

C++ DATA STRUCTURES PROGRAMMERS GUIDE LESSON 1

File:	cppdsGuideL1.doc
Date Started:	Jan 21, 2000
Last Update:	Dec 22, 2001
Status:	draft

INDUCTION, TIMING AND LOOP INVARIANTS**INDUCTION**

We need to prove that theorems are true. Proof by induction is usually used. Proof by induction is a little confusing. What they are trying to do is prove a theorem in steps. They think if the first step is true, and if they can prove subsequent steps are true then the theorem is true.

step 1 prove a base case: n = known value

This step merely confirms that the theorem is true for known values

step 2 induction hypotheses: k

The theorem is assumed to be true for all cases up to some known value k. Using this assumption then the theorem is tried to be proven correct for the next value k + 1 using the induction step.

step 3 induction step: n = k+1

Prove that the theorem is true for the next value n = k + 1.

step 4 induction conclusion

If the theorem is true for k+1 then the theorem is true for n

example 1

Prove that the series $1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$ is true using induction for $n \geq 1$

$$\boxed{\sum_{i=1}^n i} = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

step1: base case prove for n = 1 the sum must be 1

$$\boxed{\sum_{i=1}^1 i} = \frac{1(1+1)}{2} = \frac{1(2)}{2} = \frac{2}{2} = 1$$

step 2 induction hypotheses assume that the theorem is true when $n = k$

$$\sum_{i=1}^k i = \frac{k(k+1)}{2}$$

$$n = k$$

Wherever there was an n we substitute k

step 3 induction step prove that the theorem is true for the next value $(k + 1)$

(n is now equal to $k + 1$)

$$n = k + 1$$

$$\begin{aligned} \sum_{i=1}^{k+1} i &= \sum_{i=1}^k i + (k+1) = \frac{k(k+1)}{2} + (k+1) = \frac{k(k+1) + 2(k+1)}{2} \\ &= \frac{k^2 + k + 2k + 2}{2} = \frac{k^2 + 3k + 2}{2} = \frac{(k+1)(k+2)}{2} \end{aligned}$$

we have substituted $\frac{k(k+1)}{2}$ for $\sum_{i=1}^k i$

To complete the proof we must now substitute n for $k + 1$

$$\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2} = \frac{(k+1)((k+1)+1)}{2} = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

You should now be able to realize and see the proof now.

step 4 induction conclusion If the theorem is true for $n+1$ then the theorem is true for n

We substituted the next value $(k + 1)$ into the theorem now n is equal to $k + 1$. We solved the theorem equation for $(k + 1)$. From the answer we substituted n for $(k + 1)$ we got back the theorem. Thus we have concluded if the theorem can be proven true for the next value $(k + 1)$ the theorem is true for n .

Try for various n : $n = 1$ $1 = \frac{n^2 + n}{2} + \frac{1^2 + 1}{20} = 1$

$$n = 3 \quad 1 + 2 + 3 = \frac{n^2 + n}{2} + \frac{3^2 + 3}{2} = \frac{9 + 3}{2} = \frac{12}{2} = 6$$

using mathematical notation

We can use mathematical notation to represent the above proof.

Let S represent our Theorem. (S represents summation)

We want to prove for all $n \geq 1$ that S(n) is true

Here is our proof:

Base Case:

Prove that S(1) is true

Induction Hypotheses:

For any value k where $n = k$ then assume S(k) is true

Induction Step:

Prove that the next value (k + 1) is true. Let $n = (k + 1)$ Prove S(k+1) is true.

Induction Conclusion:

if (k + 1) is true then assume for all $n \geq 1$ then S(n) is true

The mathematical approach using notation gives a proof for any theorem.

weak induction

The above inductive step uses weak induction. Weak induction means we just test the base case and the next case if they are both true then we assume our theorem is true. If the base case is true $n = 1$ and if the next value is true $n = (k + 1)$ then the Theorem is true for all n.

strong induction

With strong induction you use the induction hypotheses to assume every value from the base case to k is true if $1 \leq k \leq n$ is true then the theorem is true for all n. With strong induction you are testing **every case k**.

LESSON 1 EXERCISE 1

Prove that the series

$$\boxed{\sum_{i=1}^n i^2} = 1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{2} \text{ is true using induction for } n \geq 1$$

PROOF BY CONTRADICTION

Proof by contradiction works by assuming a theorem is false and then prove some known property of the theorem false. (an opposite view point) If a false theorem is false then it got to be true ! You see you contradict your self. An easy example to understand is "There is no largest integer"

step 1 : assume the theorem is false

Assume the largest integer is X

step 2: prove the theorem false

Let $Y = X + 1$.

Now $Y > X$ therefore X cannot be the largest integer, because now Y is the largest integer. There fore the theorem is true. There cannot be any largest integer. You can always find another larger number.

RECURSION AND INDUCTION

A function that is defined in terms of itself is known as recursion. In the following equation **f(x)** is on the **left** side of the equation and **f(x-1)** is on the **right** side of the equation.

$$f(x) = f(x - 1) + x$$

Start with $x = 0$:

$$f(0) = 0$$

$$f(1) = f(0) + 1 = 1$$

$$f(2) = f(1) + 2 = 1 + 2 = 3$$

$$f(3) = f(2) + 3 = 3 + 3 = 6$$

$$f(4) = f(3) + 4 = 6 + 4 = 10$$

etc.

The next value of $f(x)$ is dependent on previous values of $f(x)$

The equation uses the results from previous computations. This is called recursion.

rules of recursion

- (1) base case: solved without recursion (recursion **stops** at the base case)
- (2) recursive cases: each recursive call leads to the base case

proof of recursion using induction for $f(x) = f(x - 1) + x$

prove: $f(n) = f(n - 1) + n$ correct for $n \geq 0$

The proof of induction is identical to the algorithm description

base case: when $n = 0$

$$f(0) = f(-1) + 0 = 0 \quad (\text{f}(-1) \text{ is assumed } 0)$$

Induction Hypotheses:

For any value k where $n = k$ then assume $f(k)$ is true

Induction Step:

Prove that the next value $(k + 1)$ is true.

Let $n = (k + 1)$ Prove $f(k+1)$ is true.

$$f(k + 1) = f((k+1) - 1) + (k + 1) = f(k) + (k + 1)$$

substituting $n = (k + 1)$ for both sides

$$f(n) = f(n - 1) + n$$

Induction Conclusion:

if $(k + 1)$ is true then assume for all $n \geq 1$ then $f(n)$ is true

recursive computer program

A recursive mathematical function is easily converted into a program, you just use the equation! We use the base case when $n = 0$. For each recursive call we decrement n each value of n is stored separately. When the recursion is finished all the separate stored values of n are added up together.

```

/* f(n) = f(n - 1) + n */
int f(int n)
{
    if(n == 0)                /* use the base case when n = 0    f(0) = 0 */
        return 0;
    else
        return f(n-1) + n    /* for each recursive call n decrements by 1 */
}

```

call	n	return value
1	3	6
2	2	3
3	1	1
4	0	0

For every function call the each value of n is stored. When the base case is reached all the stored values of n are added to the return value of the function from bottom to top

example using n = 3

For each equation call the result value is stored separately.

$$\begin{array}{ll} f(n) = f(n-1) + n & f(3) = f(2) + 3 \\ f(n-1) = f(n-2) + n-1 & f(2) = f(1) + 2 \\ f(n-2) = f(n-3) + n-2 & f(1) = f(0) + 1 \\ f(n-3) = 0; & f(0) = 0 \end{array}$$

add up results from
each recursive call

when the base case is reached the stored return values are added up:

$$\begin{array}{l} f(3) = 0 + (n-2) + (n-1) + n \\ f(3) = 0 + 1 + 2 + 3 = 6 \end{array}$$

For any n we can start with the formula

$$f(n) = f(n-1) + n$$

and repeatedly substitute for the next call

$$\begin{array}{l} f(n) = f(n-1) + n \\ f(n-1) = f(n-2) + (n-1) + n \\ f(n-2) = f(n-3) + (n-2) + (n-1) + n \\ f(n-3) = f(n-4) + (n-3) + (n-2) + (n-1) + n \\ \text{etc.} \end{array}$$

We take the upper bound and assume $f(n-4)$ is zero. The result is the **binomial series**.

$$f(n) = (n-3) + (n-2) + (n-1) + n = \sum_{i=1}^n i \quad (\text{binomial series})$$

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$

LESSON 1 EXERCISE 2

Write the equations and the recursive code for the Fibonacci sequence:

$$\mathbf{Fib(n) = Fib(n-1) + Fib(n-2)}$$

There are two base cases: $Fib(0) = Fib(1) = 1$

TIMING

We need to estimate how long a program will run. Every time we run the program we are going to have different input values and the running time will vary. Since the running time will vary, we need to calculate the worst case running time. The worst case running time represents the maximum running time possible for all input values. We call the worst case timing "**big Oh**" written $O(n)$. The n represents the worst case execution time units. How do we calculate "**Big Oh**" ???

We first must know how many time units each kind of programming statement will take:

1. simple programming statement: $O(1)$

```
k++;
```

Simple programming statements are considered 1 time unit

2. linear for loops: $O(n)$

```
k=0;
for(i=0; i<n; i++)
    k++
```

For loops are considered n time units because they will repeat a programming statement n times. The term linear means the **for** loop increments or decrements by 1

3. non linear loops: $O(\log n)$

```
k=0;
for(i=n; i>0; i=i/2)
    k++;
```

```
k=0;
for(i=0; i<n; i=i*2)
    k++;
```

For every iteration of the loop counter i will divide by 2. If i starts is at 16 then then successive i 's would be 16, 8, 4, 2, 1. The final value of k would be 4. Non linear loops are **logarithmic**. The timing here is definitely $\log_2 n$ because $2^4 = 16$. Can also works for multiplication.

4. nested for loops $O(n^2)$: $O(n) * O(n) = O(n^2)$

```
k=0;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        k++
```

Nested for loops are considered n^2 time units because they represent a loop executing inside another loop. The outer loop will execute n times. The inner loop will execute n times for each iteration of the outer loop. The number of programming statements executed will be $n * n$.

5. sequential for loops: $O(n)$

```
k=0;
for(i=0; i<n; i++)
    k++;
```

```
k=0;
for(j=0; j<n; j++)
    k++;
```

Sequential for loops are not related and loop independently of each other. The first loop will execute n times. The second loop will execute n times after the first loop finished executing. The worst case timing will be:

$$O(n) + O(n) = 2 * O(n) = O(n)$$

We drop the constant because constants represent 1 time unit. The worst case timing is $O(n)$.

6. loops with non-linear inner loop: $O(n \log n)$

```
k=0;
for(i=0; i<n; i++)
    for(j=i; j>0; j=j/2)
        k++;
```

The outer loop is $O(n)$ since it increments linear. The inner loop is $O(\log n)$ and is non-linear because decrements by dividing by 2. The final worst case timing is:

$$O(n) * O(\log n) = O(n \log n)$$

7. inner loop incremter initialized to outer loop incremter: $O(n^2)$

```
k=0;
for(i=0; i<n; i++)
    for(j=i; j<n; j++)
        k++;
```

In this situation we calculate the worst case timing using both loops. For every i loop and for start of the inner loop j will be $n-1$, $n-2$, $n-3$...

$$O(1) + O(2) + O(3) + O(4) + \dots$$

which is the binomial series:

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} = O(n^2)$$

i	j
0	n-0
1	n-1
2	n-2
3	n-3
4	n-4

8. power loops: $O(2^n)$

```
k = 0;
for(i=1; i<=n; i=i*2)
    for(j=1; j<=i; j++)
        k++;
```

To calculate worst case timing we need to combine the results of both loops. For every iteration of the loop counter i will multiply by 2. The values for j will be 1, 2, 4, 8, 16 and k will be the sum of these numbers 31 which is $2^n - 1$.

9. if-else statements

With an **if else** statement the worst case running time is determined by the branch with the largest running time.

```
/* O(n) */
if (x == 5)
{
    k=0;
    for(i=0; i<n; i++)
        k++;
}

/* O(n2) */
else
{
    k=0;
    for(i=0; i<n; i++)
        for(j=i; j>0; j=j/2)
            k++;
}
```

choose branch that has largest delay

The largest branch has worst case timing of $O(n^2)$

10. recursive

From our recursive function let $T(n)$ be the running time.

```
int f(int n)
{
    if(n == 0)
        return 0;
    else
        return f(n-1) + n
}
```

recursion behaves like a loop.
The base case is the termination for recursion.

For the line: **if(n == 0) return 0;** this is definitely: $T(1)$

For the line: **else return f(n-1) + n** the time would be : $T(n-1) + T(1)$

The total time will be: $T(1) + T(n-1) + T(1) = T(n-1) + 2$ which is $O(n)$

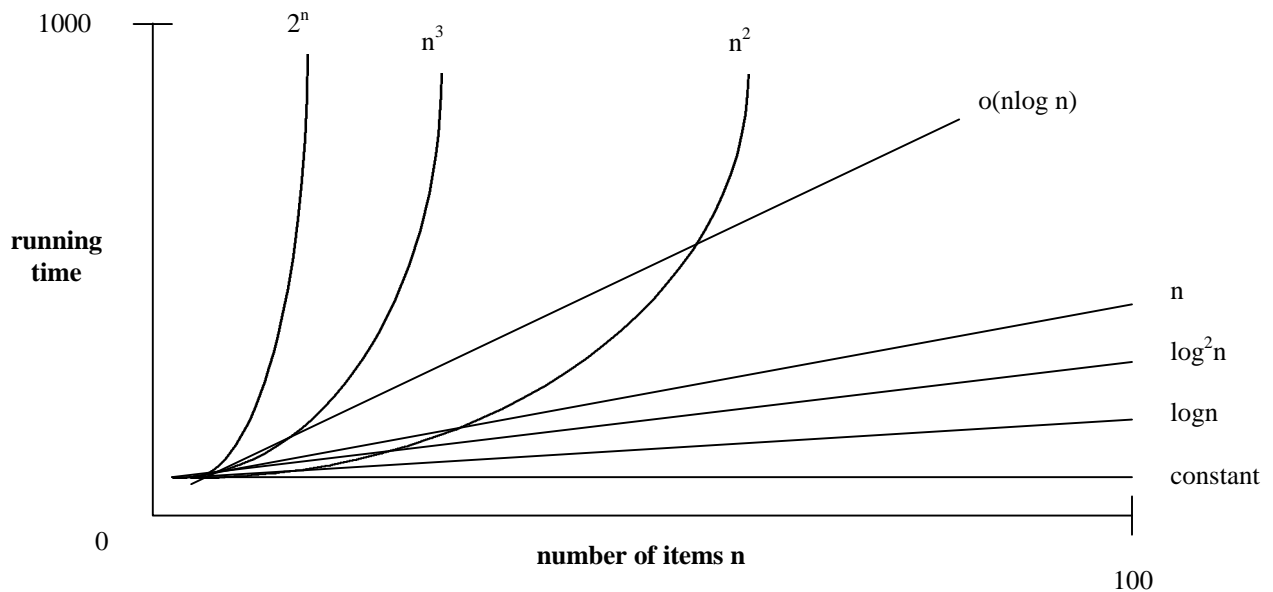
growth rates summary

The following chart lists all the possible growth rates:

c	constant
log n	logarithmic
log₂ n	log squared
n	linear
n log n	linear log squared
n²	quadratic
n³	cubic
2ⁿ	exponential

growth rate graph

The growth rates are easily visualized in the following chart. The horizontal axis **x** represent **n** and the vertical **y** axis represent running time:



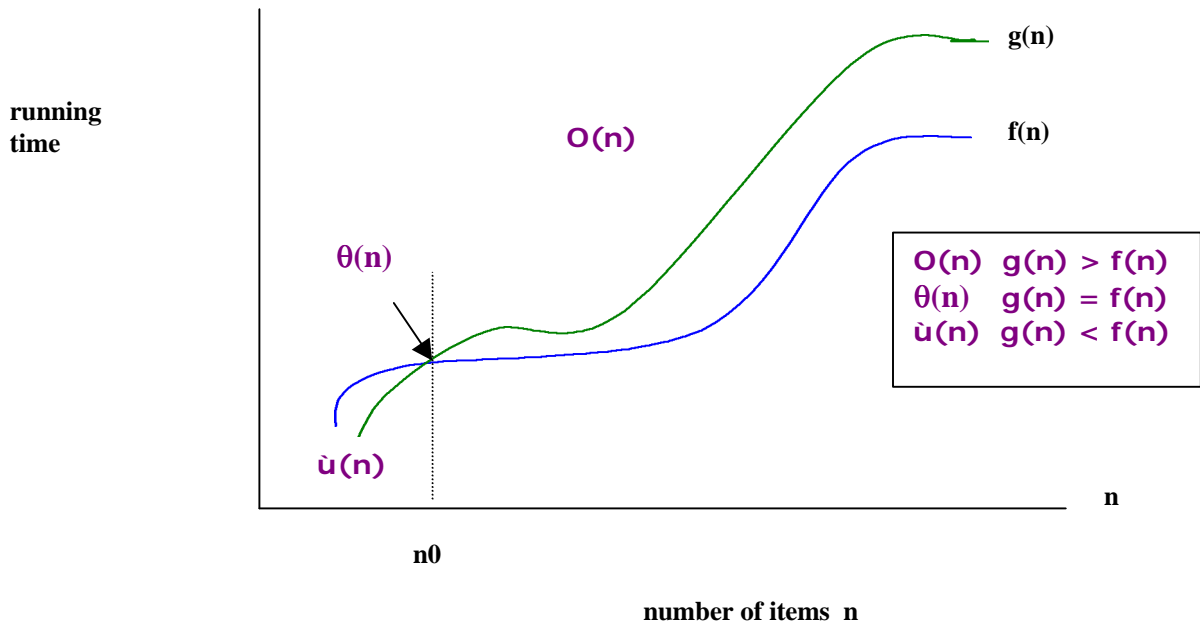
As n increases the running depends on the growth rates. For any n constant is the fastest and 2^n is the slowest. Worst case timing is also dependent on n . For small values of n we can see n^2 is much faster than $n \log n$. For larger values of n ($n \log n$) is much faster than n^2 . You must be careful in choosing your algorithms for values of n .

$$n^2 \sim n \log n$$

MATHEMATICAL THEORY

To understand what **O(n)** is all about you need to know the following mathematics. Mathematics is trying to explain what is happening.

Consider the following graph:



We have two functions $f(n)$ and $g(n)$. We are going to multiply $g(n)$ by a constant c where $c > 0$. We have arbitrary constant n_0 that is greater than or equal to 1 ($n_0 \geq 1$). "big Oh" says that there must be some constant c times $g(n)$ so that $f(n) < c * g(n)$ and n must be greater than n_0

We will call the running time $T(n)$.

if $cg(n) < f(n)$ then $T(n) = u(n)$

if $cg(n) == f(n)$ then $T(n) = \theta(n)$

if $cg(n) > f(n)$ then $T(n) = O(n)$

**O(n) is always
worst case timing**

Rules:

$T(n) + T(n) = 2 * T(n) = T(n)$
 $T(n) * T(n) = T(n^2)$

(drop constant because constants are considered 1 time unit)

LESSON 1 EXERCISE 3

What is "big Oh" ? for:

(a)

```
for(i=0;i<n*n; i++)
{
  for(j=i; j<n; j++)
    k++;
}
```

(b)

```
for(i=0; i<n; i++)
{
  for(j=i; j>0; j=j/2)
    k++;
}
```

(c)

```
for(i=0; i<n; i=i*2)
{
  for(j=i; j<n; j*j)
    k++;
}
```

SERIES

You should know the following series

$$\boxed{\sum i} = \frac{n(n+1)}{2}$$

$$\boxed{\sum i^2} = \frac{n(n+1)(2n+1)}{6}$$

$$\boxed{\sum i^k} = \frac{n^{(k+1)}}{k+1}$$

LESSON 1 EXERCISE 4

Write down the expansion for each series up to $n = 4$ and $k = 3$.

PRECONDITIONS, INVARIANTS AND POSTCONDITIONS

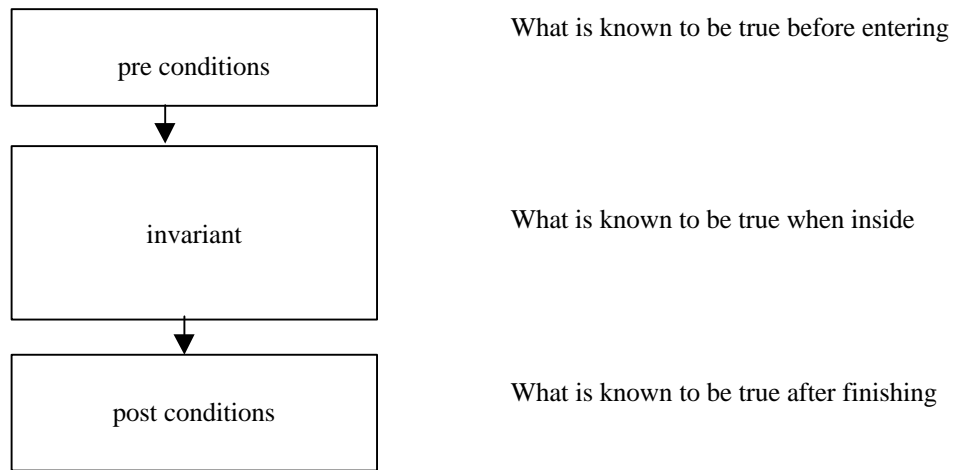
Preconditions, invariants and postconditions are used to analyze functions and loops.

The **precondition** is what is known to be true before the function or loop begins,

The **invariant** is known what is true inside the function or loop

The **post condition** is what is true after the function or loop terminates.

For the function or loop to operate correctly the **post condition** must be true after the function or loop terminates.

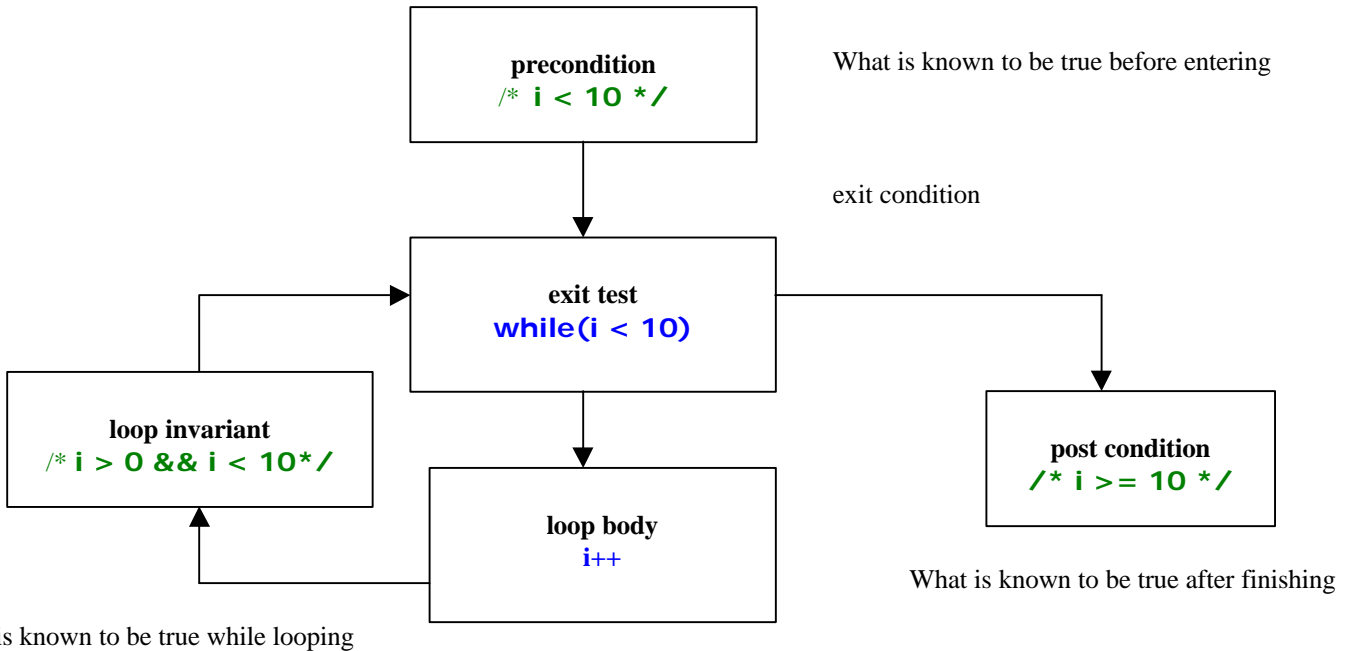


A loop has 5 sections:

section	description	example
		<code>i = 0;</code>
Precondition	what is true before entering the loop	<code>/* i = <10 */</code>
exit test	what causes the loop to exit	<code>while(i < 10)</code>
Invariant	what is true for each iteration of the loop	<code>/* i > 0 && i < 10 */</code>
body	the loop statements	<code>i++;</code>
post condition	what is true after he loop complete executiong	<code>/* i >= 10 */</code>

LOOP FLOW

It is easier to understand Preconditions, invariants and postconditions when using a loop. A loop has a precondition which is what is known to be true before we enter the loop. The loop will keep on looping while the loop invariant is true. The loop invariant must be the variables of the loop test condition that keep the loop looping. Inside the loop some operations are being performed. Every loop must have an exit condition that stops it from looping. The loop condition must test the loop invariant and exit if false. When the loop ends there are some values of variables that are true this is known as the post condition. The above preconditions, invariants and postconditions can easily be applied to functions.



Proving a loop is correct (big deal)

It's a four step process to prove that a loop is correct

step	what we have to prove
1	the precondition must be true before entering the loop
	the loop invariant must be true for the first iteration of the loop
2	the invariant is true for the next loop iteration of the loop does not exit
3	when the loop exits the post condition is true
4	the loop exits

LESSON 1 EXERCISE 4

Write a function that searches for a number in an array. Assume that the number is really in the array. The function gets the array, the length of the array and the value to search for. In your code state the **Precondition**, **exit test**, **Invariant**, **body section** and **post condition**.

USING MATHEMATICAL INDUCTION TO PROVE THAT A LOOP IS CORRECT

base case: i

Prove the invariant is true at the beginning of the loop

inductive step: $i+1$

If invariant is true at the beginning of the loop then at the beginning of the next loop it will be true again as long as the loop does not exit

proving loop termination

When the number of loop iterations reaches a certain count the loop terminates.

IMPORTANT

You should use all the material in all the lessons to do the questions and exercises. If you do not know how to do something or have to use additional books or references to do the questions or exercises, please let us know immediately. We want to have all the required information in our lessons. By letting us know we can add the required information to the lessons. The lessons are updated on a daily bases. We call our lessons the "living lessons". Please help us keep our lessons alive.

E-Mail all typos, unclear test, and additional information required to:

courses@cstutoring.com

E-Mail all attached files of your completed exercises to:

students@cstutoring.com

Order your next lesson from

www.cstutoring.com/cppds

This lesson is copyright (C) 1998-2001 by The Computer Science Tutoring Center "cstutoring"
This document is not to be copied or reproduced in any form. For use of student only