Programm- & Systemverifikation

Assertions

Georg Weissenbacher

184.741

TU WIEN

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour
- To locate bug, find transition from correct to incorrect states

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour
- To locate bug, find transition from correct to incorrect states
- How can we know what is *incorrect* or *desired*?

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour
- To locate bug, find transition from correct to incorrect states
- How can we know what is *incorrect* or *desired*?

  Requirement documents
  (Formal) specification
  Test cases
  Documentation

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour
- To locate bug, find transition from correct to incorrect states
- How can we know what is *incorrect* or *desired*?

Requirement documents
(Formal) specification
Test cases
Documentation

} not at instruction level

What happened so far

- How bugs come into being:
  - **Fault** – cause of an error (e.g., mistake in coding)
  - **Error** – incorrect state that may lead to failure
  - **Failure** – deviation from desired behaviour
- To locate bug, find transition from correct to incorrect states
- How can we know what is *incorrect* or *desired*?

  Requirement documents
  (Formal) specification
  Test cases
  Documentation
  } not at instruction level

- Want to detect deviation when it happens!

Recall very first lecture: Assertions

ECP

PLANNING AND CODING OF PROBLEMS
FOR AN
ELECTRONIC COMPUTING INSTRUMENT

BY

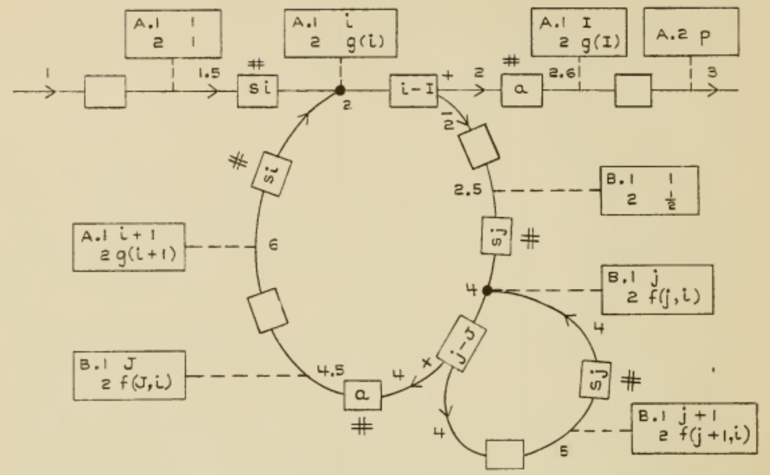
Herman H. Goldstine   John von Neumann

Report on the Mathematical and Logical aspects of an
Electronic Computing Instrument

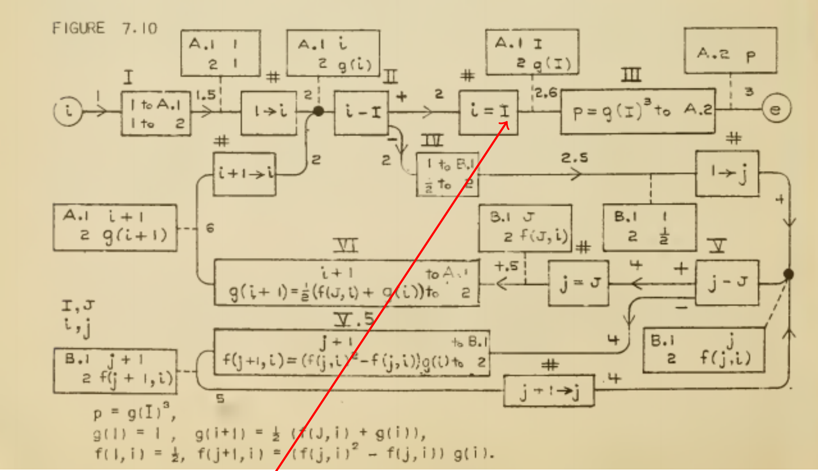
Part II, Volume I – 3

What would John von Neumann do?

FIGURE 7.9



What would John von Neumann do?



marked with #

What would John von Neumann do?

“an assertion box never requires that any specific calculations be made, it indicates only that certain relations are automatically fulfilled whenever [the program] gets to the region which it occupies”

“The contents of an assertion box are one or more relations. These may be equalities, inequalities, or any other logical expressions.”

What we know about assertions so far

- *Relations* over program variables
- Evaluate to `true` or `false`
- Have no side effect (purely theoretical construct)

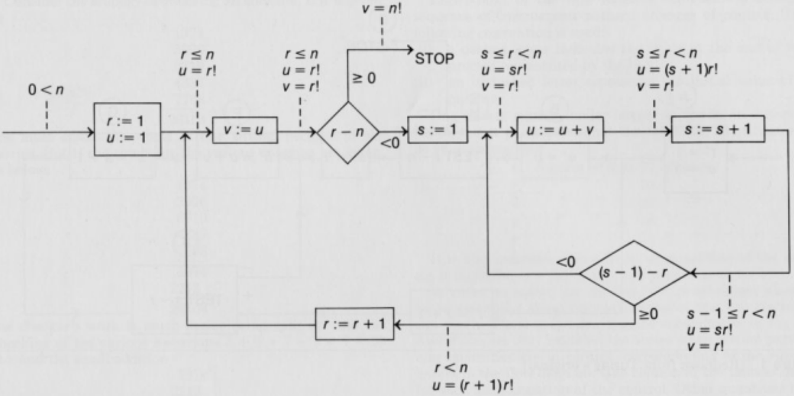
What is the purpose of assertions?

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.



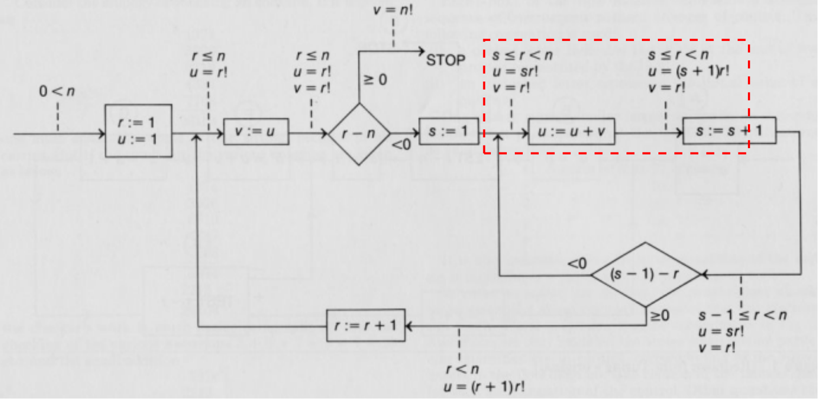
What is the purpose of assertions?

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.



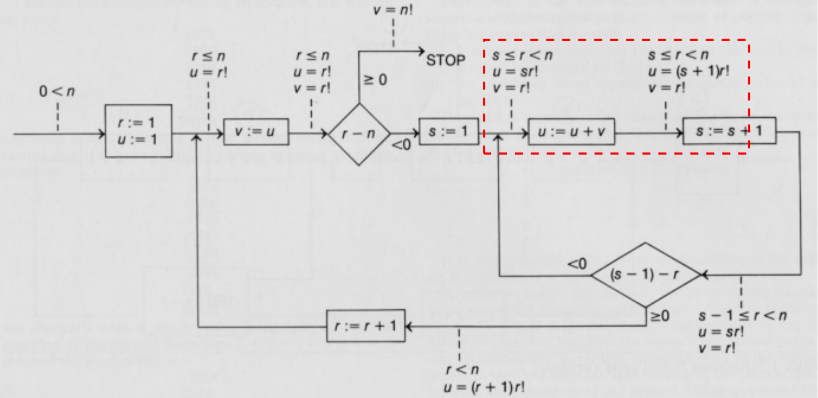
What is the purpose of assertions?

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

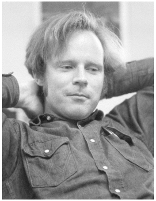


What is the purpose of assertions?

"In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows."

before	$(s \leq r < n)$ and $(u = s \cdot r!)$ and $(v = r!)$
instruction	`u := u + v`
after	$(s \leq r < n)$ and $(u = (s + 1) \cdot r!)$ and $(v = r!)$

Assigning Meanings to Programs (1967)



Robert W. Floyd

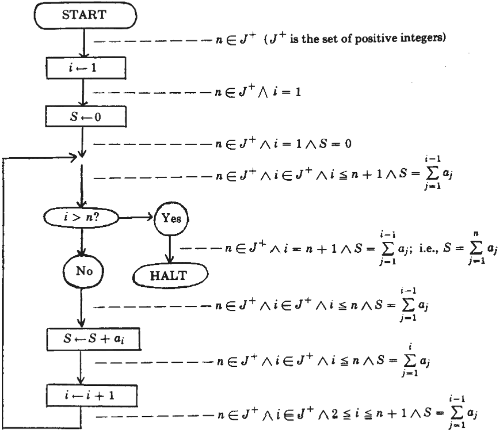
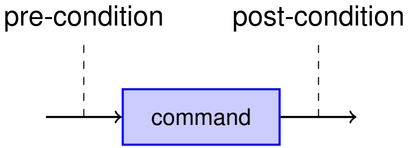


FIGURE 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j$ $(n \geq 0)$

Assertions and Program Semantics

"To prevent an interpretation from being chosen arbitrarily, a condition is imposed *on each command* of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time."



Assertions and Program Semantics

- Pre- and post-conditions mathematically rigorous
- This fixes the meaning of the instruction in between

Assertions and Program Semantics

- Pre- and post-conditions mathematically rigorous
- This fixes the meaning of the instruction in between
- Will cover this in more detail in May!
- For now, focus on more pragmatic use of assertions

Using Assertions for Debugging

```c
#include <assert.h>
#include <stdio.h>
#include <string.h>

int findlast (char* str, char elem)
{
  int i;
  for (i = strlen(str)-1; i >= 0; i--)
  {
    if (str[i] == elem)
      break;
  }
  assert (i == -1 || str[i] == elem);
  return i;
}

int main(int argc, char** argv)
{
  printf ("%d\n", findlast ("xyz", 'x'));
  printf ("%d\n", findlast ("abc", 'x'));
}
```

Using Assertions for Debugging

- We use assertion to state our intention:
  - either the result is -1
  - or the returned index points to the element in question
- Does not restrict *how* result is computed
- The assertion is not a *complete* specification
  - doesn't assert that `i` points to *last* occurrence!
  - -1 is actually always a correct answer

Assertions as Specifications

- “Light weight” specifications
- Immediate benefit for debugging
- But: do not guarantee (full) correctness of program

Assertions are Specifications

- Assertions are *partial* specifications
  - Not a complete description of program behaviour
- Simpler than *full* specification:

Assertions are Specifications

- Assertions are *partial* specifications
  - Not a complete description of program behaviour
- Simpler than *full* specification:

$$
\begin{array}{l}
(\texttt{i} = -1) \wedge (\,\nexists j \in [0, \mathsf{strlen}(s))\,.\,\texttt{str}[j] = \texttt{elem}) \\
\vee\ (0 \leq \texttt{i} < \mathsf{strlen}(s)) \wedge \left(\begin{array}{c} \texttt{str}[\texttt{i}] = \texttt{elem} \\ \wedge \\ \forall j \in (i, strlen(s))\,.\,\texttt{str}[j] \neq \texttt{elem} \end{array}\right)
\end{array}
$$

  - Requires more expressive logic (and educated programmers)
  - (Almost) as complicated as implementation

Assertions are Specifications

- Assertions are *partial* specifications
  - Not a complete description of program behaviour
- Simpler than *full* specification:

$$(\mathtt{i} = -1) \wedge (\nexists j \in [0, \mathsf{strlen}(s)) \,.\, \mathtt{str}[j] = \mathtt{elem})$$
$$\vee\ (0 \leq \mathtt{i} < \mathsf{strlen}(s)) \wedge \begin{pmatrix} \mathtt{str[i]} = \mathtt{elem} \\ \wedge \\ \forall j \in (i, \mathit{strlen}(s)) \,.\, \mathtt{str}[j] \neq \mathtt{elem} \end{pmatrix}$$

  - Requires more expressive logic (and educated programmers)
  - (Almost) as complicated as implementation
  - Which *logical language* is used?

Specification Language in Assertions

- *Expressions* of the programming language
  - C, C++, Java, ...

Specification Language in Assertions

- *Expressions* of the programming language
  - C, C++, Java, ...
- Expressions defined by ISO/IEC 14882:2011, §5
  - e.g., syntax for *multiplicative expressions*:

    multiplicative-expression:

    pm-expression (e.g., a variable)

    multiplicative-expression * *pm-expression*

    multiplicative-expression / *pm-expression*

    multiplicative-expression % *pm-expression*

  - semantics (meaning) of multiplicative operators:
    - "3 The binary $*$ operator indicates multiplication"
    - "4 The binary / operator yields the quotient, and the binary % operator yields the remainder from the division of the first expression by the second. If the second operand of / or % is zero the behavior is undefined. [...]"

Specification Language in Assertions

- Expressions in assertions are *predicates*
  - Map values of variables to Boolean values (`true`, `false`)

Specification Language in Assertions

- Expressions in assertions are *predicates*
  - Map values of variables to Boolean values (`true`, `false`)
- In fact, I just lied to you. . .
  - *pm-expression* can be a *unary-expression* (§5.3):

unary-expression:
  postfix-expression
  ++*cast-expression*
  −−*cast-expression*
  . . .
  new-expression
  delete-expression

Specification Language in Assertions

- Expressions in assertions are *predicates*
  - Map values of variables to Boolean values (`true`, `false`)
- In fact, I just lied to you...
  - *pm-expression* can be a *unary-expression* (§5.3):

    unary-expression:
        postfix-expression
        ++*cast-expression*
        −−*cast-expression*
        ...
        new-expression
        delete-expression

- *unary-expression*s can have side effects!

Specification Language in Assertions

- Expressions in assertions are *predicates*
  - Map values of variables to Boolean values (`true`, `false`)
- In fact, I just lied to you...
  - *pm-expression* can be a *unary-expression* (§5.3):

    unary-expression:
        postfix-expression
        ++*cast-expression*
        −−*cast-expression*
        ...
        new-expression
        delete-expression

- *unary-expression*s can have side effects!
- Expression maps *program state* to a new *state* and a value

Examples of expressions with side-effects

- Increment: `++value`
- Allocation: `p=(char*)malloc(5*sizeof(char))`
- Function call: fwrite(str, 1, sizeof(str), fp)

Specification Language in Assertions

Examples of expressions with side-effects

- Increment: `assert( ++value )`
- Allocation: `assert( p=(char*)malloc(5*sizeof(char)) )`
- Function call: `assert(` fwrite(str, 1, sizeof(str), fp) `)`

Specification Language in Assertions

```
#include <stdlib.h>
#include <assert.h>

int main(int argc, char** argv)
{
  char *p;
  assert (p = malloc (5 * sizeof (char)));

  char i;
  for (i=0; i < 5; i++)
    *(p+i) = i;

  return 0;
}
```

Specification Language in Assertions

```
#include <stdlib.h>
#include <assert.h>

int main(int argc, char** argv)
{
  char *p;
  assert (p = malloc (5 * sizeof (char)));

  char i;
  for (i=0; i < 5; i++)
    *(p+i) = i;

  return 0;
}
```

- Assertions can be turned *off*: `gcc -DNDEBUG badassert.c`

Specification Language in Assertions

```c
#include <stdlib.h>
#include <assert.h>

int main(int argc, char** argv)
{
  char *p;
  assert (p = malloc (5 * sizeof (char)));

  char i;
  for (i=0; i < 5; i++)
    *(p+i) = i;

  return 0;
}
```

- Assertions can be turned *off*: `gcc -DNDEBUG badassert.c`
- Result: `Segmentation fault`

Specification Language in Assertions

- Side effects in assertions are bad idea
- T.f., we assume assertions are side-effect free predicates

Other Examples of Assertions

```
int x;

...

if (x % 2 == 0)
{
  ...
}
else
{
  assert (x % 2 == 1);
  ...
}
```

- Makes assumption explicit (`x % 2` can only be 0 or 1)
- Note: this assertion may fail (how?)

Other Examples of Assertions

- But: not every assertion is useful
- The following one is redundant and a sign of paranoia:

```
do {
  x--;
} while (x > 0);
assert (x <= 0);
```

Other Examples of Assertions

- Not redundant in the following setting:

```
do {
   ...
   if (x == 42)
      break;
   ...
} while (x > 0);
assert (x <= 0);
```

Asserting Unreachability

```
enum gender { MALE, FEMALE };
...
switch (gender) {
  case MALE:
    ...
    break;
  case FEMALE:
    ...
    break;
  default:
    assert (0);
}
```

- Assertion fails if uncovered case is reached

Asserting Unreachability

```
enum gender { MALE, FEMALE };
...
switch (gender) {
  case MALE:
    ...
    break;
  case FEMALE:
    ...
    break;
  default:
    assert (0);
}
```

- Assertion fails if uncovered case is reached
  - e.g., after type change

Assertions in a constantly changing world…

- Assertions document your assumptions
- Changes in the program may break them!
  - (turn them on for regression testing)

Asserting Correctness of Results

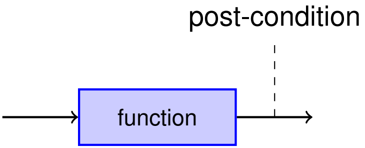
- Previously used assertions to ensure correct results:

```
assert (i == -1 || str[i] == elem);
```

Asserting Correctness of Results

- Previously used assertions to ensure correct results:

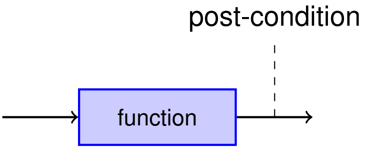
```
assert (i == -1 || str[i] == elem);
```



Asserting Correctness of Results

- Previously used assertions to ensure correct results:

```
assert (i == -1 || str[i] == elem);
```



- But result may depend on input!

Asserting Correctness of Results

```
float sqrt (float x)
{
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

- Asserts *expected result*, now *how* it is computed!

Asserting Correctness of Results

```
float sqrt (float x)
{
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

- Asserts *expected result*, now *how* it is computed!
- But what if `x` is changed?

Asserting Correctness of Results

```
float sqrt (float x)
{
  float result;
  ...
  x = x / 2; // this causes a problem
  ...

  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

Asserting Correctness of Results

```
float sqrt (float x)
{
  float result;
  ...
  x = x / 2; // this causes a problem
  ...

  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

- Solution: store `x` in “history” variable

Asserting Correctness of Results

```
float sqrt (float x)
{
  float result;
  ...
  x = x / 2; // this causes a problem
  ...

  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

- Solution: store `x` in "history" variable
  - Also known as "shadow" or "auxiliary" variables

Asserting Correctness of Results

```
float sqrt (float x)
{
  const float h_x = x;
  float result;
  ...
  x = x / 2; // this causes a problem
  ...

  assert (abs((result * result) - h_x) < EPSILON);
  return result;
}
```

- Stores *original* value of `x` before execution of `sqrt`

History Variables

- Memorise the past of the program execution
- Should have no side effect on
  - control flow
  - data flow

  of the original program!

History Variables

- Memorise the past of the program execution
- Should have no side effect on
  - control flow
  - data flow

  of the original program!
- Control flow: history variables must never influence branching
  - history variables must never occur in *conditions* (other than assertions)

History Variables

- Memorise the past of the program execution
- Should have no side effect on
  - control flow
  - data flow

  of the original program!
- Control flow: history variables must never influence branching
  - history variables must never occur in *conditions* (other than assertions)
- Data flow: values must never “flow” from history variables to program variables
  - history variables must never occur on right-hand side of assignments

History Variables

- Memorise the past of the program execution
- Should have no side effect on
  - control flow
  - data flow

  of the original program!
- Control flow: history variables must never influence branching
  - history variables must never occur in *conditions* (other than assertions)
- Data flow: values must never “flow” from history variables to program variables
  - history variables must never occur on right-hand side of assignments
- Program must still function correctly if eliminate auxiliary variables + assertions

Data Dependencies

Let $\mathtt{stmt}_j$ be an instruction in a program, and let

$R(\mathtt{stmt}_j)$	...	memory locations read by $\mathtt{stmt}_j$
$W(\mathtt{stmt}_j)$	...	memory locations written by $\mathtt{stmt}_j$

Assume there is a feasible execution path from $\mathtt{stmt}_i$ to $\mathtt{stmt}_j$

Flow (data) dependence	(RAW)	$W(\mathtt{stmt}_i) \cap R(\mathtt{stmt}_j) \neq \emptyset$
Anti-dependence	(WAR)	$R(\mathtt{stmt}_i) \cap W(\mathtt{stmt}_j) \neq \emptyset$
Output-dependence	(WAW)	$W(\mathtt{stmt}_i) \cap W(\mathtt{stmt}_j) \neq \emptyset$

Control Dependencies

$\texttt{stmt}_j$ is control-dependent on a preceding $\texttt{stmt}_i$ if $\texttt{stmt}_i$ determines whether $\texttt{stmt}_j$ is executed.

Control Dependencies

$\texttt{stmt}_j$ is control-dependent on a preceding $\texttt{stmt}_i$ if $\texttt{stmt}_i$ determines whether $\texttt{stmt}_j$ is executed.

- Let *G* be a control-flow graph with unique entry point `entry` and exit point `exit`
- $\texttt{stmt}_j$ **post-dominates** $\texttt{stmt}_i$ if $\texttt{stmt}_j$ appears on *every* path from $\texttt{stmt}_i$ to `exit`

Control Dependencies

> `stmt`$_j$ is control-dependent on a preceding `stmt`$_i$
> if `stmt`$_i$ determines whether `stmt`$_j$ is executed.

- Let G be a control-flow graph with unique entry point `entry` and exit point `exit`
- `stmt`$_j$ **post-dominates** `stmt`$_i$ if `stmt`$_j$ appears on *every* path from `stmt`$_i$ to `exit`

Statement `stmt`$_j$ is **control-dependent** on `stmt`$_i$ if

- there exists an edge from `stmt`$_i$ to `stmt`$_k$
- `stmt`$_j$ post-dominates `stmt`$_k$
- `stmt`$_j$ does *not* post-dominate `stmt`$_i$

History Variables

- Also possible to use “helper” code:

```
// assert: integer array a is sorted
bool sorted = true;
for (unsigned i = 1;
     i < sizeof(a)/sizeof(int);
     i++)
  sorted &= (a[i-1] < a[i]);
assert(sorted);
```

History Variables

- Also possible to use “helper” code:

```
// assert: integer array a is sorted
bool sorted = true;
for (unsigned i = 1;
      i < sizeof(a)/sizeof(int);
      i++)
  sorted &= (a[i-1] < a[i]);
assert(sorted);
```

- Conditions:
  - must not change original control flow
  - must not change original data flow
  - auxiliary code *must terminate*
- Primary objective: minimise *probe effect!*

Asserting Correctness of Results (Revisited)

Let's have another look at the `sqrt` function!

```
float sqrt (float x)
{
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

Isn't there a problem with this assertion?

Asserting Correctness of Results (Revisited)

Let's have another look at the `sqrt` function!

```
float sqrt (float x)
{
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

Isn't there a problem with this assertion?

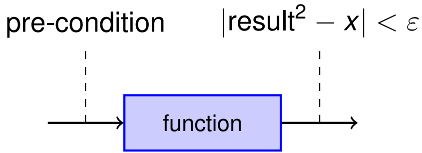
- What if $x < 0$?

Asserting Correctness of Results

- `sqrt` works only for certain inputs!

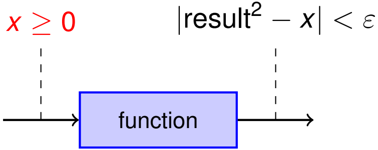
Asserting Correctness of Results

- `sqrt` works only for certain inputs!



Asserting Correctness of Results

- `sqrt` works only for certain inputs!



Design by Contract [Bertrand Meyer]

- Pre- and post-conditions represent a “contract”
- *Caller* must establish pre-condition
- *Callee* guarantees post-condition if pre-condition holds
- If pre-condition does not hold
  - callee released from contractual obligations!

> Violation of pre-condition releases callee from contractual obligations!

- Radical, but:
  - Enforces clear distribution of responsibilities
  - No “double-checking”
- The Eiffel programming language supports contracts directly:
  - `require` – `ensure`

Design by Contract

Is it a good idea to assert the pre-condition?

```
float sqrt (float x)
{
  assert (x >= 0);
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

Design by Contract

Is it a good idea to assert the pre-condition?

```
float sqrt (float x)
{
  assert (x >= 0);
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

- If we have full control over caller, yes

Design by Contract

Is it a good idea to assert the pre-condition?

```
float sqrt (float x)
{
  assert (x >= 0);
  float result;
  ...
  assert (abs((result * result) - x) < EPSILON);
  return result;
}
```

- If we have full control over caller, yes
- In general, however, no.

Rule of thumb:

> Use assertions if you can control whether they hold or not

Rule of thumb:

Use assertions if you can control whether they hold or not

- Assertions are a debugging tool!
  - Use it to find your *own* bugs

Rule of thumb:

Use assertions if you can control whether they hold or not

- Assertions are a debugging tool!
  - Use it to find your *own* bugs
- For everything else use exceptions/error codes

Design by Contract

```
float sqrt (float x)
{
  if (x < 0)
    return nanf(); // Not a number
  float result;
  const float h_x = x;
  ...
  assert (abs((result * result) - h_x) < EPSILON);
  return result;
}
```

Notes on Java:

- Java provides
  - `IllegalArgumentException`
  - `NullPointerException`
  - `IllegalStateException`
- C++ provides instances of `logic_error` (in `<stdexcept>`):
  - `domain_error`
  - `invalid_argument`
  - `length_error`
  - `out_of_range`
- *Assert* pre-conditions (only) in private methods

Design by Contract in Java

```java
/**
 * @param value Percentage between 0 and 100
 */
public setPercentage (int value)
{
  if (value < 0 || value > 100) {
     throw new
     IllegalArgumentException
          (Integer.toString(value));
  }
  this.value = value;
}
```

- Question for Java specialists: why no `throws` clause?

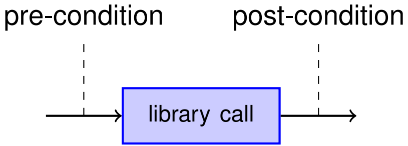
Design by Contract in Java

```java
/**
 * @param value Percentage between 0 and 100
 */
public setPercentage (int value)
{
  if (value < 0 || value > 100) {
      throw new
      IllegalArgumentException
           (Integer.toString(value));
  }
  this.value = value;
}
```

- Question for Java specialists: why no `throws` clause?
  - Unchecked exception
  - Unlikely to be caught (indicates severe bug in program)

Assert Results of Library Calls

- Assertions can be used to *check result* of external library
  - e.g., if we don't trust the library



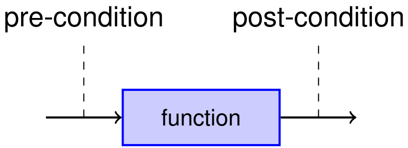
- Assert that
  - we satisfy the pre-condition of the library function
  - that the library function returns a correct result

Assert Results of Library Calls

- For example: We still don't trust `sqrt`

```
...
float x = sqrt (y); // square root of x
assert (x >= 0);
```

Weakening and Strengthening Assertions



- Pre-condition can be *strengthened* to allow *fewer* states
  - e.g., $(\mathtt{x} \geq 10)$ instead of $(\mathtt{x} \geq 0)$
- Post-condition can be *weakened* to allow *more* states
  - e.g., $(|\mathtt{result}^2 - \mathtt{x}| < \varepsilon) \; || (\mathtt{result} == \mathtt{NaN})$
- Contract will *still* be satisfied!

Conditional Assertions

- Assertion could also be conditional

```
if (h_x > 0)
    assert (abs((result * result) - h_x) < EPSILON);
```

- Can also be written just as an assertion:

```
h_x <= 0 || abs((result * result) - h_x) < EPSILON
```

- Note that $(\neg A \lor B)$ corresponds to $(A \Rightarrow B)$.

Invariant Assertions

- Assertions checked at individual points during execution
- If assertions occur in loops, they must hold *repeatedly*

Invariant Assertions

```
// compute q = x / y, r = x % y
unsigned q = 0; unsigned r = x;
while (r >= y)
{
  r = r - y;
  q = q + 1;
  assert (x == q * y + r);
}
```

- `x == q * y + r` holds throughout the loop!
- After termination: `(x == q * y + r) && (r < y)`

Invariant Assertions

- We can even *prove* this!

```
 r = r - y;

 q = q + 1;
assert (x == q * y + r);
```

Invariant Assertions

- We can even *prove* this!

```
 r = r - y;
assert (x == (q + 1) * y + r);
 q = q + 1;
assert (x == q * y + r);
```

Invariant Assertions

- We can even *prove* this!

```
assert (x == (q + 1) * y + (r - y));
 r = r - y;
assert (x == (q + 1) * y + r);
 q = q + 1;
assert (x == q * y + r);
```

Invariant Assertions

- We can even *prove* this!

```
assert (x == (q + 1) * y + (r - y));
 r = r - y;
assert (x == (q + 1) * y + r);
 q = q + 1;
assert (x == q * y + r);
```

- Assertion holds throughout the loop!

Invariant Assertions

- We can even *prove* this!

```
assert (x == (q + 1) * y + (r - y));
 r = r - y;
assert (x == (q + 1) * y + r);
 q = q + 1;
assert (x == q * y + r);
```

- Assertion holds throughout the loop!
- Assertion holds at the end of the loop!

Proofs with Assertions

What just happened? (Once more, a bit slower)

- `(x == q * y + r)` holds *after* assignment `q = q + 1`

Proofs with Assertions

What just happened? (Once more, a bit slower)

- `(x == q * y + r)` holds *after* assignment `q = q + 1`
- But the “new” `q` is the “old” `q` plus 1!

Proofs with Assertions

What just happened? (Once more, a bit slower)

- `(x == q * y + r)` holds *after* assignment `q = q + 1`
- But the "new" `q` is the "old" `q` plus 1!
- Therefore, `(x == (q + 1) * y + r)` holds for the "old" `q`

Proofs with Assertions

What just happened? (Once more, a bit slower)

- `(x == q * y + r)` holds *after* assignment `q = q + 1`
- But the "new" `q` is the "old" `q` plus 1!
- Therefore, `(x == (q + 1) * y + r)` holds for the "old" `q`
- If

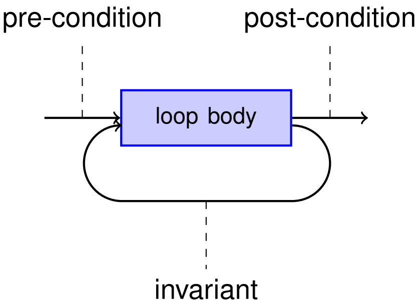
  ```
  (x == (q + 1) * y + r)
  ```

  holds before assignment `q = q + 1`, then

  ```
  (x == q * y + r)
  ```

  holds afterwards!

Invariant Assertions



Invariant Assertions

- Does the assertion hold at the beginning of the loop, too?

```
 q = 0;

 r = x;
assert (x == q * y + r);
```

Invariant Assertions

- Does the assertion hold at the beginning of the loop, too?

```
 q = 0;
assert (x == q * y + x);
 r = x;
assert (x == q * y + r);
```

Invariant Assertions

- Does the assertion hold at the beginning of the loop, too?

```
assert (x == 0 * y + x);
 q = 0;
assert (x == q * y + x);
 r = x;
assert (x == q * y + r);
```

Invariant Assertions

- Does the assertion hold at the beginning of the loop, too?

```
assert (x == 0 * y + x);
 q = 0;
assert (x == q * y + x);
 r = x;
assert (x == q * y + r);
```

- If

  `x == x`

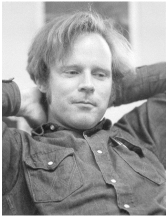
  holds before `q = 0; r = x;` then

  `(x == q * y + r)`

  holds afterwards!

Inductive Invariant Assertions

Robert W. Floyd, “Assigning Meanings to Programs”, 1967



“Then, by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at the time. By this means, we may prove certain properties of programs, . . . ”

Induction

Mathematical induction proves that a statement involving a natural number n holds for all values of n.

- *Base case.* Show that claim holds for $n = 0$.
- *Induction hypothesis*. Assume claim holds for n.
- *Induction step.* Show: claim holds for $n \Rightarrow$ it holds for $n + 1$

Induction

Mathematical induction proves that a statement involving a natural number n holds for all values of n.

- *Base case.* Show that claim holds for $n = 0$.
- *Induction hypothesis*. Assume claim holds for n.
- *Induction step.* Show: claim holds for $n \Rightarrow$ it holds for $n + 1$
- *Conclusion.* Claim holds for all $n \in \mathbb{N}$.

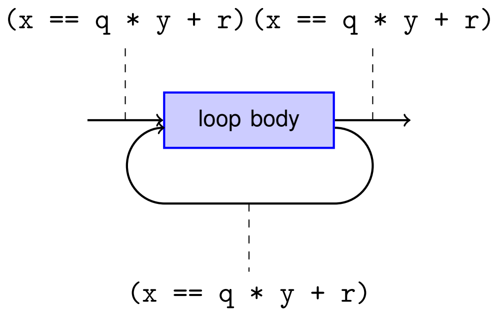
Induction

Mathematical induction proves that a statement involving a natural number n holds for all values of n.

- *Base case.* Show that claim holds for $n = 0$.
- *Induction hypothesis*. Assume claim holds for n.
- *Induction step.* Show: claim holds for $n \Rightarrow$ it holds for $n + 1$
- *Conclusion.* Claim holds for all $n \in \mathbb{N}$.

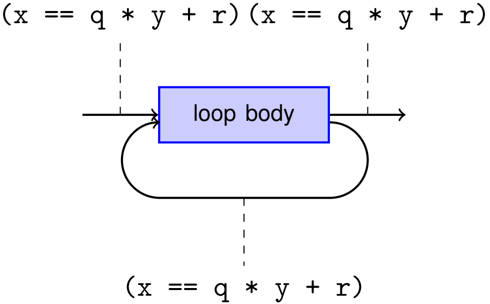
In our case: n is the number of *loop iterations*.

Inductive Invariant Assertions



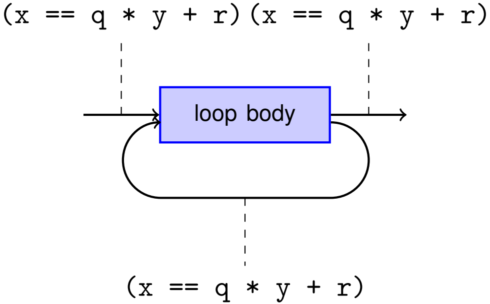
- `(x == q * y + r)` is an inductive invariant of the loop

Inductive Invariant Assertions



- `(x == q * y + r)` is an inductive invariant of the loop
- `(x == q * y + r) && (r < y)` holds after loop

Inductive Invariant Assertions



- `(x == q * y + r)` is an inductive invariant of the loop
- `(x == q * y + r) && (r < y)` holds after loop
- We have an inductive correctness proof!

Another *Invariant* Assertions

- Division? Meh. Let's try something more interesting.
- How many bits of a variable `x` are set to 1?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  y = y & (y-1);
  c = c+1;
}
```

Another *Invariant* Assertions

- Division? Meh. Let's try something more interesting.
- How many bits of a variable `x` are set to 1?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  y = y & (y-1);
  c = c+1;
}
```

- How does this work?

Another *Invariant* Assertions

- Division? Meh. Let's try something more interesting.
- How many bits of a variable `x` are set to 1?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  y = y & (y-1);
  c = c+1;
}
```

- How does this work?
- `y = y & (y-1)` deletes *least significant* bit

Another *Invariant* Assertions

- Division? Meh. Let's try something more interesting.
- How many bits of a variable `x` are set to 1?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  y = y & (y-1);
  c = c+1;
}
```

- How does this work?
- `y = y & (y-1)` deletes *least significant* bit
- But how??

Peter Wegener's Bit-Counting Trick

```
y = y & (y-1);
```

- We know: `y > 0` (because of loop exit condition)

Peter Wegener's Bit-Counting Trick

```
y = y & (y-1);
```

- We know: `y > 0` (because of loop exit condition)
- Assume `y` is binary $b_n \dots b_2 b_1 1$
  - then `(y-1)` is binary $b_n \dots b_2 b_1 0$
  - $(b_n \dots b_2 b_1 1 \;\&\; b_n \dots b_2 b_1 0) \quad = \quad b_n \dots b_2 b_1 0$

Peter Wegener's Bit-Counting Trick

```
y = y & (y-1);
```

- We know: `y > 0` (because of loop exit condition)
- Assume `y` is binary $b_n \ldots b_2 b_1 1$
  - then `(y-1)` is binary $b_n \ldots b_2 b_1 0$
  - $(b_n \ldots b_2 b_1 1 \;\&\; b_n \ldots b_2 b_1 0) \quad = \quad b_n \ldots b_2 b_1 0$
- Assume `y` is binary $b_n \ldots b_i 100$
  - then `(y-1)` is

	b_n	...	b_i	1	0	0	
+	1	...	1	1	1	1	(-1 in 2's complement)
	b_n	...	b_i	0	1	1	

Peter Wegener's Bit-Counting Trick

```
y = y & (y-1);
```

- We know: `y > 0` (because of loop exit condition)
- Assume `y` is binary $b_n \dots b_2 b_1 1$
  - then `(y-1)` is binary $b_n \dots b_2 b_1 0$
  - $(b_n \dots b_2 b_1 1 \;\&\; b_n \dots b_2 b_1 0) \quad = \quad b_n \dots b_2 b_1 0$
- Assume `y` is binary $b_n \dots b_i 100$
  - then `(y-1)` is

$$\begin{array}{cccccccl} & b_n & \dots & b_i & 1 & 0 & 0 & \\ + & 1 & \dots & 1 & 1 & 1 & 1 & \text{(-1 in 2's complement)} \\ \hline & b_n & \dots & b_i & 0 & 1 & 1 & \end{array}$$

  - $(b_n \dots b_i 100 \dots \;\&\; b_n \dots b_i 011 \dots) \quad = \quad b_n \dots b_i 000 \dots$

Wegener's Bit-Counting Algorithm

- Let's add an assertion!

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{

  y = y & (y-1);

  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Let's add an assertion!

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{

  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Let's add an assertion!

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Let's add an assertion!
- Assertion holds in first iteration
  - `(y & (y -1)) < x`, since `y != 0`

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Let's add an assertion!
- Assertion holds in first iteration
  - `(y & (y -1)) < x`, since `y != 0`

  But is the assertion inductive?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Is the assertion inductive?

```
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Is the assertion inductive?

```
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);

}
```

Wegener's Bit-Counting Algorithm

- Is the assertion inductive?
  - Does (`x != y`) and (`y != 0`) imply (x != (y & (y-1)))?

```
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);

}
```

Wegener's Bit-Counting Algorithm

- Is the assertion inductive?
  - Does (`x != y`) and (`y != 0`) imply (x != (y & (y-1)))?
  - No! Counterexample: `x`=0, `y`=1

```
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);

}
```

Wegener's Bit-Counting Algorithm

- Assertion holds in every iteration of the program!
- But is *not* an inductive invariant!

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Assertion holds in every iteration of the program!
- But is *not* an inductive invariant!
- Is there something wrong with the program?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert (x != (y & (y-1)));
  y = y & (y-1);
  assert (x != y);
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Let's try another assertion!

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{

  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Let's try another assertion!

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Does this hold in the first iteration?

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Does this hold in the first iteration?

yes, since `y!=0`

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- Does this hold in the first iteration?

yes, since y!=0

yes, since x == y

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?

```
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?
  - Does

    `(y != 0)` and `(x != 0) && (y <= (x-1))`

    imply

    `(x != 0) && ((y & (y-1)) <= (x-1))`

```
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?
  - Does

    `(y != 0)` and `(x != 0) && (y <= (x-1))`

    imply

    ```
    (x != 0) && ((y & (y-1)) <= (x-1))
    ```

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?
  - Does

    `(y != 0)` and `(x != 0) && (y <= (x-1))`

    imply

    `(x != 0) && ((y & (y-1)) <= (x-1))`
  - We know: `((y & (y-1)) < y` unless `y == 0`
    (since the assignment *deletes* a bit)

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?
  - Does

    `(y != 0)` and `(x != 0) && (y <= (x-1))`

    imply

    `(x != 0) && ((y & (y-1)) <= (x-1))`
  - We know: `((y & (y-1)) < y` unless `y == 0`
    (since the assignment *deletes* a bit)
  - But `y` is already smaller or equal `x-1`!

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?
  - Does

    `(y != 0)` and `(x != 0) && (y <= (x-1))`

    imply

    `(x != 0) && ((y & (y-1)) <= (x-1))`
  - We know: `((y & (y-1)) < y` unless `y == 0` (since the assignment *deletes* a bit)
  - But `y` is already smaller or equal `x-1`!
  - Therefore `((y & (y-1)) <= (x-1)`

Wegener's Bit-Counting Algorithm

- What about subsequent iterations?
  - Does

    `(y != 0)` and `(x != 0) && (y <= (x-1))`

    imply

    `(x != 0) && ((y & (y-1)) <= (x-1))`
  - We know: `((y & (y-1)) < y` unless `y == 0`
    (since the assignment *deletes* a bit)
  - But `y` is already smaller or equal `x-1`!
  - Therefore `((y & (y-1)) <= (x-1)`
  - And `x` doesn't change, so `x != 0`

Wegener's Bit-Counting Algorithm

- `(x != 0) && (y <= (x-1))` is an *inductive invariant*

```
unsigned y = x;
unsigned c = 0;
while (y != 0)
{
  assert ((x != 0) && ((y & (y-1)) <= (x-1)));
  y = y & (y-1);
  assert ((x != 0) && (y <= (x-1)));
  c = c+1;
}
```

Wegener's Bit-Counting Algorithm

- `(x != 0) && (y <= (x-1))` is an *inductive invariant*
- But so is `(y <= (x-1))`. So what's `(x != 0)` for?

Wegener's Bit-Counting Algorithm

- `(x != 0) && (y <= (x-1))` is an *inductive invariant*
- But so is `(y <= (x-1))`. So what's `(x != 0)` for?
- If $\mathtt{y} \leq (\mathtt{x} - 1)$ then $\mathtt{y} < \mathtt{x}$
  - unless $\mathtt{x} = 0$, in which case $\mathtt{x} - 1$ *underflows*

Wegener's Bit-Counting Algorithm

- `(x != 0) && (y <= (x-1))` is an *inductive invariant*
- But so is `(y <= (x-1))`. So what's `(x != 0)` for?
- If $y \leq (x - 1)$ then $y < x$
  - unless $x = 0$, in which case $x - 1$ *underflows*
- If $y < x$ then $y \neq x$

Wegener's Bit-Counting Algorithm

- `(x != 0) && (y <= (x-1))` is an *inductive invariant*
- But so is `(y <= (x-1))`. So what's `(x != 0)` for?
- If $y \leq (x - 1)$ then $y < x$
  - unless $x = 0$, in which case $x - 1$ *underflows*
- If $y < x$ then $y \neq x$

The new assertion implies the original one!

Wegener's Bit-Counting Algorithm

- `(x != 0) && (y <= (x-1))` is an *inductive invariant*
- But so is `(y <= (x-1))`. So what's `(x != 0)` for?
- If $\mathtt{y} \leq (\mathtt{x} - 1)$ then $\mathtt{y} < \mathtt{x}$
  - unless $\mathtt{x} = 0$, in which case $\mathtt{x} - 1$ *underflows*
- If $\mathtt{y} < \mathtt{x}$ then $\mathtt{y} \neq \mathtt{x}$

The new assertion implies the original one!

This proves that $(\mathtt{x} \neq \mathtt{y})$ holds throughout the loop

Summary Inductive Invariants

- Loop invariants hold in every loop iteration
- Inductive loop invariants:
  - if it holds in one iteration, we can deduce that it holds in the next one, too
- Any consequence of an inductive invariant is an invariant
  - but not vice versa!

Summary

- Assertions express intent of the programmer
- Powerful debugging technique
- Enable “design by contract”
- Can even be used to prove programs correct