
Core C++ 2026
20-21 Oct. 2026 :: Tel-Aviv
Is C++ Turing Complete?
Michael Daum

Hi, I’m Mike.

Software developer · Founder, Coobeeyon Development

C++ every day from 1992–2022, professionally from 1998

Founder of C++TO and CppNorth

Freelance custom software development, consulting, and training. Hire me!!



Alan Turing

1936 Invented the Turing machine.

WWII Helped break the Enigma cipher at Bletchley Park.

1950 Asked “Can machines think?”: the Turing test.

1952 Turing patterns: how chemistry makes spots and stripes.

Widely regarded as the greatest computer scientist to have never won the Turing Award



1936. Turing is 23.

What can a person with pencil and paper compute?

His answer: an imaginary machine.

States & rules

carry — still adding 1

halt — stop

IF		THEN		
STATE	READ	WRITE	MOVE	NEXT
carry	1	0	←	carry
carry	0	1	–	halt
carry	_	1	–	halt

Running machine



the answer ⟶ 11000 = 24

Tape = symbols in squares (_ is blank) · ▲ = head that reads and writes. The state and symbol select a rule. This example adds one: 10111 (23) → 11000 (24).

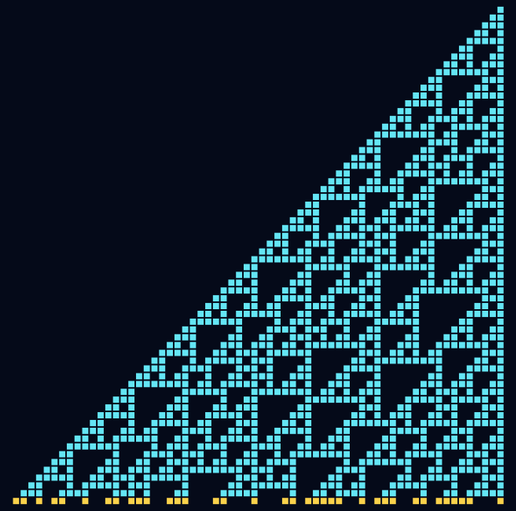
A system is Turing complete if…

it can simulate any Turing machine.

mov

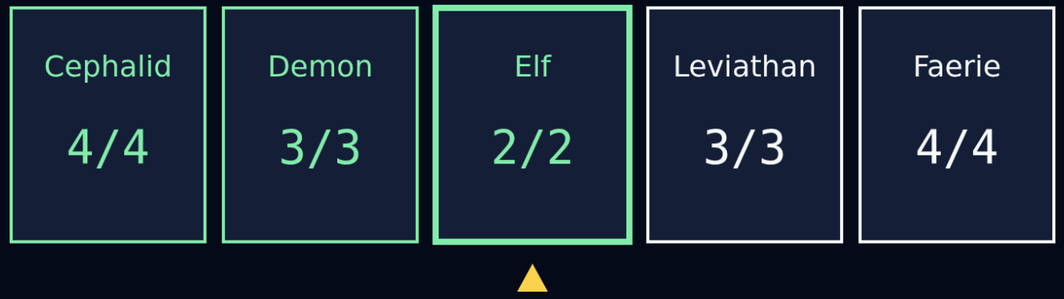
```
movw   alu_inv16(,%ecx,2), %cx
movl   alu_add16(,%eax,4), %edx
movl   (%edx,%ecx,4), %edx
movl   alu_add16(,%edx,4), %edx
movl   (alu_c), %ecx
movl   (%edx,%ecx,4), %edx
movw   %dx, (alu_t+0)
movl   %edx, (alu_c-2)
movw   (alu_x+2), %ax
movw   (alu_y+2), %cx
movw   alu_inv16(,%ecx,2), %cx
movl   alu_add16(,%eax,4), %edx
```

Rule 110

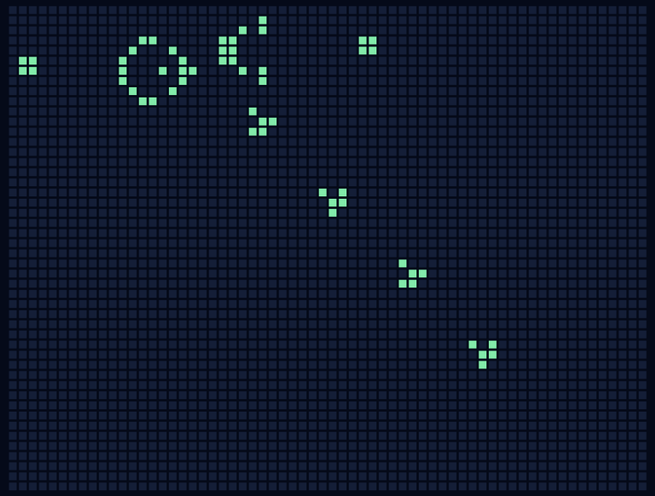


Magic: The Gathering

q2



Conway’s Game of Life



Minsky (two counter) Machine

We'll rebuild this same machine with macros, templates, deduction guides and contracts.

A	B
0, 1, 2, …	0, 1, 2, …

Two whole numbers. No fixed upper limit.



Program counter: pc

The current instruction in a finite program.

Execute it; follow its jump; repeat.

r is A or B; j and k are instruction numbers.

`INC(r, j)`

Add 1 to r. Go to j.

`DEC(r, j)`

Subtract 1 from r (zero stays zero). Go to j.

`JZ(r, j, k)`

If r is zero: go to j. Otherwise: go to k.

`HALT`

Stop.

Build this machine → you can simulate any Turing machine.

So: is C++ Turing complete?



But let us count the ways!!

Six Turing-complete subsystems of C++

01

`while (…)`

Runtime

ordinary execution

02

`#define`

Preprocessor

macro expansion, external clock

03

`template<…>`

Template meta-programming

instantiation

04

`consteval`

Constant evaluation

execution during translation

05

`-> T<…>`

CTAD

deduction guides

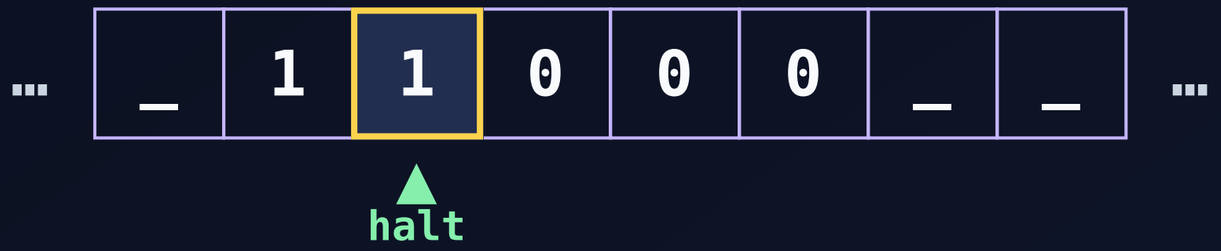
06

`pre(…)`

Contracts

semantics chosen by flags

Remember this machine? Plain runtime C++ runs it.



Rule table

IF		THEN		
state	read	write	move	next
carry	1	0	←	carry
carry	0	1	–	halt
carry	_	1	–	halt

```
while (state != "halt") {
  const Rule& r = rule_for(state, tape[head]);
  tape[head] = r.write;
  head += r.move;
  state = r.next;
  print(step++, tape, head, state);
}
```

```
$ ./turing_machine
step 0 _1011[1]__ carry
step 1 _101[1]0__ carry
step 2 _10[1]00__ carry
step 3 _1[0]000__ carry
step 4 _1[1]000__ halt
halt 11000 = 24
```

Six Turing-complete subsystems of C++

01

`while (…)`

Runtime

ordinary execution

02

`#define`

Preprocessor

macro expansion, external clock

03

`template<…>`

Template meta-programming

instantiation

04

`consteval`

Constant evaluation

execution during translation

05

`-> T<…>`

CTAD

deduction guides

06

`pre(…)`

Contracts

semantics chosen by flags

Same machine. Macros do the work.

```
#define P0 JZ(B, 3, 1)
#define P1 DEC(B, 2)
#define P2 INC(A, 0)
#define P3 HALT
```

Halted after 11 passes: A is 5, B is 0. B was added into A.

INPUT `STATE(3, (I , I , I , I,I), ())`

pc · A = 5 · B = 0

↓ PASS 11 `P3 = HALT: stop`

OUTPUT `HALTED((I , I , I , I,I), ())`

feed it back in: the next pass runs the next instruction

Six Turing-complete subsystems of C++

01

`while (…)`

Runtime

ordinary execution

02

`#define`

Preprocessor

macro expansion, external clock

03

`template<…>`

Template meta-programming

instantiation

04

`consteval`

Constant evaluation

execution during translation

05

`-> T<…>`

CTAD

deduction guides

06

`pre(…)`

Contracts

semantics chosen by flags

In 1994, at a C++ standards meeting, Erwin Unruh handed round a program that doesn’t compile.

The error messages were the output.

```
E "primes.cpp",L11/C25(#416): prim | Type `enum{}´ can´t be converted to txpe `D<2>´
E "primes.cpp",L11/C25(#416): prim | Type `enum{}´ can´t be converted to txpe `D<3>´
E "primes.cpp",L11/C25(#416): prim | Type `enum{}´ can´t be converted to txpe `D<5>´
E "primes.cpp",L11/C25(#416): prim | Type `enum{}´ can´t be converted to txpe `D<7>´
E "primes.cpp",L11/C25(#416): prim | Type `enum{}´ can´t be converted to txpe `D<11>´
…
E "primes.cpp",L11/C25(#416): prim | Type `enum{}´ can´t be converted to txpe `D<29>´
```

Each template asks for the next. The errors spell out the primes.

```
1 template <int i> struct D { D(std::false_type); };  // accepts only "not prime"
2 template <int p, int i> struct is_prime {
3   enum { prim = (p == 2) || (p % i) && is_prime<(i > 2 ? p : 0), i - 1>::prim };
4 };
5 template <int i> struct Prime_print {
6   Prime_print<i - 1> a;
7   enum { prim = is_prime<i, i - 1>::prim };
8   void f() { a.f(); D<i> d = std::bool_constant<prim>{}; }  // primes can't convert
9 };
```

GCC 14 and Clang 19, today: errors at `D<29>`, `D<23>`, `D<19>` … `D<2>`.

Erwin Unruh, 1994, showing his prime-number program to the committee:

“…he … appeared surprised that I (and others) thought that marvelous rather than scary.”

Bjarne Stroustrup, “Evolving a language in and for the real world: C++ 1991–2006”, HOPL III (2007), §7.2.2

“To kill template-metaprogramming, all I would have had to do was to say nothing. Instead my comment was along the lines ‘Wow, that’s neat! We mustn’t compromise it. It might prove useful.’”

Bjarne Stroustrup, FAQ

Primes aren’t a proof.

Let’s build a Minsky machine with template metaprogramming.

A state value becomes a template argument.

```
struct state { int pc, a, b; };
```

Instructions are tag types.

```
template <typename R, int j>          struct INC {};
template <typename R, int j>          struct DEC {};
template <typename R, int j, int k> struct JZ  {};
struct HALT {};
```

Counter ID’s Too!

```
struct A {};  struct B {};
```

Specialization implements the instruction.

```
1 template <typename I, state s> struct exec;
2
3 template <int j, state s>
4 struct exec<INC<A, j>, s> {
5   static constexpr state next{j, s.a + 1, s.b};
6 };
```

```
exec<INC<A, 0>, state{2, 2, 2}>::next → state{0, 3, 2}
```

The program is a tuple of instructions.

Add B into A

```
using program = std::tuple<
  JZ<B, 3, 1>,   // 0: if B is 0 go to 3, else go to 1
  DEC<B, 2>,     // 1: B - 1, go to 2
  INC<A, 0>,     // 2: A + 1, go to 0
  HALT>;         // 3: stop
```

EXAMPLE 3 · TEMPLATE METAPROGRAMMING

Execution by chaining through inheritance.

```cpp
template <typename P, state s>
using instruction = std::tuple_element_t<s.pc, P>;

template <typename P, state s>
constexpr state step = exec<instruction<P, s>, s>::next;

template <typename P, state s>
struct run : run<P, step<P, s>> {};

template <typename P, state s> requires std::is_same_v<instruction<P, s>, HALT>
struct run<P, s> { static constexpr state result = s; };
```

Single step

`step` is the state after the instruction at `s.pc`.

Chain by inheritance

The base class is the next `run`. It must be instantiated too.

Stop

The HALT specialization has no next base class.

```
static_assert(run<program, state{0, 2, 3}>::result == state{3, 5, 0});
```

GCC’s instantiated state types · recorded replay

```
state{0, 2, 3}
state{1, 2, 3}
state{2, 2, 2}
state{0, 3, 2}
state{1, 3, 2}
state{2, 3, 1}
state{0, 4, 1}
state{1, 4, 1}
state{2, 4, 0}
state{0, 5, 0}
state{3, 5, 0}  HALT
```

Add B into A · start A=2, B=3

A = 5 

B = 0

B moved into A. All during compilation.

Six Turing-complete subsystems of C++

01

`while (…)`

Runtime

ordinary execution

02

`#define`

Preprocessor

macro expansion, external clock

03

`template<…>`

Template meta-programming

instantiation

04

`consteval`

Constant evaluation

execution during translation

05

`-> T<…>`

CTAD

deduction guides

06

`pre(…)`

Contracts

semantics chosen by flags

EXAMPLE 4 · CONSTANT EVALUATION

`constexpr` all the things.

The runtime machine from earlier: same rules, same loop, now `constexpr`.

```
constexpr std::string run(std::string tape, std::size_t head) {
  std::string_view state = "carry";
  while (state != "halt") {
    const Rule* r = rule_for(state, tape[head]);
    tape[head] = r->write;
    head += r->move;
    state = r->next;
  }
  return tape;
}
```

```
static_assert(run("_10111__", 5) == "_11000__");
```

Six Turing-complete subsystems of C++

01

`while (…)`

Runtime

ordinary execution

02

`#define`

Preprocessor

macro expansion, external clock

03

`template<…>`

Template meta-programming

instantiation

04

`consteval`

Constant evaluation

execution during translation

05

`-> T<…>`

CTAD

deduction guides

06

`pre(…)`

Contracts

semantics chosen by flags

Class template argument deduction

```
std::pair p{1, 2.0};   // std::pair<int, double>
```

You write the values. The compiler finds the type.

Deduction guides tell it what to deduce.

```
template <typename T> struct box { box(T); };
box b1{"hello"};   // box<const char*>  (implicit guide)
```

```
box(const char*) -> box<std::string>;
box b2{"hello"};   // box<std::string>  (your guide wins)
```

A guide can compute.

```
template <int n> struct int_t {};   // a number, carried by a type
template <int n> struct next { next(auto); };
template <int n> next(int_t<n>) -> next<n + 1>;   // arithmetic

decltype(next{int_t<41>{}})   // next<42>
```

Let’s build a Minsky machine with deduction guides.

The machine’s state is the same struct we used for the metaprogramming example.

```
struct state { int pc, a, b; };
```

Carried by a type, like `int_t<41>`.

```
template <state s> struct state_t {};
```

Instructions are the same tag types we used for the metaprogramming example.

```
template <typename R, int j>          struct INC {};
template <typename R, int j>          struct DEC {};
template <typename R, int j, int k> struct JZ  {};
struct HALT {};
struct A {};  struct B {};
```

Guided CTAD implements the instruction.

```
1 template <state s> struct exec {
2   static constexpr state next = s;
3   template <typename I, typename S> exec(I, S);
4 };
5
6 template <int j, state s>
7 exec(INC<A, j>, state_t<s>) -> exec<state{j, s.a + 1, s.b}>;
```

```
decltype(exec{INC<A, 0>{}, state_t<state{2, 2, 2}>{}}) → exec<state{0, 3, 2}>
```

The program: the same one we used for the metaprogramming example.

Add B into A

```
using program = std::tuple<
  JZ<B, 3, 1>,   // 0: if B is 0 go to 3, else go to 1
  DEC<B, 2>,     // 1: B – 1, go to 2
  INC<A, 0>,     // 2: A + 1, go to 0
  HALT>;         // 3: stop
```

Running the Program: one step

```
template <typename P, state s>
using instruction = std::tuple_element_t<s.pc, P>;

template <typename P, state s>
using step =
  state_t<decltype(exec{instruction<P, s>{}, state_t<s>{}})::next>;
```

Look up instruction `s.pc` in the program, and deduce `exec` from it.

Guides chained through guides.
The default argument is the link.

```
1 template <typename T> struct run {
2   using type = T;
3   template <typename... U> run(U...);
4 };
5
6 template <typename P, state s,
7           typename R = typename decltype(run{P{}, step<P, s>{}})::type>
8 run(P, state_t<s>) -> run<R>;
```

Each guide builds the next state, then **run** again.

Running the Program: HALT stops

```
template <typename P, state s>
  requires std::is_same_v<instruction<P, s>, HALT>
run(P, state_t<s>) -> run<state_t<s>>;
```

At `HALT`, this guide applies instead, and asks for nothing more.

```
using result = decltype(run{program{}, state_t<state{0, 2, 3}>{}});
static_assert(std::is_same_v<result, run<state_t<state{3, 5, 0}>>>);
```

GCC’s deduced states · recorded replay

```
state{0, 2, 3}
state{1, 2, 3}
state{2, 2, 2}
state{0, 3, 2}
state{1, 3, 2}
state{2, 3, 1}
state{0, 4, 1}
state{1, 4, 1}
state{2, 4, 0}
state{0, 5, 0}
state{3, 5, 0}  HALT
```

Add B into A · start A=2, B=3

A = 5 

B = 0

B moved into A. No constructor ran.

Six Turing-complete subsystems of C++

01

`while (…)`

Runtime

ordinary execution

02

`#define`

Preprocessor

macro expansion, external clock

03

`template<…>`

Template meta-programming

instantiation

04

`consteval`

Constant evaluation

execution during translation

05

`-> T<…>`

CTAD

deduction guides

06

`pre(…)`

Contracts

semantics chosen by flags

Contracts: promises in the code.

```
int abs_value(int x)
  pre(x != INT_MIN)                // on entry: the caller's promise
  post(r: r >= 0)                  // on return: r is the result
{
  int result = x < 0 ? -x : x;
  contract_assert(result >= x); // right here
  return result;
}
```

Four checking modes.

MODE	CHECKS THE CONDITION?	IF IT FAILS
`ignore`	no	nothing happens
`observe`	yes	call the violation handler, then carry on
`enforce`	yes	call the violation handler, then terminate
`quick-enforce`	yes	terminate at once, no handler

Standardization status

Feb 2025	Hagenberg: P2900 voted into the C++26 draft.
Mar 2026	Croydon: the committee finishes C++26, contracts included.
Next	ISO approval, still to come.

> *“...ignores about 50 years of progress involving a combination of static analysis and improved implementation of range checks.”*

> *“...an existential threat to C++...”*

Stroustrup, Garcia, Falco, Spicer, Voutilainen: P4334R0, “P2900 Contracts’ fundamental flaws” (August 2026).

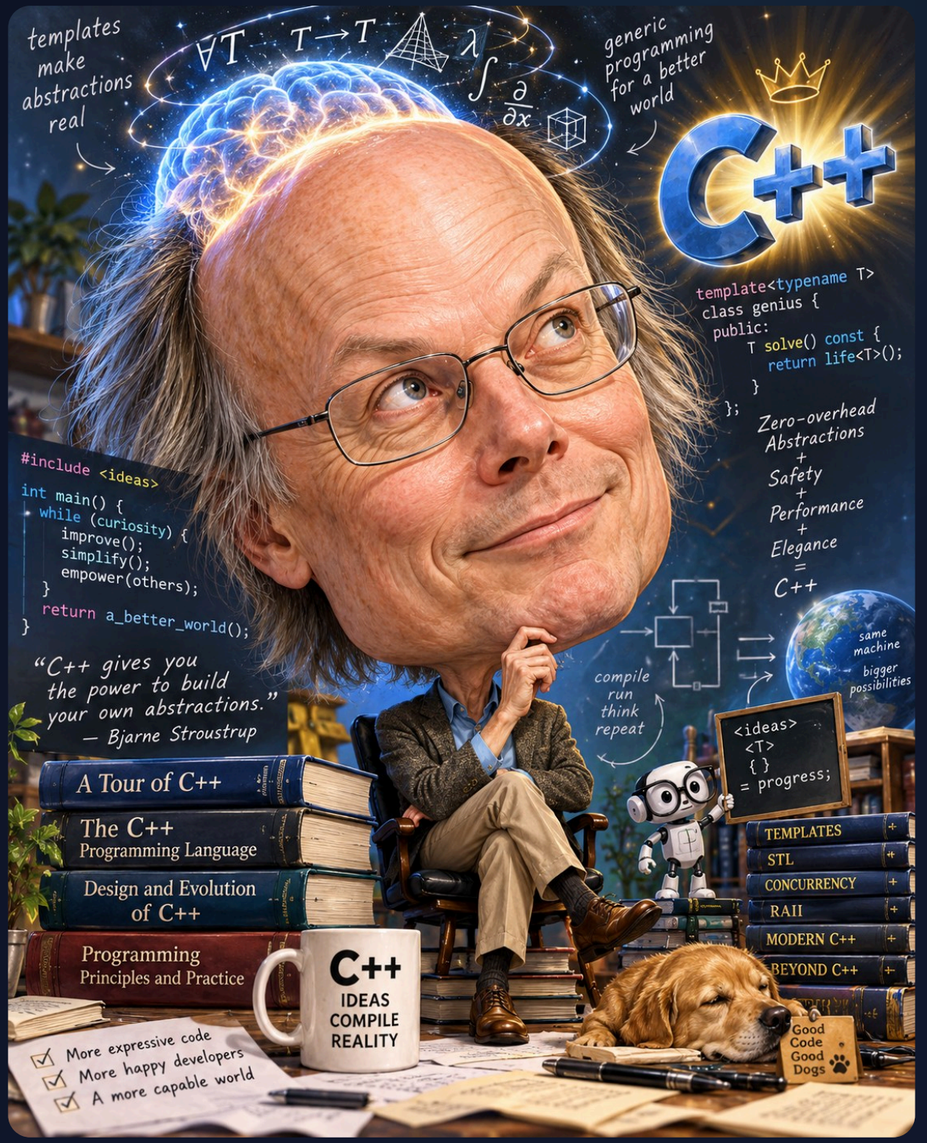
Contracts can run any code.

```
void f()
  pre(run("_10111__", 5) == "_11000__");   // the Turing machine
```

So contracts are Turing complete, trivially, just like `constexpr`.

But Bjarne is smart.

What did he mean by static analysis…?



Oooh, we can make a program that returns any of 4 values just by changing the flags!

```
1  int stage_bits = 0, mode_if_stopped = 3;
2  int* const stages = &stage_bits;
3
4  int probe()
5    pre((*stages |= 1, true))
6    post(r: (*stages |= 4, r != 0))
7  { contract_assert((*stages |= 2, true)); return 0; }
8
9  void handle_contract_violation(const std::contracts::contract_violation&)
10 { mode_if_stopped = 2; }
11 void on_terminate() { std::_Exit(mode_if_stopped); }
12
13 int main() {
14   std::set_terminate(on_terminate);
15   probe();
16   std::_Exit(stage_bits == 0 ? 0 : 1);
17 }
```

Did any check run?

no → 0 `ignore`

Did the call return?

yes → 1 `observe`

Did the handler run before it stopped?

yes → 2 `enforce`

no → 3 `quick-enforce`

The exit code is the answer: four answers from one run, so two bits.

We need 7, so let’s split it into 2 passes.
Pass 1 picks the instruction, pass 2 the counter.

PASS 1 ↓ · PASS 2 →	0 · IGNORE → A	1 · OBSERVE → B
0 · ignore	INC(A, j)	INC(B, j)
1 · observe	DEC(A, j)	DEC(B, j)
2 · enforce	JZ(A, j, k)	JZ(B, j, k)
3 · quick-enforce	HALT	HALT

Four checking modes, four instructions. Jump targets `j` and `k` come on the command line.

We “compile” the program into GCC flags.

LINE	INSTRUCTION	PASS 1 BUILT WITH	PASS 2 BUILT WITH	J K
0	JZ(B, 3, 1)	enforce	observe	3 1
1	DEC(B, 2)	observe	observe	2 0
2	INC(A, 0)	ignore	ignore	0 0
3	HALT	quick-enforce	ignore	0 0

Add B into A: the same program as macros, templates and deduction guides.

Each line: two builds, two runs.

The state is one line of text: **S 0 2 3 0 -**

pc **A** **B** **halted?** **op**

Example: line 0, **JZ(B, 3, 1)** → **enforce**, **observe**, **j k = 3 1**

```
$ g++ -fcontracts -fcontract-evaluation-semantic=enforce machine.cpp
$ echo "S 0 2 3 0 -" | ./a.out 3 1
S 0 2 3 0 2
$ g++ -fcontracts -fcontract-evaluation-semantic=observe machine.cpp
$ echo "S 0 2 3 0 2" | ./a.out 3 1
S 1 2 3 0 -
```

Pass 2: the build with **observe** fires the violation, the program prints **S 0 2 3 0 2**, and exits 0. Pass 1: the build with **enforce** fires the violation, the program aborts before printing, and exits 1. Either way, **JZ** writes **op** = 2, **JZ**.

```
1  state s;  // the state line, read from stdin
2  int target_j = 0, target_k = 0;  // jump targets, from the command line
3
4  int stage_bits = 0, mode_if_stopped = 3;
5  int* const stages = &stage_bits;
6
7  int probe()
8    pre((*stages |= 1, true))
9    post(r: (*stages |= 4, r != 0))
10 { contract_assert((*stages |= 2, true)); return 0; }
11
12 void handle_contract_violation(const std::contracts::contract_violation&)
13 { mode_if_stopped = 2; }
14 void on_terminate() { finish(mode_if_stopped); }
15
16 int main(int argc, char** argv) {
17     std::tie(s, target_j, target_k) = read_input(argc, argv);
18     std::set_terminate(on_terminate);
19     probe();
20     finish(stage_bits == 0 ? 0 : 1);
21 }
```

Fill in `read_input` so that `probe()` returns 0 if and only if `target_j` equals `target_k`, and so that `probe()` is executed exactly once before `on_terminate`.

`finish`: store the instruction, or run it.

```
[[noreturn]] void finish(int mode) {
    if (s.op == none) {     // pass 1: the mode is the instruction
        s.op = mode;
    } else {                // pass 2: the mode is the counter
        execute(s, mode, target_j, target_k);
    }
    print_state(s);
    std::_Exit(mode);
}
```

Every instruction finishes the same way: it hands `finish` a mode, and `finish` decides what that mode means. On pass 1 the mode is the instruction, so `finish` stores it in `s.op` and returns. On pass 2 the mode is the counter, so `finish` executes the instruction on it.

The four instructions are ordinary C++.

```cpp
void execute(state& s, int reg, int target_j, int target_k) {
    int& r = reg == 0 ? s.a : s.b;
    switch (s.op) {
        case 0: ++r; s.pc = target_j; break;                        // INC(r, j)
        case 1: if (r != 0) --r; s.pc = target_j; break;            // DEC(r, j)
        case 2: s.pc = r == 0 ? target_j : target_k; break;         // JZ(r, j, k)
        case 3: s.halted = true; break;                             // HALT
    }
    s.op = none;
}
```

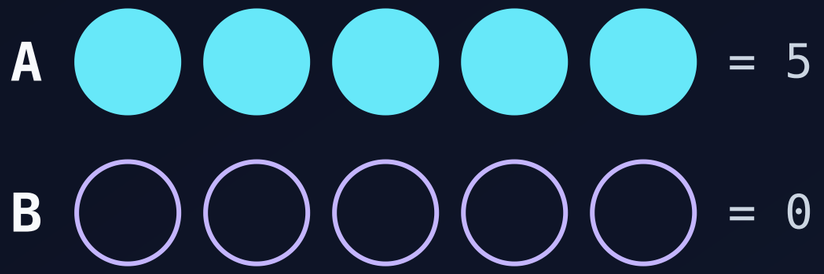
The contracts choose the instruction and the counter. This switch carries it out.

Let’s watch it work.

pass 1	`-fcontract-evaluation-semantic=`**`quick_enforce`**	**exit 3**
pass 2	`-fcontract-evaluation-semantic=`**`ignore`**	**exit 0**

HALT

```
0 JZ B 3 1
1 DEC B 2
2 INC A 0
3 HALT
```



HALT · 11 instructions · 22 compiles

One frozen source file. Two compile settings encode each instruction: 4 × first exit + second exit. The recorded run moves r0 = 3 into r1 in 28 compiles.

When contracts are approved, C++ will finally be Turing complete.

Without changing a single line of code!

The final C++ program.

```
state s;  // the state line, read from stdin
int target_j = 0, target_k = 0;  // jump targets, from the command line

// The two-bits program: which semantics was this built with?
int stage_bits = 0, mode_if_stopped = 3;
int* const stages = &stage_bits;

int probe()
  pre((*stages |= 1, true))
  post(r: (*stages |= 4, r != 0))
{ contract_assert((*stages |= 2, true)); return 0; }

// Run the stored instruction on counter reg, and use it up.
void execute(state& s, int reg, int target_j, int target_k) {
    int& r = reg == 0 ? s.a : s.b;
    switch (s.op) {
        case 0: ++r; s.pc = target_j; break;                    // INC(r, j)
        case 1: if (r != 0) --r; s.pc = target_j; break;        // DEC(r, j)
        case 2: s.pc = r == 0 ? target_j : target_k; break;     // JZ(r, j, k)
        case 3: s.halted = true; break;                         // HALT
    }
    s.op = none;
}
```

```
[[noreturn]] void finish(int mode) {
    if (s.op == none) {     // pass 1: the mode is the instruction
        s.op = mode;
    } else {                // pass 2: the mode is the counter
        execute(s, mode, target_j, target_k);
    }
    print_state(s);
    std::_Exit(mode);
}

void handle_contract_violation(const std::contracts::contract_violation&)
{ mode_if_stopped = 2; }
void on_terminate() { finish(mode_if_stopped); }

int main(int argc, char** argv) {
    std::tie(s, target_j, target_k) = read_input(argc, argv);
    std::set_terminate(on_terminate);
    probe();
    finish(stage_bits == 0 ? 0 : 1);
}
```

100% backward compatible.
Forever.

Thank you.