value from user
    std::cout << "Enter result (1 = pass, 2 = fail): ";
    int result;
    std::cin >> result;

    // if...else is nested in the while statement
    if (result == 1) {
      passes = passes + 1;
    }
    else {
      failures = failures + 1;
    }

    // increment studentCounter so loop eventually terminates
    studentCounter = studentCounter + 1;
  }
```

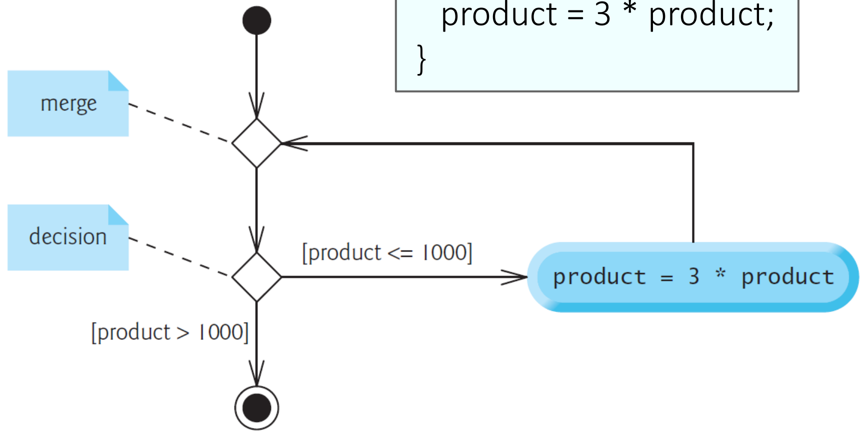
```cpp
  // termination phase; prepare and display results
  std::cout << "Passed: " << passes << "\nFailed: " << failures << std::endl;

  // determine whether more than 8 students passed
  if (passes > 8) {
    std::cout << "Congratulations!" << std::endl;
  }
}
```

UML Activity Diagram for a while Statement



Example:

```
int product{3};
while (product <= 100) {
  product = 3 * product;
}
```

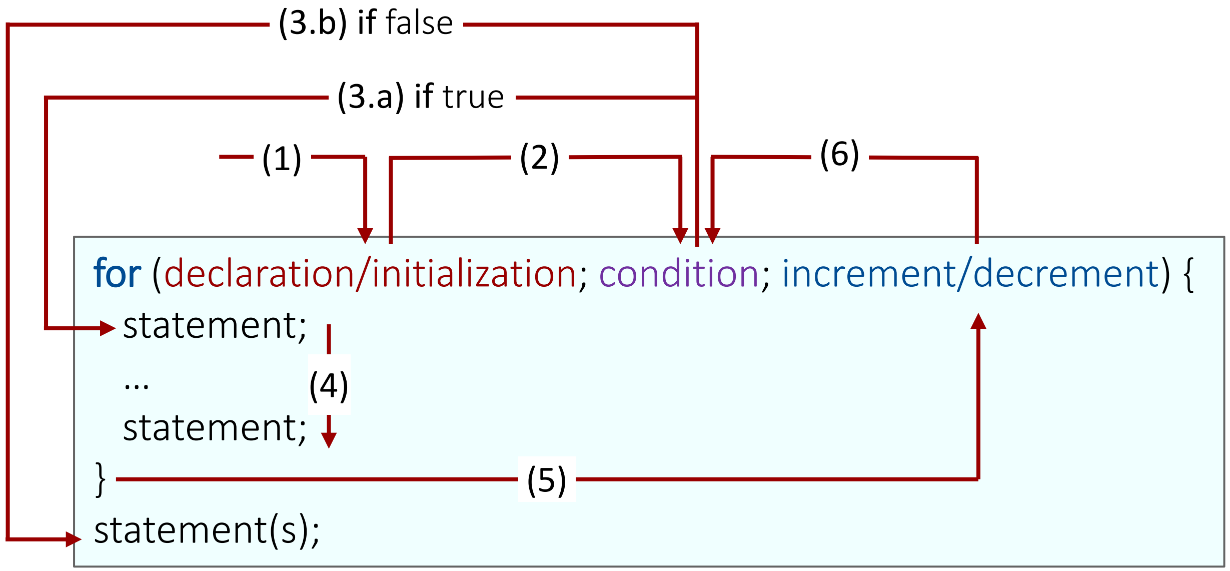


- The **merge symbol** joins two flows of activity into one.
- The decision and merge symbols can be distinguished by the number and direction of “incoming” and “outgoing” transition arrows. In addition, each transition arrow pointing out of a decision symbol has a guard condition next to it.

for Iteration Statement

for Iteration Statement

for statements repeat an action (or group of actions) in their bodies while a condition remains true. When the condition is false, the iteration terminates, and the first statement after the body of for will execute. If the condition is initially false, the action (or group of actions) will not execute. Conditions are usually formed by using the relational and equality operators.



```
for (declaration/initialization; condition; increment/decrement) {
  statement;
  ...
  statement;
}
statement(s);
```

⇔

Corresponding while statement:

```
declaration & initialization;
while (condition) {
  statement(s);
  increment/decrement;
}
statement(s);
```

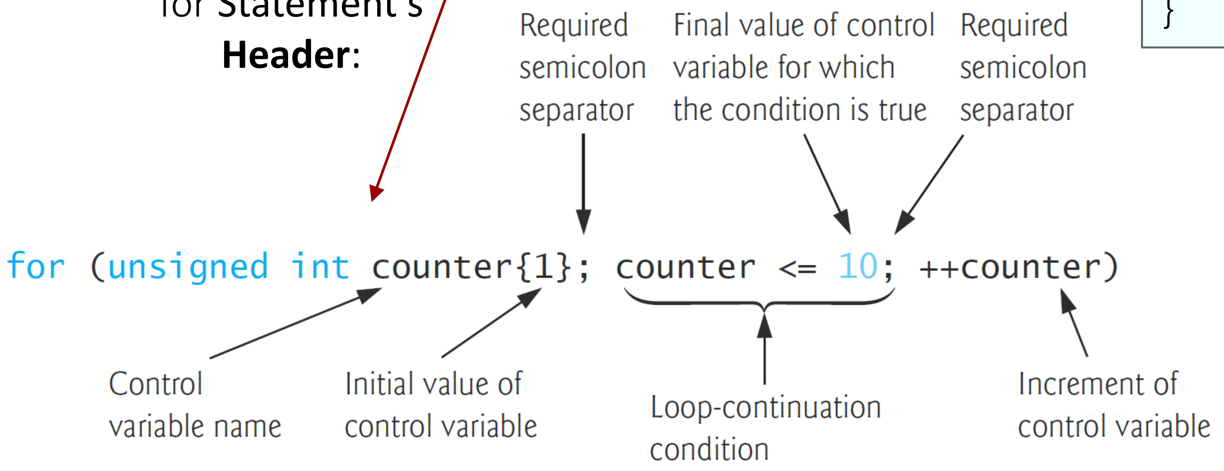
Example: Using for and while to Display Numbers from 1 to 10

```
#include <iostream>
int main() {
  for (unsigned int counter{1}; counter <= 10; ++counter) {
    std::cout << counter << "  ";
  }
}
```

⇔

```
#include <iostream>
int main() {
  unsigned int counter{1};
  while (counter <= 10) {
    std::cout << counter << "  ";
    ++counter;
  }
}
```

for Statement's **Header:**


Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
for (unsigned int counter{1}; counter <= 10; ++counter)
Control variable name
Initial value of control variable
Loop-continuation condition
Increment of control variable

- **while** can be used in most (but not all) cases in place of **for**. Typically, **for** statements are used for counter-controlled iteration and **while** statements for sentinel-controlled iteration.

Expressions in a for Header

- If a for statement's control variable is declared in the initialization section of the for's header, it can be used only in that for's body, not beyond it (variable's scope).

- If the program declares/initializes the control variable before the loop, declaration/initialization can be omitted.

- If the program calculates the increment/decrement in the loop's body or if no increment/decrement is needed, this expression can be omitted.

```
#include <iostream>
int main() {
  unsigned int i{1};
  for ( ; i <= 10; ) {
    std::cout << i << "  ";
    ++i;
  }
}
```

- If the loop-continuation condition is omitted, C++ assumes that the condition is always true, thus, creating an **infinite loop**.

Expressions in a for Header (cont.)

- The increment/decrement expression in a for acts as if it were a standalone statement at the end of the for's body. Therefore, the following increment/decrement expressions are equivalent in a for statement:

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

or

```
i = i - 1
i -= 1
--i
i--
```

(preferred)

- While using decrement expression, the loop **counts downward**. In this case, using unsigned int may result in an infinite loop.

```
#include <iostream>
int main() {
  for (int i{10}; i >= 0; i -= 1) {
    std::cout << i << "  ";
  }
}
```

Can we use unsigned int here?

Reason:

```
#include <iostream>
#include<climits>
int main() {
  unsigned int i{1};
  std::cout << i << "\n";
  i = -1;
  std::cout << i << "\n";
  std::cout << "UINT_MAX: " << UINT_MAX;
}
```

- Placing a semicolon immediately to the right of the right parenthesis of a for header makes that for's body an empty statement. This is normally a logic error.

Expressions in a for Header (cont.)

- Arithmetic expressions can be placed everywhere in a for statement's header.

```
int y{10};
int x{2};
for (int j = x; j <= 4 * x * y; j += y / x)
```

- Using an incorrect relational operator or an incorrect final value of a loop counter in the loop-continuation condition of for statement can cause a common logic error called an **off-by-one** error. For example:

```
for (unsigned int i{1}; i < 11; ++i)
```

$\equiv$

```
for (unsigned int i{1}; i <= 10; ++i)
```

$\equiv$

```
for (unsigned int i{0}; i < 10; ++i)
```

(10 iterations)

$\not\equiv$

```
for (unsigned int i{1}; i < 10; ++i)
```

(9 iterations)

- If a program must modify the control variable's value in the loop's body, use while rather than for.

Sample Program: Compound-Interest Calculations (Method 1: Floating-Point-Based Calculations)

A person invests $1,000 in a savings account yielding 5% annual interest rate. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula:

$$a = p(1 + r)^n$$

p: original amount invested (i.e., the principal)
r: annual interest rate (e.g., use 0.05 for 5%)
n: number of years
a: amount on deposit at the end of the nth year.

```
#include <iostream>
#include <iomanip>
#include <cmath> // for pow function
int main() {
  // set floating-point number format
  std::cout << std::fixed << std::setprecision(2);
  double principal{1000}; // initial amount before interest
  double rate{0.05}; // interest rate
  std::cout << "Initial principal: " << principal << "\n";
  std::cout << "    Interest rate:    " << rate << "\n";
  // display headers
  std::cout << "\nYear" << std::setw(20) << "Amount on deposit" << "\n";
  // calculate on deposit for each of ten years
  for (unsigned int year{1}; year <= 10; year++) {
    double amount = principal * std::pow(1 + rate, year);
    // display the year and the amount
    std::cout << std::setw(4) << year << std::setw(20) << amount << "\n";
  }
}
```

- Standard Library Function pow(x, y) from header <cmath> calculates the value of x^y.

Formatting with setw and Justification with left, right

setw(n) is a parameterized stream manipulator which specifies that the next value output should appear in a field width of at least n character positions.

- If the output value is less than n character positions wide, the value is right justified in the field by default.
- If the output value is more than n character positions wide, the field width is extended with additional character positions to the right to accommodate the entire value.

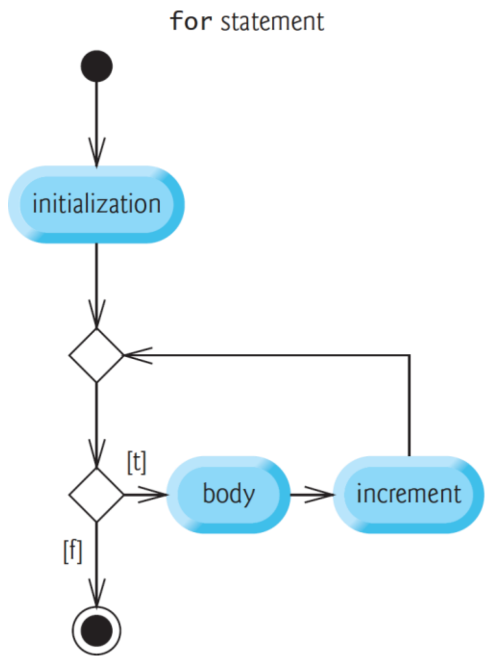
```
#include <iostream>
#include <iomanip>
int main() {
  std::cout << "Default positioning:\n"
         << std::setw(9) << "Print" << '\n';

  std::cout << "Left positioning:\n" << std::left
         << std::setw(9) << "Print" << '\n';

  std::cout << "Right positioning:\n" << std::right
         << std::setw(9) << "Print" << '\n';
}
```

```
Default positioning:
    Print
Left positioning:
Print
Right positioning:
    Print
```

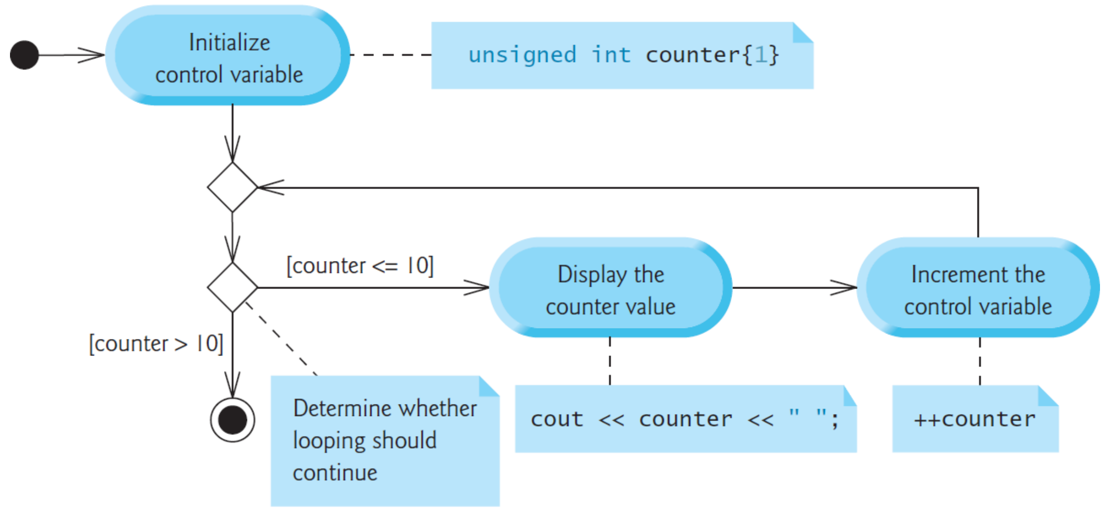
- To indicate that values should be output left justified, simply output stream manipulator left.
- Right justification can be restored by outputting stream manipulator right.

- setw is defined in <iomanip> header file and left, right are defined in <iostream> header file.

UML Activity Diagram for for Statement



Example:


Initialize control variable
unsigned int counter{1}
[counter <= 10]
Display the counter value
Increment the control variable
[counter > 10]
Determine whether looping should continue
cout << counter << " ";
++counter

```
#include <iostream>
int main() {
  for (int counter{1}; counter <= 10; counter++) {
    std::cout << counter << "  ";
  }
  std::cout << std::endl;
}
```

Comma Operator

Comma as a Separator and Operator

- **Comma as a Separator** is used to separate multiple variables in a variable declarations, multiple elements in array declaration and initialization, and multiple arguments/ parameters in function calls and definitions, enum declarations, and constructs.

```
int a{1}, b{2}, c;
```

```
void setNumbers(int X, int Y, int Z)
```

- **Comma as an Operator** between the expressions allows to evaluate multiple expressions wherever a single expression is allowed. It guarantees that a list of expressions evaluates from left to right (and returns the value/type of the rightmost expression, if required).

```
int a{1}, b{2}, c{3}; // Comma as a Separator
int i = (a, b); // Comma as an Operator, Result: i=2
int j = (a, b, c); // Comma as an Operator, Result: j=3
int k = (a += 2, a + b); // Comma as an Operator, Result: k=5
int l = a, b; // evaluates as "(l=a), b", i.e., l gets assigned
              // the value of a, and b is evaluated and discarded.
```

```
#include <iostream>
int main() {
   int x{ 1 };
   int y{ 2 };
   std::cout << (++x, ++y) << '\n';
  // increment x and y, evaluates to the right operand
}
```

Applications of Comma Operator

- Application in Condition: It can be used within a condition (of an if, while, do…while, or for) to allow auxiliary computations, e.g., doing arithmetic operations or calling a function and using the result by a comma-separated list.

```
if (y = f(x), y > x) {
   ... // statements involving x and y
}
```

⟺

```
if ( (y = f(x)) > x) {
   ... // statements involving x and y
}
```

- Application in for Statements: It can be used to allow multiple initialization expressions and/or multiple increment/decrement expressions by comma-separated lists.

```
for (int lower = 0, upper = 10; lower < upper; ++lower, --upper){
   ... // statements involving lower and upper
}
```

- Application in Avoiding a Block and Its Associated Braces:

```
if (condition)
   x = 2, y = 3;
```

⟺

```
if (condition) {
   x = 2;
   y = 3;
}
```

Sample Program: Summing Even Integers

Problem: Using a for statement, write a program to sum the even integers from 2 to 20 and print the result.

```
// Summing integers with the for statement.
#include <iostream>
int main() {
  unsigned int total{0};
  // total even integers from 2 through 20
  for (unsigned int number{2}; number <= 20; number += 2) {
    total += number;
  }
  std::cout << "Sum is " << total << std::endl;
}
```

You could merge the statement's body into the increment portion by using a **comma operator**.

```
#include <iostream>
int main() {
  unsigned int total{0};
  for (unsigned int number{2}; number <= 20; total += number, number += 2){
  }
  std::cout << "Sum is " << total << std::endl;
}
```

do...while Iteration Statement

do...while Iteration Statement

do...while statements are similar to the while statements, however, the do...while statements test the loop-continuation condition after executing the loop's body; thus, the body always executes at least once. When the condition is false, the iteration terminates, and the first statement after the body of do...while will execute. Conditions are usually formed by using the relational and equality operators.

```
do {
  statement;
  ...
  statement;
  } while (condition);
```

";" here.

Example: Using a do...while to output the numbers 1–10:

```
#include <iostream>
int main() {
  unsigned int counter{1};

  do {
    std::cout << counter << "  ";
    ++counter;
  } while (counter <= 10);

  std::cout << std::endl;
}
```

- Not providing in the body of a do...while statement an action that eventually causes the condition to become false results in a logic error called an **infinite loop** (the loop never terminates).

do...while Iteration Statement

Note: Since scope of a variable declared in a block {} is just within the block, a variable declared in body of a do...while cannot be used in the loop condition, which is outside that scope.

✗

```
#include <iostream>
int main() {
  int i{0};
  do {
    int j;
    j = i * 2;
    std::cout << j << " ";
    i++;
  } while (j < 100);
}
```

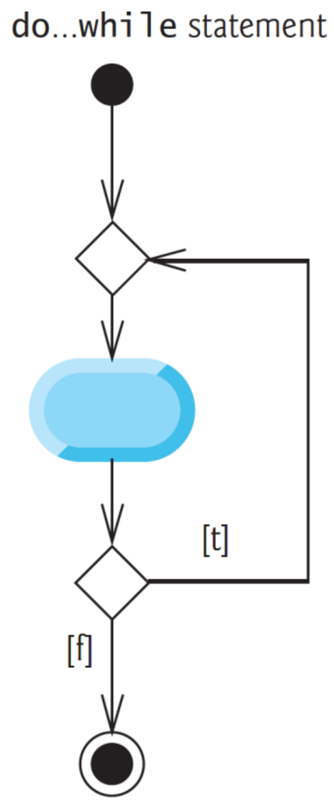
j declared inside do...while's body. Thus, it cannot be used in condition.

✓

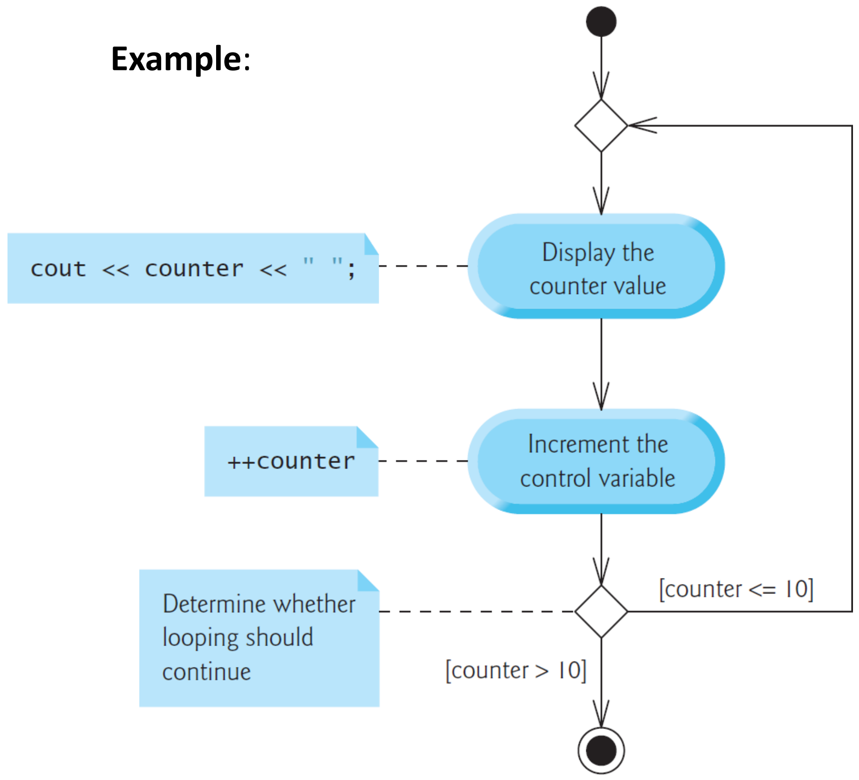
```
#include <iostream>
int main() {
  int i{0};
  int j;
  do {
    j = i * 2;
    std::cout << j << " ";
    i++;
  } while (j < 100);
}
```

j declared outside do...while's body. Thus, it can be used in condition.

UML Activity Diagram for do...while Statement



Example:


cout << counter << " ";
Display the counter value
++counter
Increment the control variable
Determine whether looping should continue
[counter <= 10]
[counter > 10]

```
#include <iostream>
int main() {
  unsigned int counter{1};

  do {
    std::cout << counter << "  ";
    ++counter;
  } while (counter <= 10);

  std::cout << std::endl;
}
```

switch Multiple-Selection Statement

switch Multiple-Selection Statement

switch Multiple-Selection Statement selects among many different actions (or groups of actions), depending on the value of a variable or expression.

break statement at the end of a case causes control to exit the switch statement immediately.

break statement is not required for the last case (or the optional default case, when it appears last).

```
switch (an expression or variable) {
  case label1:
    statement(s);
    break;
  case label2:
    statement(s);
    break;
  ...
  default:
    statement(s);
    break;
}
```

switch statement does not require braces ({}) around multiple statements in a case.

switch's controlling expression/variable is evaluated to produce a value.

- If the expression's value is equal to the value after any of the **case labels**, the statements after the matching case label are executed.
- If no matching value can be found and a **default case** exists, the statements after the default case are executed instead, otherwise, execution continues after the end of the **switch** block.

switch vs if...else

```
#include <iostream>
int main() {
  int x;
  std::cout << "Enter a Number: ";
  std::cin >> x;
  if (x == 1)
    std::cout << "One";
  else if (x == 2)
    std::cout << "Two";
  else if (x == 3)
    std::cout << "Three";
  else
    std::cout << "Unknown";
}
```

x is evaluated up to three times, which is inefficient.

$\Leftrightarrow$

```
#include <iostream>
int main() {
  int x;
  std::cout << "Enter a Number: ";
  std::cin >> x;
  switch (x) {
    case 1:
      std::cout << "One";
      break;
    case 2:
      std::cout << "Two";
      break;
    case 3:
      std::cout << "Three";
      break;
    default:
      std::cout << "Unknown";
      break;
  }
}
```

x is evaluated only once, which is more efficient.

Remarks

- The **controlling expression/variable** must evaluate to a signed or unsigned integral type (bool, char, int, long, long long, or enumerated types that evaluates to a constant integer value), but not floating-point types or strings.
- The value after the case labels must either match the type of the controlling expression /variable or must be convertible to that type.
- There is no practical limit to the number of case labels you can have, but all case labels in switch statement must be unique.

```
switch (x) {
  case 54:
    ...
  case 54:  // error: already used value 54!
    ...
  case '6': // error: '6' converts to integer value 54, which is already used
    ...
}
```

- The **default** label is **optional**, and there can only be one **default** label per **switch** statement. By convention, the **default** case is placed last in the **switch** statement.

Remarks

- switch statement does not provide a mechanism for testing ranges of values. Therefore, every value you need to test must be listed in a separate case label.
- When break statement is omitted for a case, each time a match occurs, the statements for that case and subsequent cases execute until a break statement or the end of the switch is encountered. This is called "**falling through**" to the statements in subsequent cases.

```
#include <iostream>
int main() {
  char n{'B'};
  switch (n) {
    case 'A':
      std::cout << 'A' << "\n"; // Skipped
    case 'B': // Match!
      std::cout << 'B' << "\n"; // Execution begins here
      std::cout << 'B' << "\n";
    case 'C':
      std::cout << 'C' << "\n"; // This is also executed
      break;
    case 'D':
      std::cout << 'D' << "\n"; // Skipped
      break;
    default:
      std::cout << 'E' << "\n"; // Skipped
      break;
  }
}
```

ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

The digits at the left of the table are the left digits of the decimal equivalents (0–127) of the character codes, and the digits at the top of the table are the right digits of the character codes. For example, the character code for “F” is 70, and the character code for “&” is 38.

Sample Program: Using a switch Statement in Member Function of a Class

Day.h file

```
class Day {
public:
  void set_data() {
    std::cout<<"Enter number of day: ";
    std::cin>>day;
  }
  void display_day()  {
    switch (day) {
      case 1:
        std::cout<<"MONDAY";
        break;
      case 2:
        std::cout<<"TUESDAY";
        break;
      case 3:
        std::cout<<"WEDNESDAY";
        break;
      case 4:
        std::cout<<"THURSDAY";
        break;
      case 5:
        std::cout<<"FRIDAY";
        break;
      case 6:
        std::cout<<"SATURDAY";
        break;
      case 7:
        std::cout<<"SUNDAY";
        break;
      default:
        std::cout<<"INVALID INPUT";
        break;
    }
  }
private:
  int day;
};
```

main.cpp file

```
#include<iostream>
#include"Day.h"
int main() {
   Day d;
   d.set_data();
   d.display_day();
}
```

Sample Program: Using a switch Statement to Count Letter Grades

Problem: Calculate the class average of a set of numeric grades entered by the user, and uses a switch statement to determine the number of students who received an A, B, C, D or F.

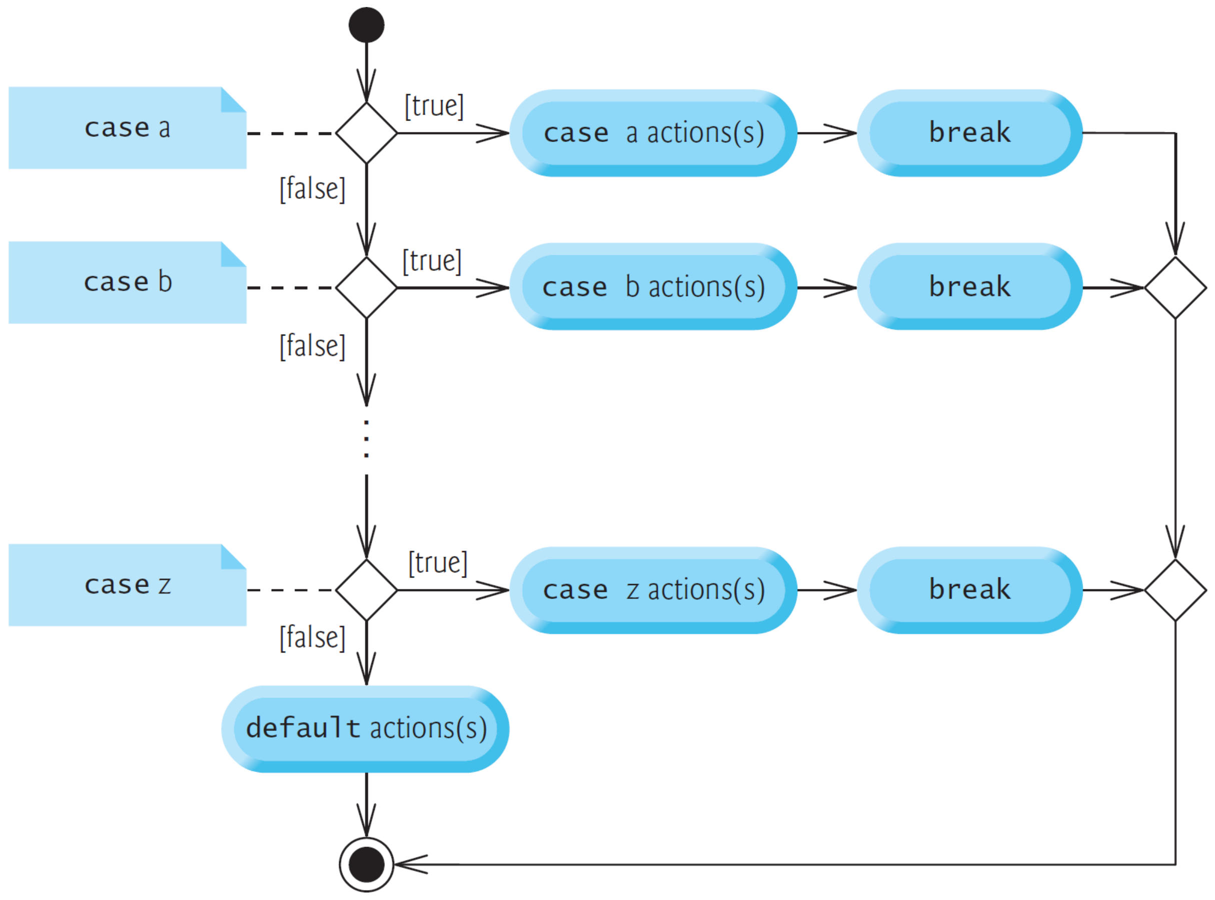
```
// Using a switch statement to count letter grades.
#include <iostream>
#include <iomanip>

int main() {
  int total{0}; // sum of grades
  unsigned int gradeCounter{0}; // number of grades entered
  unsigned int aCount{0}; // count of A grades
  unsigned int bCount{0}; // count of B grades
  unsigned int cCount{0}; // count of C grades
  unsigned int dCount{0}; // count of D grades
  unsigned int fCount{0}; // count of F grades

  std::cout << "Enter the integer grades in the range 0-100.\n"
    << "Type the end-of-file indicator to terminate input:\n"
    << "   On UNIX/Linux/Mac OS X type <Ctrl> d then press Enter\n"
    << "   On Windows type <Ctrl> z then press Enter\n";

  int grade;
```

```
// loop until user enters the end-of-file indicator
while (std::cin >> grade) {
  total += grade; // add grade to total
  ++gradeCounter; // increment number of grades

  //  increment appropriate letter-grade counter
  switch (grade / 10) {
    case 10:  // grade was 100
    case 9: // grade was between 90 and 99
      ++aCount;
      break; // exits switch
    case 8: // grade was between 80 and 89
      ++bCount;
      break; // exits switch
    case 7: // grade was between 70 and 79
      ++cCount;
      break; // exits switch
    case 6: // grade was between 60 and 69
      ++dCount;
      break; // exits switch
     default: // grade was less than 60
      ++fCount;
      break; // optional; exits switch anyway
  } // end switch
} // end while
```

Sample Program: Using a switch Statement to Count Letter Grades (cont.)

```
  // set floating-point number format
  std::cout << std::fixed << std::setprecision(2);

  // display grade report
  std::cout << "\nGrade Report:\n";

  // if user entered at least one grade...
  if (gradeCounter != 0) {
    // calculate average of all grades entered
    double average{static_cast<double>(total) / gradeCounter};

    // output summary of results
    std::cout << "Total of the " << gradeCounter << " grades entered is "
      << total << "\nClass average is " << average
      << "\nNumber of students who received each grade:"
      << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
      << "\nD: " << dCount << "\nF: " << fCount << std::endl;
  }
  else { // no grades were entered, so output appropriate message
    std::cout << "No grades were entered" << std::endl;
  }
}
```

UML Activity Diagram for switch Statement



break and continue Statements

break Statement

The break statement, when executed in a while, for, do...while, or switch, causes immediate exit from that loop, and execution continues with the first statement after the control statement.

- Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch.

```
#include <iostream>
int main() {
  for (unsigned int count{1}; count <= 10; ++count) { // loop 10 times
    if (count == 5) {
      break; // terminates loop if count is 5
    }
    std::cout << count << " ";
  }
  std::cout << "\nBroke out of loop at count = 5" << std::endl;
}
```

```
1 2 3 4
```

continue Statement

The continue statement, when executed in a while, for, or do…while, skips the remaining statements in the loop body and proceeds with the next iteration of the loop.

- In while and do…while statements, after the continue statement executes, the program evaluates the loop-continuation test immediately.
- In a for statement, after the continue statement executes, the increment/decrement expression executes, then the program evaluates the loop-continuation test.

```
#include <iostream>
int main() {
  for (unsigned int count{1}; count <= 10; ++count) { // loop 10 times
    if (count == 5) {
      continue; // skip remaining code in loop body if count is 5
    }
    std::cout << count << " ";
  }
  std::cout << "\nUsed continue to skip printing 5" << std::endl;
}
```

```
1 2 3 4 6 7 8 9 10
```

while does not execute in the same manner as for

```
#include <iostream>
int main() {
  for (unsigned int count{1}; count <= 10; ++count) {
    if (count == 5) {
      continue;
    }
    std::cout << count << " ";
  }
}
```

```
1 2 3 4 6 7 8 9 10
```

```
#include <iostream>
int main() {
  unsigned int count{1};
  while (count <= 10) {
    if (count == 5) {
      continue;
    }
    std::cout << count << " ";
    ++count;
  }
}
```

```
infinite loop!
```

```
#include <iostream>
int main() {
  unsigned int count{1};
  while (count <= 10) {
    ++count;
    if (count == 5) {
      continue;
    }
    std::cout << count << " ";
  }
}
```

```
2 3 4 6 7 8 9 10 11
```

```
#include <iostream>
int main() {
  unsigned int count{0};
  while (count <= 9) {
    ++count;
    if (count == 5) {
      continue;
    }
    std::cout << count << " ";
  }
}
```

```
1 2 3 4 6 7 8 9 10
```

Constants

Literal Constants

Literal Constants are unnamed values inserted directly into the code. All literals have a type. The type of a literal is deduced from the literal's value by compiler.

Literal Value	Examples	Default Literal Type
integer value	5, 0, -3	int
Boolean value	true, false	bool
floating point value	1.2, 0.0, 3.4	double (not float!)
character	'a', '\n'	char
C-style string	"Hello, world!"	const char*

- If the default type of a literal is not as desired, you can change the type of a literal by adding a **suffix**:

Data type	Literal Suffix	Meaning
integral	u or U	unsigned int
integral	l or L	long
integral	ul, uL, Ul, UL, lu, lU, Lu, or LU	unsigned long
integral	ll or LL	long long
integral	ull, uLL, Ull, ULL, llu, llU, LLu, or LLU	unsigned long long
integral	z or Z	The signed version of std::size_t (C++23)
integral	uz or UZ	std::size_t (C++23)
floating point	f or F	float
floating point	l or L	long double
string	s	std::string
string	sv	std::string_view

- Suffixes are not case sensitive.

Literal Constants

```
#include <iostream>

int main() {
   std::cout << 5; // 5 (no suffix) is type int (by default)
   std::cout << 5u; // 5u is type unsigned int
   std::cout << 5L; // 5L is type long
   std::cout << 5.0; // 5.0 (no suffix) is type double (by default)
   std::cout << 5.0f; // 5.0f is type float
   float f{4.1f}; // use 'f' suffix so the literal is a float and matches variable type of float
   double d{4.1}; // type double matches the literal type double
}
```

const Variables

A variable whose value can not be changed is called a **constant variable**. Defining a variable as a constant (using const keyword in the variable's declaration) helps ensure that this value is not accidentally changed.

```
const var_type var_name
```
(preferred style)

or

```
var_type const var_name
```

- Constant variables must be initialized when declaring them, and then, that value can not be changed, e.g., via assignment.

```
#include <iostream>
int main() {
  std::cout << "Enter your age: ";
  int age{};
  std::cin >> age;
  const int constAge {age}; // initialize const variable using non-const value
  // int const constAge {age}; // "east const" style, okay but not preferred
  age = 5; // age is non-const, so we can change its value
  constAge = 6; // error: constAge is const, so we cannot change its value
}
```

Compile-time Constants

- A **Compile-time Constant** is a constant whose value is known at compile-time. Examples:
  - Literals (e.g., 1, 2.3, ‘A’, and “Hello, world!”).
  - A const variable only if its initializer is a constant expression.

An expression that all the values in it are known at compile-time, and it is evaluated by the compiler at compile-time.

For example, in `int x{3+4};`

3+4 is a constant expression, and a modern compiler will replace it with the resulting value 7 at compile-time.

```
#include <iostream>
int main() {
   const int x { 3 };  // x is a compile-time const
   const int y { 4 };  // y is a compile-time const
   const int z { x + y }; // x + y is a constant expression, so z is compile-time const
   const int a { 1 + 2 }; // 1 + 2 is a constant expression, so a is compile-time const
}
```

Run-time Constants

Runtime Constants are const variable whose initialization values are not known until run-time (i.e., they are initialized with a non-constant or run-time expression).

```
#include <iostream>
int getNumber() {
    std::cout << "Enter a number: ";
    int y{};
    std::cin >> y;
    return y;
}
int main() {
    const int x{ 3 };          // x is a compile time constant
    const int y{ getNumber() }; // y is a runtime constant
    const int z{ x + y };       // x + y is a runtime expression
    std::cout << z << '\n';     // a runtime expression
}
```

Therefore, depending on the initializer, const variables could end up as either a compile-time const or a runtime const.

constexpr Variables

There are a few cases where C++ requires a compile-time constant instead of a run-time constant.

By using the constexpr (short for "constant expression") keyword instead of const in a variable's declaration, we can ensure that the variable is a compile-time constant. Thus, if the initialization value of a constexpr variable is not a constant expression, the compiler will error.

```
#include <iostream>
int five() {
   return 5;
}
int main() {
   constexpr double gravity { 9.8 }; // ok: 9.8 is a constant expression
   constexpr int a { 4 + 5 };      // ok: 4 + 5 is a constant expression
   constexpr int b { a}; // ok: a is a constant expression
   std::cout << "Enter your age: ";
   int age{};
   std::cin >> age;
   constexpr int myAge { age }; // compile error: age is not a compile-time constant expression
   constexpr int f { five() }; // compile error: return value of five() is not a compile-time constant expression
}
```

Type Deduction (auto keyword)

Type Deduction

Because C++ is a strongly-typed language, we are required to provide an explicit type for all objects. **Type Deduction** (also sometimes called **Type Inference**) is a feature that allows the compiler to deduce the type of an object from the object's initializer. To use type deduction, the auto keyword is used in place of the variable's type.

```
int add(int x, int y) {
  return x + y;
}
int main() {
  auto d{ 5.0 }; // 5.0 is a double literal, so d will be type double
  auto i{ 1 + 2 }; // 1 + 2 evaluates to an int, so i will be type int
  auto x{ i }; // i is an int, so x will be type int too
  auto sum{ add(5, 6) }; // add() returns an int, so sum's type will be deduced to int

  const int a{ 5 };  // a has type const int
  auto b{ a };       // b will be type int (const is dropped)
  const auto c{ a }; // c will be type const int (const is reapplied)

  auto z1; // Error: The compiler is unable to deduce the type of z1
  auto z2{ }; // Error: The compiler is unable to deduce the type of z2
}
```

Type deduction will drop the const qualifier from deduced types. If you want a deduced type to be const, you must use the const keyword in conjunction with the auto keyword.

Type Deduction Using Literal Suffixes

```
#include <iostream>

int main() {
   auto x1{0}; // int (by default)
   auto x2{0L}; // long
   auto x3{0LL}; // long long
   auto x4{0.0f}; // float
   auto x5{0.0}; // double (by default)
   auto x6{0.0L}; // long double
   std::cout << "x1 is " << sizeof(x1) << " bytes\n";
   std::cout << "x2 is " << sizeof(x2) << " bytes\n";
   std::cout << "x3 is " << sizeof(x3) << " bytes\n";
   std::cout << "x4 is " << sizeof(x4) << " bytes\n";
   std::cout << "x5 is " << sizeof(x5) << " bytes\n";
   std::cout << "x6 is " << sizeof(x6) << " bytes\n";
}
```

Type Deduction for String Literals

If you want the type deduced from a string literal to be std::string or std::string_view, you must use the s or sv literal suffixes:

```
#include <string>
#include <string_view> // Since C++17
int main() {
    auto a { "Hello" }; // a will be type const char*, not std::string

    using namespace std::literals; // easiest way to access the s and sv suffixes, since C++14
    auto a1 { "Hello"s };  // "Hello"s is a std::string literal, so a1 will be deduced as a std::string

    // Since C++17:
    auto a2 { "Hello"sv }; // "Hello"sv is a std::string_view literal, so a2 will be deduced as a std::string_view
}
```