1 Statements

1.1 Structured Programming
   1.1.1 Static versus Dynamic Properties
   1.1.2 Flat Control Structure
   1.1.3 Structured Programming

1.2 Syntax
   1.2.1 Statement Lists
   1.2.2 If-then-else syntax
   1.2.3 Bracketing

1.3 Invariants
   1.3.1 Who cares?
   1.3.2 Pre- and Postconditions

Imperative Languages

Imperative programming languages are characterized by

- assignment
- control flow

1. Statements
2. Data Structures
3. Procedure Activation

1.1 Structured Programming

1.1.1 Static versus Dynamic Properties

Consider a simple sequence of statements:
\[
\begin{align*}
a & := 0; \\
b & := a + 1; \\
\text{print}(2 \times b); 
\end{align*}
\]

In this case, the dynamic structure of the computation mirrors the static structure of the code.

- **Static** properties of code are any properties that can be determined by a “single pass” examination of the code, without actually executing or simulating the execution of the code.
- **Dynamic** properties of code are properties that can only be determined by executing (or simulating the execution of) the code.

What are some examples of static code properties?

What are some examples of dynamic code properties?

1.1.2 Flat Control Structure

A basic block of code is a sequence of consecutive statements such that all of the statements in the sequence are executed once if and only if the first is executed.

- “straight-line” code
- no jumps into the middle
- no jumps out of the middle
In basic blocks, the static and dynamic structure are essentially identical.

Clearly, as we introduce conditional and looping statements, the static and dynamic structure must diverge.

– but how much?

Consider the following FORTRAN code:

```
SUBROUTINE BINSEARCH ( A , N , X , I )
DIMENSION A ( 1 )
LOW = 1
HIGH = N
1  IF ( LOW > HIGH ) GO TO 100
   MID = ( LOW + HIGH ) / 2
   IF ( A ( MID ) - X ) 2 , 3 , 4
      2  LOW = MID + 1
      GO TO 1
      3  I = MID
      GO TO 200
   4  HIGH = MID - 1
      GO TO 1
100 I = - 1
200 CONTINUE
END
```

Early FORTRAN control flow

GO TO unconditional branch to a statement identified by a numeric label

arithmetic IF IF (⟨exp⟩) s, t, u
If ⟨exp⟩ is negative, go to s. If it is zero, go to t. If positive, go to u.

FORTRAN control flow (cont.)

logical IF IF (⟨exp⟩) ⟨stmt⟩
If ⟨exp⟩ is true, then do the ⟨stmt⟩.

DO loop DO s I = j, k, m
Define a loop from here to statement s, with I initialized to j, incremented by m at the end of each loop, and loop execution ending after \( \frac{k - j}{m} \) iterations.

“Spaghetti” Code

```
SUBROUTINE BINSEARCH (A, N, X, I)
     DIMENSION A(1)
     LOW = 1
     HIGH = N
  1  IF (LOW > HIGH) GOTO 100
     ... + 1
     GO TO 1
  3  I = MID
     GOTO 200
  4  HIGH = MID - 1
     GO TO 1
 100 I = -1
 200 CONTINUE
     END
```

1.1.3 Structured Programming

Text: A program is structured if the flow of control through the program is evident from the syntactic structure of the program text.

More common: A program is structured if it is formed from a restricted set of single-entry, single-exit control flow constructs:

- sequencing
- selection
- iteration

Sequencing

Arranging statements into basic blocks:

\[
⟨\text{stmt}⟩ ::= \text{begin} \langle\text{sl}\rangle \text{end}
\]
\[
\langle\text{sl}\rangle ::= \langle\text{stmt}\rangle ; \langle\text{sl}\rangle
\]
\[
a ::= 1 ;
b ::= a - 1 ;
\text{print} 2 * b ;
\]
Selection

Selection refers to all control-flow statements used to choose one of several alternatives: if, switch/case, etc.

- Structured languages replaced FORTRAN's flat syntax with the idea of nesting statements.

Some alternate forms of if-then-else (Pascal, C, Ada):

\[
\begin{align*}
\langle \text{stmt} \rangle & := \text{if} \langle \text{exp} \rangle \text{then} \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & := \text{if} \langle \langle \text{exp} \rangle \rangle \text{then} \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle \\
\langle \text{stmt} \rangle & := \text{if} \langle \text{exp} \rangle \text{then} \langle \text{stmt-list} \rangle \\
& \quad \quad \quad \text{elseif} \langle \text{stmt-list} \rangle \\
& \quad \quad \quad \text{else} \langle \text{stmt-list} \rangle \text{endif;}
\end{align*}
\]

All are single-entry, single-exit:

Implementation of If

if \langle \text{exp} \rangle \text{then} \langle \text{stmt}_1 \rangle \text{else} \langle \text{stmt}_2 \rangle

\[
\begin{align*}
\text{translation of} \langle \text{exp} \rangle; \\
\text{if result is false goto} \ L1; \\
\text{translation of} \langle \text{stmt}_1 \rangle; \\
L1: \quad \text{translation of} \langle \text{stmt}_2 \rangle; \\
L2: \quad \ldots
\end{align*}
\]

Of course, if there is no else part, we just get

Many languages also feature a multi-way selection:
Pascal:

\[
\begin{align*}
\text{case} \ \langle \text{exp} \rangle \ \text{of} \\
\langle \text{const}_1 \rangle & : \langle \text{stmt}_1 \rangle; \\
\langle \text{const}_2 \rangle & : \langle \text{stmt}_2 \rangle; \\
\vdots & : \\
\langle \text{const}_n \rangle & : \langle \text{stmt}_n \rangle \\
\text{end}
\end{align*}
\]

All are single-entry, single-exit:

C's version is slightly different:

\[
\begin{align*}
\text{switch} \ \langle \text{exp} \rangle \ {\{ \\
\text{case} \ \langle \text{const}_1 \rangle : \langle \text{stmt}_1 \rangle; \ break; \\
\text{case} \ \langle \text{const}_2 \rangle : \langle \text{stmt}_2 \rangle; \ break; \\
\vdots & : \\
\text{case} \ \langle \text{const}_n \rangle : \langle \text{stmt}_n \rangle; \}
\end{align*}
\]

Translation of these can be interesting.

- Treat as nested if-then-elses
- Jump table
- Hash table
case \langle \text{exp} \rangle \text{ of} \\
\langle \text{const}_1 \rangle : \langle \text{stmt}_1 \rangle ; \\
\langle \text{const}_2 \rangle : \langle \text{stmt}_2 \rangle ; \\
\vdots \\
\langle \text{const}_n \rangle : \langle \text{stmt}_n \rangle \\
\text{end}

can be treated as

tmp := \langle \text{exp} \rangle ; \\
\text{if} \ tmp = \langle \text{const}_1 \rangle \text{ then} \langle \text{stmt}_1 \rangle \\
\text{else if} \ tmp = \langle \text{const}_2 \rangle \text{ then} \langle \text{stmt}_2 \rangle \\
\text{else} \\
\vdots \\
\text{else if} \ tmp = \langle \text{const}_n \rangle \text{ then} \langle \text{stmt}_n \rangle \\
\text{end can be treated as}

tmp := \langle \text{exp} \rangle ; \\
\text{if} \ tmp = \langle \text{const}_1 \rangle \text{ then} \langle \text{stmt}_1 \rangle \\
\text{else if} \ tmp = \langle \text{const}_2 \rangle \text{ then} \langle \text{stmt}_2 \rangle \\
\text{else} \\
\vdots \\
\text{else if} \ tmp = \langle \text{const}_n \rangle \text{ then} \langle \text{stmt}_n \rangle \\
\text{end can be treated as}

\text{while} \langle \text{exp} \rangle \text{ do} \langle \text{stmt} \rangle \\
\text{while} (\langle \text{exp} \rangle) \langle \text{stmt} \rangle \\
\text{while} \langle \text{exp} \rangle \text{ loop} \langle \text{stmt-list} \rangle \text{ end loop ; }

(Pascal, C, Ada)

All are single-entry, single-exit:

\text{Definite Loops}

FORTRAN had a true definite loop:

\text{DO} \langle \text{label} \rangle \ I = \langle \text{start} \rangle , \langle \text{last} \rangle , \langle \text{increment} \rangle \\
\text{Number of iterations could be computed at loop entry, because}

1. all 3 expressions were evaluated at entry
2. programs were forbidden to change the loop variable within a loop

Later languages adapted the DO loop, but relaxed the rules, making it indefinite.

\text{FOR} \ I := \langle \text{start} \rangle \ \text{TO} \ \langle \text{last} \rangle \ \text{BY} \ \langle \text{increment} \rangle \\
\text{for} \ I \ \text{in} \ \langle \text{start} \rangle \ldots \langle \text{last} \rangle \\

(Pascal, Ada)

C's is particularly flexible:

\text{for} (\langle \text{stmt}_1 \rangle ; \langle \text{exp} \rangle ; \langle \text{stmt}_2 \rangle ) \langle \text{stmt}_3 \rangle \\
\text{translates as}

\langle \text{stmt}_1 \rangle \\
\text{while} ( ! \langle \text{exp} \rangle ) \\
\{ \\
\langle \text{stmt}_3 \rangle \\
\langle \text{stmt}_2 \rangle \\
\}

\text{procedure BinSearch (}
\var A : \text{array}[1..1000] \text{ of real} ; \\
N : \text{integer} ; X : \text{real} ; \ \text{var} \ I : \text{integer} ; \}
\text{var \{Pascal version\}} \\
\text{Mid} , \text{Low} , \text{High} : \text{integer} ; \\
\text{begin} \\
\text{Low} := 1 ; \text{High} := N ; \\
\text{Mid} := (\text{Low} + \text{High}) / 2 ; \\
\text{while} (\text{Low} <= \text{High}) \\
\text{and} (A[\text{Mid}] \neq X) \text{ do begin} \\
\text{if} \ A[\text{Mid}] < X \text{ then} \\
\text{Low} := \text{Mid} + 1 \\
\text{else} \\
\text{High} := \text{Mid} - 1 ; \\
\text{Mid} := (\text{Low} + \text{High}) / 2 \\
\text{end} ; \\
\text{if} (\text{Low} <= \text{High}) \\
\text{I} := \text{Mid} \\
\text{else} \\
\text{I} := -1 ; \\
\text{end} ;
void binSearch (double *A, int N, double X, int *I)
{ /* C version */
   int mid, low, high;
   low = 0; high = N - 1;
   mid = (low + high) / 2;
   while ((low <= high)
      && (a[mid] <> X)) {
      if (a[mid] < X)
         low = mid + 1;
      else
         high = mid - 1;
         mid = (low + high) / 2;
   }
   I = (low <= high)? mid : -1;
}

procedure BinSearch (A: in array[<>] of Real;
                     N: in Integer; X: in Real;
                     I: out Integer) is
begin  // Ada version
   Low := 1; High := N;
   Mid := (Low + High) / 2;
   while (Low <= High) and (A[Mid] <> X) do begin
      if A[Mid] < X then
         Low := Mid + 1
      else
         High := Mid - 1;
         Mid := (Low + High) / 2;
      end if;
   end loop;
   if (Low <= High)
      I := Mid
   else
      I := -1;
   end if;
end BinSearch;

To see the significance of the structured programming philosophy,
compare the flows of control between these structured versions and
the older FORTRAN version.

“Spaghetti” Code

SUBROUTINE BINSEARCH (A, N, X, I)
DIMENSION A(1)
LOW = 1
HIGH = N
1 IF (LOW > HIGH) GOTO 100
MID = (LOW + HIGH) / 2
IF (A(MID) - X) 2, 3, 4
2 LOW = MID + 1
GO TO 1
3 I = MID
GOTO 200
4 HIGH = MID - 1
GO TO 1
100 I = -1
200 CONTINUE
END

“Structured” Code
begin
   Low := 1; High := N;
   Mid := (Low + High) / 2;
   while (Low <= High) and (A[Mid] <> X) do begin
      if A[Mid] < X then
         Low := Mid + 1
      else
         High := Mid - 1;
         Mid := (Low + High) / 2;
      end if;
   end loop;
   if (Low <= High)
      I := Mid
   else
      I := -1;
   end if;
end BinSearch;

1.2 Syntax
Special problems arise determining where some statements end.

- **Statement Lists**
- **If-then-else syntax**
- **Bracketing**
1.2.1 Statement Lists

Pascal:

\[
\langle \text{stmt} \rangle ::= \text{BEGIN} \langle \text{sl} \rangle \text{END} | \langle \text{if-stmt} \rangle | \ldots
\]

\[
\langle \text{if-stmt} \rangle ::= \text{IF} \langle \text{exp} \rangle \text{THEN} \langle \text{stmt} \rangle [\text{ELSE} \langle \text{stmt} \rangle]
\]

\[
\langle \text{sl} \rangle ::= \langle \text{stmt} \rangle | \langle \text{stmt} \rangle ; \langle \text{sl} \rangle
\]

This allows

**procedure** p;
**begin**
  if a > b then
  a := b;
  writeln (a);
**end**;

Arguably, it even looks better this way!

To avoid this problem, Pascal later added

\[
\langle \text{stmt} \rangle :=
\]

The empty string is an acceptable statement!

So now this is legal:

**procedure** p;
**begin**
  if a > b then
  a := b;
  writeln (a);
**end**;

How many statements in the outer statement list?

This is also legal Pascal:

**procedure** p;
**begin**
  if a > b then
  a := b;
  writeln (a);
  writeln (a);
**end**;

Adding the empty statement encourages another error, however.

**begin**
  if x >= 0 then
  write ('A');
  else
  write ('B');
**end**;

This leads to a syntax error, because the ';' puts two statements between the **then** and the **else**.

Pascal used ';' as a separator.

Most languages avoid Pascal's error-proneness by using ';' as a terminator, e.g., C:

\[
\langle \text{stmt} \rangle ::= \{ \langle \text{sl} \rangle \} | \langle \text{if-stmt} \rangle | \langle \text{exp} \rangle ;\]

\[
\langle \text{if-stmt} \rangle ::= \text{if} (\langle \text{exp} \rangle) \langle \text{stmt} \rangle [\text{else} \langle \text{stmt} \rangle]
\]

\[
\langle \text{sl} \rangle ::= \langle \text{stmt} \rangle | \langle \text{stmt} \rangle \langle \text{sl} \rangle
\]

1.2.2 If-then-else syntax

The simplest way to capture the structure of C (and Pascal) **if** statements is:

\[
\langle \text{stmt} \rangle ::= \langle \text{if-stmt} \rangle | \ldots
\]

\[
\langle \text{if-stmt} \rangle ::= \text{if} (\langle \text{exp} \rangle) \langle \text{stmt} \rangle | \text{if} (\langle \text{exp} \rangle) \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle
\]

But how can we parse

**if** (a) **if** (b) s1 **else** s2
This simple grammar:

\[
\langle \text{stmt} \rangle ::= \langle \text{if-stmt} \rangle | \ldots
\]
\[
\langle \text{if-stmt} \rangle ::= \text{if} \langle \text{exp} \rangle \text{then} \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle | \text{if} \langle \text{exp} \rangle \text{then} \langle \text{stmt} \rangle \text{endif;}
\]

turns out to be ambiguous.

Although the grammar can be rewritten to remove the ambiguity

- it’s not easy
- the resulting productions are unintuitive
- the ambiguity of the grammar may well reflect a weakness in the language design

If terminators

More modern languages resolve this problem by terminating the if with a reserved word.

\[
\langle \text{stmt} \rangle ::= \langle \text{if-stmt} \rangle | \ldots
\]
\[
\langle \text{if-stmt} \rangle ::= \text{if} \langle \text{exp} \rangle \text{then} \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle \text{endif;}
\]

1.2.3 Bracketing

The if...end if is an example of bracketing.

Bracketing helps define the nested structure of languages.

Pascal and C use a single bracketing construct (BEGIN...END and {...}).

Mismatched Brackets

What happens if a programmer leaves out a closing bracket?

```c
{ /* C version */
int mid, low, high;
low = 0; high = N - 1;
mid = (low + high) / 2;
while ((low <= high)
    && (a[mid] <> X) {
    if (a[mid] < X)
        low = mid + 1;
else
    high = mid - 1;
    mid = (low + high) / 2;
/* missing */
    I = (low <= high)? mid : -1;
}

void nextFunction ()
{

Ada adopted distinct bracketing schemes for each construct.

procedure BinSearch (\ldots) is
begin // Ada version
    Mid, Low, High: integer;
    Low := 1; High := N;
    while (Low <= High) loop
        Mid := (Low + High) / 2;
        ...
Imperative Programming – Statements

```c
exit when A[Mid] = X;
if A[Mid] < X then
    Low := Mid + 1
else
    High := Mid - 1;
end if;
end loop;
if (Low <= High)
    I := Mid
else
    I := -1;
end if;
end BinSearch;
```

This allows most bracketing errors to be caught at the end of the next bracket.

1.3 Invariants

An invariant is a property that holds every time execution reaches a designated location in the code.

Invariants are useful in
- proving algorithms correct
- designing them correctly in the first place
- documenting the details of their operation

1.3.1 Who cares?

How do you know when your code is ready to release?
- when you’re done testing it

Who cares? (cont.)

How do you know when you’re done testing it?
- when you’re bored with testing
- when you can’t think of any more good tests
- when you’ve exhausted your budget for testing
- when it’s the day before the Marketing dept. announced the product would be released
- when you’ve satisfied a scientifically chosen testing criterion
- when the code is bug-free

Testing is a sampling process, and no matter how well done, will not guarantee the code to be bug-free.
So what do you do?
- Use good testing practices, and hope for the best
- Employ statistical models of software reliability to predict when the code is “good enough”
- Prove the code correct
- Try to avoid bugs in the first place.

Editorial opinion:

Proving programs correct is most useful only where other techniques can’t be applied (e.g., security)
But understanding how a program could be proven correct is an aid to designing it correctly.

1.3.2 Pre- and Postconditions

A precondition is an invariant attached just before the start of a construct. It describes the conditions under which that construct is supposed to work correctly.

A postcondition is an invariant attached just after a construct. It describes what that construct was supposed to accomplish.

You may have seen these ideas applied to documenting subroutines:

```c
void binSearch (double *A, int N, double X, int *I)
/*
 * pre: A contains N elements, sorted into ascending order
 * post: if X is in A[0..N-1], then A[I] == X
 *       if X is not in A[0..N-1], then I < 0
 */
{...
```

These illustrate some of the cardinal rules of invariants:
- every invariant is a boolean condition
- every invariant should be checkable

I personally would recommend
Try to express every invariant in a form from which it would be
easy to write code to check it.

Invariants should describe dynamic behavior, not repeat obvious
static information.

```c
void binSearch ( double *A, int N,
    double X, int *I )
/*
 * post: if there exists j. 0<=j<N,
 *   such that A[j]==X,
 * then A[I] == X
 * else I<0
*/
{...}
```

**Statement-Level Invariants**

If we write invariants between statements, then one invariant may
serve as

- the pre-condition for the following statement and
- the postcondition for the following one

```c
mid = (low + high) / 2 ;
while ( (low <= high) && (a[mid] <> X) ) {
    /* { (low <= high) && (a[mid] <> X) 
        && (mid == (low + high) / 2 } */ 
    if ( a[mid] < X)
```

**Proof Rules for Partial Correctness**

A program is (totally) correct if, given any input that satisfies the
program’s precondition, the program eventually halts with output that
satisfies the program’s postcondition.

A program is partially correct if, given any input that satisfies the
program’s precondition and for which the program eventually halts,
the output satisfies the program’s postcondition.

For example, we might write

```
x > 0
{ y > 0} y = x + y; { y > 0}
```

**Composition (Stmt Lists)**

```
{P} S_1 {Q}, {Q} S_2 {R}
{P} S_1; S_2 {R}
```

In other words,

- if
  - when P is a precondition for statement S_1, Q is a postcondition, and
  - when Q is a precondition for statement S_2, R is a postcondition,
- then when P is a precondition for the statement sequence S_1; S_2,
  R is a postcondition.

**Selection (Conditionals)**

```
{P ∧ E} S_1 {Q}, {P ∧ ¬E} S_2 {Q}
{P} if E then S_1 else S_2 {Q}
```

**While Loops**

```
{P ∧ E} S {P}
{P} while E do S {P ∧ ¬E}
```

P is a loop invariant, an invariant condition that is true upon entry
to the loop, at the start of each iteration, and upon exit.
Finding a loop invariant is considered the key to understanding
what a loop actually does.

**Designing With Invariants**

So what does all this buy us? let’s look at the binary search code:

```c
procedure BinSearch ( 
    var A: array[1..1000] of real;
    N: integer; X: real; var I: integer );
var { Pascal version }
    Mid, Low, High: integer;
begin
    Low := 1; High := N;
```
Mid := (Low + High) / 2;
while (Low <= High)
    and (A[Mid] <> X) do begin
    if A[Mid] < X then
        Low := Mid + 1
    else
        High := Mid - 1;
    Mid := (Low + High) / 2;
end;
if (Low <= High)
    I := Mid
else
    I := -1;
end:

Are you absolutely sure that Low and High will never get “stuck” at the same values on two successive iterations? Not even when
• Low = High
• Low + 1 = High
• Low + 2 = High
• each of the above, and Low is even? odd?
• each of the above, and High is even? odd?

procedure BinSearch (  
    var A: array[1..1000] of real;
    N: integer; X: real; var I: integer);
var {Pascal version}
    Mid, Low, High: integer;
begin
    Low := 1; High := N;
    /* {X is not in A, or
     (A[j] == X for some Low<=j<=High)} */
    Mid := (Low + High) / 2;
    /* {X is not in A, or
     (A[j] == X for some Low<=j<=High))
     and (Mid = (Low + High / 2) } */
    while (Low <= High)
        and (A[Mid] <> X) do begin
    if A[Mid] < X then
        Low := Mid + 1
    else
        High := Mid - 1;
    Mid := (Low + High) / 2;
end;
    if (Low <= High)
        I := Mid
    else
        I := -1;
end:

The condition
/* {X is not in A, or
 (A[j] == X for some Low<=j<=High))
 and (Mid = (Low + High / 2) } */
is our loop invariant.
For shorthand, call it Inv.

procedure BinSearch (  
    var A: array[1..1000] of real;
    N: integer; X: real; var I: integer);
var {Pascal version}
    Mid, Low, High: integer;
begin
    Low := 1; High := N;
    /* {X is not in A, or
     (A[j] == X for some Low<=j<=High)} */
    Mid := (Low + High) / 2;
    /* {Inv} */
    while (Low <= High)
        and (A[Mid] <> X) do begin
    /* {Inv and (Low <= High)
 and A[Mid] <> X} */
    if A[Mid] < X then
Low := Mid + 1
else
High := Mid - 1;
Mid := (Low + High) / 2
end;
/* { Inv and (Low > High
or A[Mid] = X) } */
if (Low <= High)
I := Mid
else
I := -1;
end;
The post-loop invariant makes it clear that, if we exit the loop, the program will return the correct value in I:
/* { (X is not in A, or
(A[j] == X for some Low<=j<=High))
and (Mid = (Low + High / 2)
and (Low > High
or A[Mid] = X) } */
simplifies to
/* { (X is not in A and Low > High)
(Low <= High and A[Mid] = X) } */

Looking at the middle of the loop, we have

while (Low <= High)
and (A[Mid] <> X) do begin
/* { Inv and (Low <= High)
and A[Mid] <> X } */
if A[Mid] < X then
Low := Mid + 1
else
High := Mid - 1;
Mid := (Low + High) / 2
end;
Now because Low <= Mid, we have Low < Mid + 1. So we can argue that

while (Low <= High)
and (A[Mid] <> X) do begin
/* { Inv and (Low <= High)
and A[Mid] <> X
and d = High−Low } */
if A[Mid] < X then
Low := Mid + 1
/* { d > High−Low } */
else
High := Mid - 1;
Mid := (Low + High) / 2
end;

We can make a similar argument for High:

while (Low <= High)
and (A[Mid] <> X) do begin
/* { Inv and (Low <= High)
and A[Mid] <> X
and d = High−Low } */
if A[Mid] < X then
Low := Mid + 1
/* { d > High−Low } */
else
High := Mid - 1;
/* { d > High−Low } */
Mid := (Low + High) / 2
end;

and then the rule for conditionals lets us say:

while (Low <= High)
and (A[Mid] <> X) do begin
/* { Inv and (Low <= High)
and A[Mid] <> X
and d = High−Low } */
if A[Mid] < X then
Low := Mid + 1
/* { d > High−Low } */
else
High := Mid - 1;
/* { d > High−Low } */
/* { d > High−Low } */

Let d denote the distance between High and Low at the beginning of the loop.

while (Low <= High)
and (A[Mid] <> X) do begin
/* { Inv and (Low <= High)
and A[Mid] <> X
and d = High−Low } */
if A[Mid] < X then
Low := Mid + 1
/* { d > High−Low } */
Imperative Programming – Statements

Mid := (Low + High) / 2
eend;

So at the end of each loop iteration, we are guaranteed that the
distance between Low and High has actually been reduced.
This means that our loop will exit eventually.

OK, who are you trying to kid?
No, I don’t do that in detail when I design algorithms.
But I do think in those terms:

• What is the loop invariant?

• What critical quantities do I need to track to tell if things are
  working properly?

  – Should those quantities be increasing, decreasing, or staying
    the same?

Invariants and Debugging
Programmers use invariants during testing and debugging by turning
them into assertions, programming language code that checks an
invariant and issues an error message if the check fails.

The assert(e) macro in C and C++ converts to

if (!e)
  error("assertion violated in line ...

unless the compiler flag NDEBUG is set.
If that flag is set, the assertion is converted into a comment.

void binSearch (double *A, int N,
double X, int *I)
{
  /* C version */
  int mid, low, high, d;
  low = 0; high = N−1;
  mid = (low + high) / 2;
  while ((low <= high) && (a[mid] < X)) {
    assert ((d = high−low) >= 0);
    if (a[mid] < X)
      low = mid + 1;
    else
      high = mid − 1;
    assert (d > high−low);
    mid = (low + high) / 2;
  }
  I = (low <= high)? mid : −1;
}