Cinquecento
A Programming Language for Debugging

Dan Ridge and Vic Zandy
IDA/CCS

Overview

- Audience: low-level HPC hackers
  - Developers of HPC tools, runtimes, schedulers, filesystems, ...
  - E.g., paradyn, dyninst, upc, xcpu, bproc, proverb, rocks, ckpt, ...
- Cinquecento: New language for debugging systems
  - Programs that observe and analyze execution of programs
  - For: reproducing and diagnosing bugs; validation; regression tests
  - A tool for people who build tools

Why a Language?

HPC hacker purgatory:

1. System does the wrong thing.
2. Immediate cause: insane state.
3. How did that happen?
4. Re-run, *periodically verifying consistency of data structures*.
5. Resolve bug; repeat.

Why a Language?

HPC hacker purgatory:

1. System does the wrong thing.
2. Immediate cause: insane state.
3. How did that happen?
4. Re-run, *periodically verifying consistency of data structures*.
5. Resolve bug; repeat.

We want to automate step #4.

Idea #1: Speak Target Language

Speak the language of the target program

- Complex data is most easily traversed in native language
- When life gets hard, we usually compile in new debugging code
- Our native language: **C**

Idea #1: Speak Target Language

Speak the language of the target program

- Complex data is most easily traversed in native language
- When life gets hard, we usually compile in new debugging code
- Our native language: **C**

```
dtab[c->type]->dc != ’M’
```

Idea #1: Speak Target Language

Speak the language of the target program

- Complex data is most easily traversed in native language
- When life gets hard, we usually compile in new debugging code
- Our native language: **C**

```
dtab[c->type]->dc != 'M'
```

```
/* get size of result */
size = 0;
i = 0;
do {
        cmd = *data++;
        size |= (cmd&~0x80)<<i;
        i += 7;
} while((cmd&0x80) && data < top);
```

Idea #2: Domains

- Most systems have multiple, distributed processes
  - we need a way to interact with each one
- Many systems are heterogeneous
  - in executables, OS, architecture, quirks of debug interface, ...
  - too much variety to bake assumptions into tool or language
- *Domain*: first-class, programmable representation of target program
  - first-class: many, named, stored in data structures
  - programmable: you can implement new ones *in Cinquecento!*

Idea #2: Domains

- Most systems have multiple, distributed processes
  - we need a way to interact with each one
- Many systems are heterogeneous
  - in executables, OS, architecture, quirks of debug interface, ...
  - too much variety to bake assumptions into tool or language
- *Domain*: first-class, programmable representation of target program
  - first-class: many, named, stored in data structures
  - programmable: you can implement new ones *in Cinquecento!*

Key design in Cinquecento: C-based interface to domains

Example #1

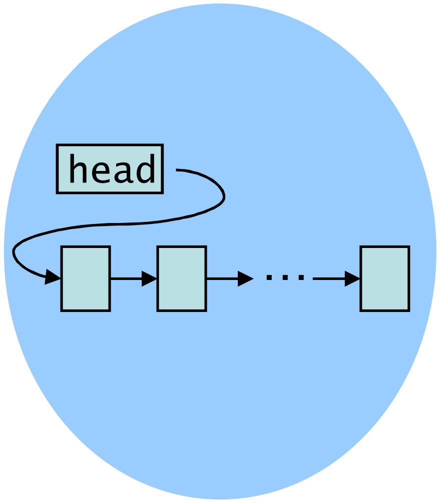
Your data structure (C type):

```
typedef struct Node Node;
struct Node {
   int v;
   Node *next;
};
```

Your data (C symbol):

```
Node *head;  // a list
```

Your process:



Example #1

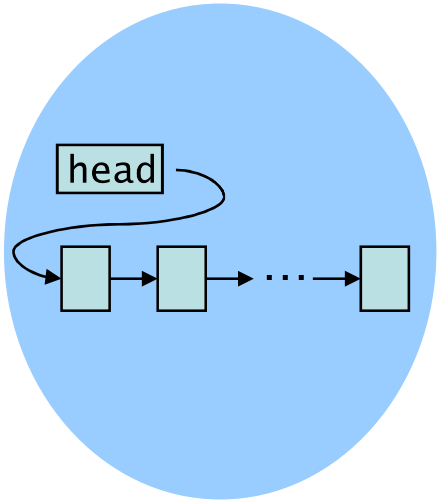
Your data structure (C type):

```
typedef struct Node Node;
struct Node {
   int v;
   Node *next;
};
```

Your data (C symbol):

```
Node *head;   // a list
```

Your process:




```
p = head;
while(p){
   printf("%d\n", p->v);
   p = p->next;
}
```

← C code that prints the list

Example #1

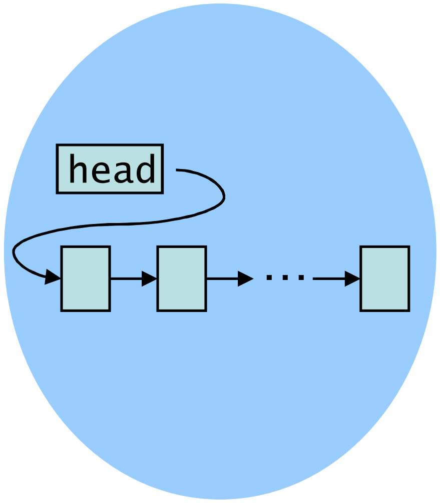
Your data structure (C type):

```
typedef struct Node Node;
struct Node {
   int v;
   Node *next;
};
```

Your data (C symbol):

```
Node *head;  // a list
```

Your process:




```
p = myproc`head;
while(p){
   printf("%d\n", p->v);
   p = p->next;
}
```

← Cinquecento code that prints the list

Example #1

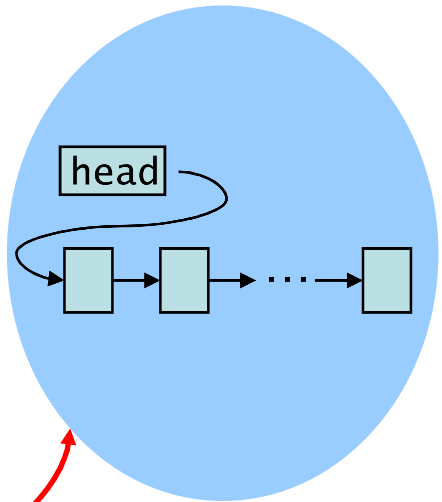
Your data structure (C type):

```
typedef struct Node Node;
struct Node {
   int v;
   Node *next;
};
```

Your data (C symbol):

```
Node *head;  // a list
```

Your process:




```
p = myproc`head;
while(p){
   printf("%d\n", p->v);
   p = p->next;
}
```

The domain myproc connects Cinquecento to the process

Example #2

Your data structure (C type):

```
typedef struct Node Node;
struct Node {
   int v;
   Node *next;
};
```

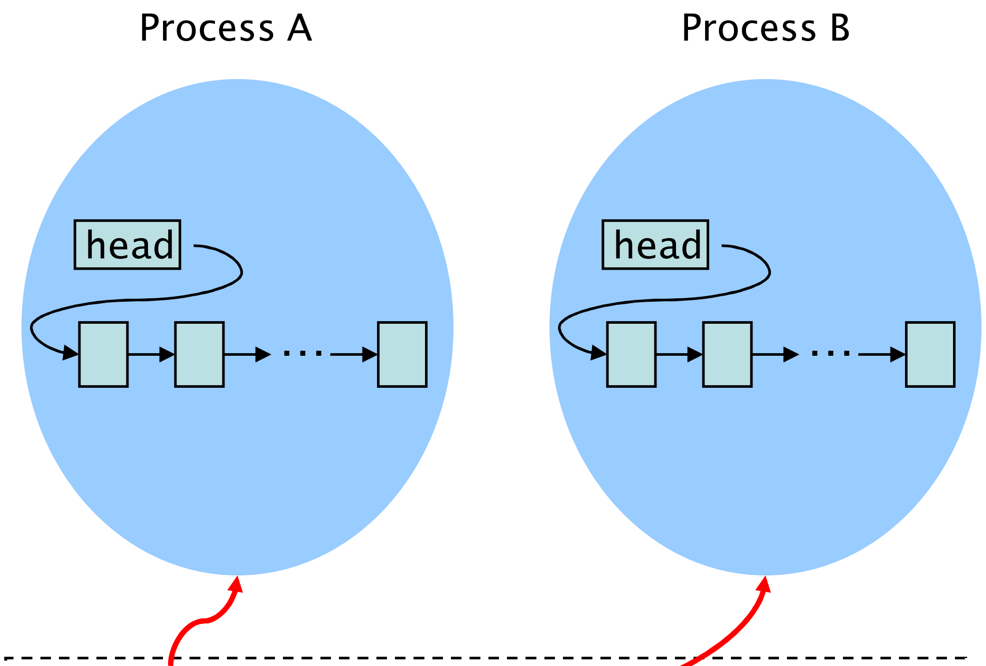
Your data (C symbol):

```
Node *head;  // a list
```

```
ap = head of list A
bp = head of list B
while(ap || bp){
   if(!ap || !bp || ap->v != bp->v)
        error("mismatch!");
   ap = ap->next, bp = bp->next;
}
```

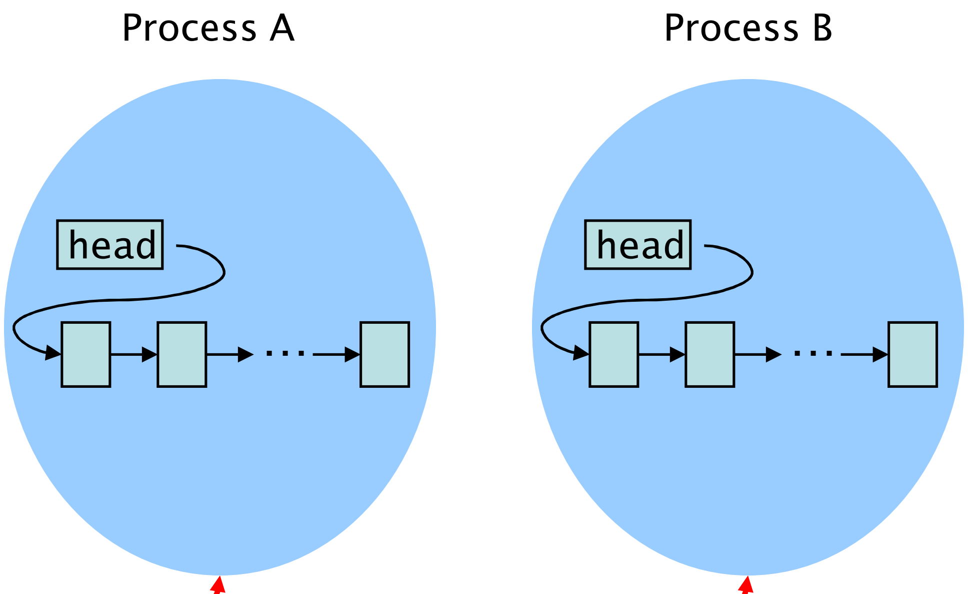
← C idiom for comparing two lists

Example #2




```
ap = A`head;
bp = B`head;
while(ap || bp){
   if(!ap || !bp || ap->v != bp->v)
        error("mismatch!");
   ap = ap->next, bp = bp->next;
}
```

Example #2




```
ap = A`head;
bp = B`head;
while(ap || bp){
   if(!ap || !bp || ap->v != bp->v)
        error("mismatch!");
   ap = ap->next, bp = bp->next;
}
```

Possible variations on A vs. B:

- Same input,
  different revisions of program
- Same program,
  different inputs
- 32-bit x86 vs. 64-bit MIPS
- Same process,
  different points in time
- Live process vs. core dump

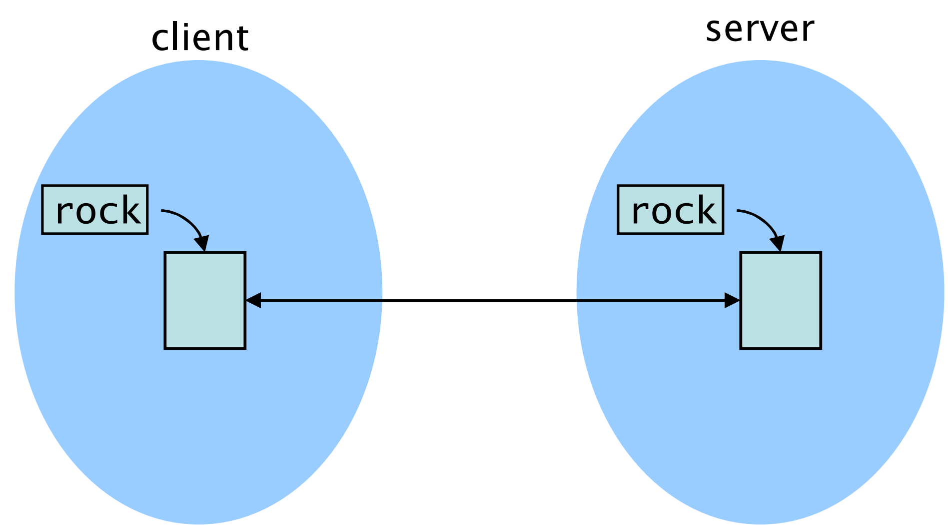
Example #3

Your data structure (C type):

```
typedef struct Rock Rock;
typedef struct Heartbeat Heartbeat;
struct Rock {
    unsigned rcvseq, sndseq;
    Heartbeat hb;
    ...
};
struct Heartbeat {
    unsigned missed, limit;
    ...
};
```

Your data (C symbol):

```
Rock *rock;  // a rock
```



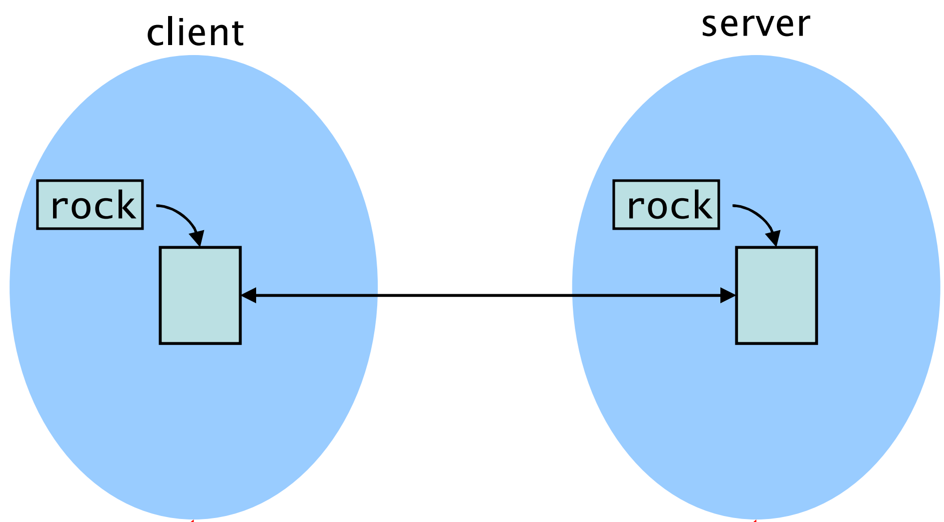
Example #3

Your data structure (C type):

```
typedef struct Rock Rock;
typedef struct Heartbeat Heartbeat;
struct Rock {
    unsigned rcvseq, sndseq;
    Heartbeat hb;
    ...
};
struct Heartbeat {
    unsigned missed, limit;
    ...
};
```

Your data (C symbol):

```
Rock *rock;  // a rock
```

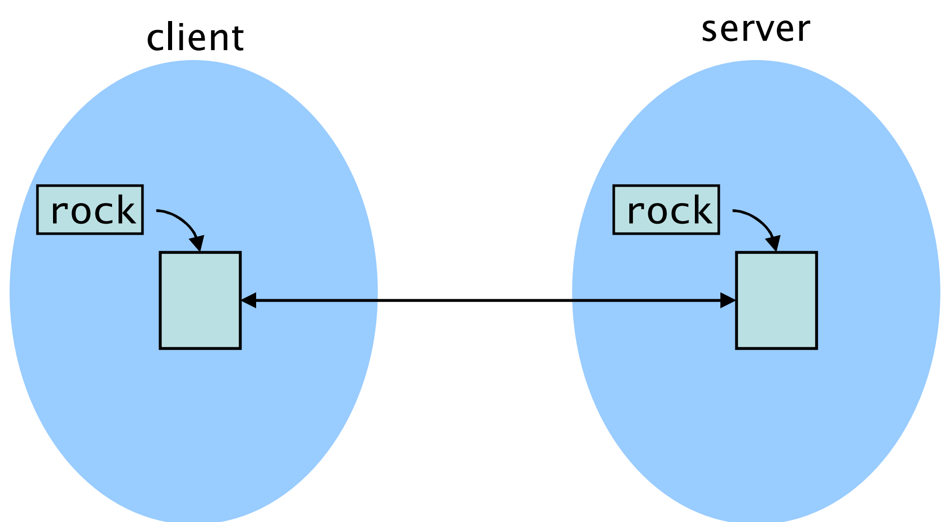



```
c = client`rock;
s = server`rock;
if(c->rcvseq >= s->sndseq || s->rcvseq >= c->sndseq)
   error("inconsistent rock state");
```

Example #3

Your data structure (C type):

```
typedef struct Rock Rock;
typedef struct Heartbeat Heartbeat;
struct Rock {
     unsigned rcvseq, sndseq;
     Heartbeat hb;
     ...
};
struct Heartbeat {
     unsigned missed, limit;
     ...
};
```



Your data (C symbol):

```
Rock *rock;   // a rock
```

```
c = client`rock;
s = server`rock;
if(c->rcvseq >= s->sndseq || s->rcvseq >= c->sndseq)
   error("inconsistent rock state");

c->hb->missed = c->hb->limit; // inject client fault
```

C in Cinquecento

- Expression syntax and semantics are generally standard (C99)
  - Evaluation conforms to machine of target program

- Identifiers and type names extended with *domain resolution operator*

  - `dom`x` (variable reference)
  - `dom`MaxBuf` (enum constant)
  - `struct dom`Node` (tagged type name)
  - `dom`void*` (base type name)
  - `dom`Node` (`typedef` type name)

- Control statements are standard (`if`, `switch`, `for`, `while`, `do`)
- Types are defined in an extension of C (`@names`)
- Real difference: Functions and variables work like Scheme, not C

C Values

```
typedef struct Node Node;
struct Node {
     int v;
     Node *next;
};
Node *head;
```

- Values read from domains are called *cvalues*

```
p = myproc`head;        // p is a cvalue
```

- Like ordinary C rvalues, cvalues have a C type and value

```
print(p);               // prints some address
print(typeof(p));       // prints the type name "Node*"
```

- Cvalues also have a reference to the domain they came from

```
print(domof(p));        // prints "myproc"
```

- This idea minimizes domain clutter in expressions and functions

```
p = p->next;           // chase pointers in myproc
q = (`Obj*)p;          // cast to Obj* as defined in myproc
```

The rest of the language

- Debugging is incremental, interactive, prototype driven
  – We want a scripting language
  – An *excellent* scripting language: **Scheme**!
- C-over-domains embedded in simple, functional language
  – Based on semantics (not syntax) of Scheme
- Our Scheme heritage:
  – Lambda: closures (functions) over lexically scoped variables
  – Dynamic typing
  – Proper tail recursion
  – Automatic memory management
  – Decades of high-performance implementation know-how
  – (And someday...) Macros
- Library of standard data structures (lists, strings, vectors, dictionaries)

Recap: Domains

- Domains represent target processes
- They encapsulate
  - definitions of types (including size, encoding, layout down to bit)
  - definitions of symbols (name x type x location)
  - interface to target memory
  - control interface
- Domains are *contexts for evaluating C expressions*

The Structure of Domains

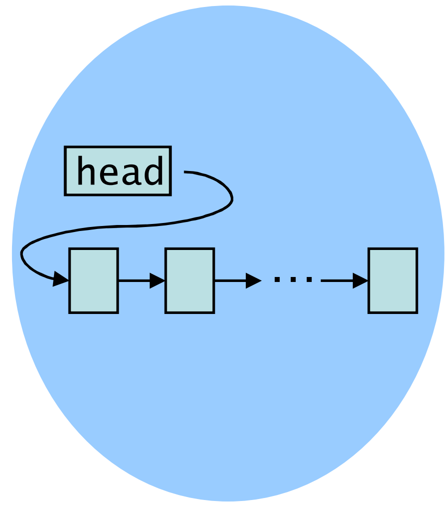
Your data structure (C type):

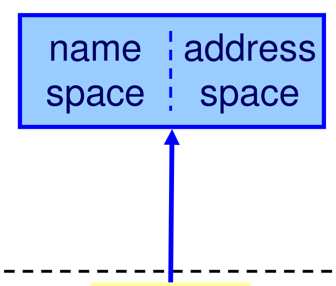
```
typedef struct Node Node;
struct Node {
   int v;
   Node *next;
};
```

Your data (C symbol):

```
Node *head;  // a list
```

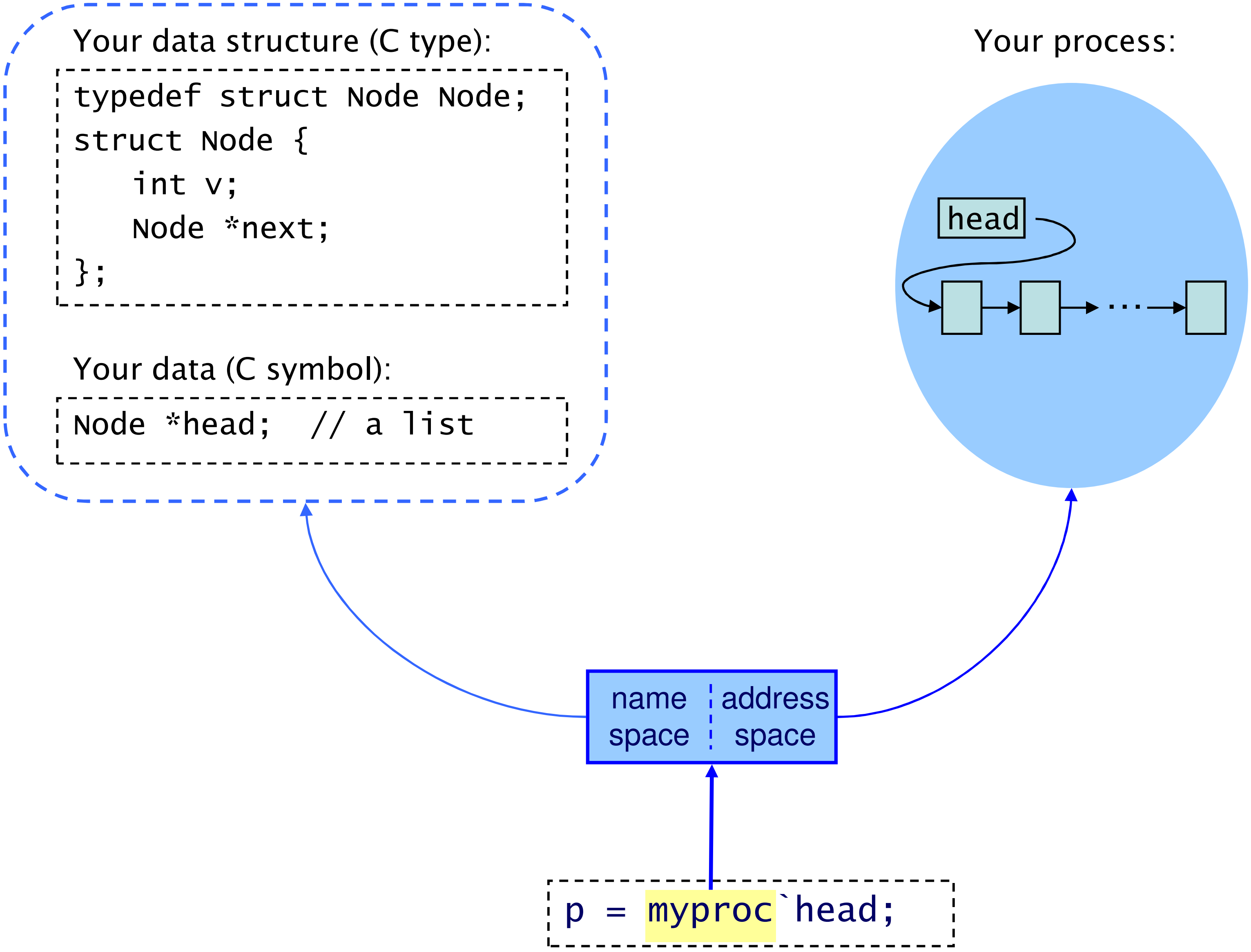
Your process:






```
p = myproc`head;
```

The Structure of Domains



The Structure of Domains

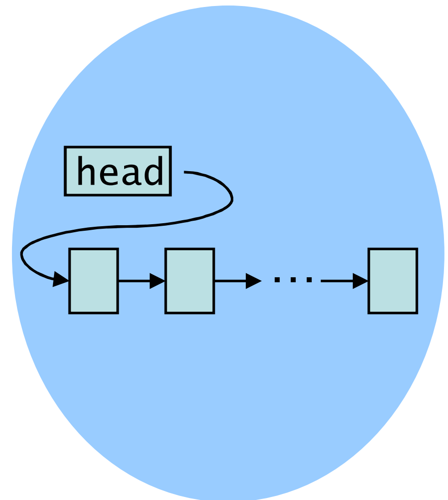
Your data structure (C type):

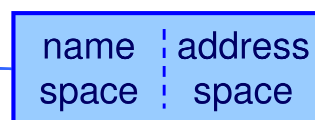
```
typedef struct Node Node;
struct Node {
    int v;
    Node *next;
};
```

Your data (C symbol):

```
Node *head;  // a list
```

Your process:






```
p = myproc`head;
```

The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

The Structure of Domains

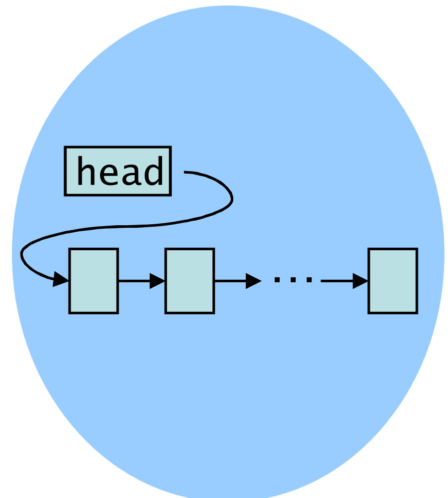
Your data structure (C type):

```
typedef struct Node Node;
struct Node {
    int v;
    Node *next;
};
```

Your data (C symbol):

```
Node *head;  // a list
```

Your process:



The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

name space	address space

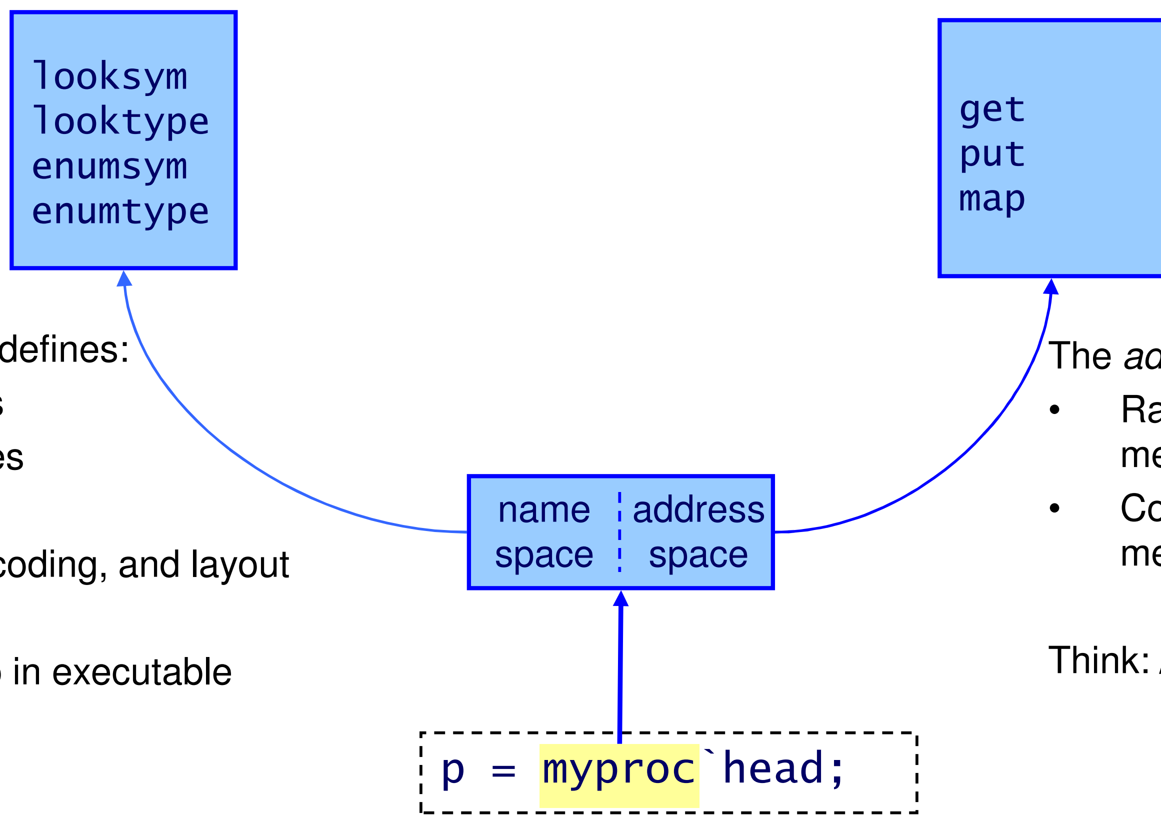
```
p = myproc`head;
```

The *address space* defines:

- Range of addressable memory
- Contents of addressable memory

Think: /proc/*pid*/maps + ptrace

The Structure of Domains



The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

The *address space* defines:

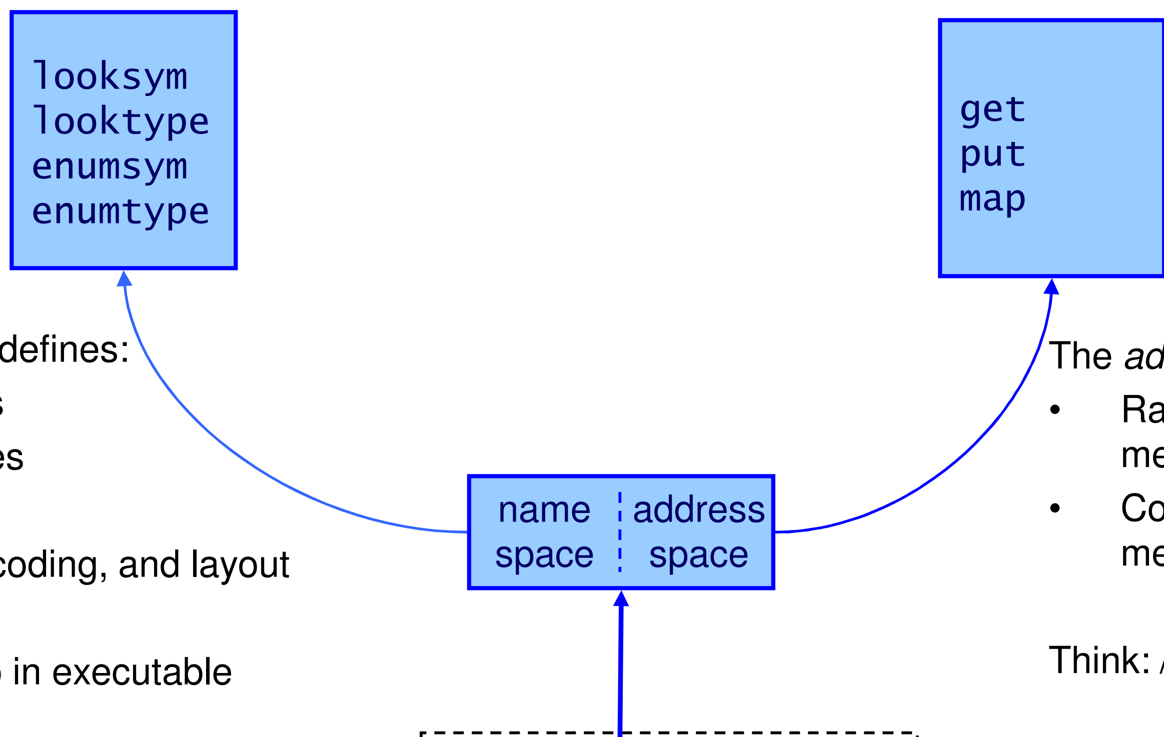
- Range of addressable memory
- Contents of addressable memory

Think: /proc/*pid*/maps + ptrace

The Structure of Domains

```
p = myproc`head;
```

```
sym = myproc.looksym("head");
addr = symaddr(sym);
type = symtype(sym);
bytes = myproc.get(addr, sizeof(type));
p = cval(bytes, type, myproc);
```



The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

The *address space* defines:

- Range of addressable memory
- Contents of addressable memory

Think: /proc/*pid*/maps + ptrace

```
p = myproc`head;
```

Domain Construction Primitives

- `mkns`: Construct a name space

```
define looksym(id) {
    ...
}
define looktype(typename) {
    ...
}
...
ns = mkns({ “looksym” : looksym, “looktype” : looktype, ... });
```

- `mkas`: Construct an address space

```
define get(addr, size) {
    ...
}
...
as = mkas({ “get” : get, ... });
```

- `mkdom`: Construct a domain

```
myproc = mkdom(ns, as);
```

@names: Name Space Syntax

You can specify name spaces in their native language:

```
ns = @names c32le {
    // type definitions
    typedef struct Node Node;
    struct Node {
          @0x0     int v;
          @0x4     Node *next;
          @0x8;
    };

    // symbol definitions
    @0x800012c    Node *head;
};
```

`@names`: Name Space Syntax

You can specify name spaces in their native language:

`@names` extends existing name spaces.

Root name spaces define base C types:

- `c32le` 32-bit little-endian
- `c32be` 32-bit big-endian
- `c64le` 64-bit little-endian
- `clp64le` 64-bit little-endian (long ptr)

```
ns = @names c32le {
    // type definitions
    typedef struct Node Node;
    struct Node {
        @0x0     int v;
        @0x4     Node *next;
        @0x8;
    };

    // symbol definitions
    @0x800012c    Node *head;
};
```

`@names`: Name Space Syntax

You can specify name spaces in their native language:

`@names` extends existing name spaces.

Root name spaces define base C types:

- `c32le` 32-bit little-endian
- `c32be` 32-bit big-endian
- `c64le` 64-bit little-endian
- `clp64le` 64-bit little-endian (long ptr)

Fields of structs/unions may be omitted:

```
struct Node {
        @0x4        Node *next;
        @0x8;
};
```

They may also overlap.

```
ns = @names c32le {
    // type definitions
    typedef struct Node Node;
    struct Node {
        @0x0     int v;
        @0x4     Node *next;
        @0x8;
    };

    // symbol definitions
    @0x800012c    Node *head;
};
```

`@names`: Name Space Syntax

You can specify name spaces in their native language:

`@names` extends existing name spaces.
Root name spaces define base C types:

- `c32le` 32-bit little-endian
- `c32be` 32-bit big-endian
- `c64le` 64-bit little-endian
- `clp64le` 64-bit little-endian (long ptr)

Fields of structs/unions may be omitted:

```
struct Node {
        @0x4        Node *next;
        @0x8;
};
```

They may also overlap.

Offsets and sizes may be any expression:

```
@loadbase+0x012c    Node *head;
```

```
ns = @names c32le {
    // type definitions
    typedef struct Node Node;
    struct Node {
        @0x0      int v;
        @0x4      Node *next;
        @0x8;
    };

    // symbol definitions
    @0x800012c    Node *head;
};
```

@names: Name Space Syntax

You can specify name spaces in their native language:

@names extends existing name spaces.
Root name spaces define base C types:

- c32le 32-bit little-endian
- c32be 32-bit big-endian
- c64le 64-bit little-endian
- clp64le 64-bit little-endian (long ptr)

Fields of structs/unions may be omitted:

```
struct Node {
        @0x4        Node *next;
        @0x8;
};
```

They may also overlap.

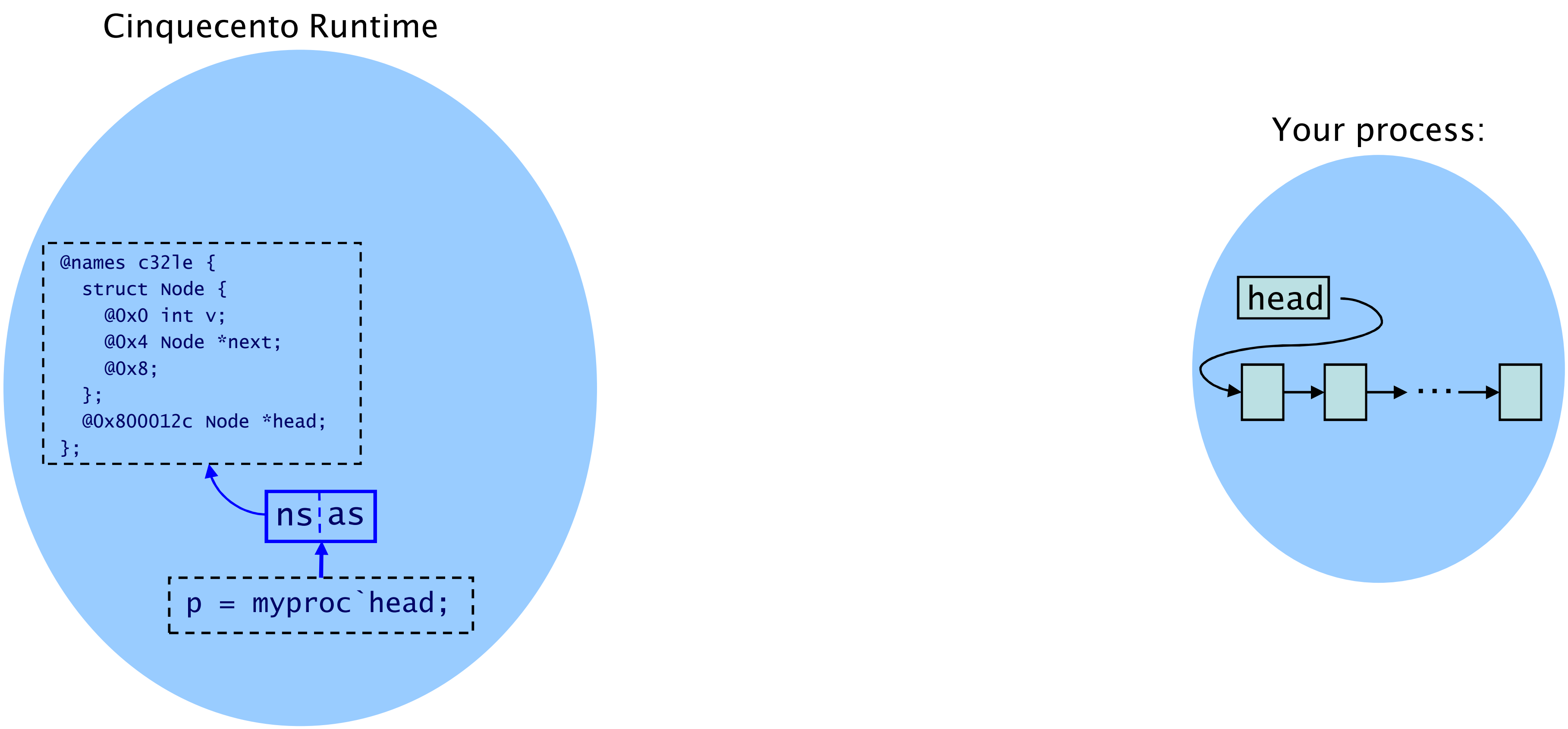
Offsets and sizes may be any expression:

```
@loadbase+0x012c    Node *head;
```

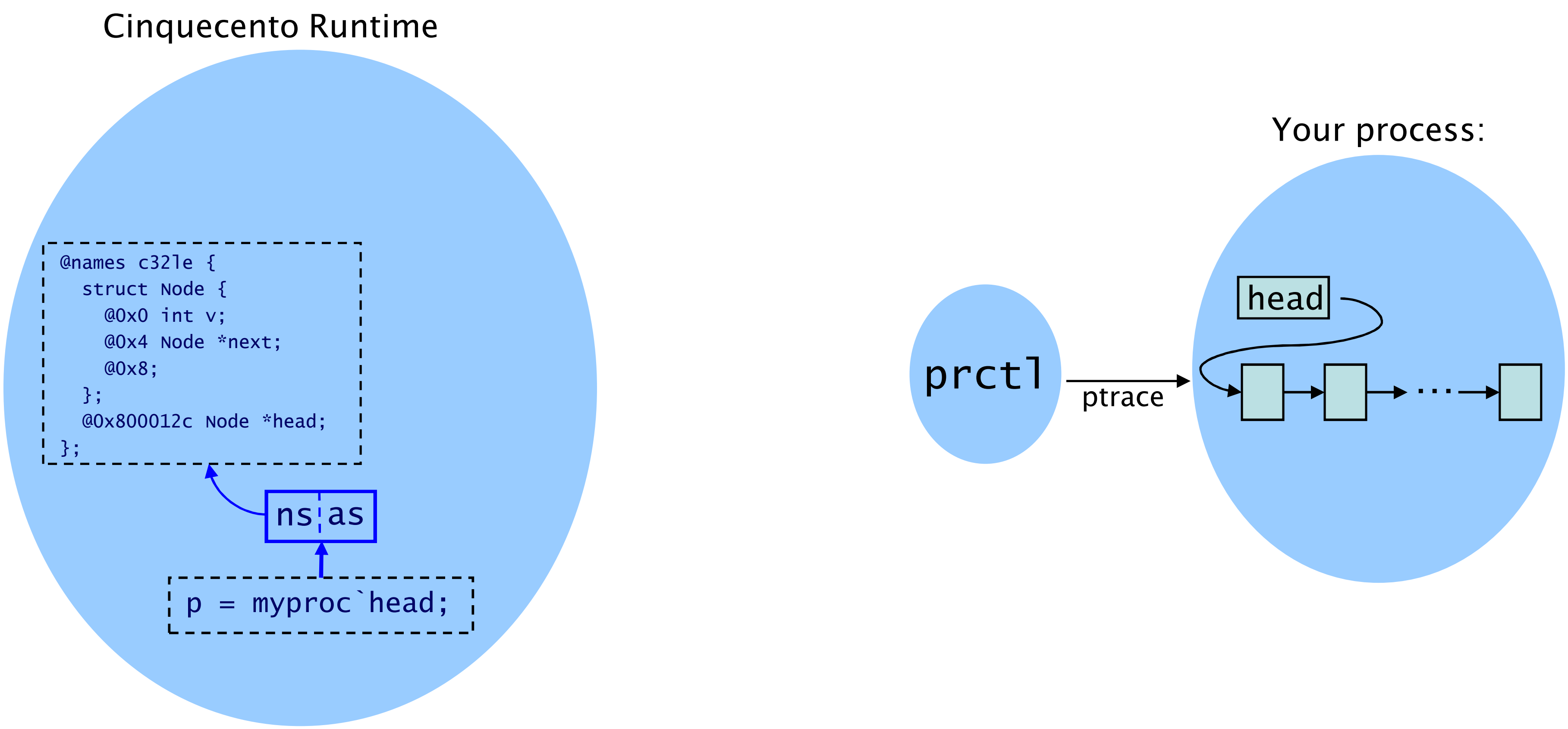
```
ns = @names c32le {
    // type definitions
    typedef struct Node Node;
    struct Node {
        @0x0      int v;
        @0x4      Node *next;
        @0x8;
    };

    // symbol definitions
    @0x800012c    Node *head;
};
```

dwarf2cqct: generates Cinquecento @names from executable

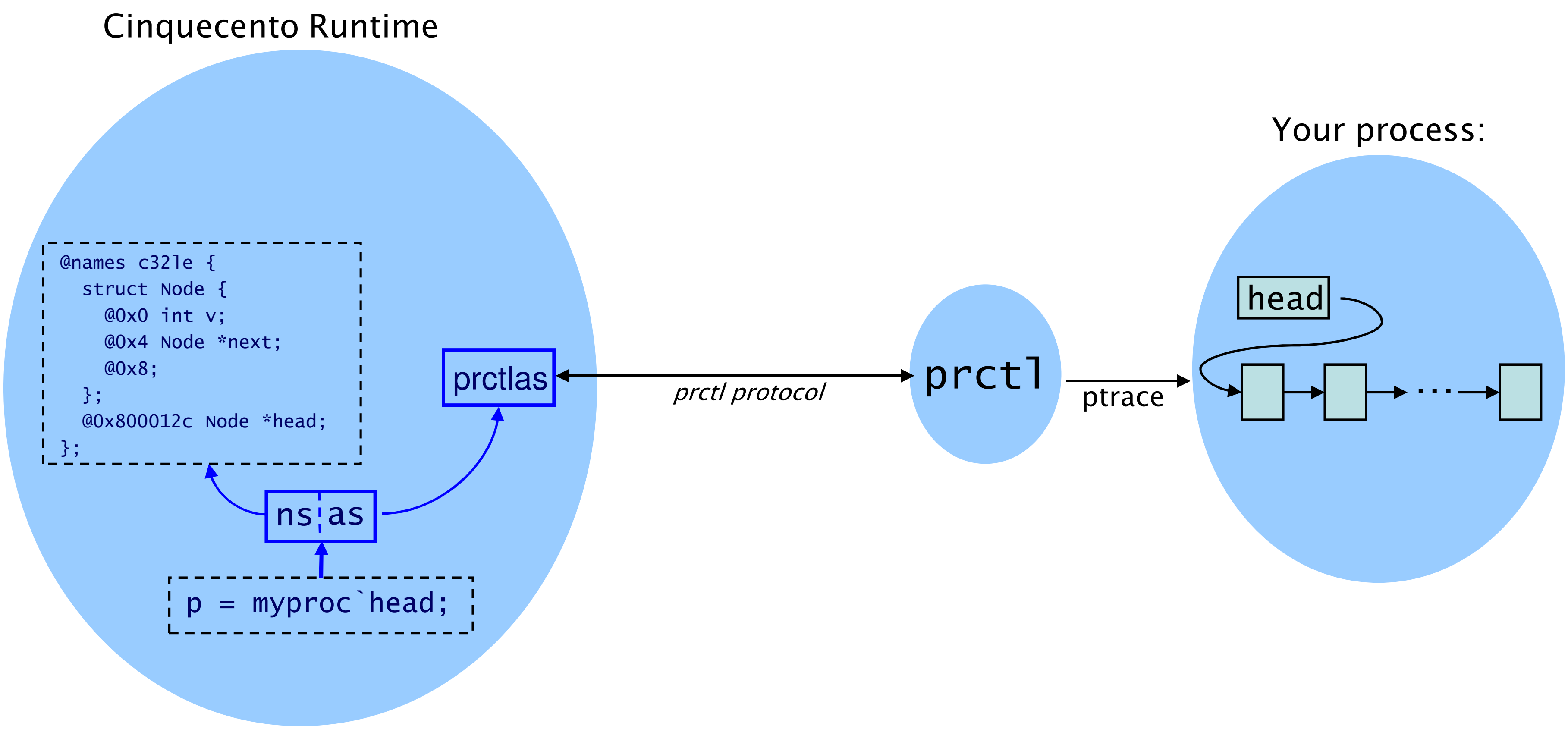
prctl: Linux Process Address Space



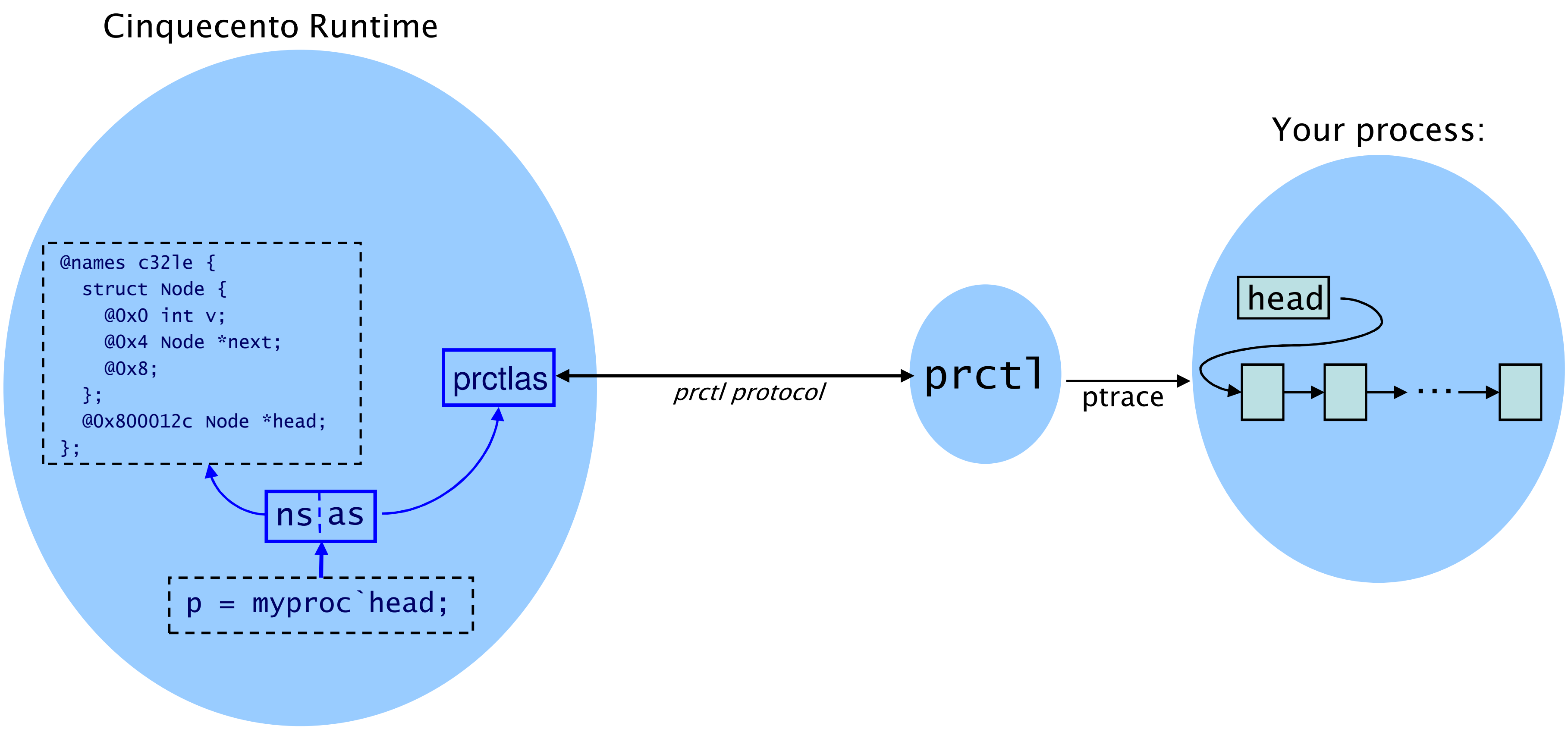
prctl: Linux Process Address Space



prctl: Linux Process Address Space

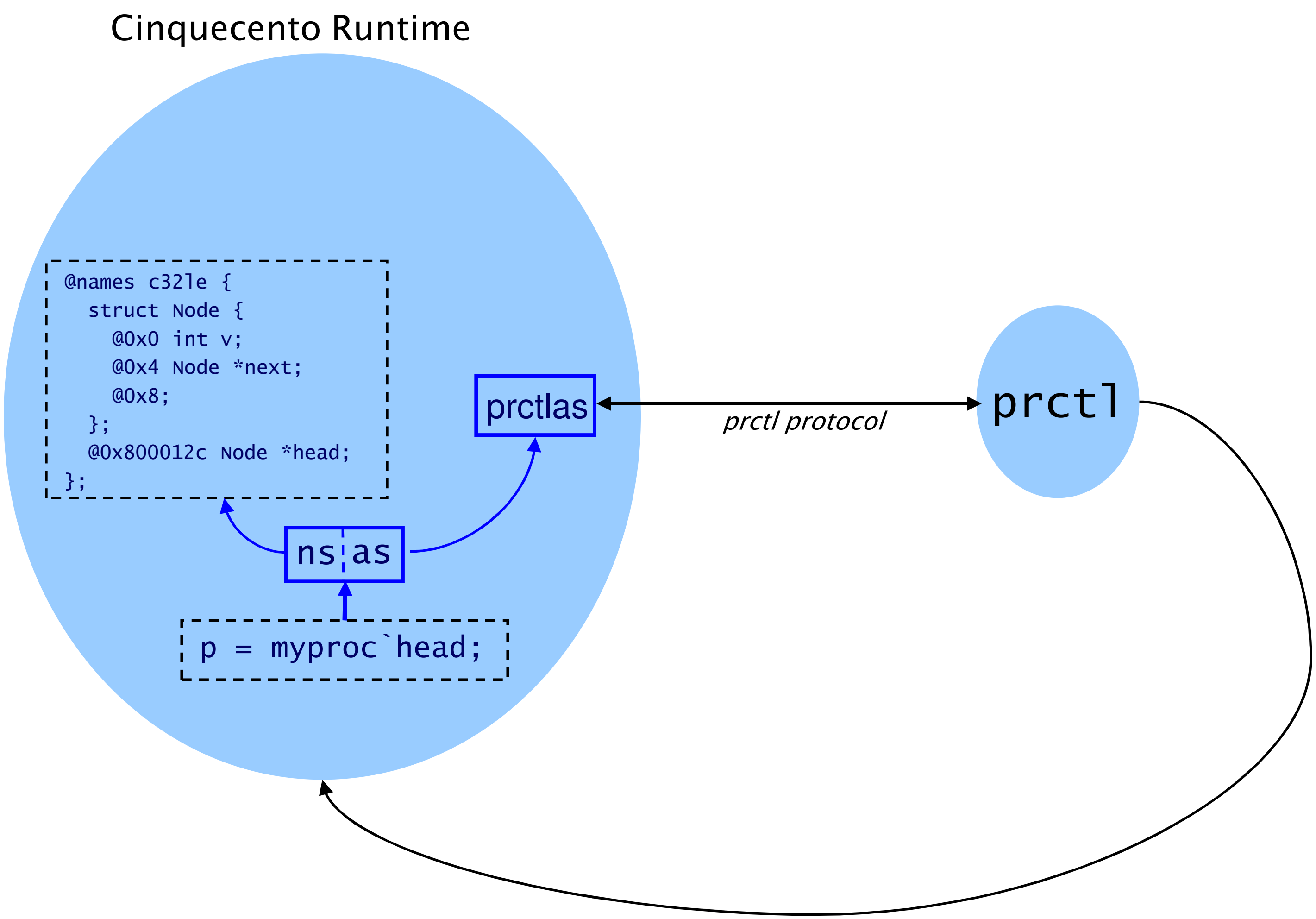


prctl: Linux Process Address Space




```
@include <prctl.cqct>
as1 = launchlocalproc("/bin/myprog", "-c", "arg");
as2 = attachlocalproc(19120);
as3 = attachremoteproc("asiago:30000", 10210);
```

prctl: Linux Process Address Space



Process Control

Process control is exposed through address space interface:

<u>Continue, pause, step</u>
```
as.xcont()
as.xbreak()
as.xstep()
```

<u>Read and write registers</u>
```
as.geteax()
as.seteax(val)
```

<u>Set and clear breakpoints</u>
```
as.bpset(addr, function)
as.bpdel(id)
```

This design is *convenient*

- Process control and memory access go through related OS interfaces

But not quite *right*: many unresolved issues

- multi-threaded targets
- orchestrating control of multiple targets
- register naming
- breakpoint abstractions vs. implementations

Likely future: a separate control abstraction in the domain

Local Variables

We want to reference local variables in breakpoint handlers.

Solution: Breakpoint handler constructs *local domain*

- Same address space
- New name space, defining lexically visible symbols

More general: a local domain per stack frame

- `unwind()` returns list of local domains
- Enables code to compare state in different frames

Systems Programming in Cinquecento

Address spaces are not just for processes.

We make address spaces out of

- anonymous memory (think: mmap)
- strings
- files
- streaming data (network connections)

Example: prctl protocol (`@include <prctl.cqct>`)

- Need to marshal, unmarshal, and validate messages
- This is the sort of thing C is good for
- So: why not do it in Cinquecento?

```
protns = @names c32le {
   struct Tlaunch{
            @0          uint8 op;
            @1          uint64 narg;
            @9;
   };

   struct Rlaunch{
            @0          uint8 op;
            @1          uint64 id;
            @9          Reg reg;
            @153;
   }
};
```

```
fd = opentcp(addr);
fdas = mkfdas(fd);
indom = mkdom(protns, fdas[0]);
outdom = mkdom(protns, fdas[1]);
```

Systems Programming in Cinquecento

```
define launch(domas, arg){
          @local argvlen, narg, i;
          narg = length(arg);

          /* format Tlaunch message */
          argvlen = 0;
          for(i = 0; i < narg; i++)
                    argvlen += strlen(arg[i]))+1;
          xwr((`uint64)(sizeof(outdom`Tlaunch)+argvlen));
          p = (outdom`Tlaunch*)opress(sizeof(outdom`Tlaunch));
          p->op = outdom`Tlaunch;
          p->narg = narg;
          for(i = 0; i < narg; i++){
                    xwr(listref(arg, i));
                    xwr((`uint8)0);
          }
          outdom.flush();

          /* receive Rlaunch reply */
          reply = read1();
          checkreply(indom`Rlaunch, reply);
          id = reply->id;
          reg = copydata(&reply->reg);
          indom.flush();
          state = indom`Stopped;
}
```

Systems Programming in Cinquecento

- Other examples
  - executables (`@include <elf.cqct>`)
  - core files (`@include <core.cqct>`)
  - debug info (`@include <dw.cqct>`)
  - packet captures (`@include <pcap.cqct>`)
  - git blobs (`@include <git.cqct>`)
  - infocom z machine (`@include <z.cqct>`)
- Cinquecento: debugger *implementation* language
  - as well as debugger *operation* language
- Increasingly Cinquecento is replacing C in our daily lives:
  - Pointers, C types
  - Garbage collection
  - Control over mappings in the address space

Status

- Bits I left out
  - Snappoints (language interface to back-in-time debuggers)
  - First-class C types, ctype API

- Future
  - Distributed evaluation of Cinquecento expressions
  - Macros
  - C++

- Implementation
  - www.cqctworld.org
  - Manual, tech report, library
  - `l1`: C-based implementation for Linux and OS-X
  - `prctl`, `dwarf2cqct`: as and ns for Linux programs
  - `cqct`: Original Scheme-based implementation

Mixed-domain expressions

- You can combine and compare cvalues of different domains

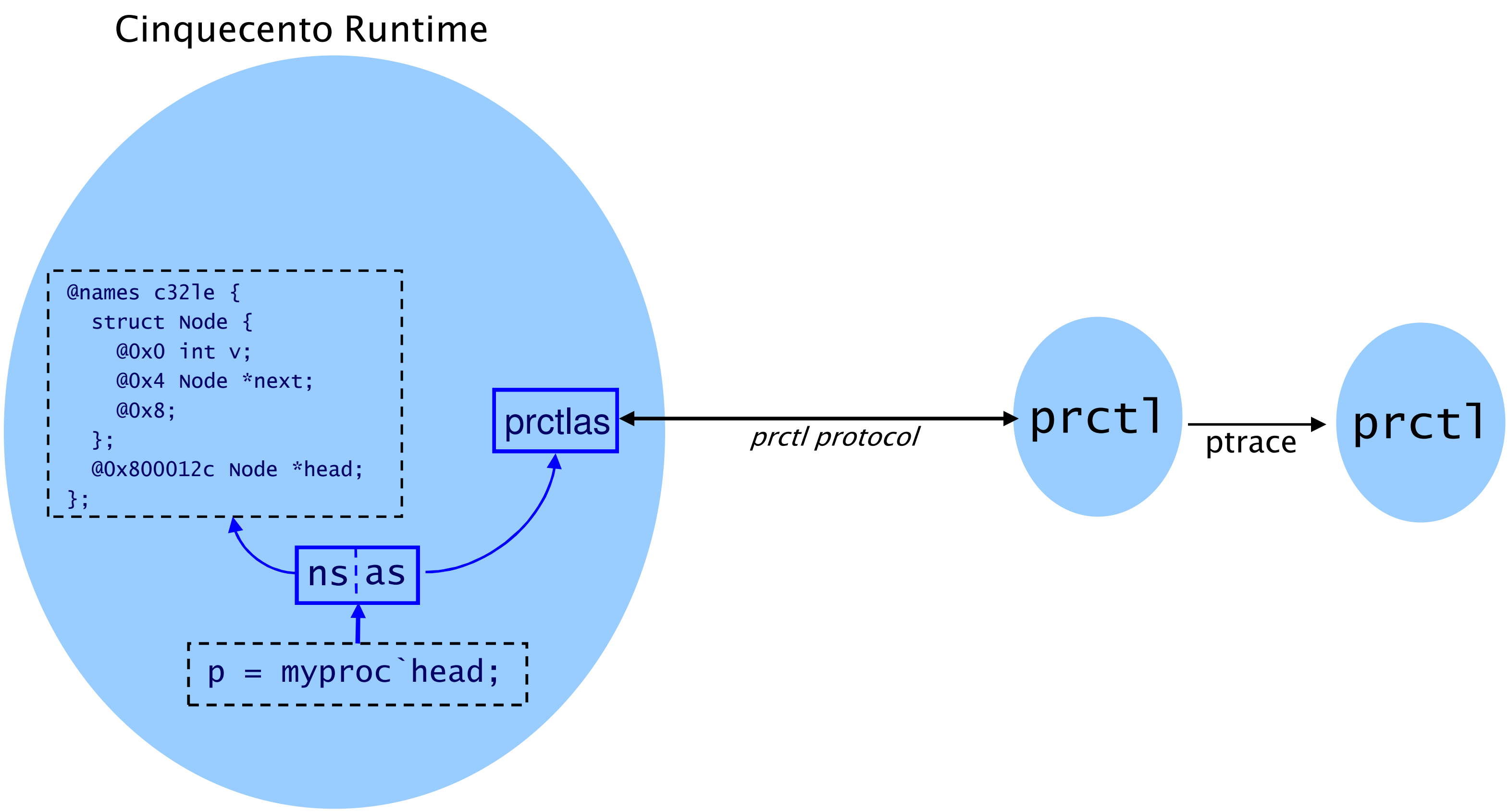
```
ap = A`head;
bp = B`head;
if(ap->v != bp->v)
  error("mismatch");
```

- Even if the domains have different type definitions (e.g., 32-bit vs. 64-bit `long`)
  - Idea: extend C "usual conversions" to promote operands to common domain
  - Subtle semantics: see tech report

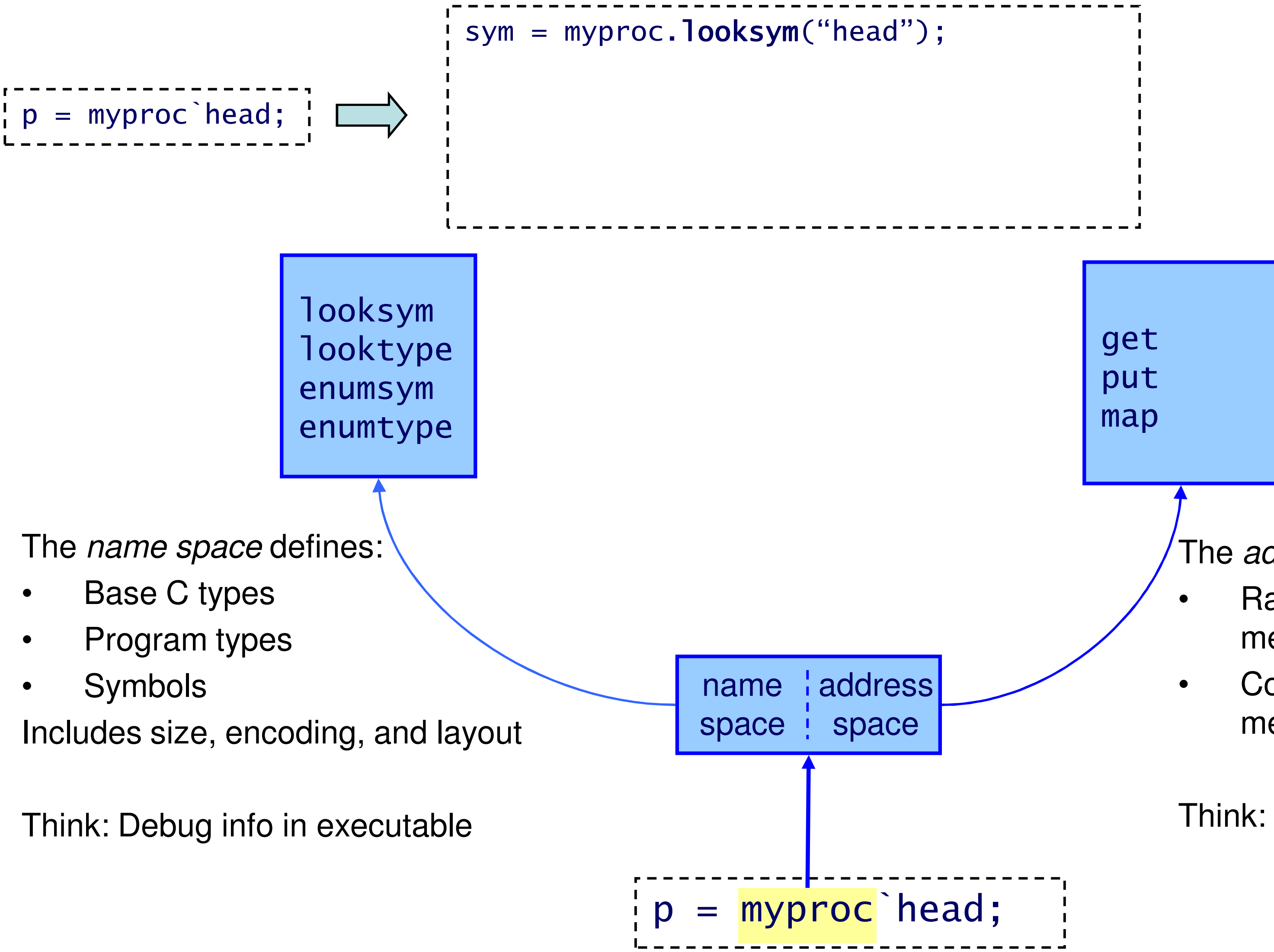
- *Domain cast* operator allows explicit domain conversion

```
xp = {B}ap;  // change domain of ap to B
```

`prctl`: Linux Process Address Space



The Structure of Domains



The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

The *address space* defines:

- Range of addressable memory
- Contents of addressable memory

Think: /proc/*pid*/maps + ptrace

The Structure of Domains

```
p = myproc`head;
```

```
sym = myproc.looksym("head");
addr = symaddr(sym);
type = symtype(sym);
```

```
looksym
looktype
enumsym
enumtype
```

```
get
put
map
```

The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

The *address space* defines:

- Range of addressable memory
- Contents of addressable memory

Think: /proc/*pid*/maps + ptrace

name space | address space

```
p = myproc`head;
```

The Structure of Domains

```
p = myproc`head;
```

```
sym = myproc.looksym("head");
addr = symaddr(sym);
type = symtype(sym);
bytes = myproc.get(addr, sizeof(type));
```

```
looksym
looktype
enumsym
enumtype
```

```
get
put
map
```

The *name space* defines:

- Base C types
- Program types
- Symbols

Includes size, encoding, and layout

Think: Debug info in executable

name space | address space

The *address space* defines:

- Range of addressable memory
- Contents of addressable memory

Think: /proc/*pid*/maps + ptrace

```
p = myproc`head;
```