C++ & Python Program Design

-- Flow Control

Junjie Cao @ DLUT

Summer 2022

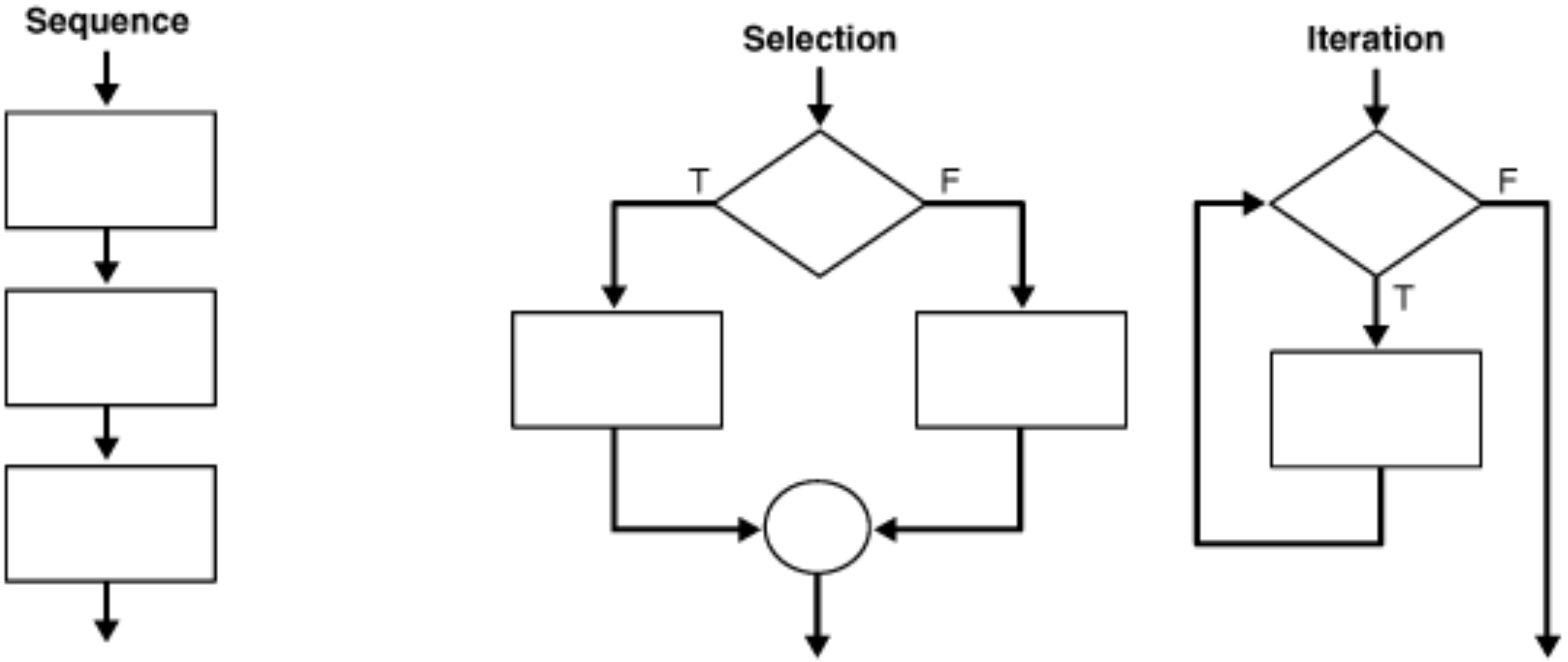
https://github.com/jjcao-school/c

Content

1. Control Structure
   - If else
   - While
   - for
2. Algorithm complexity
   - Timekeeping
   - O(n)

Control Structures

Motivation – Flow Control

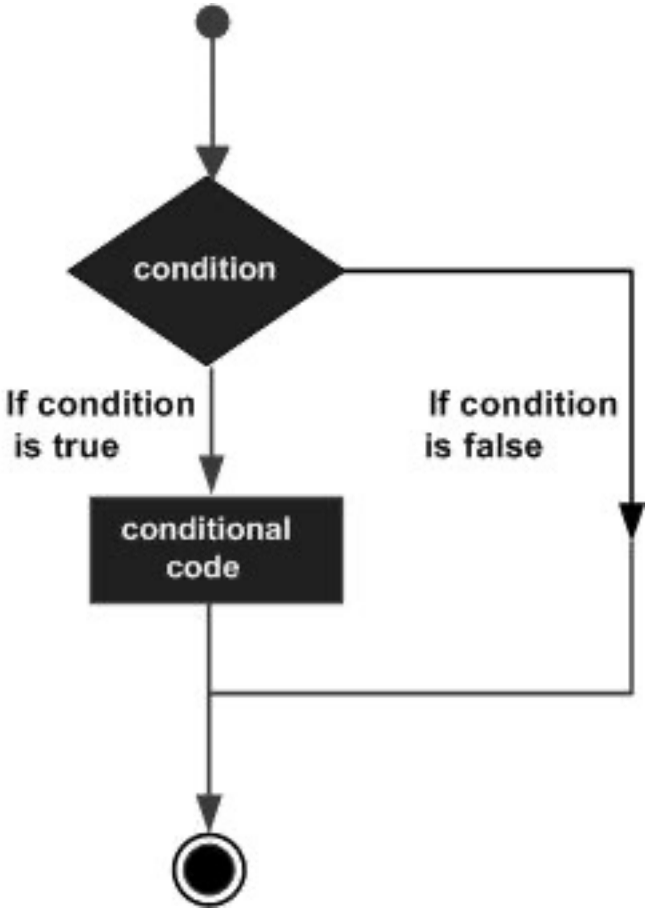


- Execute **statements语句** one by one, from first to last;
- Wouldn’t be very useful, e.g. moving in a game
- Alter the order of execution, execute or not – **control flow**

flow-of-control statements

Portions of code, depending on circumstances情况, execute in a certain way:

1. Conditionals: Check values of variables and to execute (or not execute) certain statements
   ① If
   ② Switch-case
2. Loops
   ① While
   ② for



Simple if

Which is Python?

```
if condition:
    statement1
    statement2
    ...
```

vs.

```
if (condition) {
    statement1
    statement2
    ...
}
```

If & if-else

```
if(condition)
{
        statement1
        statement2
        …
}
```

```
if(condition)
        statement
```

Block /Compound Statement

```
if(condition)
{
        statementA1
        statementA2
        …
}
else
{
        statementB1
        statementB2
        …
}
```

```
if(condition)
        statementA1
else
        statementB1
```

vs.

```
if condition:
   statement1
   statement2
   ...
else:
   statement1
   statement2
   ...
```

Else if

```
if(condition1)
{
        statementA1
        statementA2
        …
}
else if(condition2)
{
        statementB1
        statementB2
        …
}
```

- C++ does not have an elif
- May be more than one “else if”
- Once a block whose condition was met is executed, any “else if” after it are **ignored**, so:
  - Either one or no block is executed.
  - Optimize the order of “else if” for speeding up

```
grade = 85
```

```
if (grade < 60):
    print('F')
elif (grade < 70):
    print('D')
elif grade < 80:
    print('C')
elif grade < 90:
    print('B')
else:
    print('A')
```

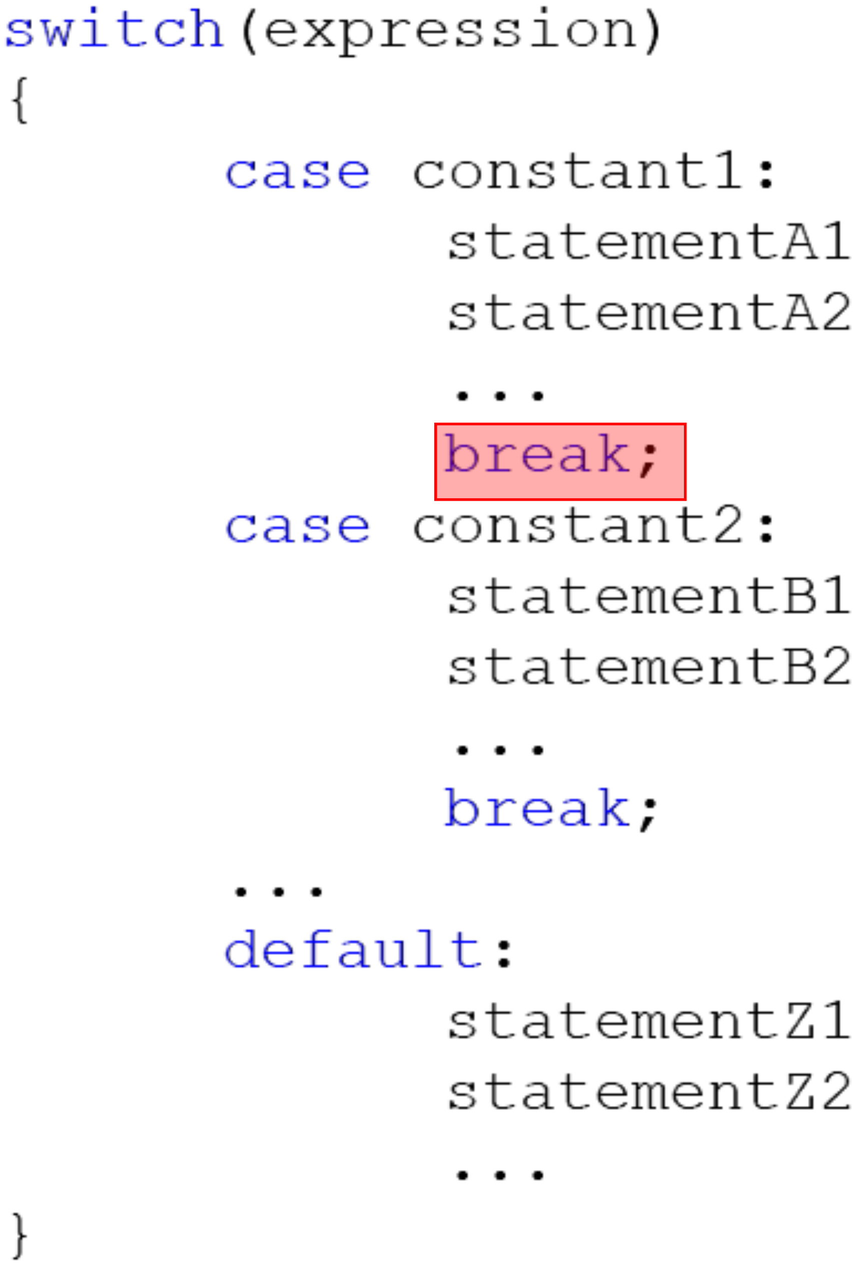
Example

```
if (grade < 60) {
    cout<<'F'<<endl;
}
else if (grade < 70) {
    cout<<'D'<<endl;
}
else if (grade < 80) {
    cout<<'C'<<endl;
}
else if (grade < 90) {
    cout<<'B'<<endl;
}
else cout<<'A'<<endl;
```

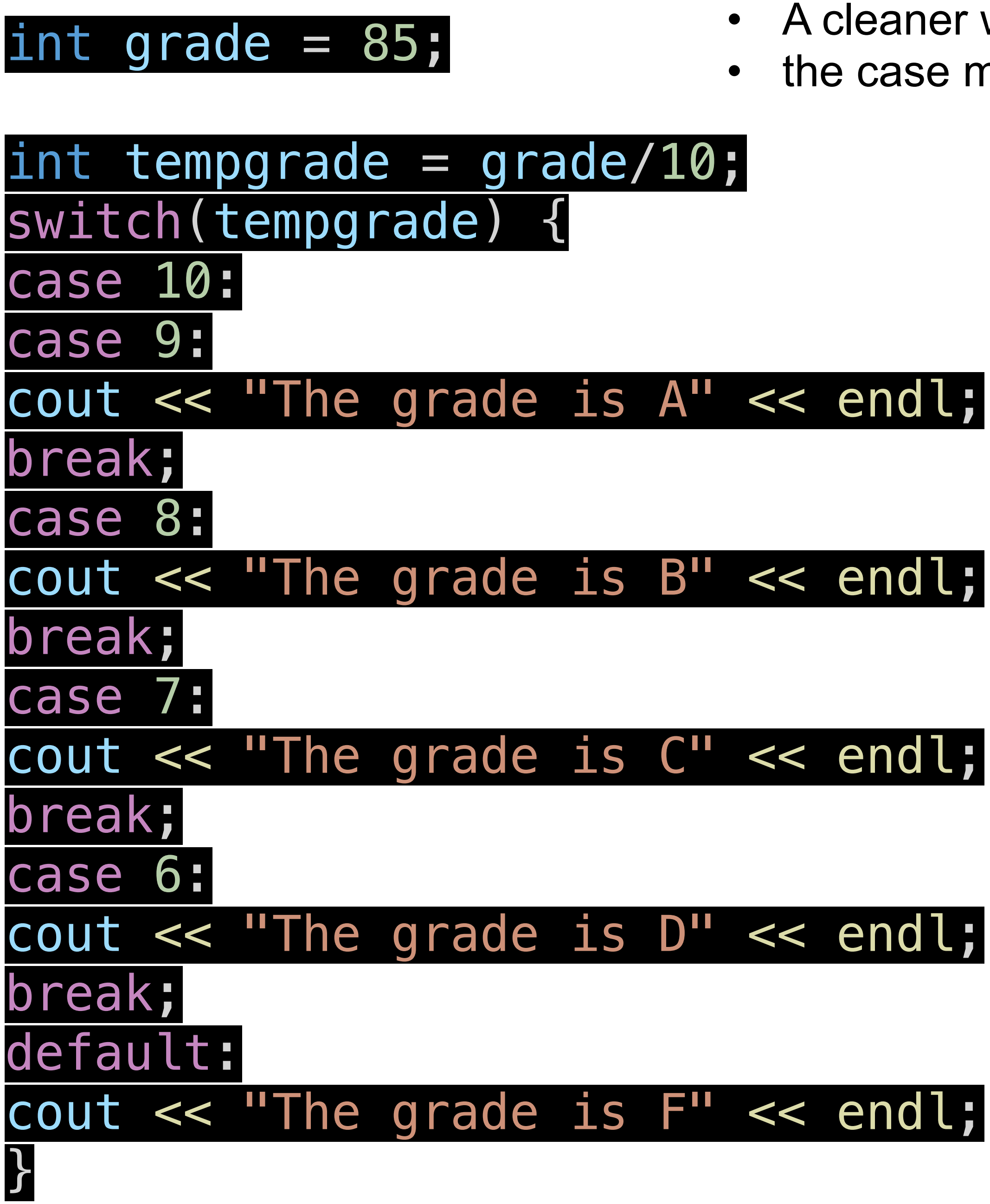
```
grade = 55

if (grade < 60):
    print('F')
elif (grade < 70):
    print('D')
elif grade < 80:
    print('C')
elif grade < 90:
    print('B')
else:
    print('A')
```

- Do you still remember how to input arguments from command line or screen?

Switch-case



```
switch(expression)
{
      case constant1:
            statementA1
            statementA2
            ...
            break;
      case constant2:
            statementB1
            statementB2
            ...
            break;
      ...
      default:
            statementZ1
            statementZ2
            ...
}
```



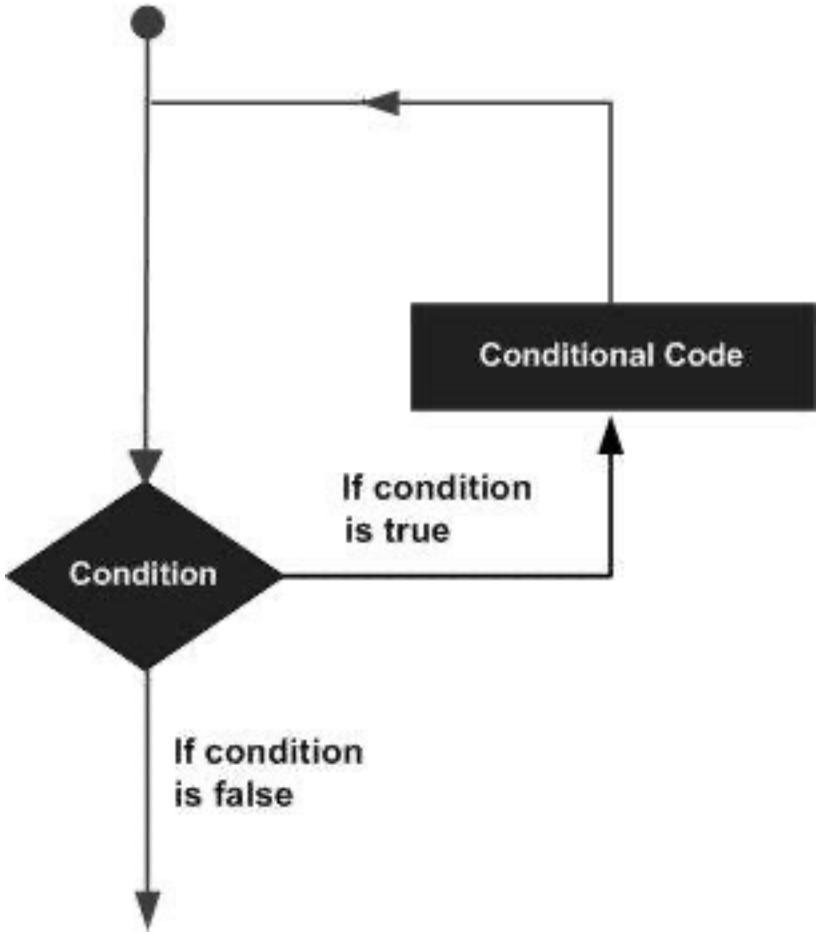
```
int grade = 85;

int tempgrade = grade/10;
switch(tempgrade) {
case 10:
case 9:
cout << "The grade is A" << endl;
break;
case 8:
cout << "The grade is B" << endl;
break;
case 7:
cout << "The grade is C" << endl;
break;
case 6:
cout << "The grade is D" << endl;
break;
default:
cout << "The grade is F" << endl;
}
```

- A cleaner way than using “if-else”
- the case must be based on integers

flow-of-control statements

Portions of code, depending on circumstances情况, execute in a certain way:

1. Conditionals: Check values of variables and to execute (or not execute) certain statements
   ① If
   ② Switch-case
2. Loops
   ① While
   ② for



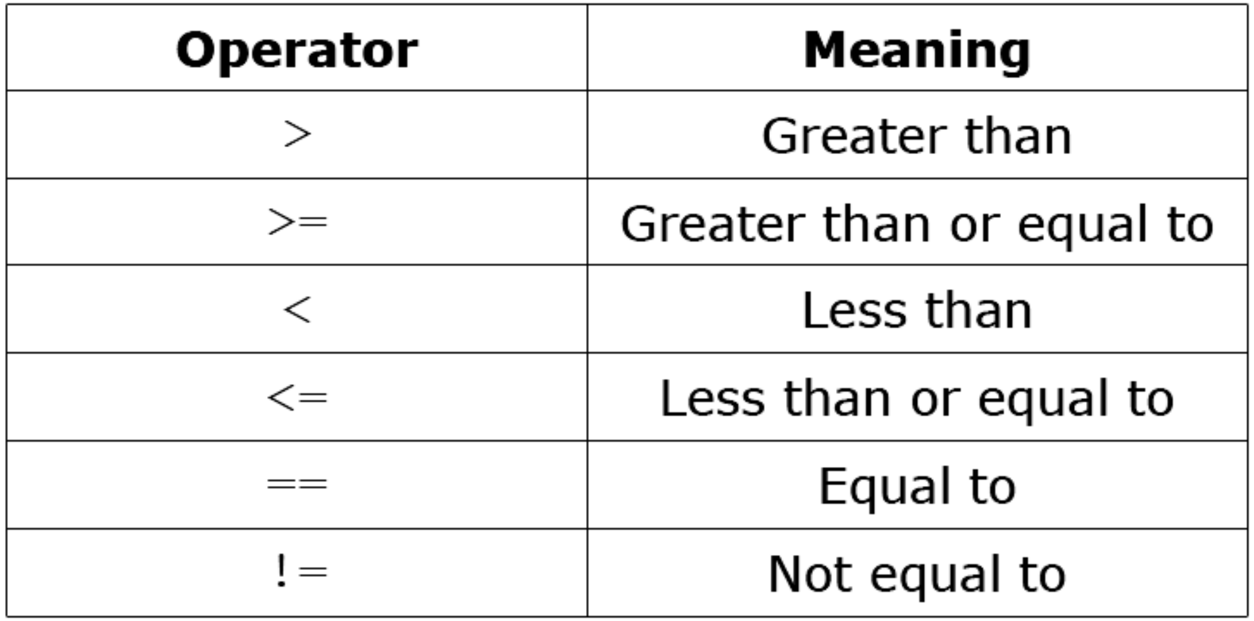
While & do-while

```
while(condition)
{
        statement1
        statement2
        …
}
do
{
        statement1
        statement2
        …
}
while(condition);
```

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

int main() {
    int x = 0;

    while(x < 10)
            x = x + 1;

    cout << “x is “ << x << “\n”;

    return 0;
}
```

Control the iteration with a compound condition

- while ((counter <= 10) && (!done)) { ...
- Boolean expression: use 2 operators: relational and logical.
- Relational operators => simple Boolean expression
  - counter <= 10

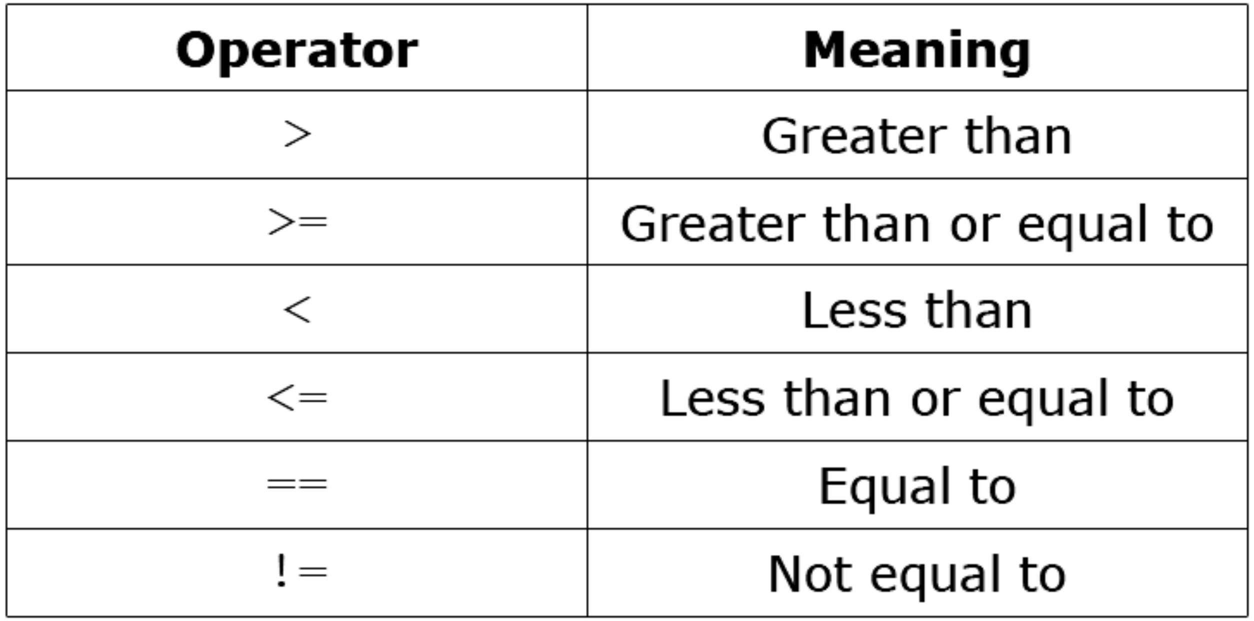


Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

Relational operators

Control the iteration with a compound condition

- while ((counter <= 10) && (!done)) { ...
- Boolean expression: use 2 operators: relational and logical.
- Relational operators => simple Boolean expression
  - counter <= 10
  - !done
  - (counter <= 10) && (!done))

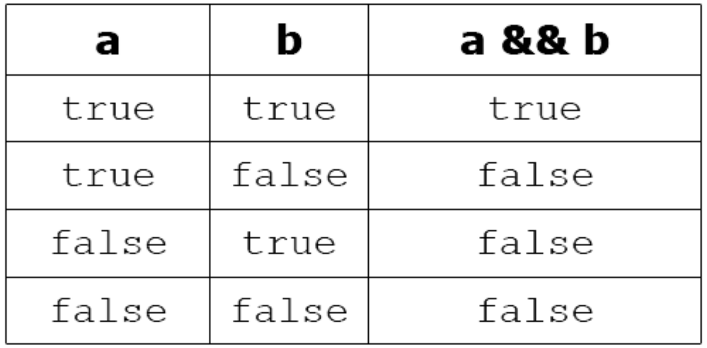


Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

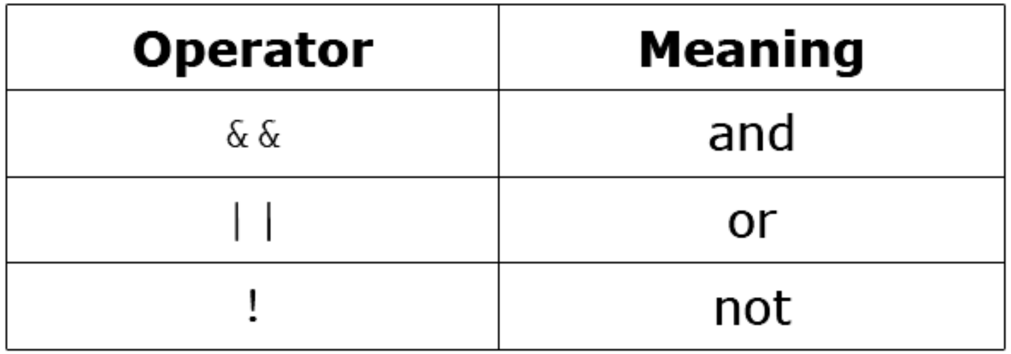
Relational operators

Control the iteration with a compound condition

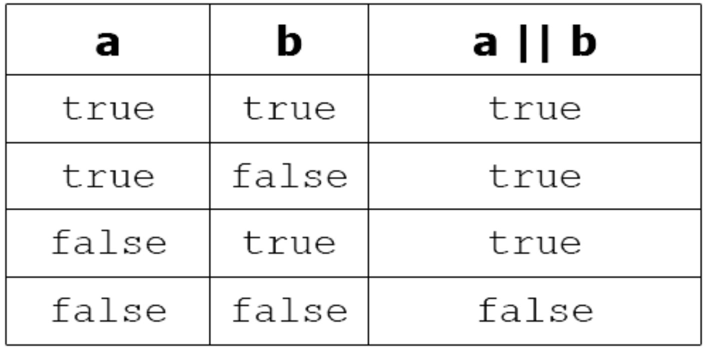
- while ((counter <= 10) && (!done)) { ...
- Boolean expression: use 2 operators: relational and logical.
  - counter <= 10
- Logical operators combine relational expressions => more complicated boolean expressions
  - !done
  - (counter <= 10) && (!done))



a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false



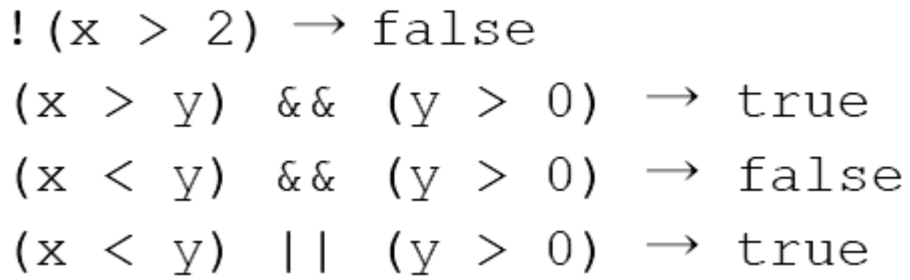
Operator	Meaning
&&	and
||	or
!	not



a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Boolean expressions

- Assume x=6 & y=2:



```
!(x > 2) → false
(x > y) && (y > 0) → true
(x < y) && (y > 0) → false
(x < y) || (y > 0) → true
```

- A quirk of C++:
  - false ⇔ 0
  - true ⇔ !0, i.e. “hello!” is true, 2 is true.

While in Python

```
count = 0
while count < 5:
    print(count)
    count += 1 # This is the same as count = count + 1
```

```
count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break
```

For in Python

- for iterating_var in sequence:

  statements(s)

```
primes = [2, 3, 5, 7] # define a list
for prime in primes:
    print(prime)
```

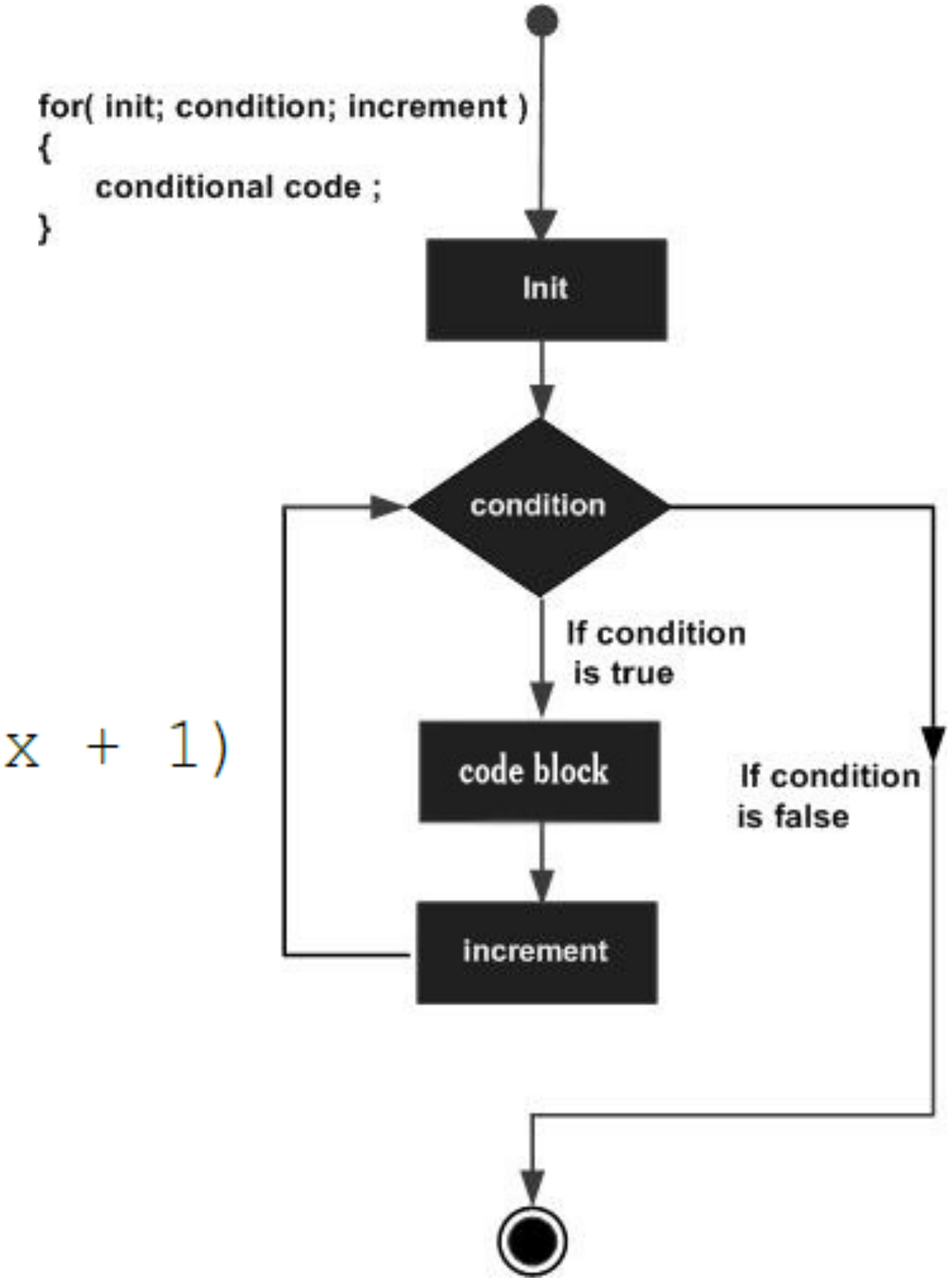
- iterate over a sequence of numbers, which can be used as Sequence Index 序列索引:

```
# Prints out the numbers 0,1,2,3,4
for x in range(5):
    print(x)
```

```
fruits = ['banana', 'apple', 'mango']
for index in range(len(fruits)):
    print('Current fruit :', fruits[index])
```

For loops

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

int main() {

    for(int x = 0; x < 10; x = x + 1)
            cout << x << "\n”;

    return 0;
}
```

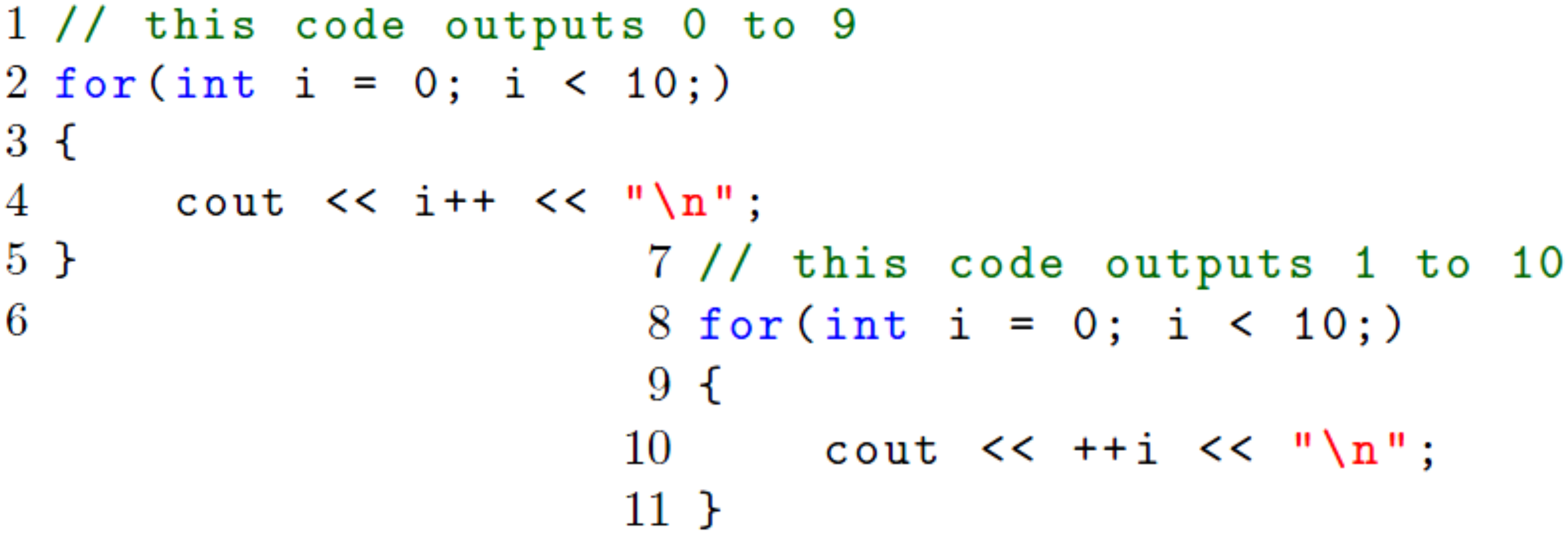


```
for( init; condition; increment )
{
    conditional code ;
}
```

Increment, Decrement operators (++, --)

a++ & ++a are shorthand for a=a+1:

- ++a will increment a and then return the value (so it will return one greater than the original value)
- a++ will return the current value and then increment

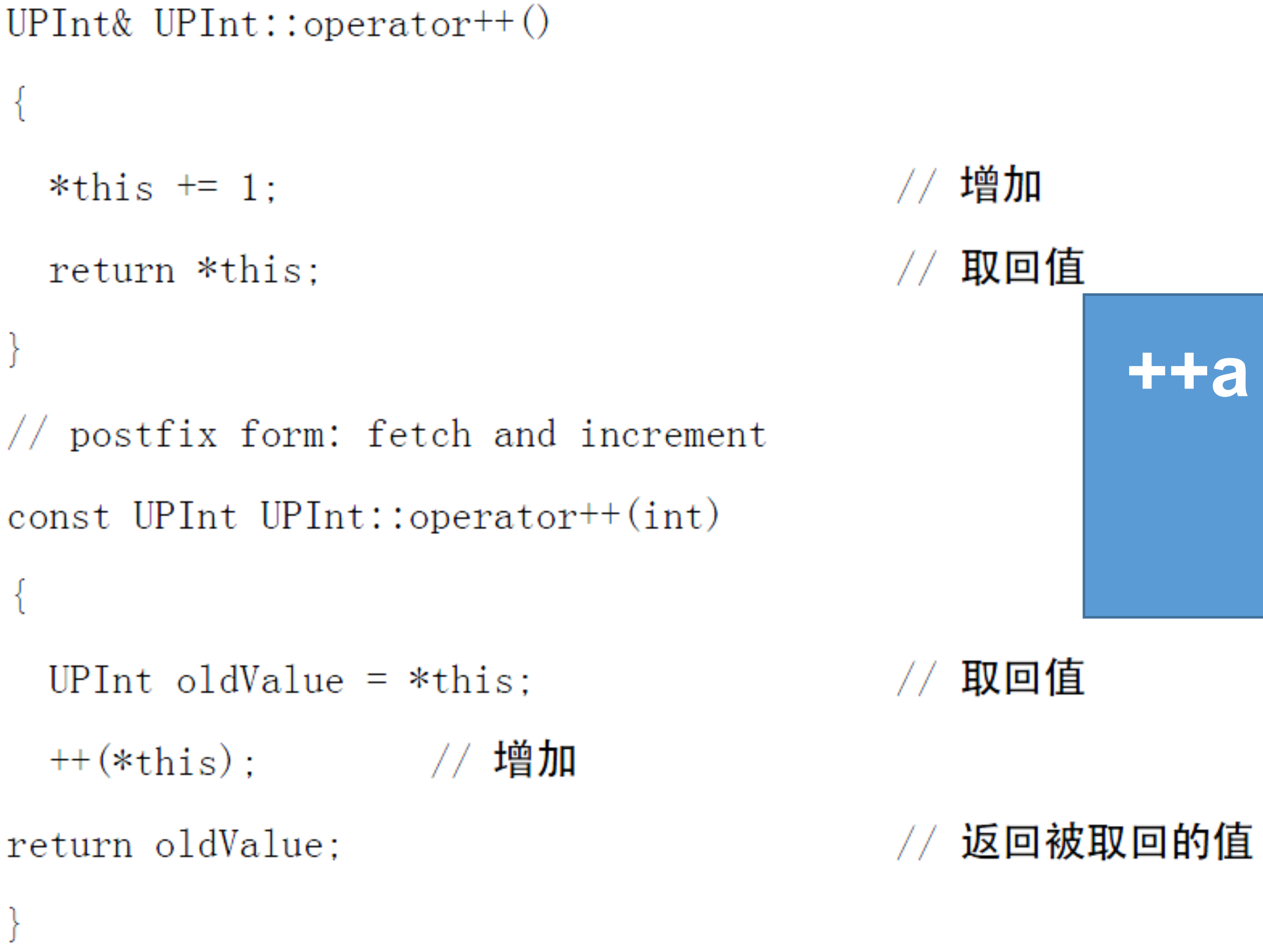


```
// this code outputs 0 to 9
for(int i = 0; i < 10;)
{
    cout << i++ << "\n";
}

```

```
// this code outputs 1 to 10
for(int i = 0; i < 10;)
{
    cout << ++i << "\n";
}
```

Prefix (++a) vs Postfix (a++) Increment operators

```
class UPInt { // "unlimited precision int"
public:
  UPInt& operator++();                    // ++ prefix
  const UPInt operator++(int);     // ++ postfix
};
```



```
UPInt& UPInt::operator++()
{
  *this += 1;                                  // 增加
  return *this;                                // 取回值
}
// postfix form: fetch and increment
const UPInt UPInt::operator++(int)
{
  UPInt oldValue = *this;                      // 取回值
  ++(*this);            // 增加
return oldValue;                               // 返回被取回的值
}
```

++a is a little more effective than a++

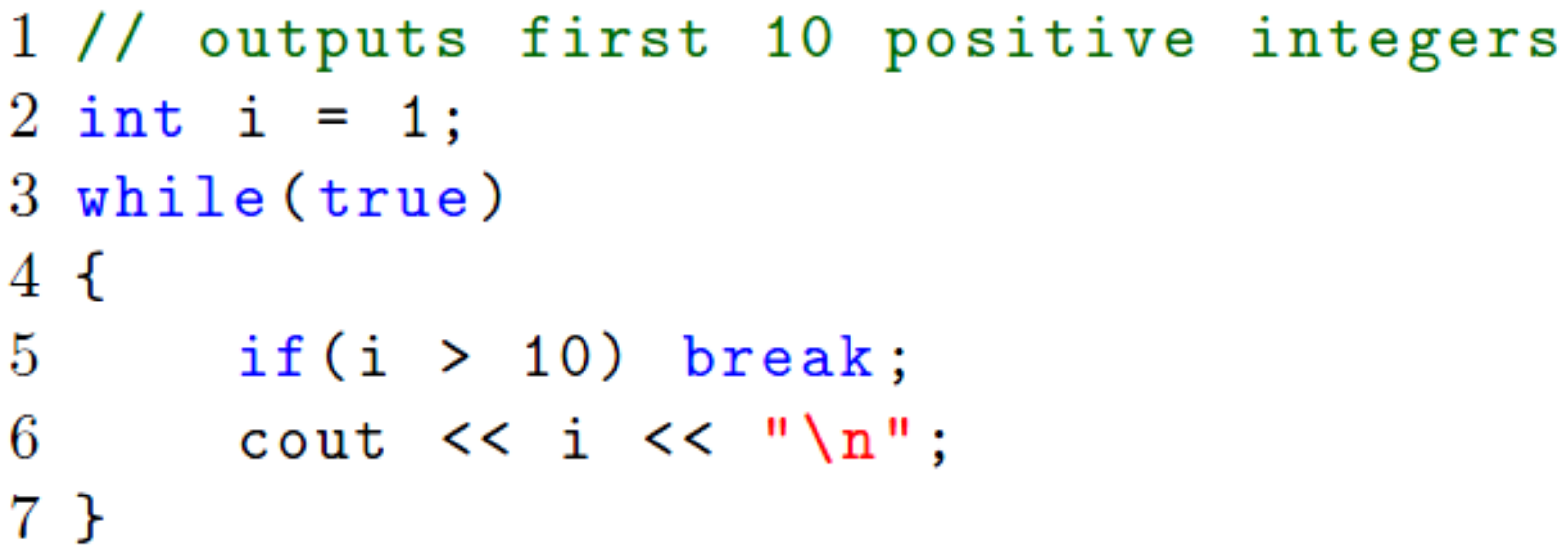
-- More effective c++, M6

Nested conditionals

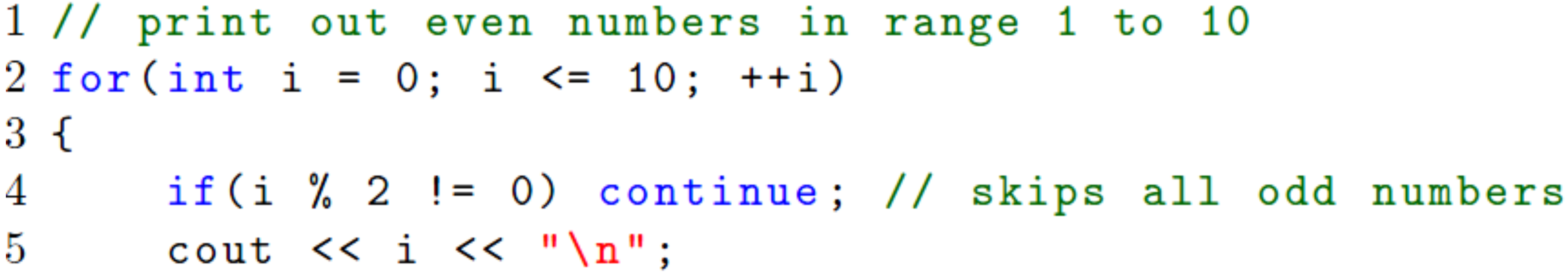
```
#include <iostream>
using namespace std;

int main() {
   int x = 6;
   int y = 0;

   if(x > y) {
         cout << “x is greater than y\n”;
         if(x == 6)
               cout << “x is equal to 6\n”;
         else
               cout << “x is not equalt to 6\n”;
   } else
         cout << “x is not greater than y\n”;

   return 0;
}
```

Break & continue

- **break**: exit a for loop or a while loop
- **continue:** skip the current block, and return to the "for" or "while" statement.



```
// outputs first 10 positive integers
int i = 1;
while(true)
{
    if(i > 10) break;
    cout << i << "\n";
}
```



```
// print out even numbers in range 1 to 10
for(int i = 0; i <= 10; ++i)
{
    if(i % 2 != 0) continue; // skips all odd numbers
    cout << i << "\n";
```

For-each loops

For-each loops

- C++11 introduces a new type of loop called a **for-each** loop

```
for (element_declaration : array)
    statement;
```

```
int main()
{
   int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 };
   for (auto number : fibonacci) // type is auto, so number has its type deduced from the fibonacci array
      std::cout << number << ' ';


   return 0;
}
```

For-each loops

```
int array[5] = { 9, 7, 5, 3, 1 };
for (auto &element: array) // The ampersand makes element a reference to the actual array element, preventing a copy from being made
{
    std::cout << element << ' ';
}
```

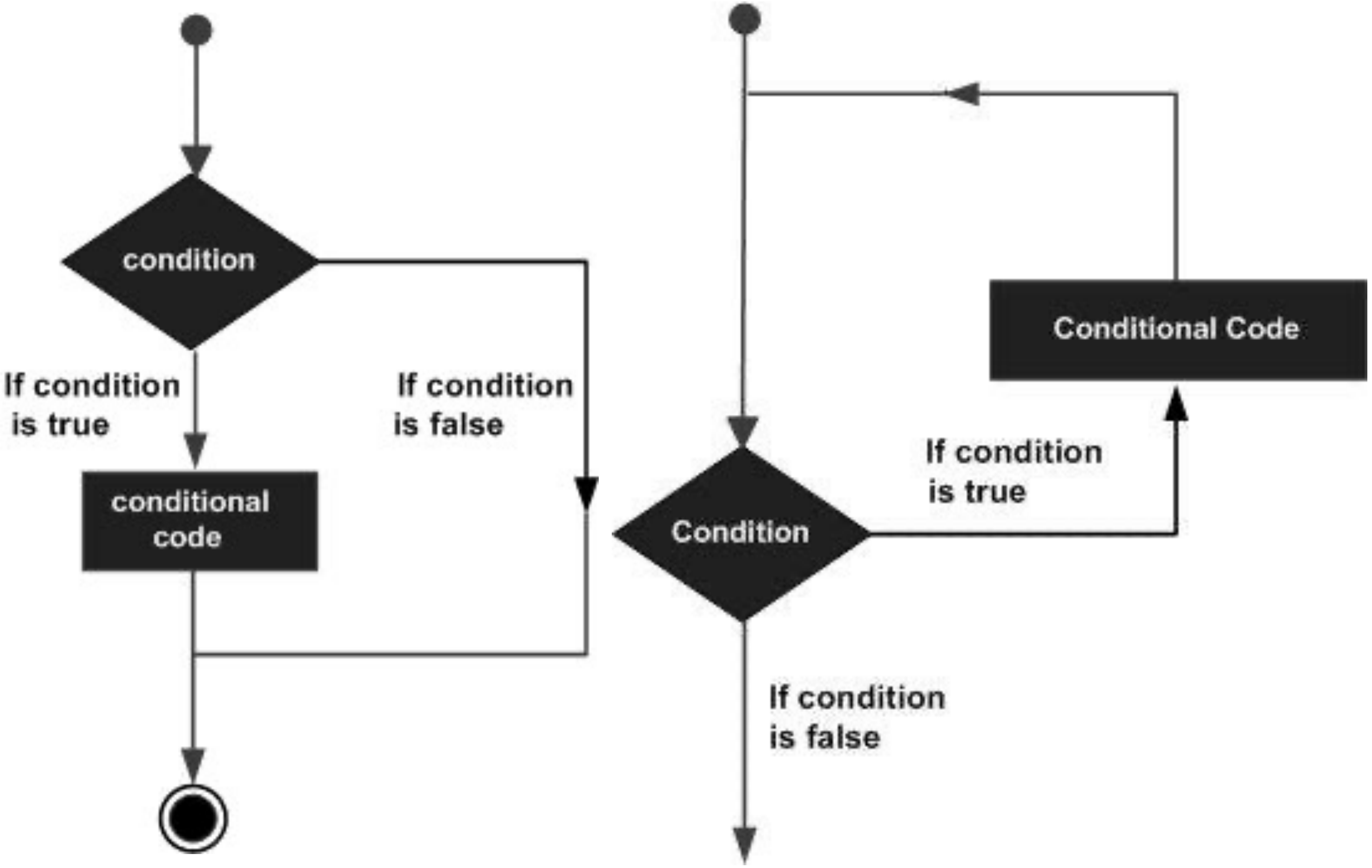
Rule: Use references or const references for your element declaration in for-each loops for performance reasons.

For-each doesn’t work with pointers to an array

```
int sumArray(int array[]){
    int sum = 0;
    for (const auto &number : array) // compile error, the size of array isn't known
        sum += number;
    return sum;
}

int main()
{
     int array[5] = { 9, 7, 5, 3, 1 };
     std::cout << sumArray(array);
     return 0;
}
```

flow-of-control statements

1. Conditionals:
   - If, else if / elif
   - Switch-case
2. Loops
   - While
   - For



- Bool expression
- Relational & Logical operators
- break & continue

Timekeeping: C++ vs Python

```
import time
sum = 0;add = 1
start = time.time()
iterations = 1000*1000*100
for i in range(iterations):
    sum += add
    add /= 2.0

end = time.time()
print("Python for Time measured: {}
seconds".format(end - start))
```

13 seconds

Timekeeping: C++ vs Python

13 vs. 0.6 seconds!! @ macbook, 2.9GHz i7, 16GB mem.

```
#include <stdio.h>
#include <chrono>
double sum(0), add(1);

auto begin = std::chrono::high_resolution_clock::now();
int iterations = 1000*1000*100;
for (int i=0; i<iterations; i++) {
sum += add; add /= 2.0;
}
auto end = std::chrono::high_resolution_clock::now();
auto elapsed =
std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin);
printf(“Result: %.20f\n”, sum); //2
printf("C++ Time measured: %.3f seconds.\n", elapsed.count() *
1e-9);
```

Analyzing an Algorithm / function

- Predicting the resources the algorithm requires

- Resources
    - **Computation Time**
    - Memory
    - Communication Bandwidth
    - …

Evaluating an algorithm

- Mike: My algorithm can sort $\mathbf{10^6}$ numbers in $\mathbf{3}$ seconds.
- Bill: My algorithm can sort $\mathbf{10^6}$ numbers in $\mathbf{5}$ seconds.

- Mike: I've just tested it on my new Pentium IV processor.
- Bill: I remember my result from my undergraduate studies (1985).

- Mike: My input was a random permutation of $\mathbf{1..\,10^6}$.
- Bill: My input was the sorted output, so I only needed to verify that it is sorted.

- Processing time is surely a bad measure!!!

- We need a ‘stable’ measure, **independent** of the **implementation**.

The RAM Model of Computation

RAM: Random Access Machine

1. Each simple operation (+, -, =, if, call) takes 1 step.
2. **Loops and subroutine** calls are **not simple** operations. They depend upon the size of the data and the contents of a subroutine. "Sort" is not a single step operation.
3. Each memory access takes exactly 1 step.

For a given problem instance:

- Running time of an algorithm = **#RAM** steps

- Useful abstraction => allow us to analyze algorithms in a machine-independent fashion.

Insertion Sort

```
for  i ← 2 to length[A]               n
    key ← A[i]                        n
    j ← i-1                           n
    while j>0 and A[j]>key            $\sum_{i=2}^{n} t_i$    $t_i$<i
        A[j+1] ← A[j]                 $\sum_{i=2}^{n} t_i$
        j ← j-1                       $\sum_{i=2}^{n} t_i$
    A[j+1] ← key                      n
```

$T(n)=4n+3\sum_{i=2}^{n} t_i$

Best case: linear function of n

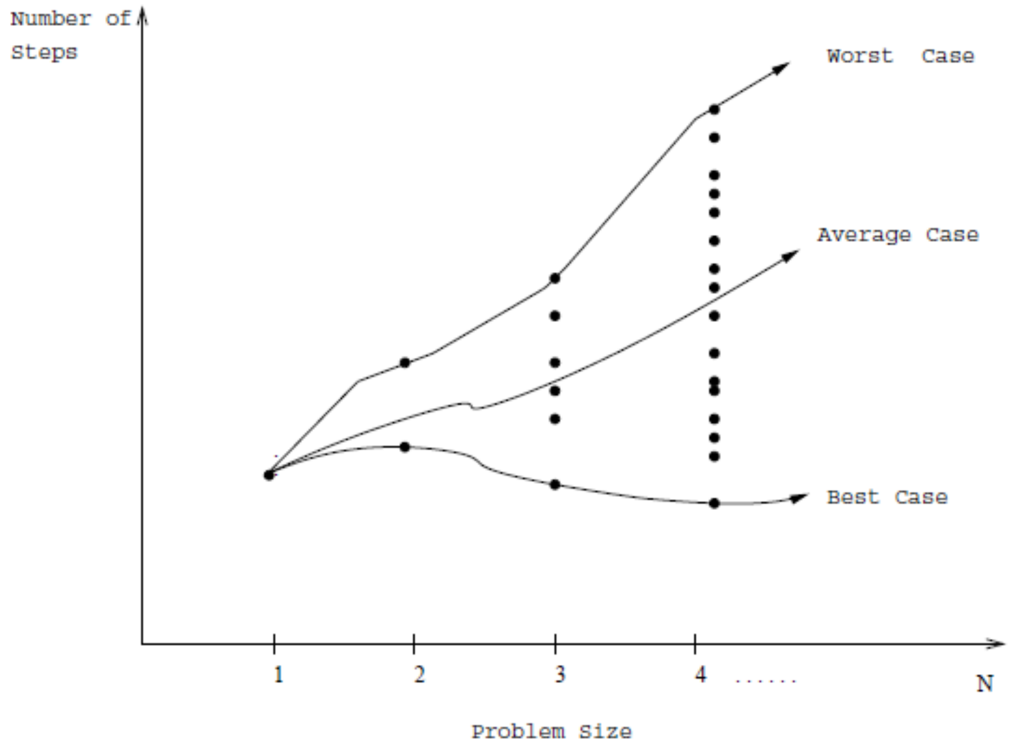
$\Rightarrow t_i$=1 & $\sum_{i=2}^{n} t_i$=n-1

$\Rightarrow$T(n)=4n+3(n-1)=7n

Worst case: quadratic function of n

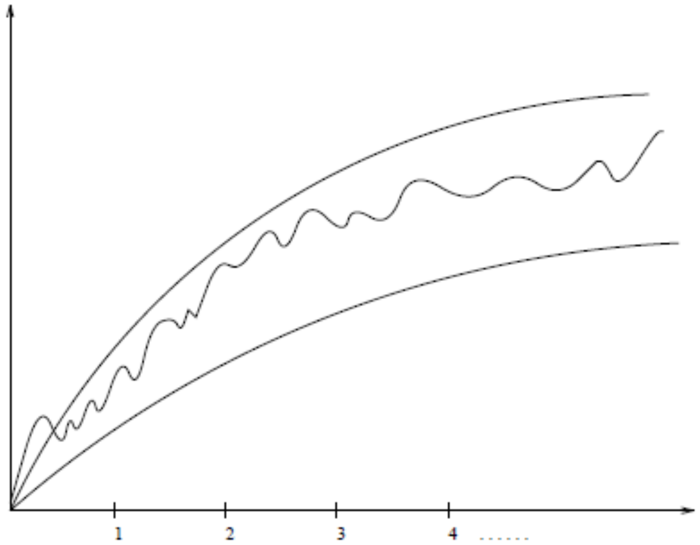
$\Rightarrow t_i$=i-1 & $\sum_{i=2}^{n} t_i$=n(n-1)/2

$\Rightarrow$T(n)=4n+3n(n-1)/2 =4n+3n^2



Exact Analysis is Hard!

- Best, worst, and average are difficult to deal with precisely because the details are very complicated:



- It easier to talk about *upper and lower bounds* of the function. **Asymptotic notation (O, Θ, Ω)** are as well as we can practically deal with complexity functions.

Order of Growth

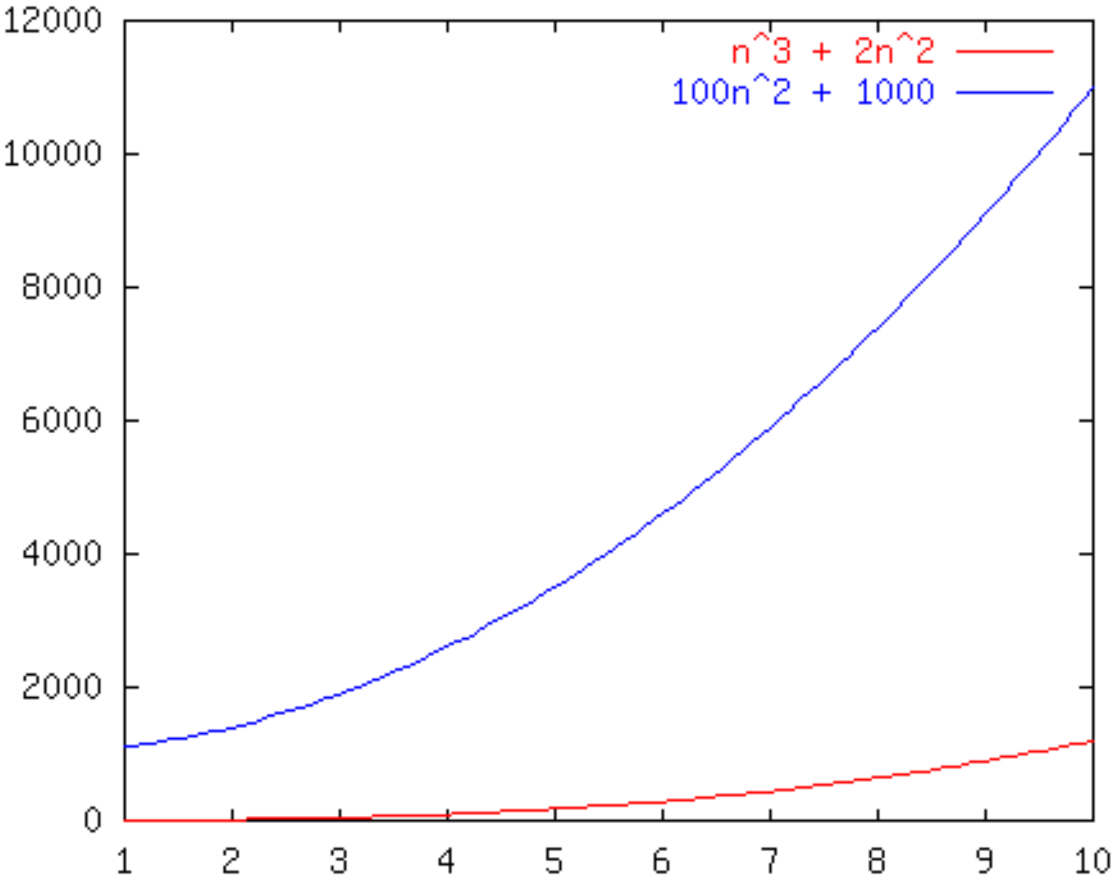
- We are interested in the type of function the running time was, not the specific function (linear, quadratic,…)

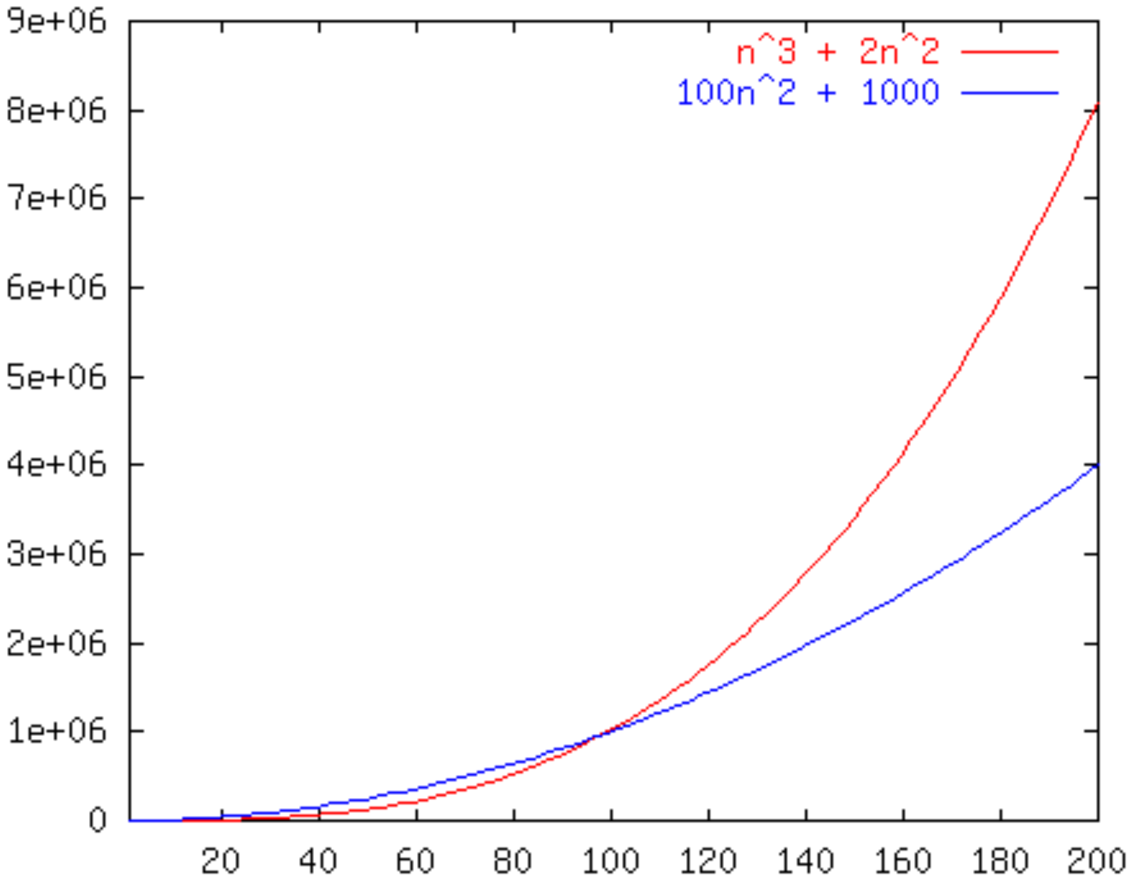
- Really interested only in the leading terms

- Mostly interested only in the Rate of Growth of the leading terms

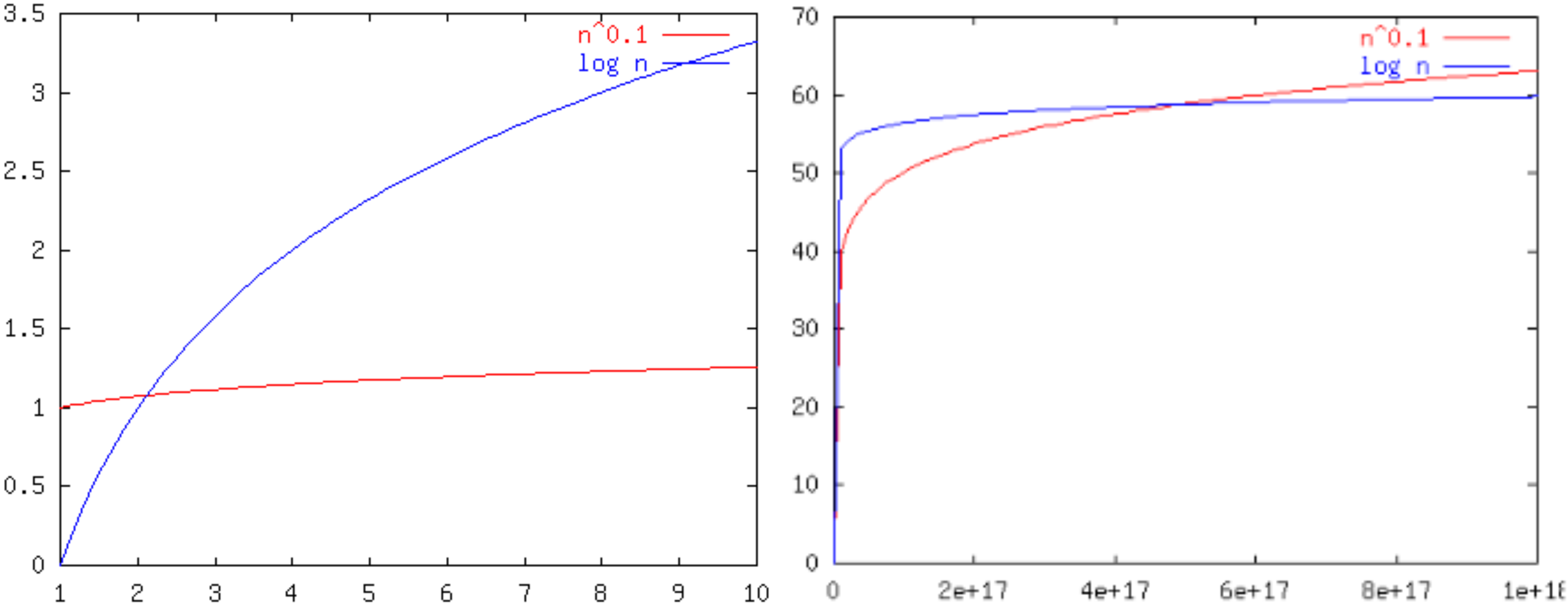
  ⇒ ignore constant coefficients

Which Function Grows Faster?

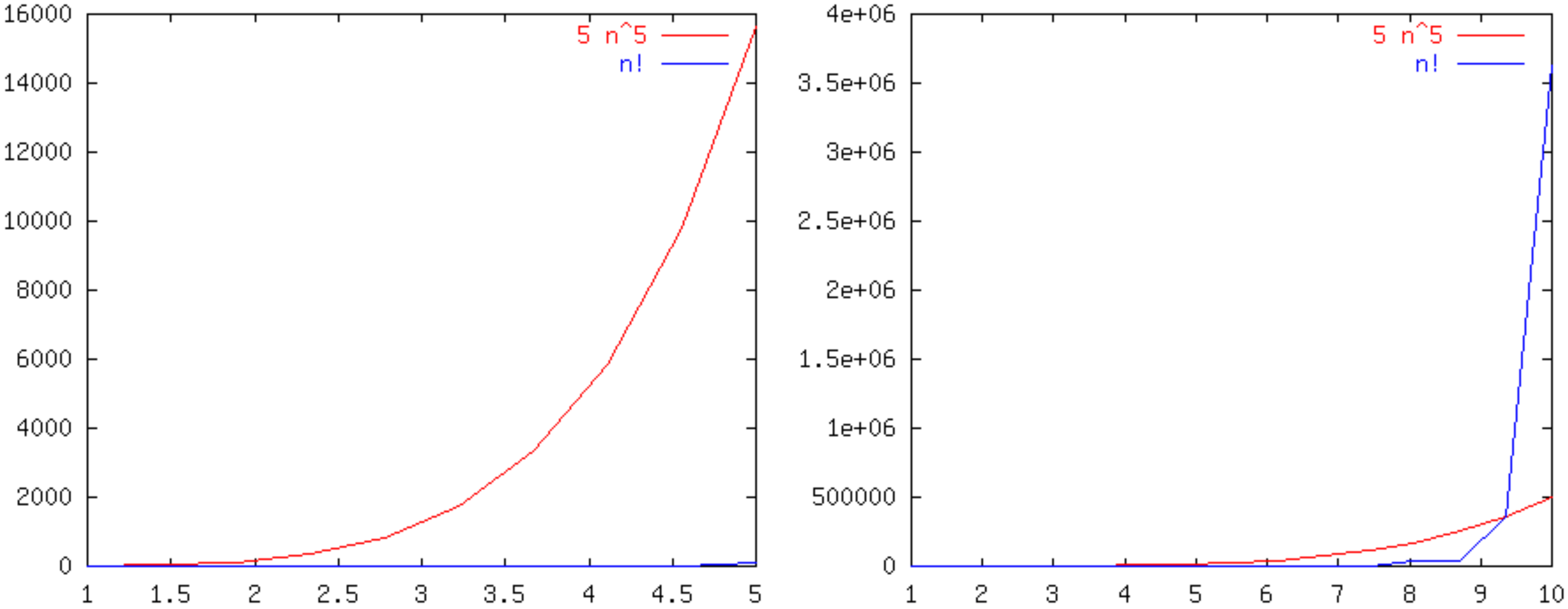




Which Function Grows Faster?



Which Function Grows Faster?



RAM cost of the function

```
import time
sum = 0;add = 1
start = time.time()
iterations = 1000*1000*100
for i in range(iterations):
    sum += add
    add /= 2.0

end = time.time()
print("Python for Time measured: {}
seconds".format(end - start))
```

n

n

n

3n in total

13 seconds

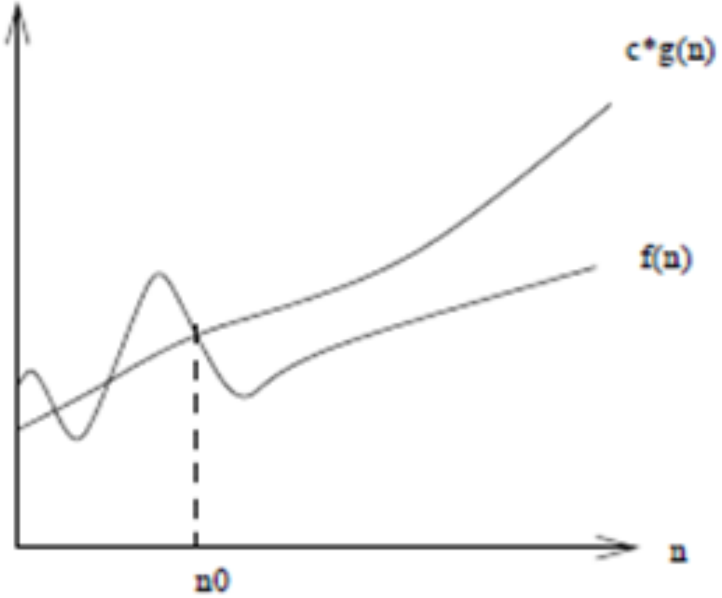
Big Oh notation
Upper Bound on Running Time

Definition: $f(n) \in O(g(n))$

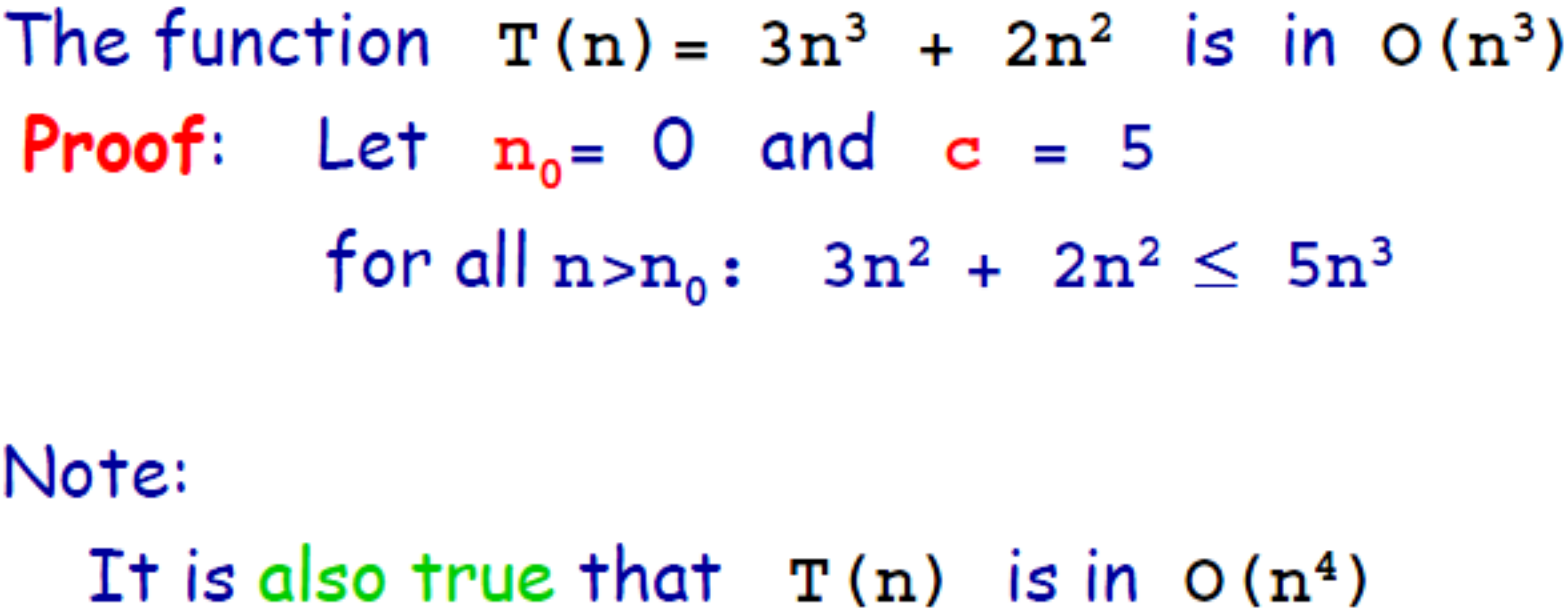
if there are $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c \cdot g(n) \quad \text{for all } n > n_0$$

Intuition: $f(n)$ is "less than" $g(n)$ when we ignore small values of n and constant multiples



Big-Oh - Example



The function $T(n) = 3n^3 + 2n^2$ is in $O(n^3)$

Proof: Let $n_0 = 0$ and $c = 5$

for all $n > n_0$: $3n^2 + 2n^2 \leq 5n^3$

Note:

It is also true that $T(n)$ is in $O(n^4)$

Complexity of the algorithm: O(n)

```python
import time
sum = 0;add = 1
start = time.time()
iterations = 1000*1000*100
for i in range(iterations):
    sum += add
    add /= 2.0

end = time.time()
print("Python for Time measured: {}
seconds".format(end - start))
```

n

n

n

3n in total => O(n)

13 seconds