Programm- & Systemverifikation

A bug's life

Georg Weissenbacher

184.741

for(syte)
Formal Methods in Systems Engineering

A problem has been detected and Windows has been shut down to prevent damage to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen, restart your computer, If this screen appears again, follow these steps:

Check to make sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.

If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x00000000,0xF86B5A89)

*** gv3.sys - Address F86B5A89 base at F86B5000, DateStamp 3dd991eb

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance.

A problem has been detected and Windows has been shut down to prevent damage to your computer.

DRIVER_IRQL_NOT_LESS_OR_EQUAL

If this is the first time you've seen this Stop error screen, restart your computer, If this scr[illegible] appears again, follow these steps:

Check to make sure any new hardwar[illegible]
If this is a new installation, ask[illegible]er for any Windows updates you might [illegible]

If problems continue, disable or r[illegible] or software. Disable BIOS memory c[illegible]
If you need to use Safe Mode to remove or disable comp[illegible]ents, restart your computer, press F8 to select Advanced Startup Opti[illegible]s, and then select Safe Mode.

Technical information:

*** STOP: 0x000000D1 (0x0000000C,0x00000002,0x00000000,0xF86[illegible])

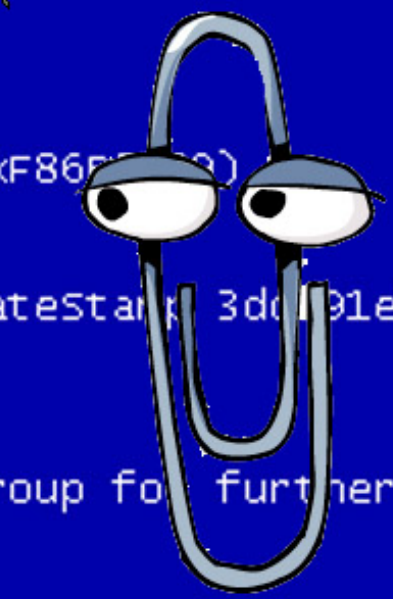
*** gv3.sys - Address F86B5A89 base at F86B5000, DateStam[illegible] 3dd[illegible]91eb

Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further assistance.

Looks like you want to know what

DRIVER_IRQL_NOT _LESS_OR_EQUAL

means ...



What went wrong?

- `gv3.sys`: Mobile processor power management
- Each driver routine runs at certain *interrupt request level*

IRQL	Description
PASSIVE_LEVEL	User threads and kernel-mode operations
APC_LEVEL	Async procedure calls and page faults
DISPATCH_LEVEL	Thread scheduler and DPCs
	...
POWER_LEVEL	Power failure
HIGH_LEVEL	Machine checks, catastrophic errors

What went wrong?

- `gv3.sys`: Mobile processor power management
- Each driver routine runs at certain *interrupt request level*

IRQL	Description
PASSIVE_LEVEL	User threads and kernel-mode operations
APC_LEVEL	Async procedure calls and page faults
DISPATCH_LEVEL	Thread scheduler and DPCs
	...
POWER_LEVEL	Power failure
HIGH_LEVEL	Machine checks, catastrophic errors

- Kernel API imposes restrictions on calls, e.g.,
- `ExAcquireFastMutex`:
  - acquires fast mutex with APCs to the current thread disabled.
  - Callers **must be running** at IRQL $\leq$ `APC_LEVEL`.

What went wrong?

- All deferred procedure calls run at `DISPATCH_LEVEL`

```
KDEFERRED_ROUTINE CustomDpc;

VOID MyDpc(
__in      struct _KDPC *Dpc,
__in_opt  PVOID DeferredContext,
__in_opt  PVOID SystemArgument1,
_in_opt   PVOID SystemArgument2
)
{
     ...
     ExAcquireFastMutex (_mutex);
     ...
     ExReleaseFastMutex (_mutex);
```

What is the output of this program?

```
#include <stdio.h>

int main (int argc, char** argv)
{
  int c = 2147483642;

  while ((c+1) > c)
  {
    printf ("%d\n", c);
    c++;
  }

  return 0;
}
```

Let's figure this out!

- `gcc -g -o overflow overflow.c`
- `./overflow`

Let's figure this out!

- `gcc -g -o overflow overflow.c`
- `./overflow`

```
2147483642
2147483643
2147483644
2147483645
2147483646
```

Let's figure this out!

- `gcc -g -o overflow overflow.c`
- `./overflow`

```
2147483642
2147483643
2147483644
2147483645
2147483646
```

- `gcc -O3 -o overflow overflow.c`
- `./overflow`

Let's figure this out!

- `gcc -g -o overflow overflow.c`
- `./overflow`

```
2147483642
2147483643
2147483644
2147483645
2147483646
```

- `gcc -O3 -o overflow overflow.c`
- `./overflow`

```
2147483642
2147483643
...
2147483646
2147483647
-2147483648
-2147483647
...
```

Let's count to a million (the fast way)!

```
#include <stdio.h>
#include <pthread.h>

int c = 0;
void *count (void *parg)
{
  for (unsigned i=0; i<5000000; i++)
    c++;
}

int main (int argc, char** argv)
{
  pthread_t thread1, thread2;
  pthread_create (&thread1, NULL, count, NULL);
  pthread_create (&thread2, NULL, count, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf ("%d\n", c);
  return 0;
}
```

Let's see how fast this really is . . .

- `gcc -pthread -o threads threads.c`
- `./threads`

Let's see how fast this really is . . .

- `gcc -pthread -o threads threads.c`
- `./threads`

  960225

Let's see how fast this really is . . .

- `gcc -pthread -o threads threads.c`
- `./threads`

  960225

- `./threads`

Let's see how fast this really is . . .

- `gcc -pthread -o threads threads.c`
- `./threads`

  960225

- `./threads`

  1000000

Let's see how fast this really is . . .

- `gcc -pthread -o threads threads.c`
- `./threads`

  960225

- `./threads`

  1000000

- `./threads`

Let's see how fast this really is . . .

- `gcc -pthread -o threads threads.c`
- `./threads`

  960225

- `./threads`

  1000000

- `./threads`

  658697

What does this program compute?

```
class Imaginary {
public:
  float r; float i;

  Imaginary (): r(0), i(0) { }
  Imaginary (Imaginary &other) { *this = other; }
  Imaginary operator= (const Imaginary other)
  {
    r = other.r; i = other.i;
  }

};

int main (int argc, char** argv)
{
  Imaginary i;
  Imaginary j = i;
  return j.i;
}
```

Let's try it out!

- `g++ -o recursion recursion.cpp`
- `./recursion`

Let's try it out!

- `g++ -o recursion recursion.cpp`
- `./recursion`

  Segmentation fault

What the . . .

What's wrong with these programs?

What the . . .

What's wrong with these programs?

(I'll tell you in a bit . . .)

Outline of Lecture

- **What is a bug?**
  - Classes of Bugs
  - Cause and Symptom
- What do we need to understand bugs?
  - Understand the Program
  - Know the Programmer's Intentions

What is a (software) bug?

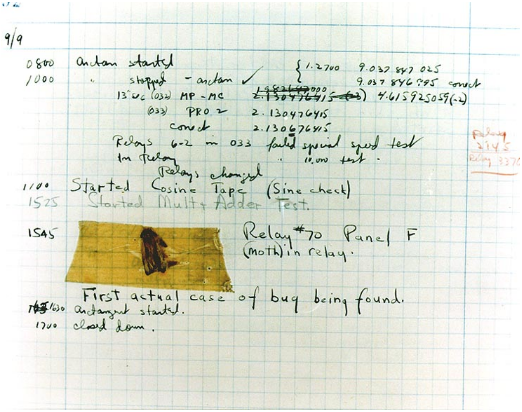
“Know your enemy”

Sun Tzu, The Art of War

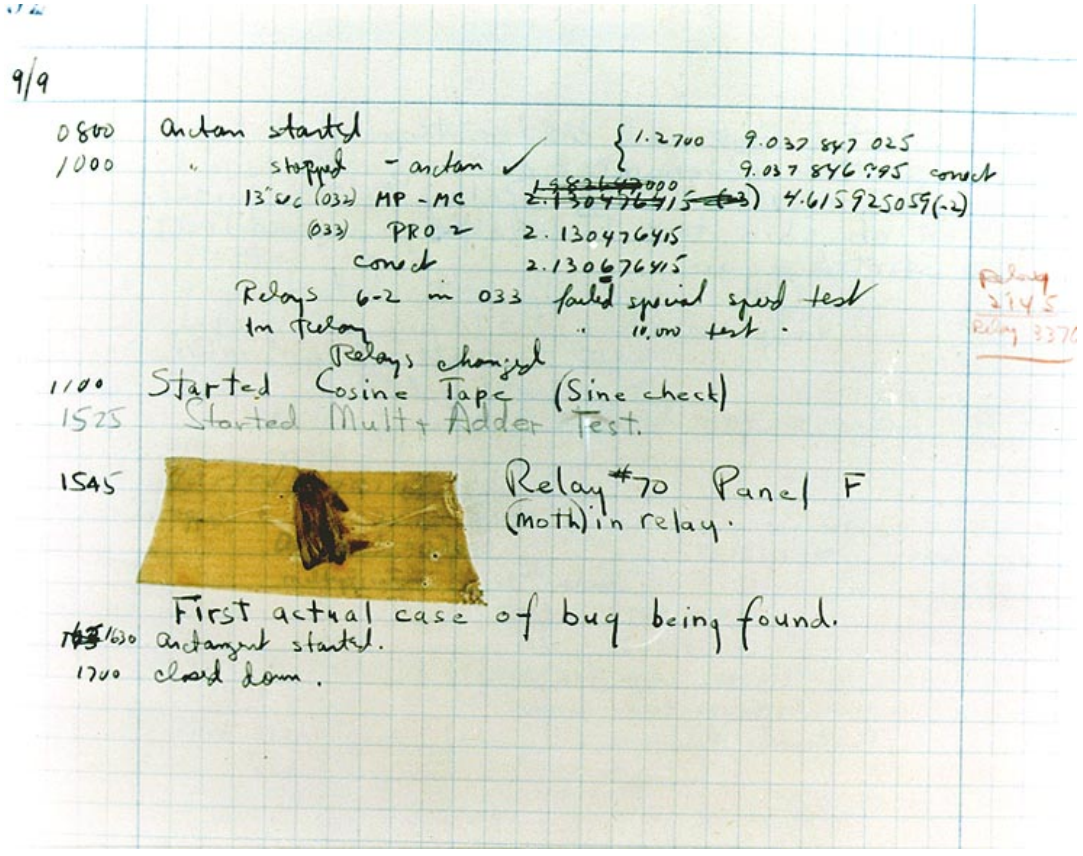
(executive summary of original quote)

孫子兵法

What is a bug?

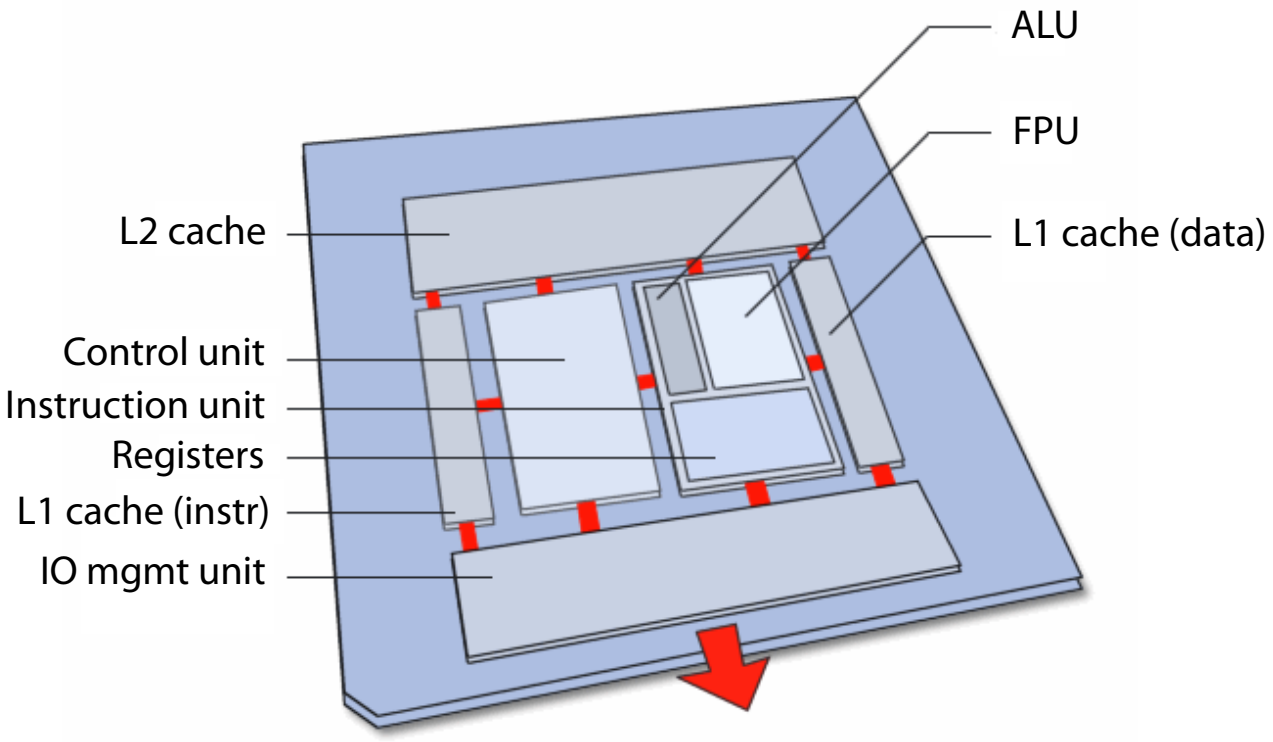

9/9
0800 arctan started
1000 " stopped - arctan ✓
13" uc (032) MP - MC
(033) PRO 2 2.130476415
const 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 arctangent started.
1700 closed down.

What is a bug?

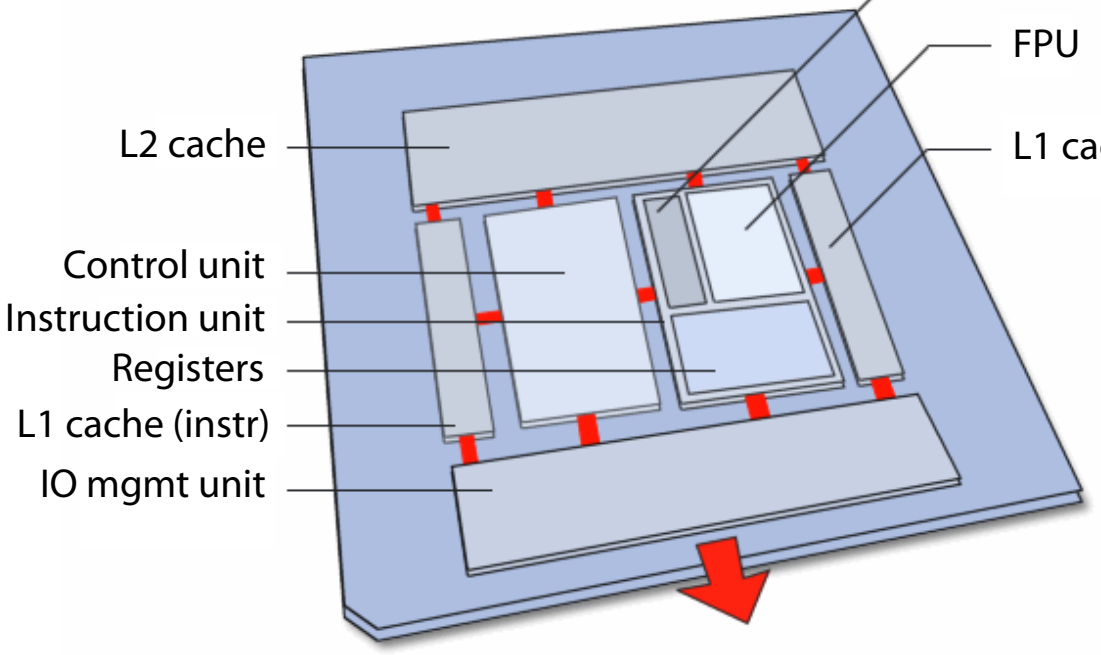


"**flaw** in a system that results in unintended behaviour"

What kinds of bugs are there?

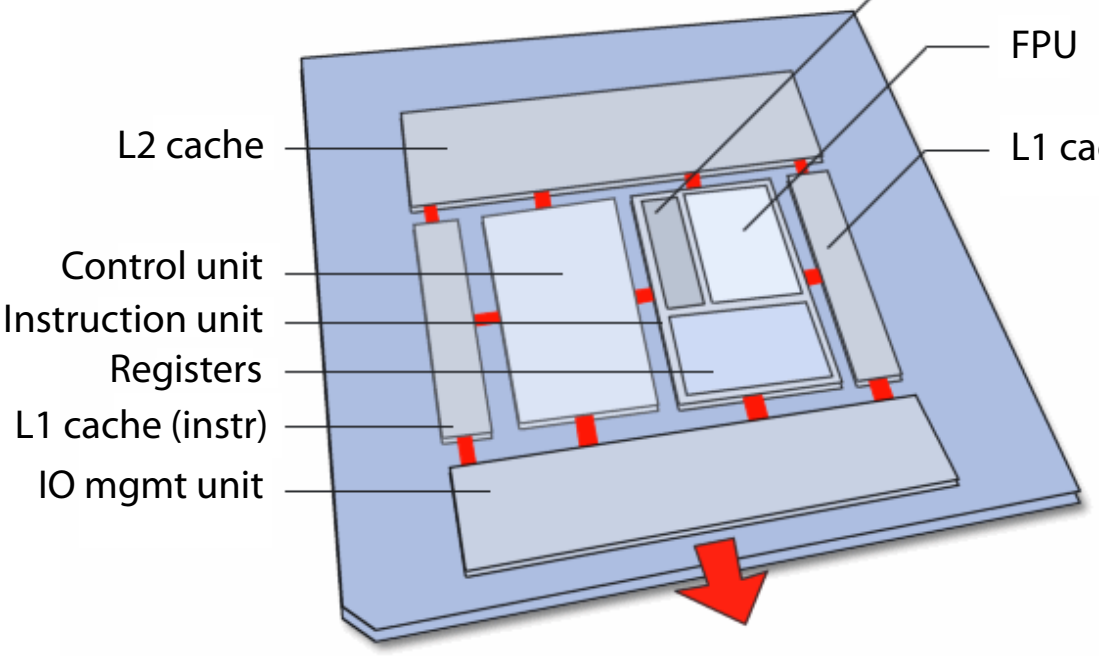


What kinds of bugs are there?



Arithmetic bugs
overflow, division by 0,
precision/conversion errors

What kinds of bugs are there?

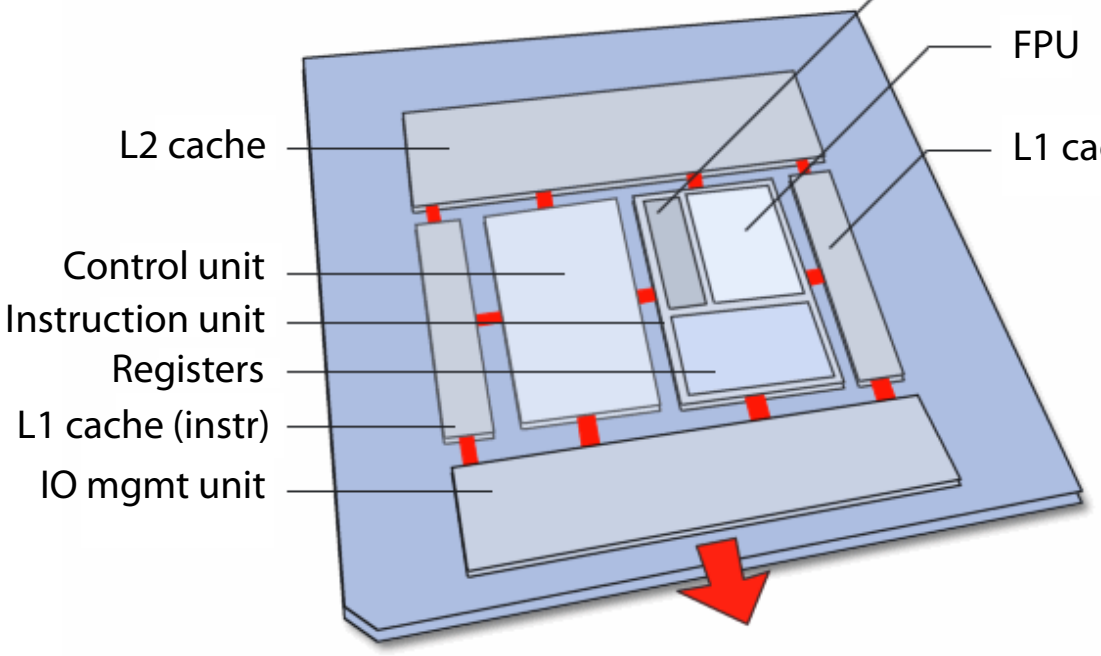


Arithmetic bugs

overflow, division by 0,
precision/conversion errors

Ariane 5, flight 501

What kinds of bugs are there?



Arithmetic bugs

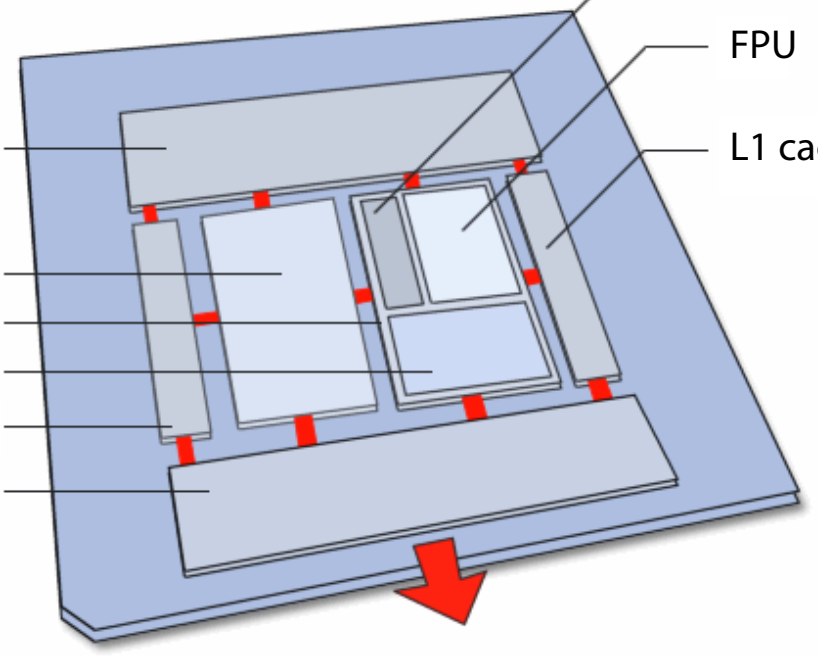
overflow, division by 0,
precision/conversion errors

Ariane 5, flight 501

`overflow.c`

What kinds of bugs are there?

Logic bugs
infinite loops,
off-by-one



Arithmetic bugs
overflow, division by 0,
precision/conversion errors

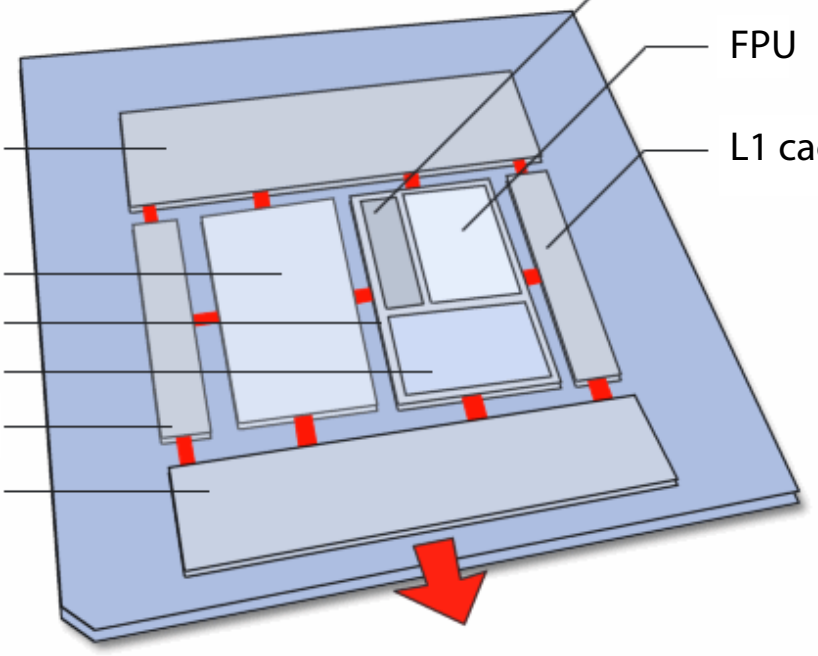
Ariane 5, flight 501

`overflow.c`

What kinds of bugs are there?

Logic bugs
infinite loops,
off-by-one

`recursion.cpp`



Arithmetic bugs
overflow, division by 0,
precision/conversion errors

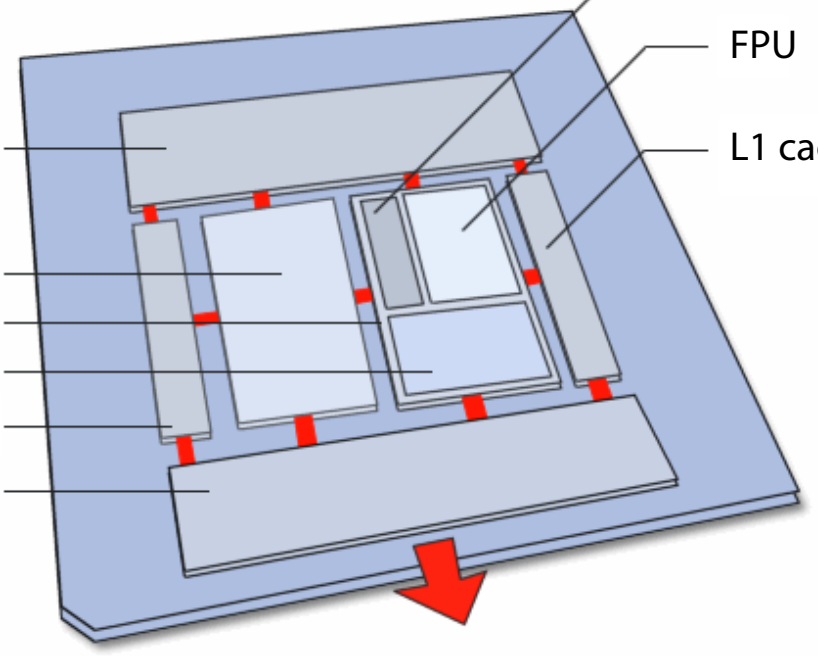
Ariane 5, flight 501

`overflow.c`

What kinds of bugs are there?

Logic bugs
infinite loops,
off-by-one

`recursion.cpp`



Arithmetic bugs
overflow, division by 0,
precision/conversion errors

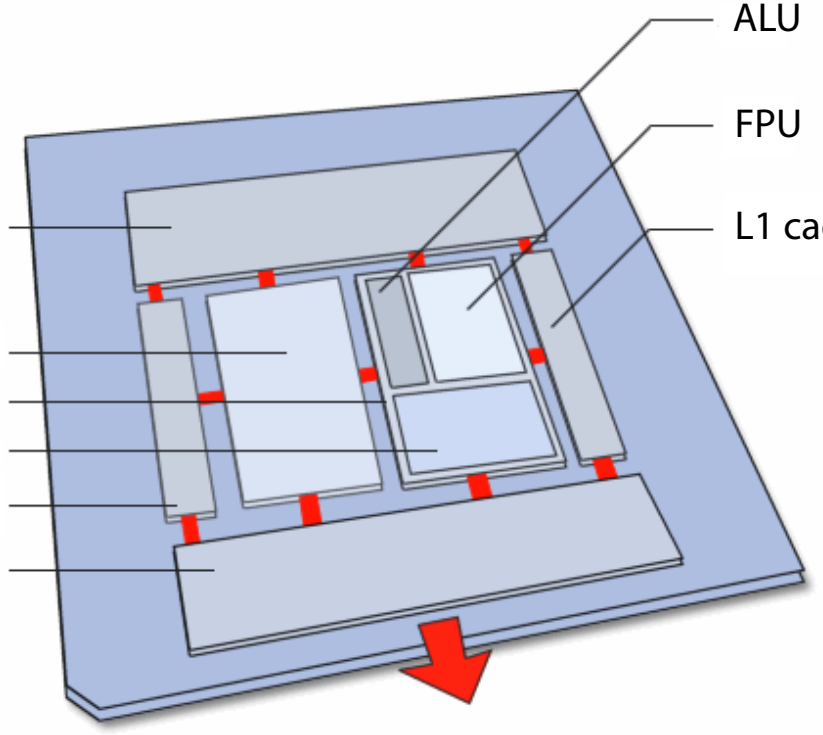
Ariane 5, flight 501

`overflow.c`

What kinds of bugs are there?

Arithmetic bugs

overflow, division by 0, precision/conversion errors



Logic bugs

infinite loops, off-by-one

`recursion.cpp`

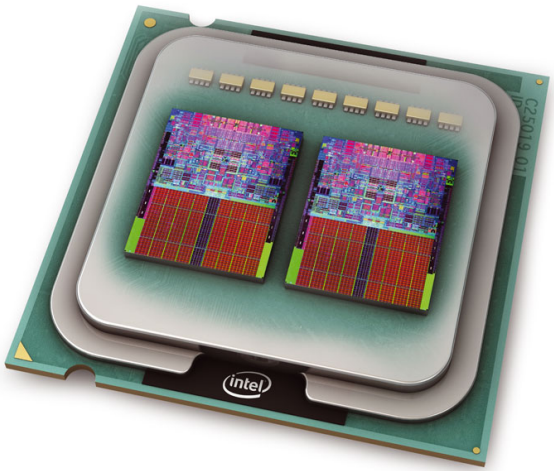
Ariane 5, flight 501

`overflow.c`

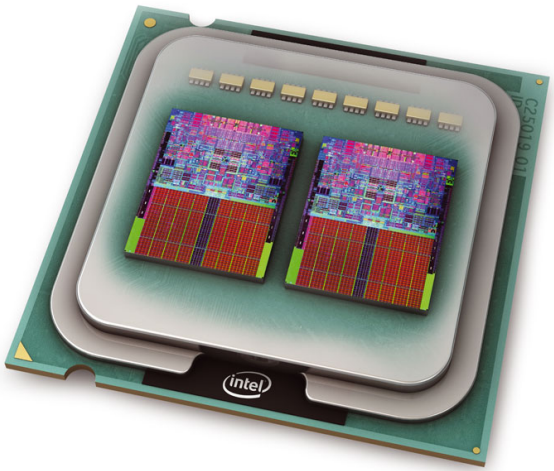
Resource bugs

NULL pointer deref, uninitialised variables, wrong data type for instruction, access violations, resource leaks, buffer overflows

What kinds of bugs are there?



What kinds of bugs are there?



Multi-Threading Bugs

- deadlock

  (two tasks wait for same resource)

- livelock/starvation

  (thread makes no progress)

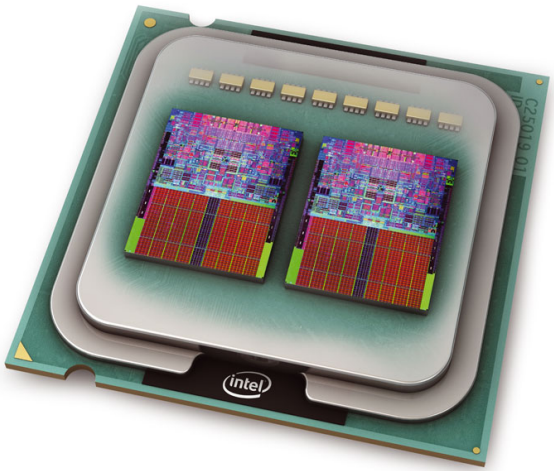
- race condition

  (two threads accessing resource at same time)

- atomicity violation

  (interruption of supposedly atomic action)

What kinds of bugs are there?



Multi-Threading Bugs

- deadlock

  (two tasks wait for same resource)

- livelock/starvation

  (thread makes no progress)

- race condition

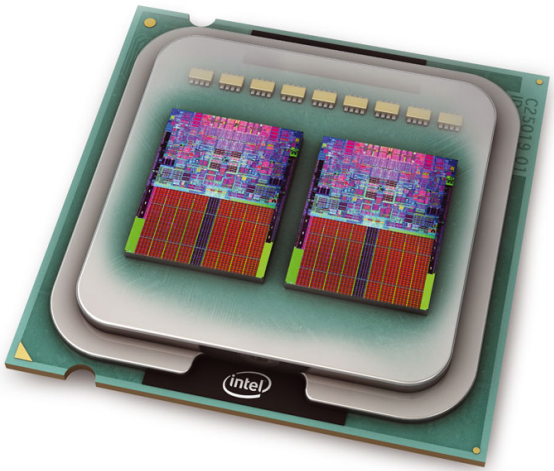
  (two threads accessing resource at same time)

  - Therac-25 bug, Northeastern Blackout, Dirty COW (last lecture)

- atomicity violation

  (interruption of supposedly atomic action)

What kinds of bugs are there?



Multi-Threading Bugs

- deadlock

  (two tasks wait for same resource)

- livelock/starvation

  (thread makes no progress)

- race condition

  (two threads accessing resource at same time)

  - Therac-25 bug, Northeastern Blackout, Dirty COW (last lecture)
  - our own `threads.c`?

- atomicity violation

  (interruption of supposedly atomic action)

What kinds of bugs are there?

Syntax/Semantics Bugs

- (Unintentional) use of wrong operator (= vs ==)
- Wrong assumptions about programming language semantics

What kinds of bugs are there?

Syntax/Semantics Bugs

- (Unintentional) use of wrong operator (= vs ==)
- Wrong assumptions about programming language semantics
  - we will hear more about this!

What kinds of bugs are there?

Interfacing Bugs

- incorrect usage of API
- incorrect protocol implementation
- incorrect hardware handling/assumptions about platform

What kinds of bugs are there?

Interfacing Bugs

- incorrect usage of API
  - the blue screen from before
- incorrect protocol implementation
- incorrect hardware handling/assumptions about platform

What kinds of bugs are there?

Performance/Timing Bugs

- timing in real-time programs
- high computational complexity
- random disk/memory access (e.g., garbage collection)

What kinds of bugs are there?

Teamworking/Development Related Bugs

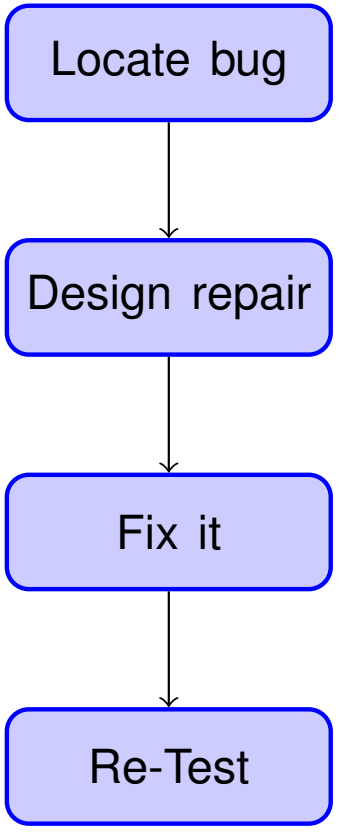
- documentation/implementation out of sync
- copy & paste errors
- wrong version of source code

A Different Classification

Bugs from a programmer's point of view . . .

- **Bohrbug** named after Bohr
  (plain and simple – like Bohr's atomic model)
- **Heisenbug** named after Heisenberg
  (disappears or alters its behavior if you try to debug it)
- **Schrödinbug** named after Schrödinger
  (code that should have never worked but did – until you looked at it)
- **Mandelbug** named after Benoît Mandelbrot
  (cause too hard to understand, bug appears chaotic)

How to Debug [Ian Sommerville, 2007]



How “Bugs” come into being

1. programmer introduces a **fault** in the code

How “Bugs” come into being

1. programmer introduces a **fault** in the code
2. fault gets excited during execution, results in **error**

How “Bugs” come into being

1. programmer introduces a **fault** in the code
2. fault gets excited during execution, results in **error**
3. error propagates, results in system **failure**

Terminology: Fault, Error, Failure

1. **fault** – cause of an error (e.g., mistake in coding)
2. **error** – incorrect state that may lead to failure
3. **failure** – deviation from specified/desired behaviour

Terminology: Fault, Error, Failure

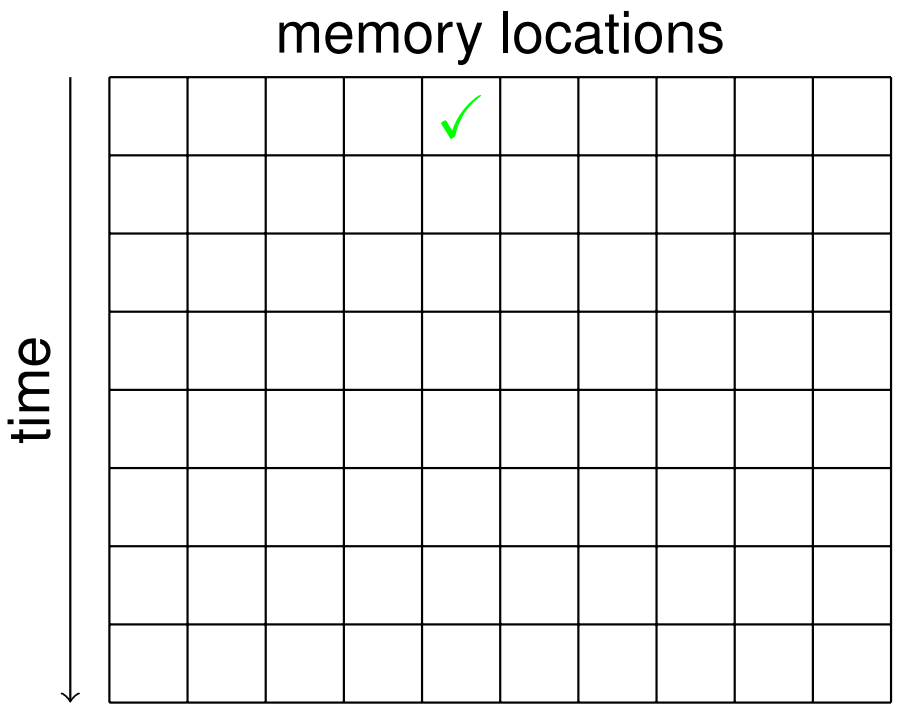
1. **fault** – cause of an error (e.g., mistake in coding)
2. **error** – incorrect state that may lead to failure
3. **failure** – deviation from specified/desired behaviour

Terminology: Fault, Error, Failure

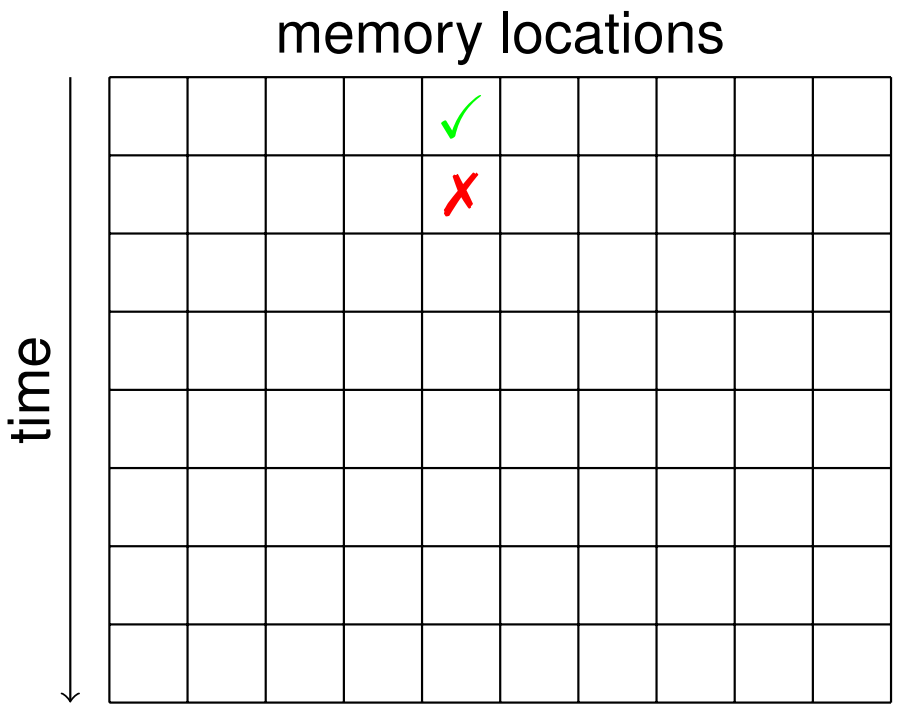
1. **fault** – cause of an error (e.g., mistake in coding)
2. **error** – incorrect state that may lead to failure
3. **failure** – deviation from specified/desired behaviour

(Standardised terminology: IEEE 610.12-1990)

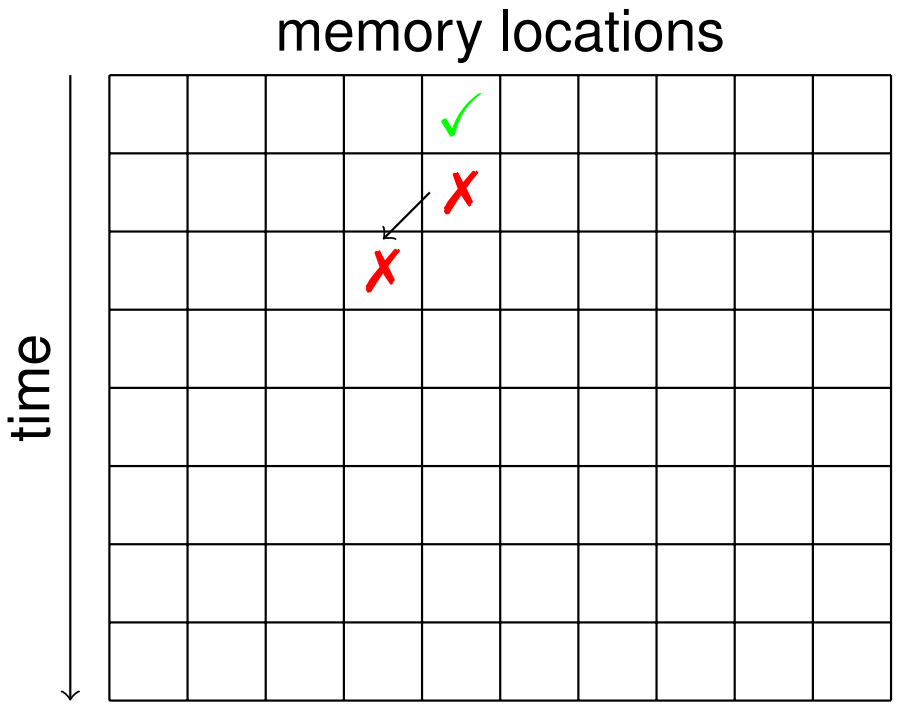
Terminology: Fault, Error, Failure



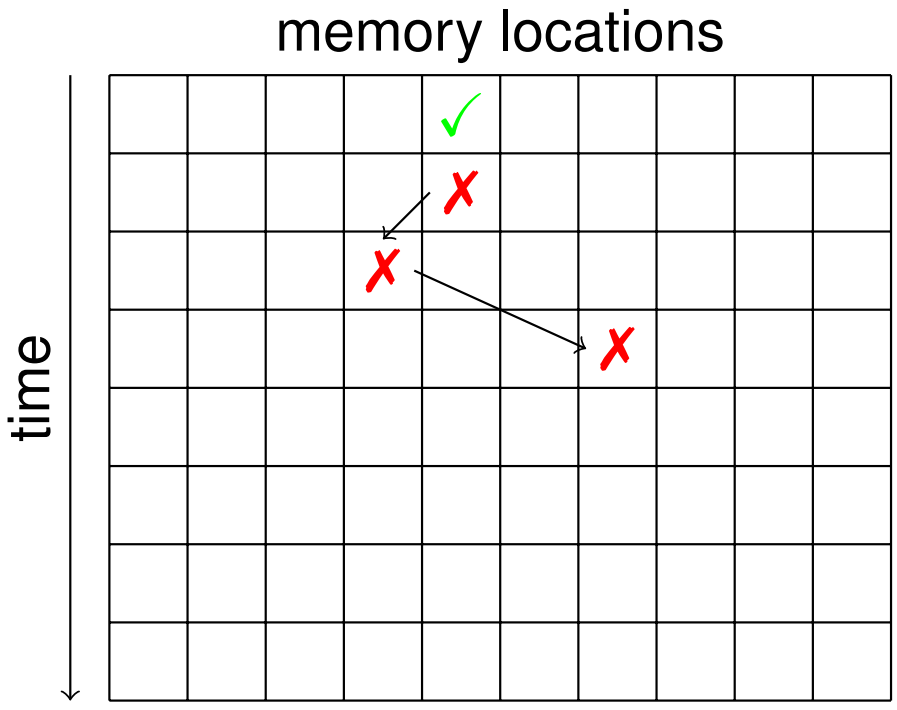
Terminology: Fault, Error, Failure



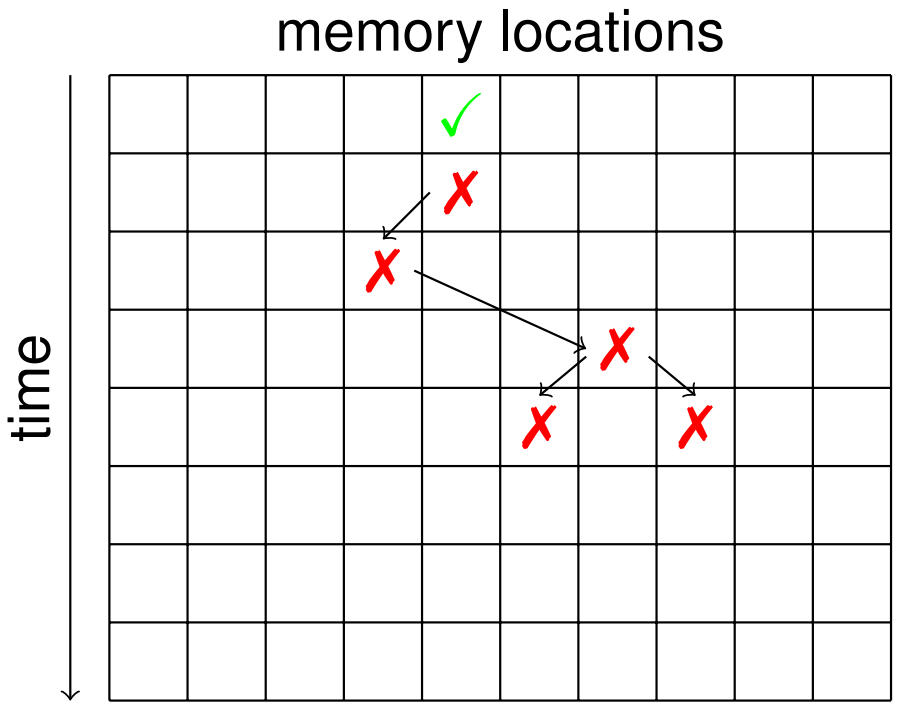
Terminology: Fault, Error, Failure



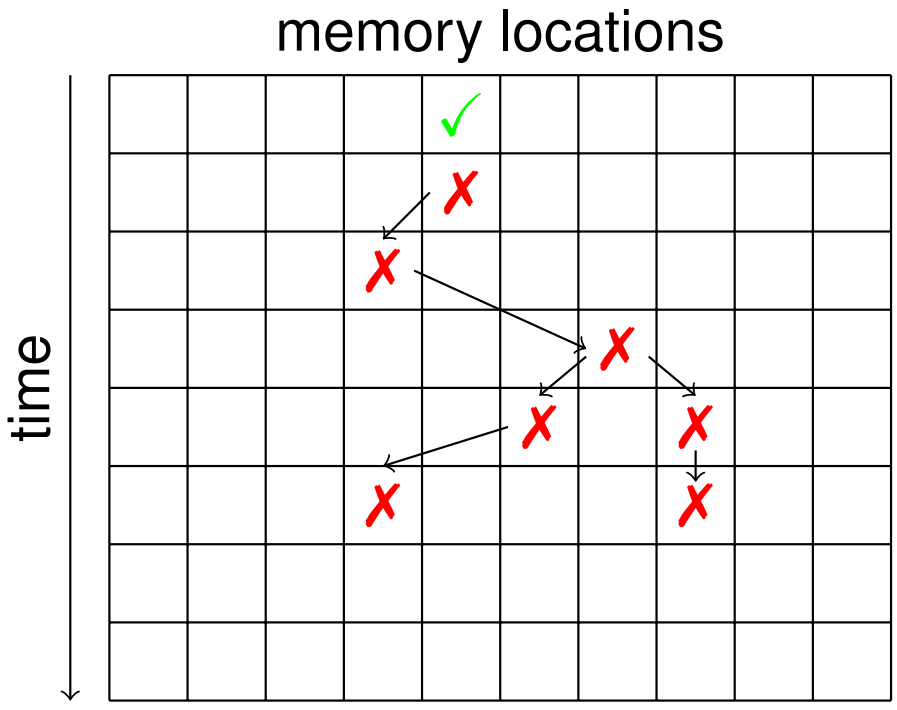
Terminology: Fault, Error, Failure



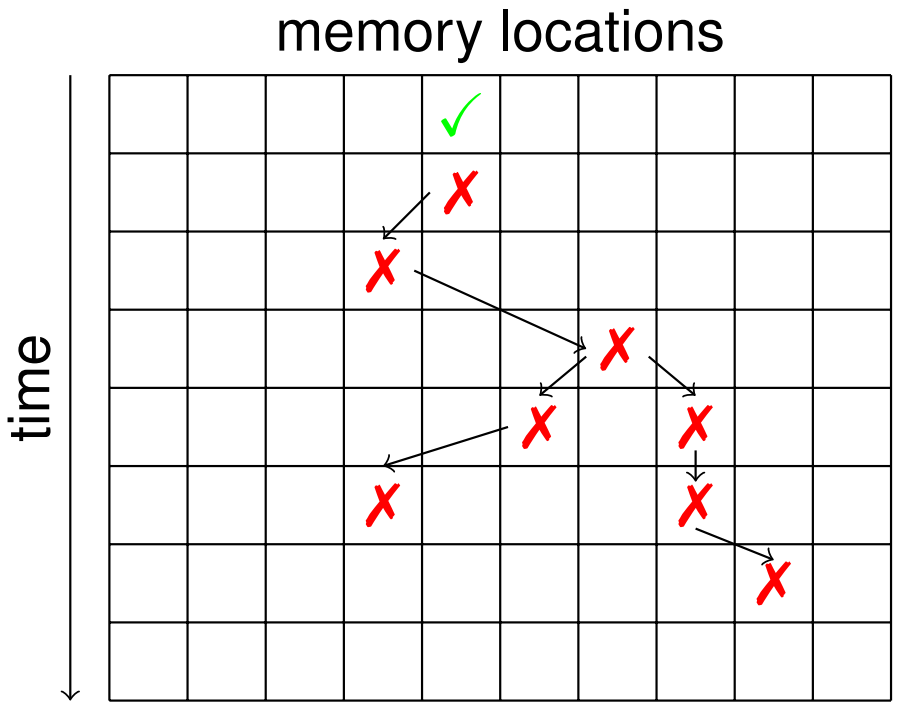
Terminology: Fault, Error, Failure



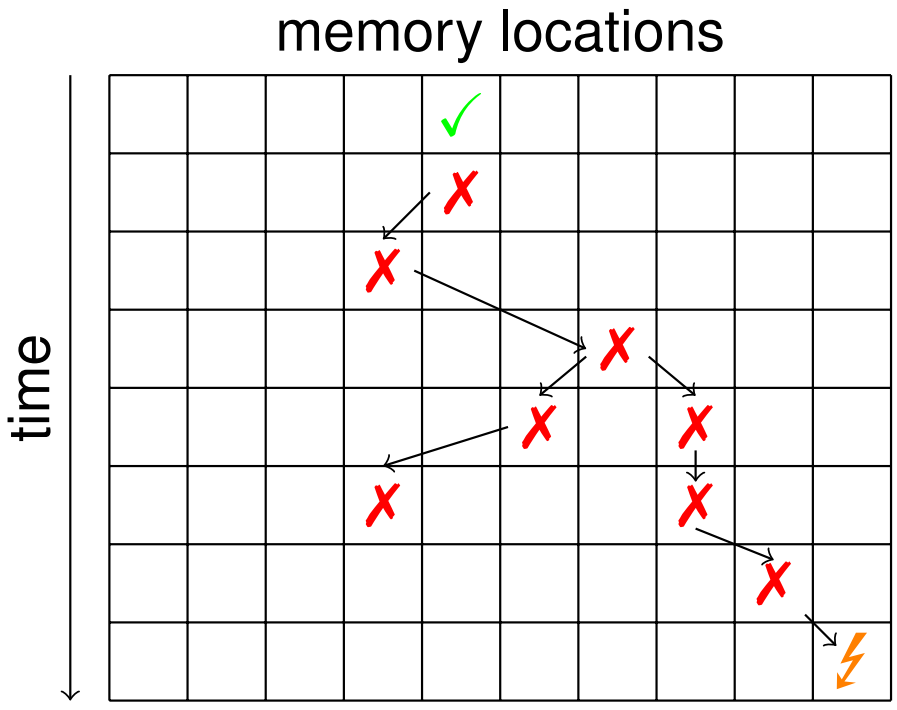
Terminology: Fault, Error, Failure



Terminology: Fault, Error, Failure



Terminology: Fault, Error, Failure



Terminology: Fault, Error, Failure

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

unsigned count (char* str, char elem)
{
  unsigned i, c=0;
  for (i = 1; i <= strlen (str); i++)
  {
    if (str[i] == elem)
      c++;
  }
  return c;
}

int main(int argc, char** argv)
{
  printf ("%d\n", count ("xyzyx", 'x'));
  return 0;
}
```

Not every fault results in failure!

```
int power (int x, int y) { int r = y * y; return r; }
```

Not every fault results in failure!

```
int power (int x, int y) { int r = y * y; return r; }
```

- `power (2, 2)` $= 2 \cdot 2 = 2^2 \checkmark$

Not every fault results in failure!

```
int power (int x, int y) { int r = y * y; return r; }
```

- `power (2, 2)` $= 2 \cdot 2 = 2^2 \checkmark$
- `power (2, 4)` $= 4 \cdot 4 = 2^4 \checkmark$

Not every fault results in failure!

```
int power (int x, int y) { int r = y * y; return r; }
```

- `power (2, 2)` $= 2 \cdot 2 = 2^2$ ✓
- `power (2, 4)` $= 4 \cdot 4 = 2^4$ ✓
- `power (1, 1)` $= 1 \cdot 1 = 1^1$ ✓

Not every fault results in failure!

```
int power (int x, int y) { int r = y * y; return r; }
```

- `power (2, 2)` $= 2 \cdot 2 = 2^2$ ✓
- `power (2, 4)` $= 4 \cdot 4 = 2^4$ ✓
- `power (1, 1)` $= 1 \cdot 1 = 1^1$ ✓
- `power (2, 5)` $= 5 \cdot 5 \neq 2^5$ ↯

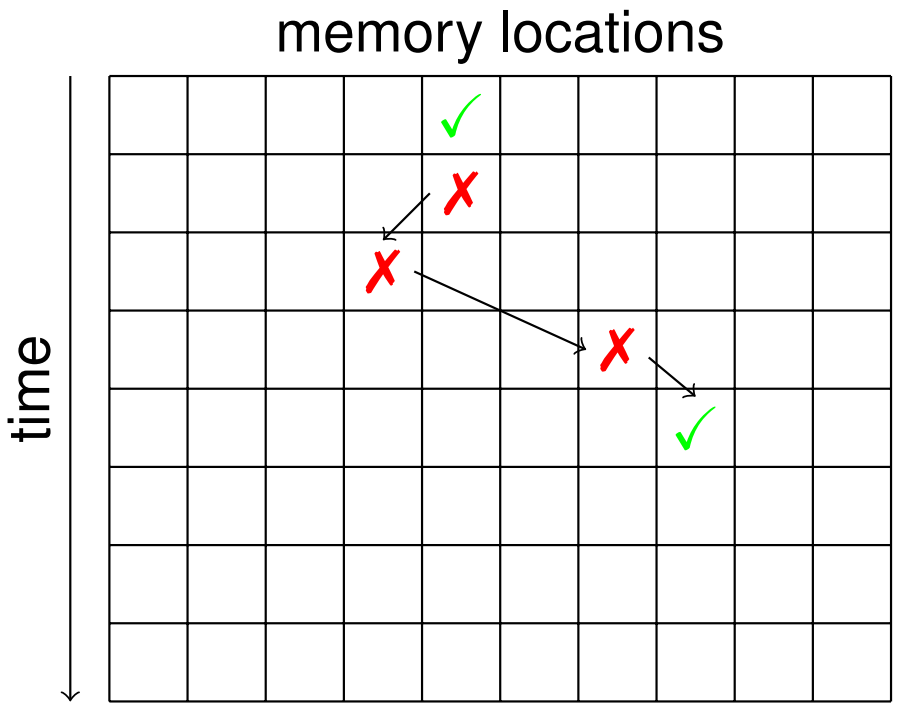
Not every fault results in failure!

```
int power (int x, int y) { int r = y * y; return r; }
```

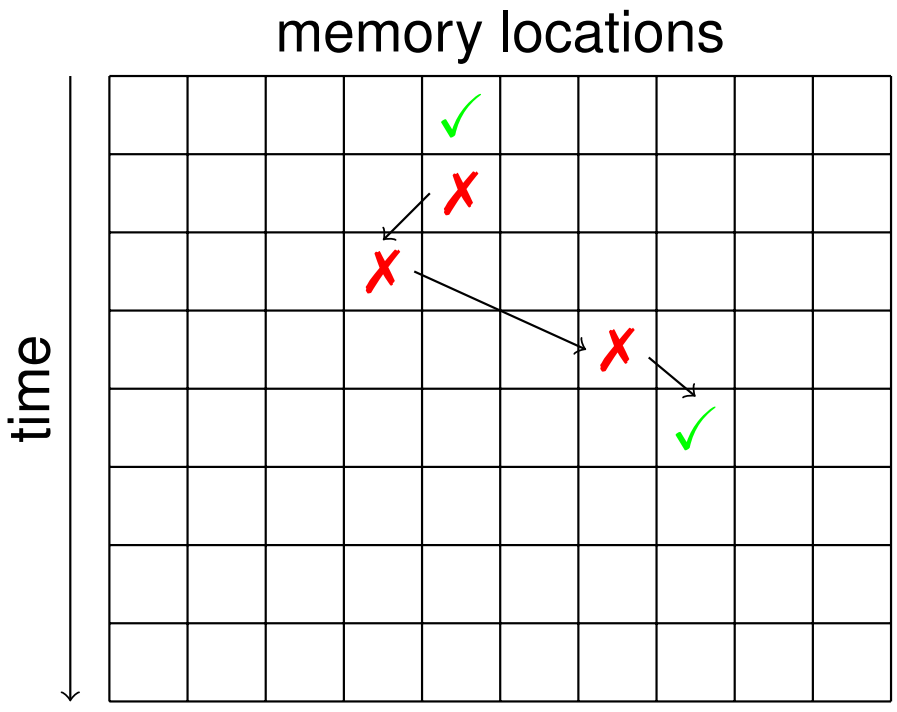
- `power (2, 2)` $= 2 \cdot 2 = 2^2$ ✓
- `power (2, 4)` $= 4 \cdot 4 = 2^4$ ✓
- `power (1, 1)` $= 1 \cdot 1 = 1^1$ ✓
- `power (2, 5)` $= 5 \cdot 5 \neq 2^5$ ↯

Fault is not *triggered* in first 3 cases!

Not every fault results in failure!



Not every fault results in failure!



Error is not *propagated*!

Causes and Symptoms

- So what exactly *causes* the problem?

cause: **fault**, symptom: { **error** / **failure**

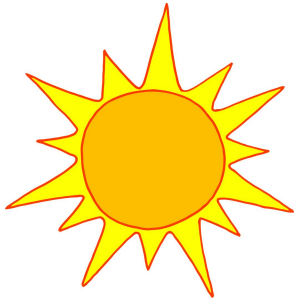
Causes and Symptoms

- Attempt of a more formal definition:
  - Event A is a necessary cause of effect B if the presence of B implies the presence of A.

Locating the cause is non-trivial

Rain in April

Campfire in April



Sun in May

Campfire in June

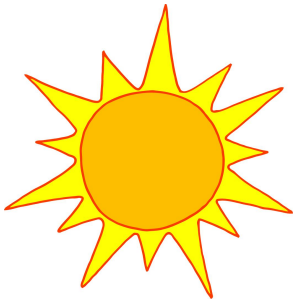
Causes and Symptoms

cause: **campfire**, symptom: { **fire spreads**, **wildfire**

Locating the cause is non-trivial

Rain in April

Campfire in April



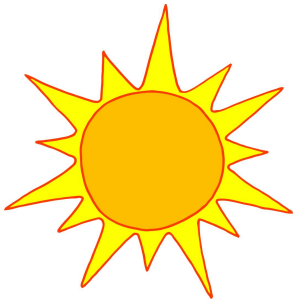
Sun in May

Campfire in June

Locating the cause is non-trivial

Rain in April

Campfire in April



Sun in May

Campfire in June

Locating the cause is non-trivial

Rain in April

Campfire in April

Wildfire (in April)

Locating the cause

- If it hadn't rained in April,
  there would not have been a wildfire in June
- Did the rain cause the wildfire in June?

Locating the cause

1. **fault** – cause of an error (e.g., mistake in coding)
2. **error** – incorrect state that may lead to failure
3. **failure** – deviation from specified/desired behaviour

Locating the cause

1. **fault** – cause of an error (e.g., mistake in coding)
2. **error** – incorrect state that may lead to failure
3. **failure** – deviation from specified/desired behaviour

Locate transition from *correct* to *incorrect*

Locating the cause

Locate transition from *correct* to *incorrect*

- What is correct, what is incorrect?
- Depends on programmer's intention (often implicit!)

Locating the cause

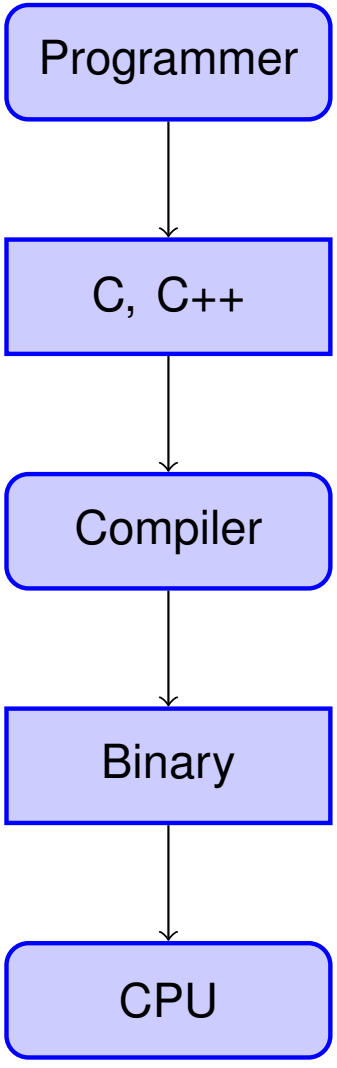
Locate transition from *correct* to *incorrect*

- What is correct, what is incorrect?
- Depends on programmer's intention (often implicit!)
- State your intention!

Outline of Lecture

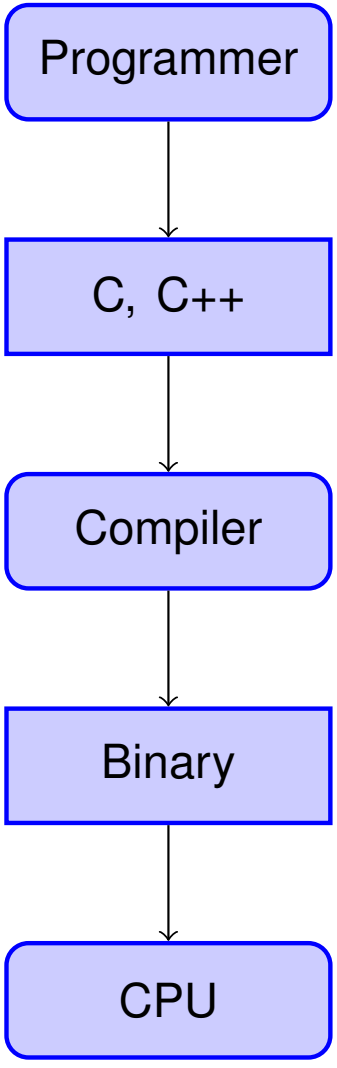
- What is a bug?
  - Classes of Bugs
  - Cause and Symptom
- **What do we need to understand bugs?**
  - Understand the Program
  - Know the Programmer's Intentions

Understanding Programs



- Programmer expresses *intention* in C/C++
- Compiler translates program to binary
- Processor executes (interprets) the binary

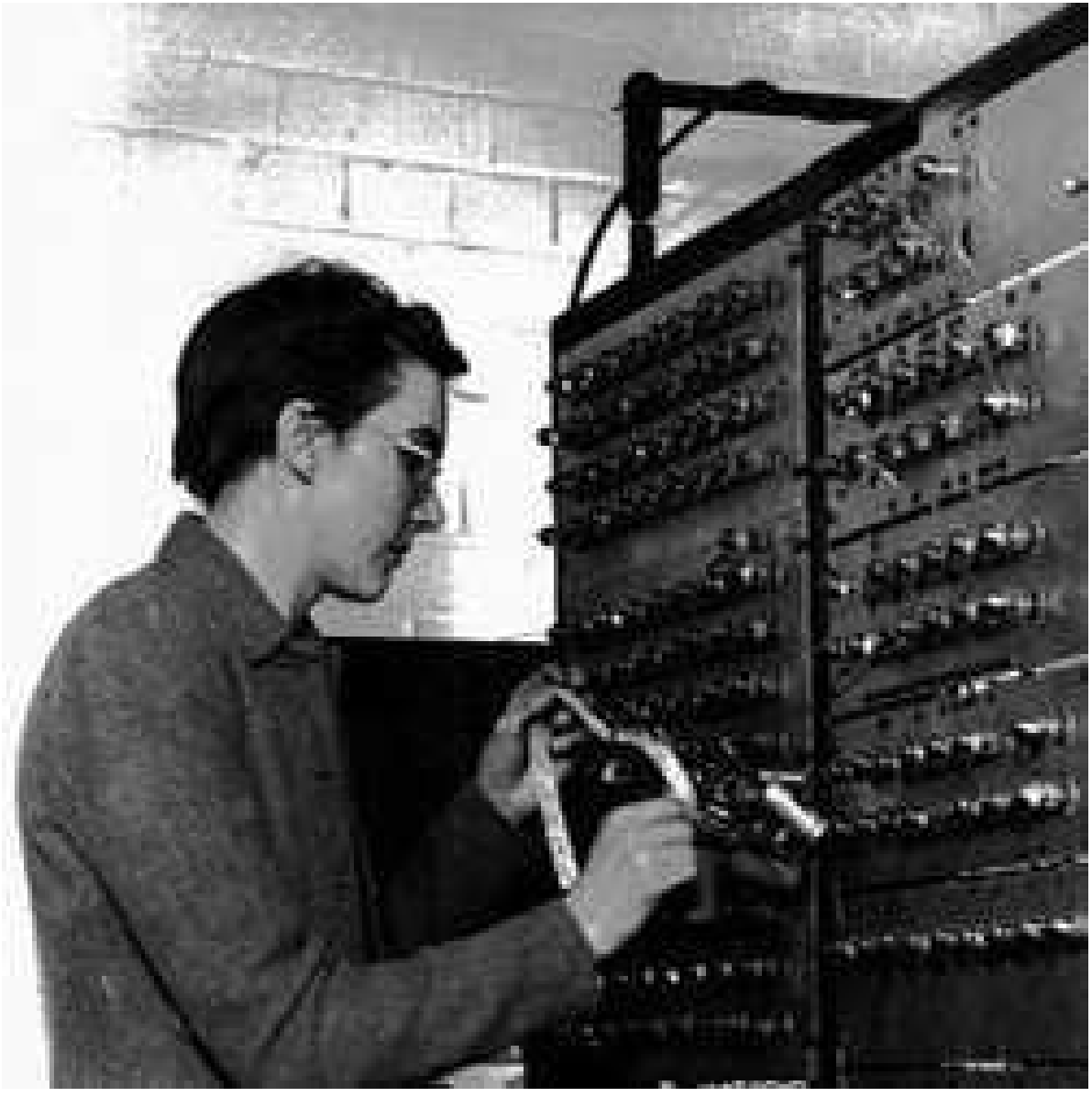
Understanding Programs



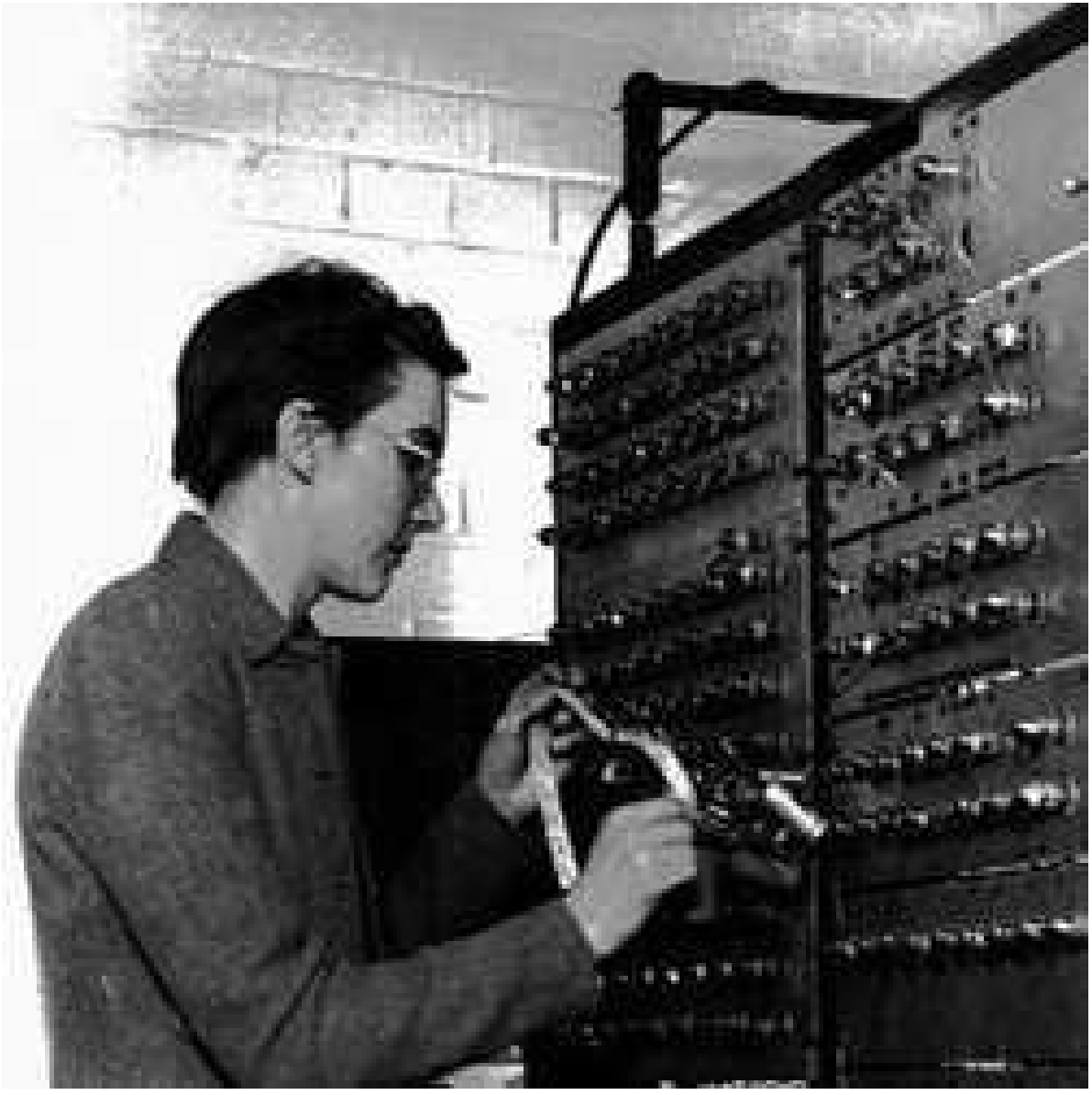
- Programmer expresses *intention* in C/C++
- Compiler translates program to binary
- Processor executes (interprets) the binary

Programmer, compiler, CPU need to agree on *semantics*

Definition of Programming Languages



Definition of Programming Languages



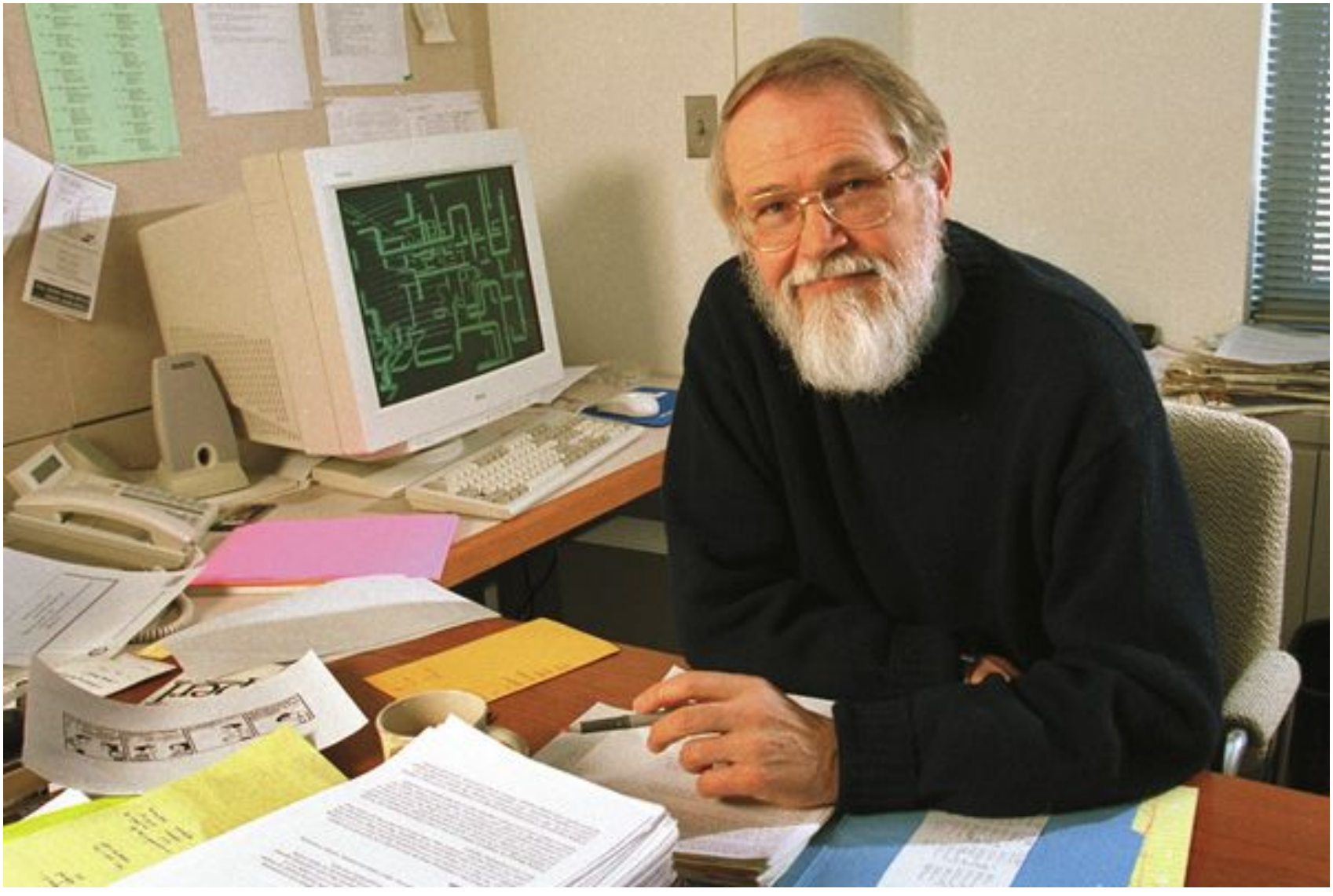
Kathleen Booth (Birbeck College): Inventor of Assembler

Definition of Programming Languages

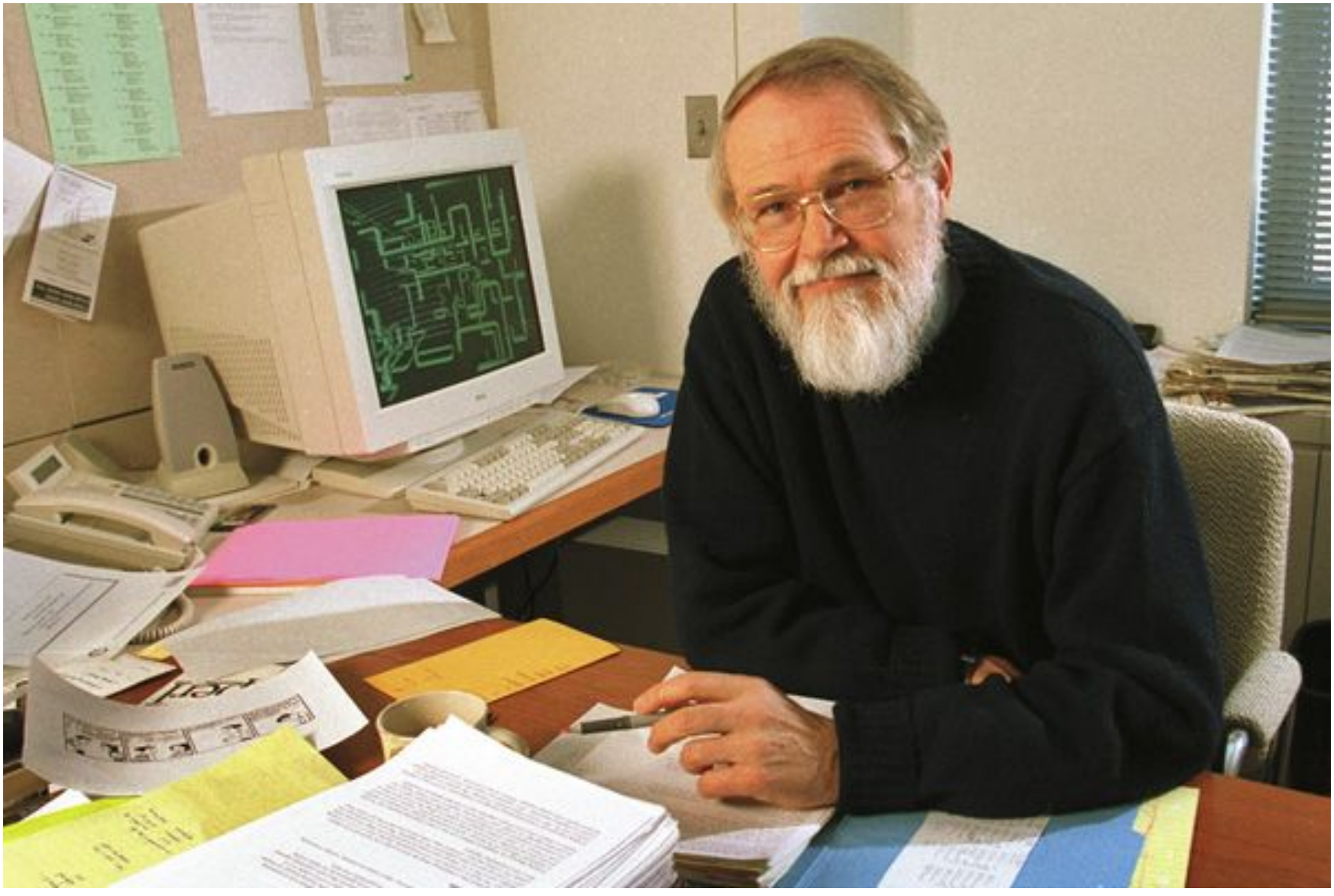
Definition of Programming Languages

Grace Murray Hopper (Navy Reserves): Inventor of COBOL

Definition of Programming Languages

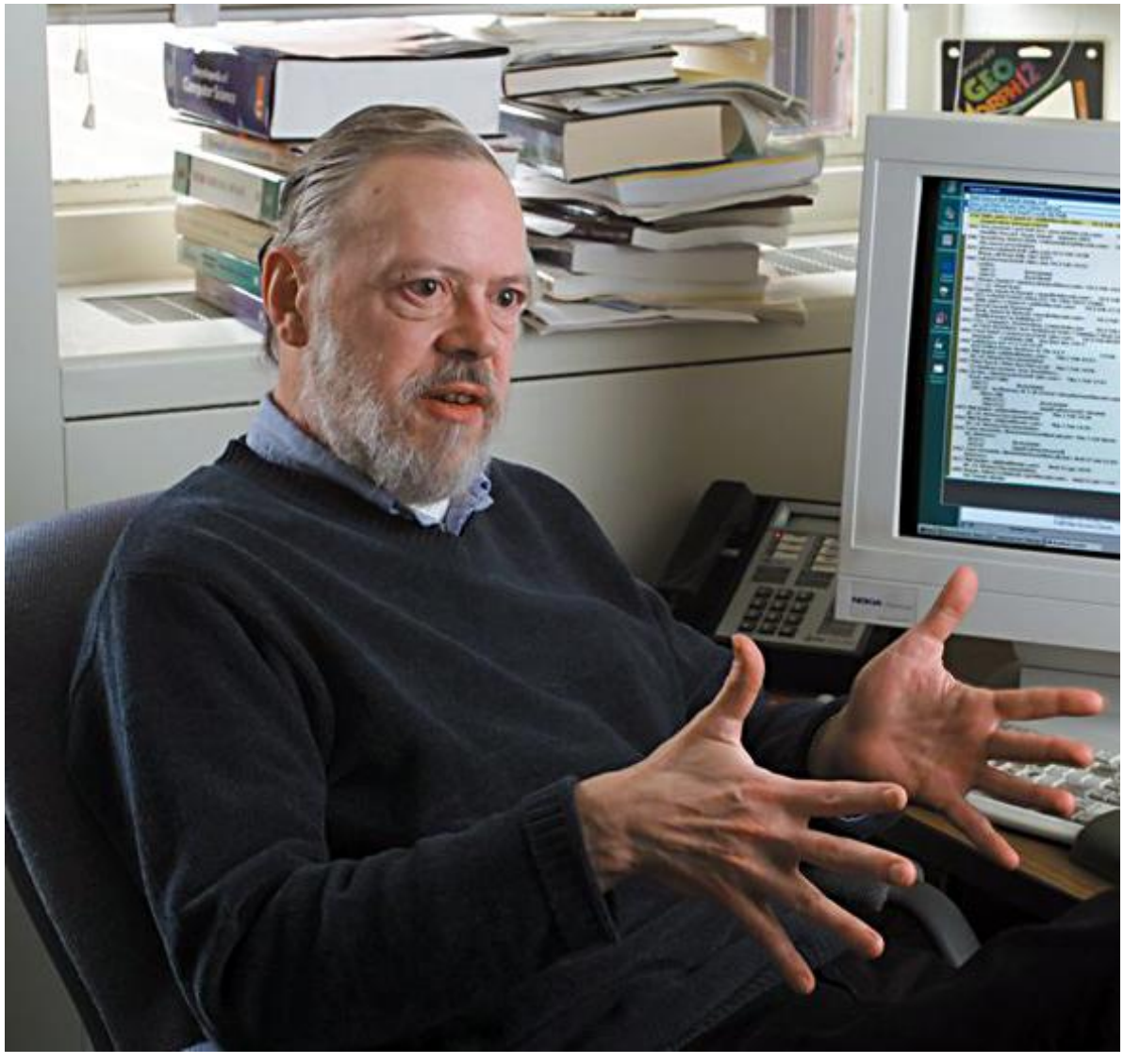


Definition of Programming Languages

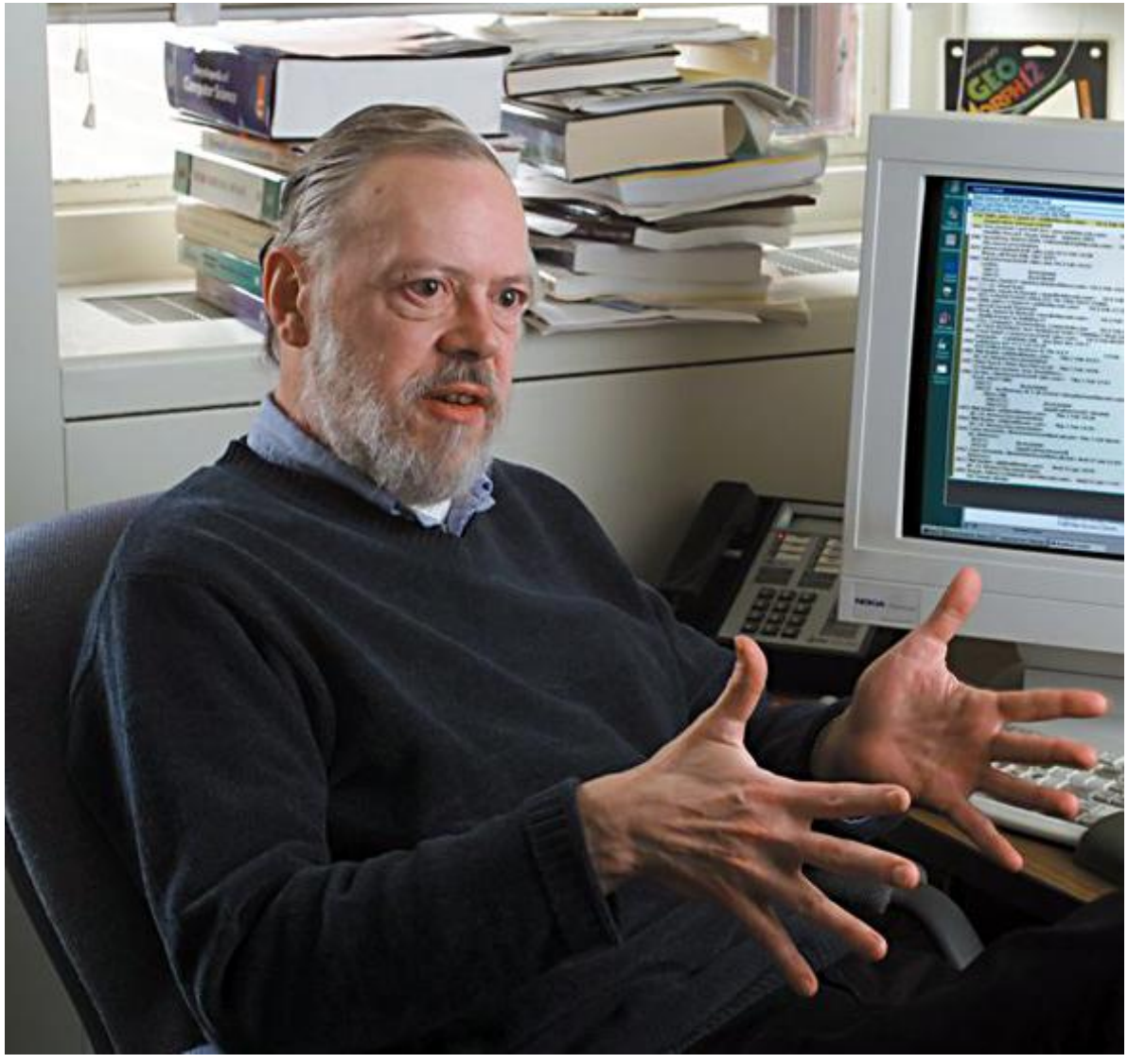


Brian Kernighan (now Princeton, then Bell Labs)

Definition of Programming Languages



Definition of Programming Languages



Dennis Ritchie, (Lucent, Bell Labs) † Oct 2011

Definition of Programming Languages

Definition of Programming Languages

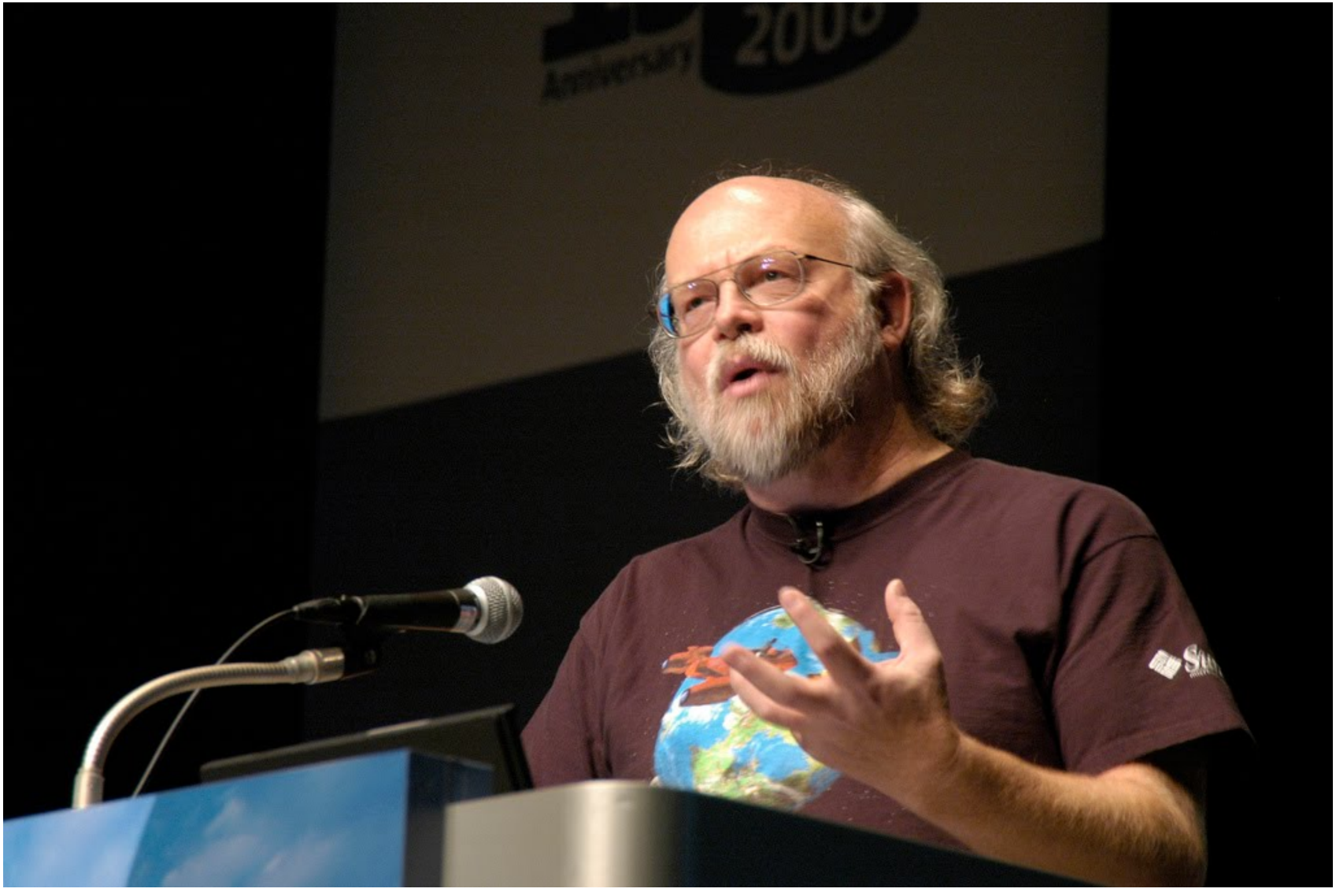
Barbara Liskov (MIT): Inventor of CLU and Argus

Definition of Programming Languages

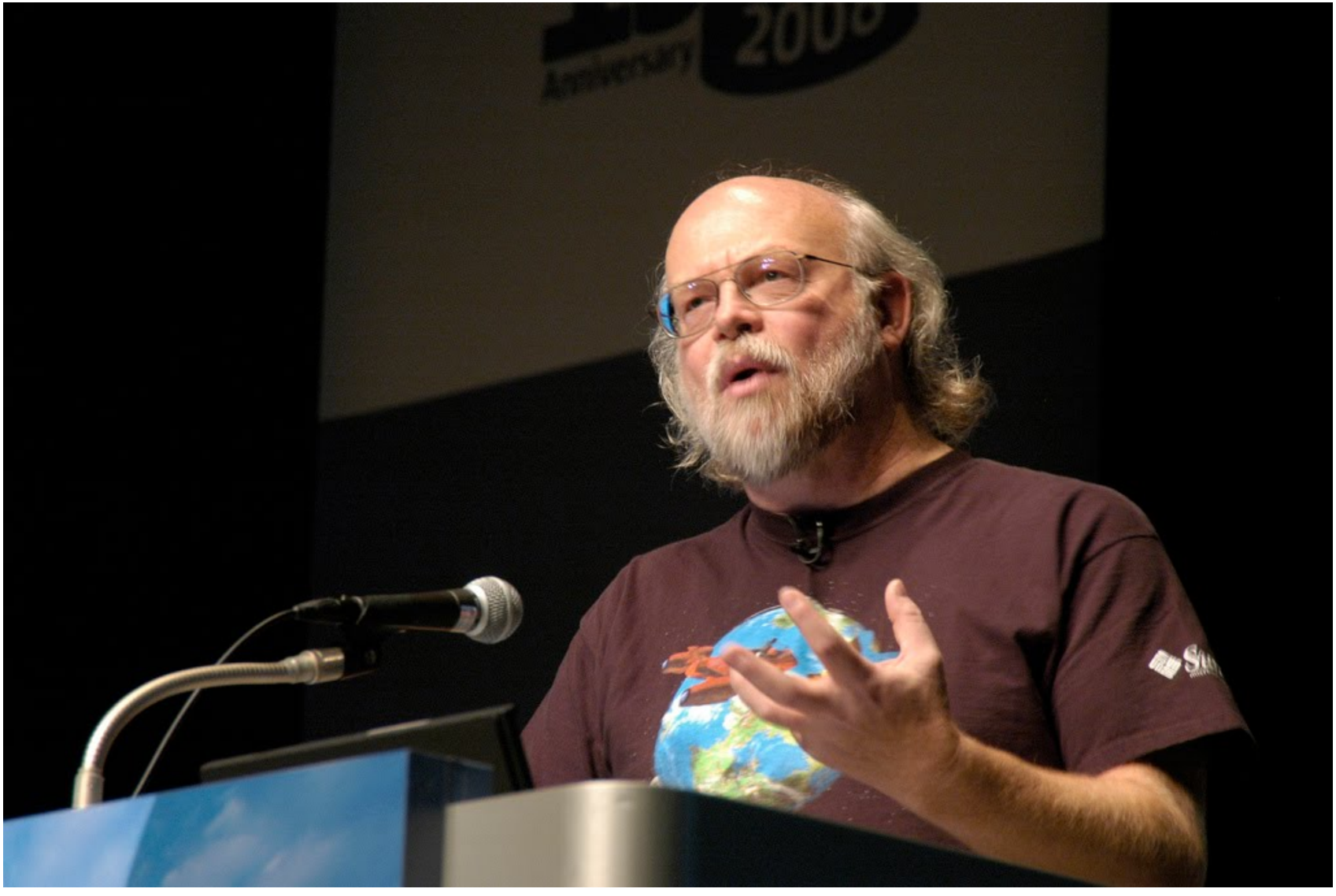
Definition of Programming Languages

Bjarne Stroustrup, now Texas A&M Univ., then AT&T

Definition of Programming Languages



Definition of Programming Languages



James Gosling, now Typesafe Inc., then Sun Microsystems

Definition of Programming Languages

Definition of Programming Languages

Anders Hejlsberg (Microsoft)

Programming Language Standards

- C (ISO/IEC 9899:2011)
  - `open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`
- C++ (ISO/IEC 14882:2011)
  - `open-std.org/JTC1/SC22/WG21/docs/papers/2011/n3242.pdf`
- Java SE 7
  - `docs.oracle.com/javase/specs/`
- C#
  - `http://www.ecma-international.org/publications/standards/Ecma-334.htm`

Programming Language Standards

- C++ expressions defined by ISO/IEC 14882:2011, §5
  - e.g., syntax for *multiplicative expressions* (§5.6):

    multiplicative-expression:
    pm-expression
    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. [. . .]"

Semantics of Programs

```c
#include <stdio.h>

int main (int argc, char** argv)
{
  int c = 2147483642;

  while ((c+1) > c)
  {
    printf ("%d\n", c);
    c++;
  }

  return 0;
}
```

Semantics of Programs

```
while ((c+1) > c)
{
  printf ("...", c);
  c++;
}
```

Semantics of Programs

```
while ((c+1) > c)
{
  printf ("...", c);
  c++;
}
```

Semantics of Programs

```
while ((c+1) > c)
{
  printf ("...", c);
  c++;
}
```

- ISO/IEC 14882:2011 §5.7 (Additive Operators)

  “3 The result of the binary + operator is the sum of the operands.”

Semantics of Programs

```
while ((c+1) > c)
{
  printf ("...", c);
  c++;
}
```

- ISO/IEC 14882:2011 §5.7 (Additive Operators)

  “3 The result of the binary + operator is the sum of the operands.”

- ISO/IEC 14882:2011 §5.9 (Relational Operators)

  “The operators $<$ (less than), $>$ (greater than), [...] all yield `false` or `true`.”

Semantics of Programs

```
while ((c+1) > c)
{
  printf ("...", c);
  c++;
}
```

- ISO/IEC 14882:2011 §5.7 (Additive Operators)

  “3 The result of the binary + operator is the sum of the operands.”
- ISO/IEC 14882:2011 §5.9 (Relational Operators)

  “The operators $<$ (less than), $>$ (greater than), [. . .] all yield `false` or `true`.”
- ISO/IEC 14882:2011 §5 (Expressions)

  “4 If during the evaluation of an expression, the result is [. . .] not in the range of representable values for its type, the behavior is undefined.”

“Undefined” semantics

- Here, undefined means “compiler-dependent” (rather than undefined at run-time)

“Undefined” semantics

- Here, undefined means “compiler-dependent”
  (rather than undefined at run-time)
- Optimiser in `gcc/g++` takes advantage under-specification
  - simplifies `((c+1)>c)` to `true`

“Undefined” semantics

- Here, undefined means “compiler-dependent”
  (rather than undefined at run-time)
- Optimiser in `gcc/g++` takes advantage under-specification
  - simplifies `((c+1)>c)` to `true`
- In debugging mode, `gcc/g++` doesn’t apply optimisations
  - `((c+1)>c)` evaluates to `false` if `c == INT_MAX`

“Undefined” semantics

- Here, undefined means “compiler-dependent”
  (rather than undefined at run-time)
- Optimiser in `gcc/g++` takes advantage under-specification
  - simplifies `((c+1)>c)` to `true`
- In debugging mode, `gcc/g++` doesn’t apply optimisations
  - `((c+1)>c)` evaluates to `false` if `c == INT_MAX`
- Turning debugger on results in *Heisenbug*

Semantics of parallel programs

```
#include <stdio.h>
#include <pthread.h>

int c = 0;
void *count (void *parg)
{
  for (unsigned i=0; i<5000000; i++)
    c++;
}

int main (int argc, char** argv)
{
  pthread_t thread1, thread2;
  pthread_create (&thread1, NULL, count, NULL);
  pthread_create (&thread2, NULL, count, NULL);
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf ("%d\n", c);
  return 0;
}
```

Semantics of parallel programs

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  “3 [. . .] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other”

Semantics of parallel programs

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  “3 [. . .] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other”

- ISO/IEC 14882:2011 §1.10
  (Multi-threaded executions and data races)

  “3 [. . .] Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location.”

Semantics of parallel programs

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  “3 [. . .] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other”

- ISO/IEC 14882:2011 §1.10 (Multi-threaded executions and data races)

  “3 [. . .] Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location.”

Semantics of parallel programs

- ISO/IEC 14882:2011 §1.7 (The C++ Memory Model)

  “3 [. . .] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other”

- ISO/IEC 14882:2011 §1.10 (Multi-threaded executions and data races)

  “3 [. . .] Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location.”

Semantics of parallel programs

- ISO/IEC 14882:2011 §1.10
  (Multi-threaded executions and data races)

  "14 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither of them happens before the other. Any such data race results in undefined behavior."

Semantics of parallel programs

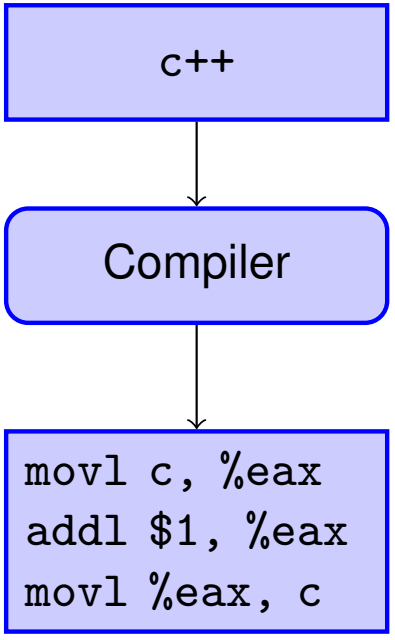
- Again, *undefined* means *compiler-dependent*

Semantics of parallel programs

- Again, *undefined* means *compiler-dependent*
- `gcc -S threads.c`

Semantics of parallel programs

- Again, *undefined* means *compiler-dependent*
- `gcc -S threads.c`



Semantics of parallel programs

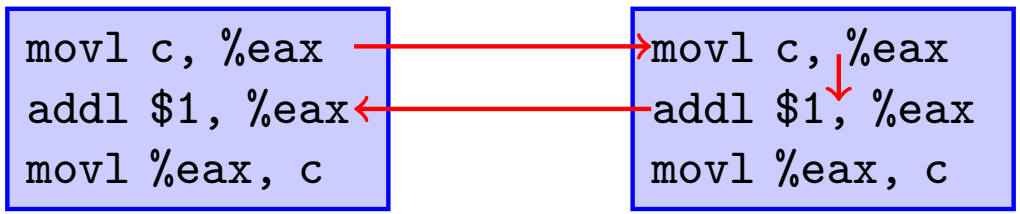
```
movl c, %eax
addl $1, %eax
movl %eax, c
```

```
movl c, %eax
addl $1, %eax
movl %eax, c
```

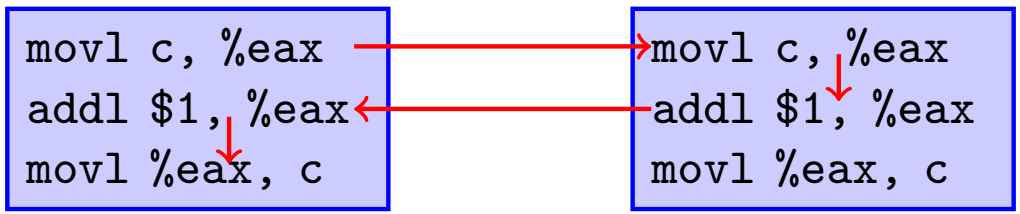
Semantics of parallel programs

Semantics of parallel programs

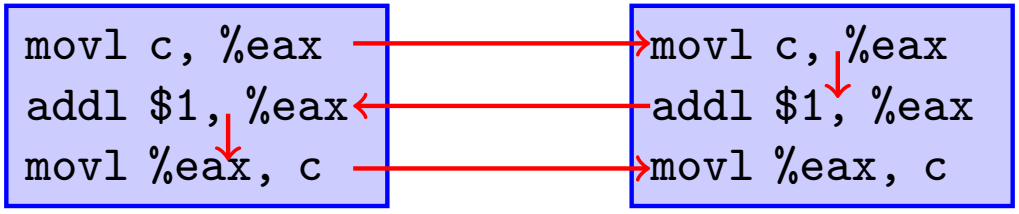
Semantics of parallel programs



Semantics of parallel programs



Semantics of parallel programs



Semantics of high-level programming languages

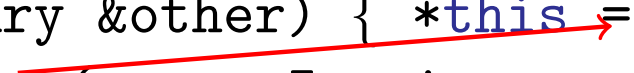
```
class Imaginary {
public:
  float r; float i;

  Imaginary (): r(0), i(0) { }
  Imaginary (Imaginary &other) { *this = other; }
  Imaginary operator= (const Imaginary other)
  {
    r = other.r; i = other.i;
  }

};

int main (int argc, char** argv)
{
  Imaginary i;
  Imaginary j = i;
  return j.i;
}
```

```
Imaginary (Imaginary &other) { *this = other; }
Imaginary operator= (const Imaginary  other)
{
  r = other.r; i = other.i;
}
```

- C++ allows redefinition of operators such as = (assignment)

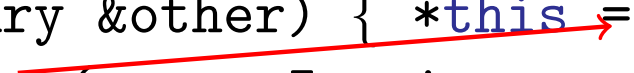
Semantics of high-level programming languages

```
Imaginary (Imaginary &other) { *this = other; }
Imaginary operator= (const Imaginary  other)
{
  r = other.r; i = other.i;
}
```



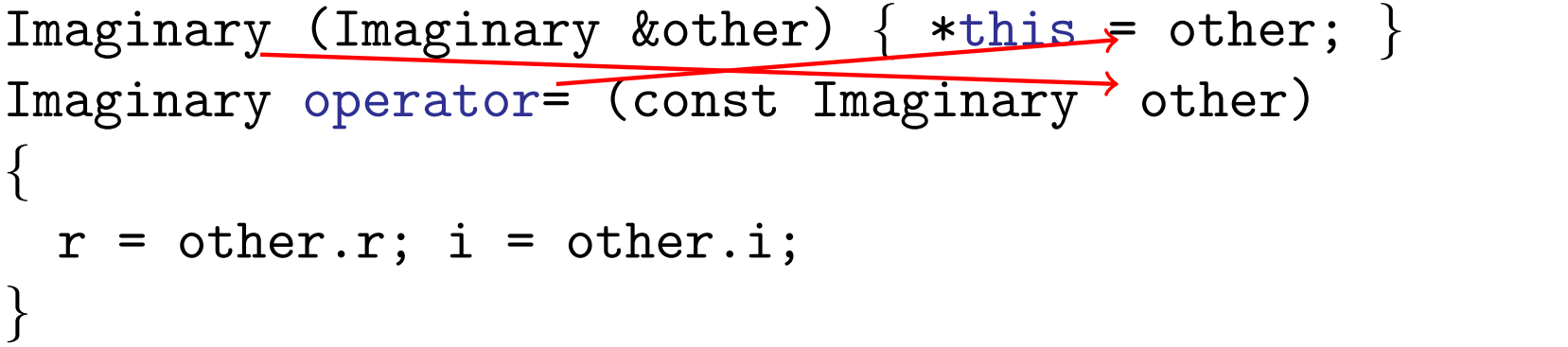
- C++ allows redefinition of operators such as = (assignment)

```
Imaginary (Imaginary &other) { *this = other; }
Imaginary operator= (const Imaginary  other)
{
  r = other.r; i = other.i;
}
```



- C++ allows redefinition of operators such as = (assignment)
- C++ allows definition *copy constructor*

Semantics of high-level programming languages



- C++ allows redefinition of operators such as = (assignment)
- C++ allows definition *copy constructor*

```
Imaginary (Imaginary &other) { *this = other; }
Imaginary operator= (const Imaginary other)
{
  r = other.r; i = other.i;
}
```

- C++ allows redefinition of operators such as = (assignment)
- C++ allows definition *copy constructor*
- Unintentional *mutual recursion*

Semantics of high-level programming languages

```
Imaginary (Imaginary &other) { *this = other; }
Imaginary operator= (const Imaginary  other)
{
  r = other.r; i = other.i;
}
```

- C++ allows redefinition of operators such as = (assignment)
- C++ allows definition *copy constructor*
- Unintentional *mutual recursion*
- Fix: use reference `&`

```
Imaginary (Imaginary &other) { *this = other; }
Imaginary operator= (const Imaginary &other)
{
  r = other.r; i = other.i;
}
```

- C++ allows redefinition of operators such as = (assignment)
- C++ allows definition *copy constructor*
- Unintentional *mutual recursion*
- Fix: use reference &

Formal Semantics of Programming Languages

Language standards describe semantics *informally* in English.
This can be ambigous.

Formal Semantics prevent ambiguity:

- **Operational.** Defines steps of the computation (on a machine) the program makes when executed (cf. *transition relation*)
- **Denotational.** Mathematical function maps program structures to respective meaning
- **Axiomatic.** Meaning determined indirectly by means of assertions (defined via axioms and rules, cf. Hoare Logic)

Operational Semantics (Example)

ISO/IEC 14882:2011 §5.7 (Additive Operators)

"3 The result of the binary + operator is the sum of the operands."

- *Structural (small-step) operational semantics* is syntax-oriented.
- Defined by means of inference rules:

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)}$$

$$\frac{}{\langle n_1 + n_2, \sigma \rangle \to n} \text{ (where } n \equiv n_1 + n_2 \pmod{64}\text{)}$$

Small-Step Semantics

- $\rightarrow$ specifies one step of the program
- $\langle c, \sigma \rangle$ is a *configuration*:
  - c is a command (or expression)
  - σ is a store
- $\langle c, \sigma \rangle \rightarrow \langle c', \sigma \rangle$ reduces configuration $\langle c, \sigma \rangle$ to $\langle c', \sigma' \rangle$
- $\rightarrow^*$ represents 0 or more steps of $\rightarrow$
- $\langle \texttt{skip}, \sigma \rangle$ is the final configuration

Small-Step Semantics Rules

Evaluation of variables:

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)}$$

Arithmetic reduction:

$$\frac{}{\langle n_1 + n_2, \sigma \rangle \to n} \text{ (where } n \equiv n_1 + n_2 \pmod{64}\text{)}$$

Rule with premises:

$$\frac{\langle x, \sigma \rangle \to n_1 \quad \langle y, \sigma \rangle \to n_2}{\langle x + y, \sigma \rangle \to \langle n_1 + n_2, \sigma \rangle}$$

Small-Step Semantics

Assignments:

$$\frac{}{\langle x := n, \sigma \rangle \to \langle \texttt{skip}, \sigma[x \mapsto n] \rangle}$$

Substitution in assignments:

$$\frac{\langle e, \sigma \rangle \to^* n}{\langle x := e, \sigma \rangle \to \langle x := n, \sigma \rangle}$$

Skip rule:

$$\frac{}{\langle \texttt{skip}; c, \sigma \rangle \to \langle c, \sigma \rangle}$$

Sequential execution:

$$\frac{\langle c_0, \sigma \rangle \to \langle c_0', \sigma \rangle}{\langle c_0;\ c_1, \sigma \rangle \to \langle c_0';\ c_1, \sigma \rangle}$$

Small-Step Semantics (Example)

Evaluate $\langle z := x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle$

Small-Step Semantics (Example)

$$\text{Evaluate } \langle z := x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle$$

1. Evaluate variables:

$$\frac{\langle x, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 3 \quad \langle y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 2}{\langle x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to \langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle}$$

Small-Step Semantics (Example)

$$\text{Evaluate } \langle z := x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle$$

1. Evaluate variables:

$$\frac{\langle x, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 3 \quad \langle y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 2}{\langle x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to \langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle}$$

2. Reduce arithmetic expression:

$$\frac{}{\langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 5}$$

Small-Step Semantics (Example)

$$\text{Evaluate } \langle z := x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle$$

1. Evaluate variables:

$$\frac{\langle x, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 3 \quad \langle y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 2}{\langle x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to \langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle}$$

2. Reduce arithmetic expression:

$$\frac{}{\langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 5}$$

3. Substitution:

$$\frac{\langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 5}{\langle z := x + y, \sigma\rangle \to \langle z := 5, \sigma\rangle}$$

Small-Step Semantics (Example)

$$\text{Evaluate } \langle z := x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle$$

1. Evaluate variables:

$$\frac{\langle x, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 3 \quad \langle y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 2}{\langle x + y, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to \langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle}$$

2. Reduce arithmetic expression:

$$\frac{}{\langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 5}$$

3. Substitution:

$$\frac{\langle 3 + 2, \{x \mapsto 3, y \mapsto 2, z \mapsto 1\}\rangle \to 5}{\langle z := x + y, \sigma\rangle \to \langle z := 5, \sigma\rangle}$$

4. Assignment:

$$\frac{}{\langle z := 5, \{\ldots, z \mapsto 1\}\rangle \to \langle \texttt{skip}, \{\ldots, z \mapsto 5\}\rangle}$$

Small-Step Semantics for Programming Constructs

Conditionals:

$$\frac{\langle b, \sigma \rangle \to b'}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma \rangle \to \langle \texttt{if } b' \texttt{ then } c_0 \texttt{ else } c_1, \sigma \rangle}$$

$$\frac{}{\langle \texttt{if true then } c_0 \texttt{ else } c_1, \sigma \rangle \to \langle c_0, \sigma \rangle}$$

$$\frac{}{\langle \texttt{if false then } c_0 \texttt{ else } c_1, \sigma \rangle \to \langle c_1, \sigma \rangle}$$

Small-Step Semantics for Programming Constructs

Conditionals:

$$\frac{\langle b, \sigma\rangle \to b'}{\langle \texttt{if } b \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle \to \langle \texttt{if } b' \texttt{ then } c_0 \texttt{ else } c_1, \sigma\rangle}$$

$$\frac{}{\langle \texttt{if true then } c_0 \texttt{ else } c_1, \sigma\rangle \to \langle c_0, \sigma\rangle}$$

$$\frac{}{\langle \texttt{if false then } c_0 \texttt{ else } c_1, \sigma\rangle \to \langle c_1, \sigma\rangle}$$

Loops:

$$\frac{}{\langle \texttt{while } b \texttt{ do } c, \sigma\rangle \to \langle \texttt{if } b \textit{ then } (c; \texttt{while } b \texttt{ do } c) \texttt{ else skip}, \sigma\rangle}$$

Small-Step Semantics

- Rules determine how to evaluate program c in given store σ
- Alternative: *Big-Step* Semantics:

$$\langle c, \sigma \rangle \Downarrow \sigma'$$

  - σ' is the store of final configuration $\langle \texttt{skip}, \sigma \rangle$
  - Relation between Big-Step and Small-Step Operational Semantics:

$$\langle c, \sigma \rangle \Downarrow \sigma' \quad \Leftrightarrow \quad \langle c, \sigma \rangle \rightarrow^* \langle \texttt{skip}, \sigma' \rangle$$

Outline of Lecture

- What is a bug?
  - Classes of Bugs
  - Cause and Symptom
- What do we need to understand bugs?
  - Understand the Program
  - **Know the Programmer's Intentions**

What is an “unintended behaviour”?

- Definition of fault/error/failure refers to “unintended behaviour”
- How do we know when/which program behaviour is “unintended”?

What is an “unintended behaviour”?

- Definition of fault/error/failure refers to “unintended behaviour”
- How do we know when/which program behaviour is “unintended”?
  - Programmer’s intentions need to be clear from the code

Making the programmer's intention clear

- Comments
- KISS (Keep it Simple, Stupid)
- Assertions

Comments

- Be concise, brief
- Document the *purpose* of your code
- Explain *what* the code is doing
  - *How* it's done should be obvious from the code!
- Formatting: dictated by the tool you use (e.g., `Doxygen`)
- Update comments when you change the code!

Coding Style: KISS

- Conform to coding standards, follow style of existing code
  - You are an engineer, not an artist!
- Avoid “nifty” language features (like overloading)
  - unless it makes code easier to understand
- Industry standards exist in some fields (e.g., automotive)
  - MISRA: Motor Industry Software Reliability Association

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain “features”

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

  - "12.10 (req): The comma operator shall not be used"

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

  - "12.10 (req): The comma operator shall not be used"

```
x = (0, 1, 2);
```

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

  - "12.10 (req): The comma operator shall not be used"

```
x = (0, 1, 2);
```

  - "13.6 (req): Numeric variables used for iteration counting in a for loop shall not be modified in the loop body"

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

  - "12.10 (req): The comma operator shall not be used"

```
x = (0, 1, 2);
```

  - "13.6 (req): Numeric variables used for iteration counting in a for loop shall not be modified in the loop body"

```
for (i = 0; i<20; i++) { ...i++; ...}
```

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

  - "12.10 (req): The comma operator shall not be used"

```
x = (0, 1, 2);
```

  - "13.6 (req): Numeric variables used for iteration counting in a for loop shall not be modified in the loop body"

```
for (i = 0; i<20; i++) { ...i++; ...}
```

  - "3.1 (req): Use of implementation-defined behaviour shall be documented" (or better: avoided)

MISRA C/C++ Standard

- C/C++ are extremely powerful languages
- MISRA standard disallows/discourages use of certain "features"
  - "12.4 (req): The right-hand operand of a logical `&&` or `||` shall not contain sided effects."

```
while (y != x && x--) ...
```

  - "12.10 (req): The comma operator shall not be used"

```
x = (0, 1, 2);
```

  - "13.6 (req): Numeric variables used for iteration counting in a for loop shall not be modified in the loop body"

```
for (i = 0; i<20; i++) { ...i++; ...}
```

  - "3.1 (req): Use of implementation-defined behaviour shall be documented" (or better: avoided)

```
if ((x+1)>x) { ...}
```

MISRA C/C++ Standard

- Can be checked using static analysers (e.g. PC-Lint)
- Easier for humans *and* static analysers to check your code

Summary so far...

- Bugs come in many flavours
- Faults may lead to errors, which may lead to failure
- Causes of failures are hard to derive:
  - detect deviation from intended behaviour instead
- We need to
  - Understand what the program does (semantics)
  - Understand what the programmer wants

Try this at home!

- GNU compiler part of your favourite Linux or BSD distribution
- For Windows:
  - Cygwin (`http://www.cygwin.org`)
  - Mininimalist GNU for Windows (`http://www.mingw.org`)
- For Mac:
  - `gcc`/`g++` part of XCode (free on AppStore for macOS)
  - MacPorts (`http://www.macports.org`)
  - Fink (`http://fink.sf.net`)
  - Homebrew (`http://brew.sh`)

Next lecture: **Assertions**