Source control

- Testing is often linked with source control
- Testing is used to drive functionality
    - Tests tell you what is currently broken
    - Modifications reduce the number of faults
    - You ***incrementally change*** the system to make it better
- Source control is used to track ***incremental changes***
    - Each modification adds more functionality
    - *Some* modifications break functionality
    - We want to keep the last working version available

Source control = version control

- Version control can be done manually
  - Keep source files with different suffices:
    - `prog_v0.cpp, prog_v1.cpp, prog_v2.cpp,` …
    - Most projects rely on more than just one source
  - Keep snapshots based on date+time:
    - `prog-2019-10-01.tar.gz, prog-2019-10-02.tar.gz,` …
    - Difficult to see what has changed between snapshots
- Source control automates version management
  - Easier to remember what you changed
  - More difficult to lose changes

Source control = backup + collaboration

- Most modern source control is ***distributed***
  - There are multiple copies of the project’s source files
  - Copies are held on many machines in many locations
  - Copies are frequently synchronised between machines

- Most modern source control is ***concurrent***
  - Lots of people work on their own copy independently
  - Changes get merges when copies are synchronised
  - Conflicts between changes are addressed while merging

We are going to use git

- Git is now ***the*** dominant method for source control
  - Though there are a few other options out there
- Used widely across all fields of software
  - Standard for open-source
  - Very common in industry
- Also used outside software to manage files
  - Common for digital design and document control
- Supported by some well-known infrastructure
  - Github, gitlab, Microsoft, …

Basic concepts in git : repositories

- ***Repository*** : a directory representing your project
  - Files within the repository will be versioned
  - Each file has it’s own history

- ***Local*** repository : the repository on your computer
- ***Remote*** repository : a repository somewhere else
  - Could be a repository on someone else’s laptop
  - Could be a repository stored in github

Learning about git

- We’re going to introduce git incrementally
- This term is all single user
    - Only one person (you) will be working in a repository
- Next term will be multiple user
    - Need to deal with conflicts
- Eventually git will be used to manage submissions

Testing and change management

- Testing is critical to getting a working program
  - In your study: making sure you pass assessments
  - In industry: making sure you deliver a working system
- Testing is part of the larger software lifecycle
  1. Requirements gathering
  2. Design
  3. Implementation
  4. Testing
  5. Maintenance
- Source control goes hand-in-hand with testing
- An example of something that needs testing is…

Pointers

Pointers : the big idea

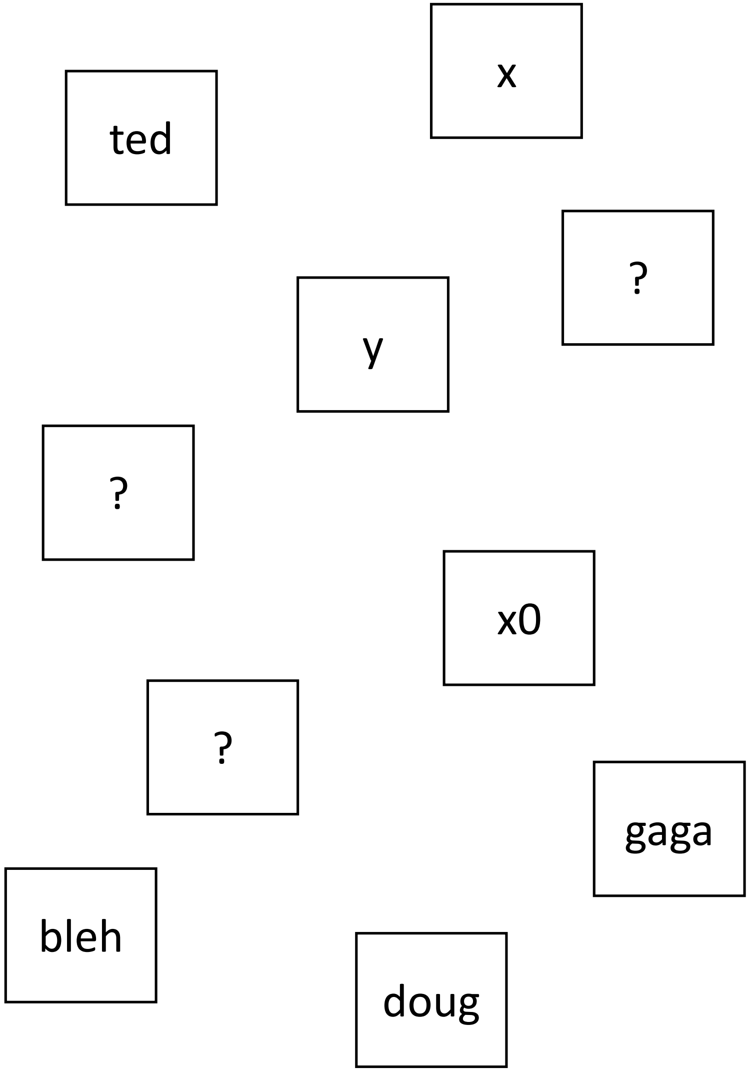
A “pointer” can do one of two things:

1. point at something
2. point at nothing

(It can’t do both at the same time)

If you have a pointer to something

then you can easily access the thing it points to



x

ted

?

y

?

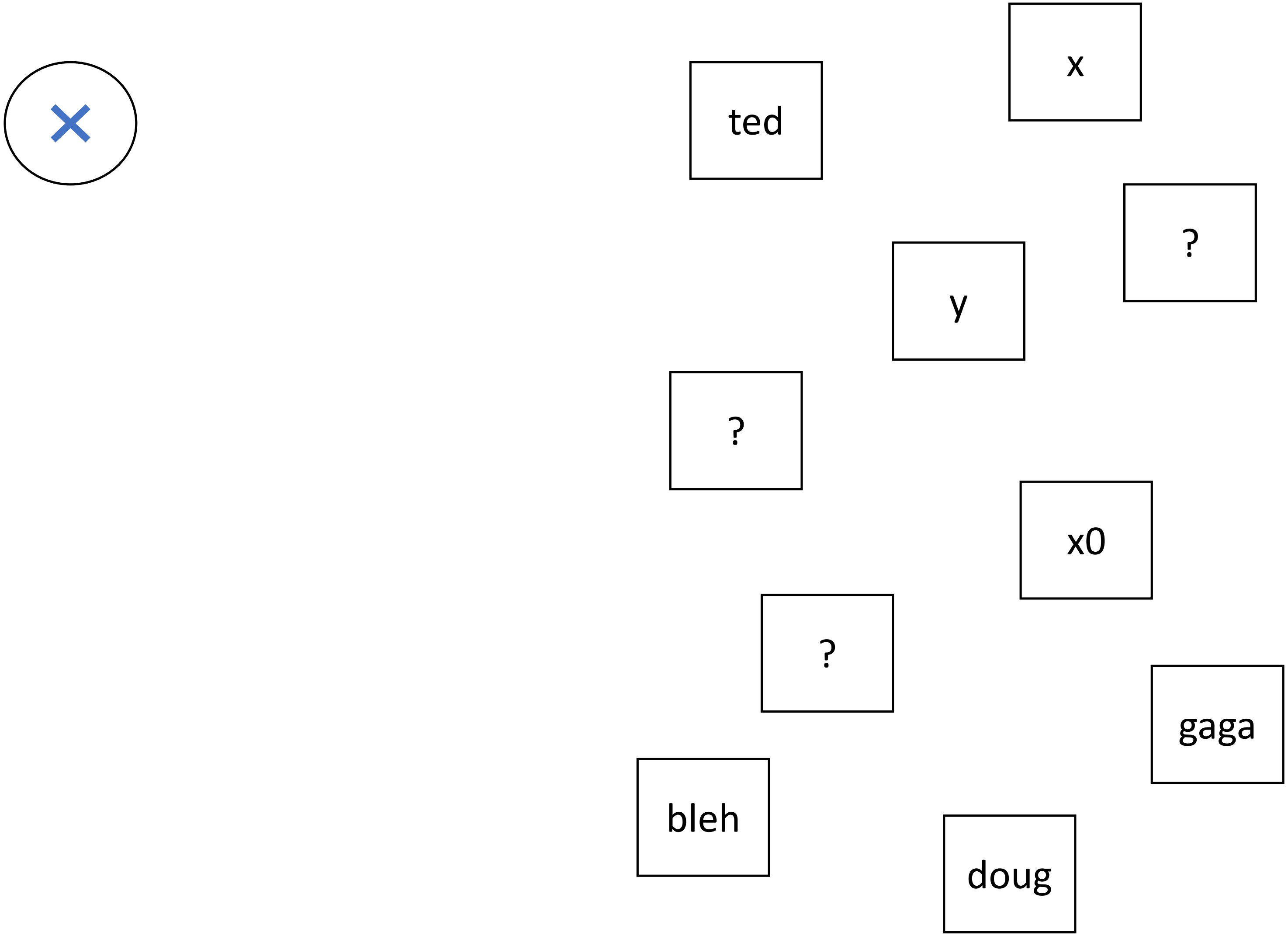
x0

?

gaga

bleh

doug



x

ted

?

y

?

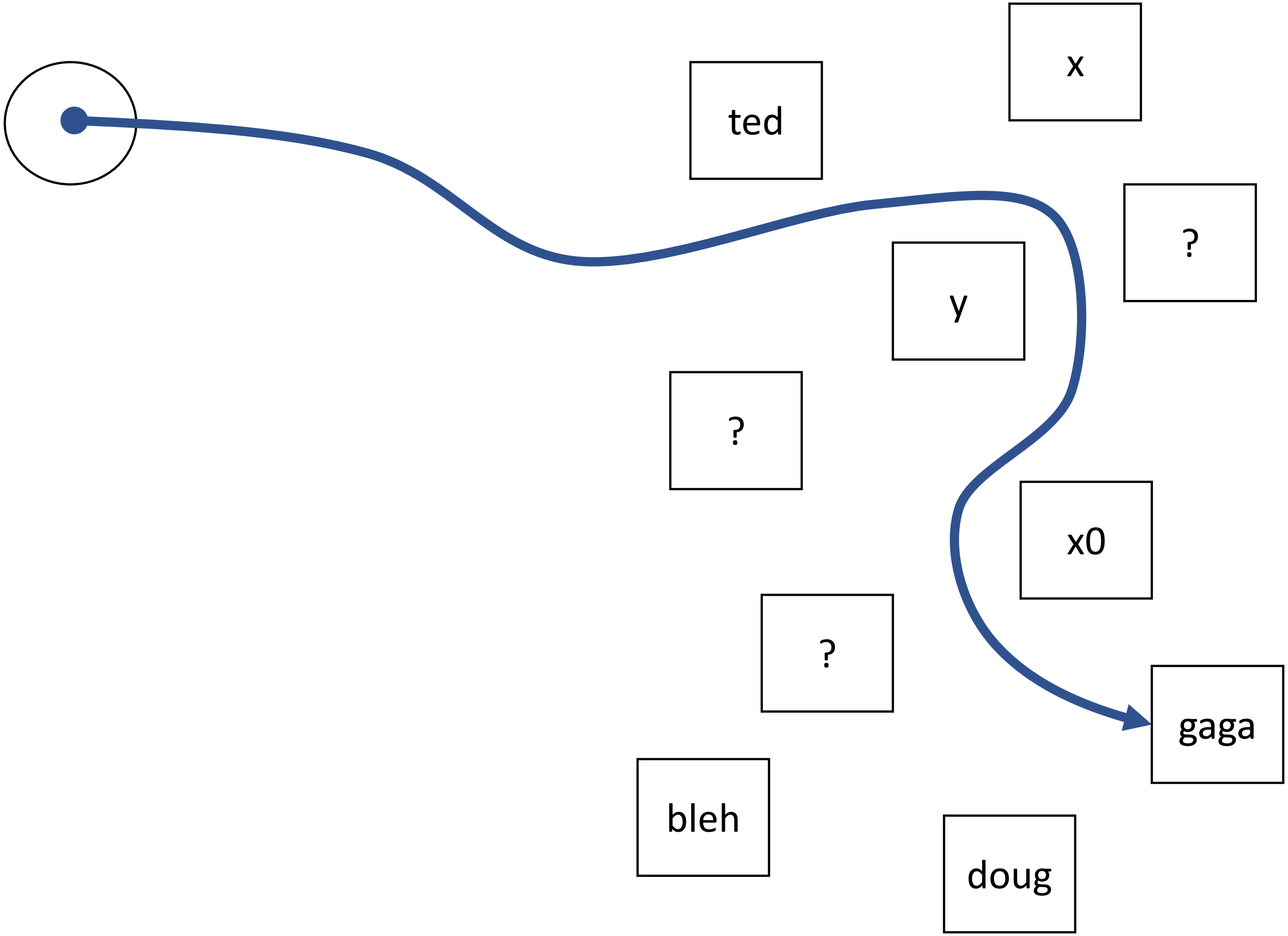
x0

?

gaga

bleh

doug



x

ted

?

y

?

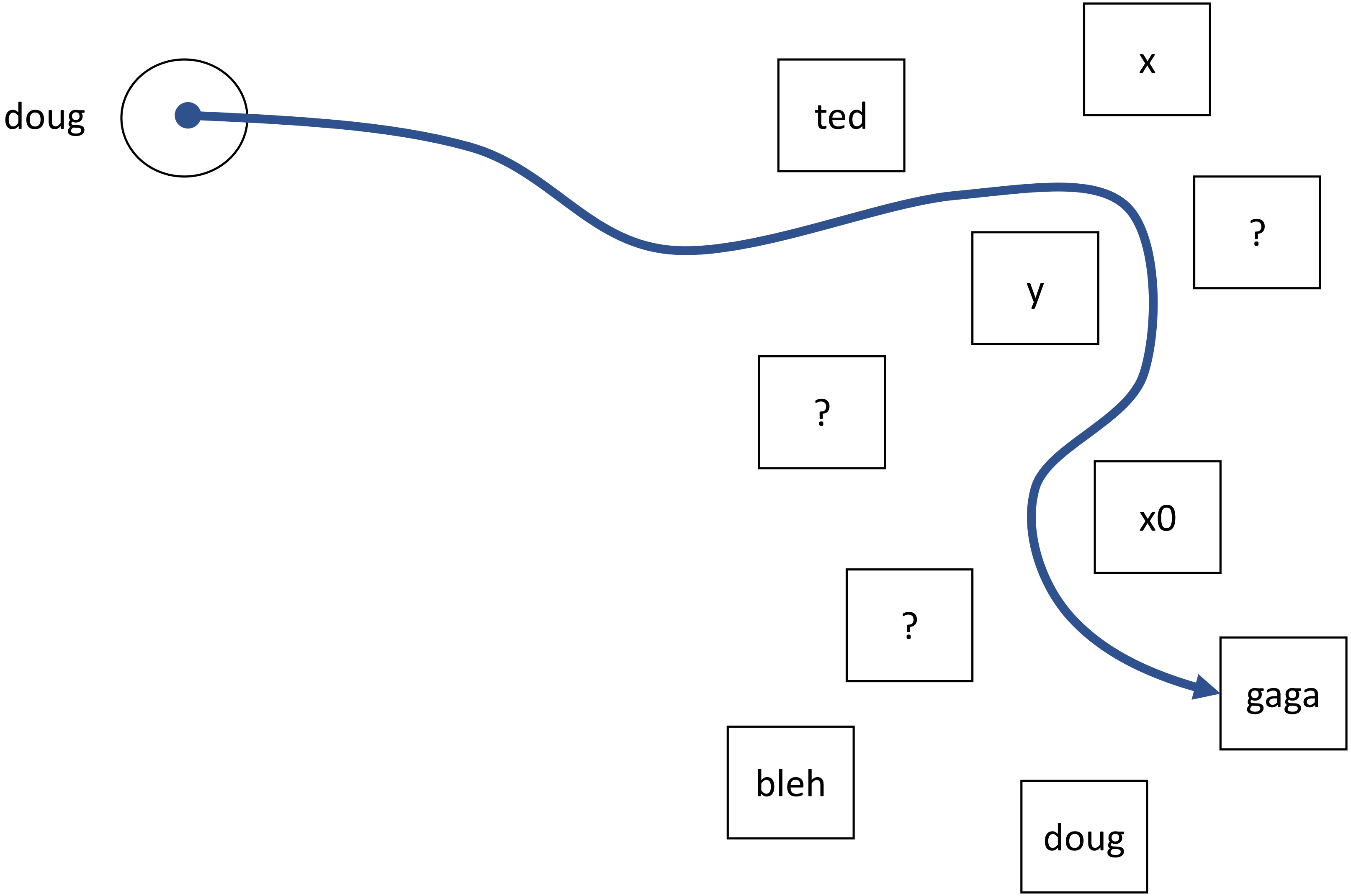
x0

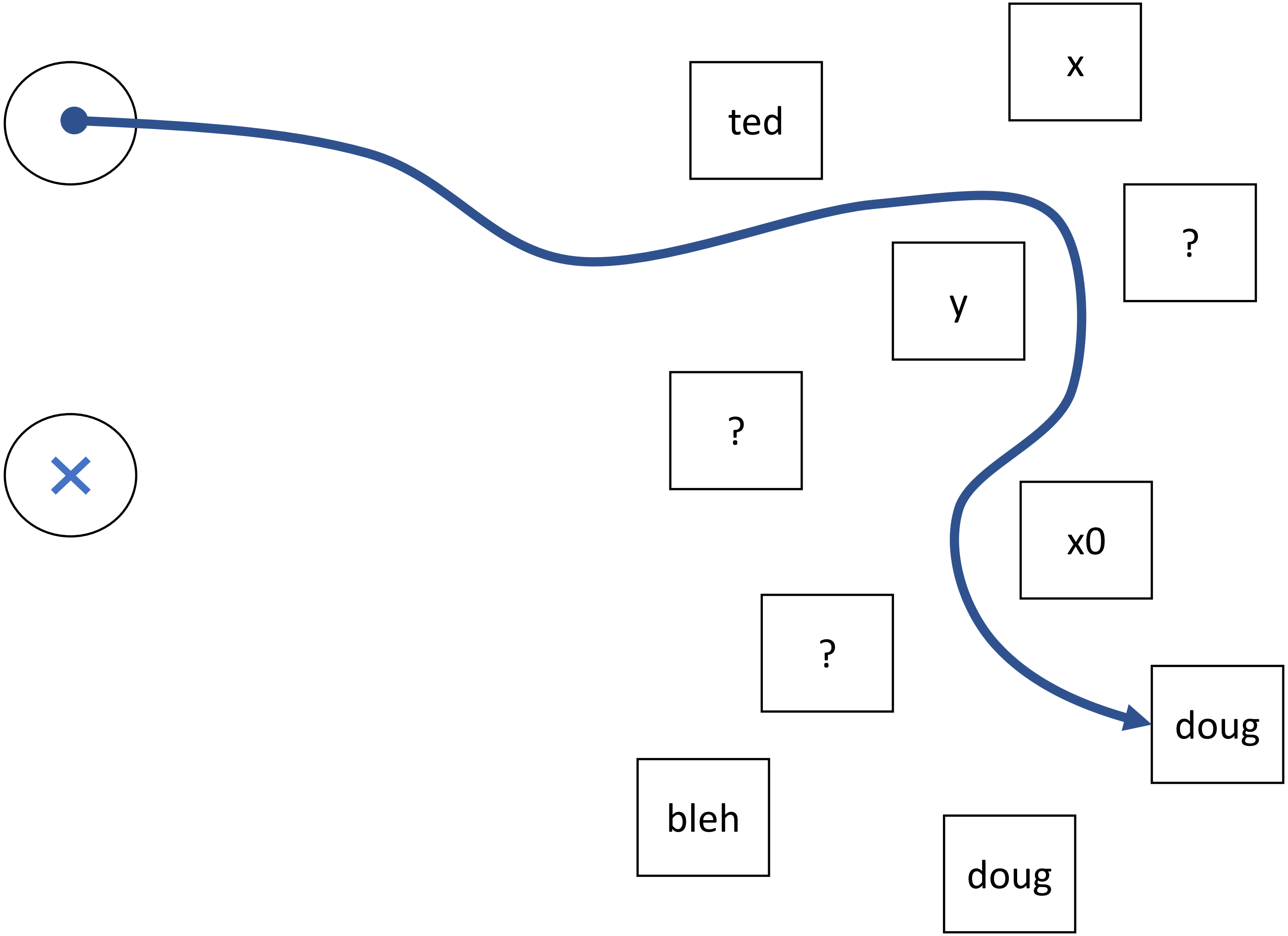
?

gaga

bleh

doug


doug
ted
x
?
y
?
x0
?
gaga
bleh
doug



ted

x

?

y

?

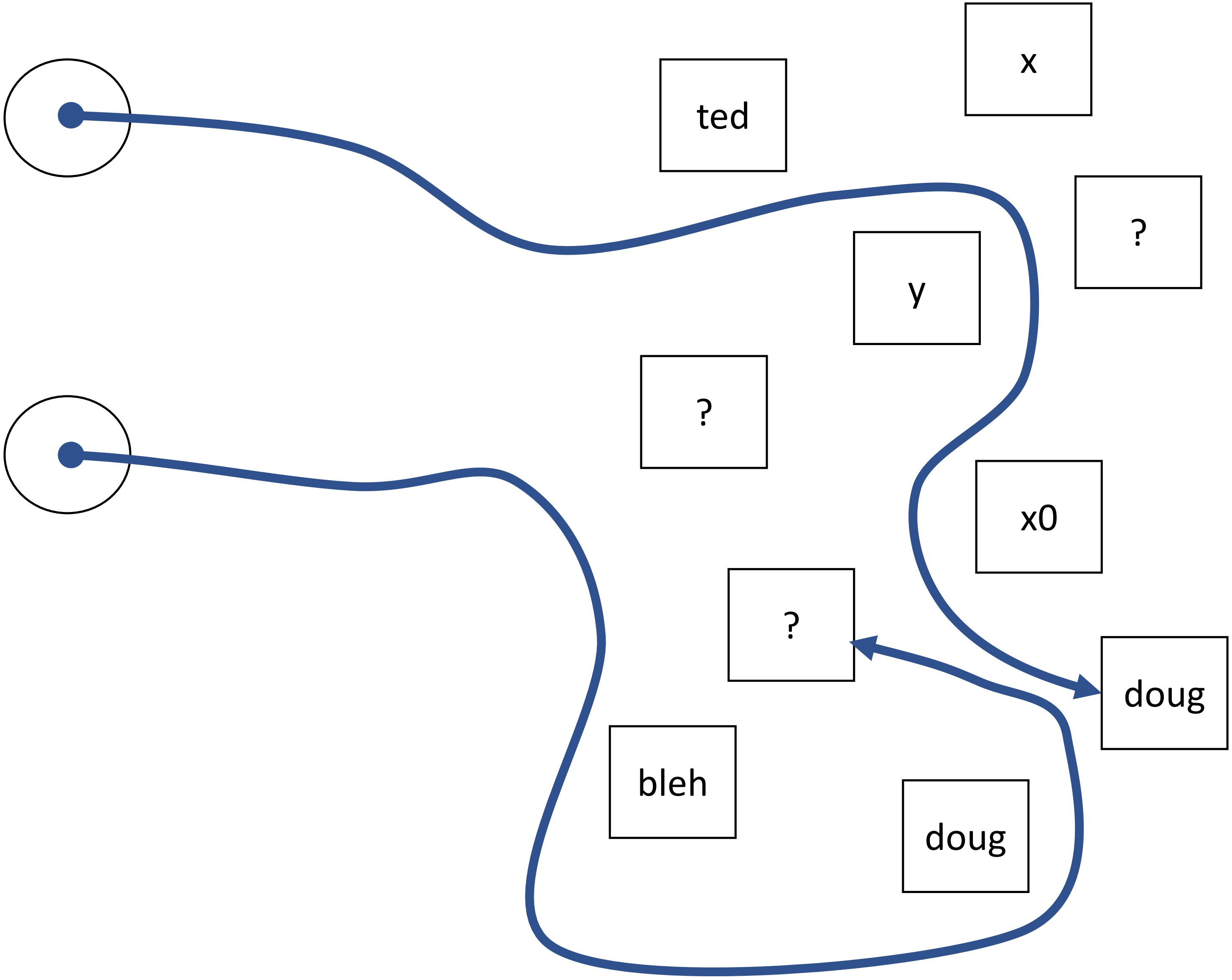
x0

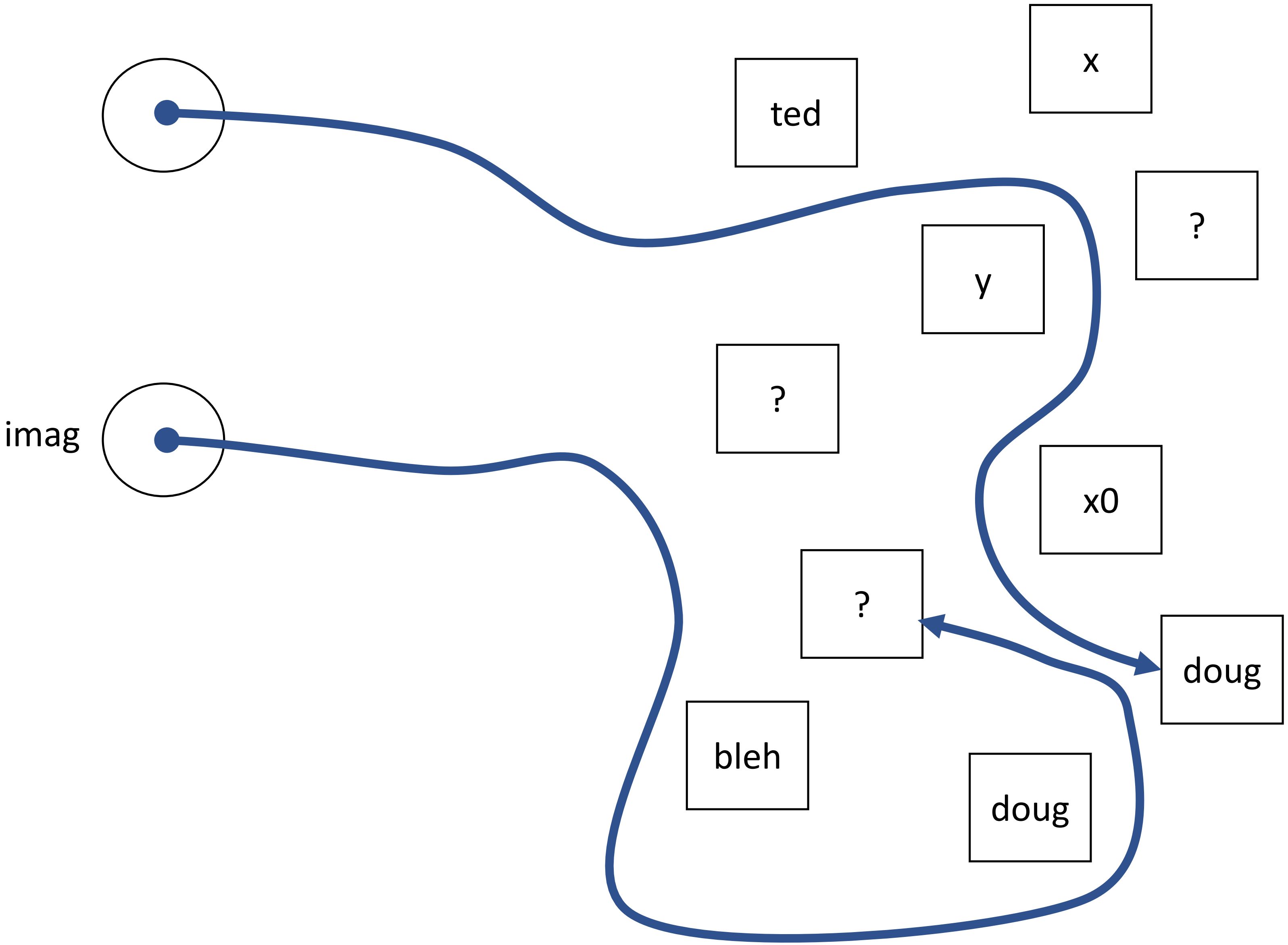
?

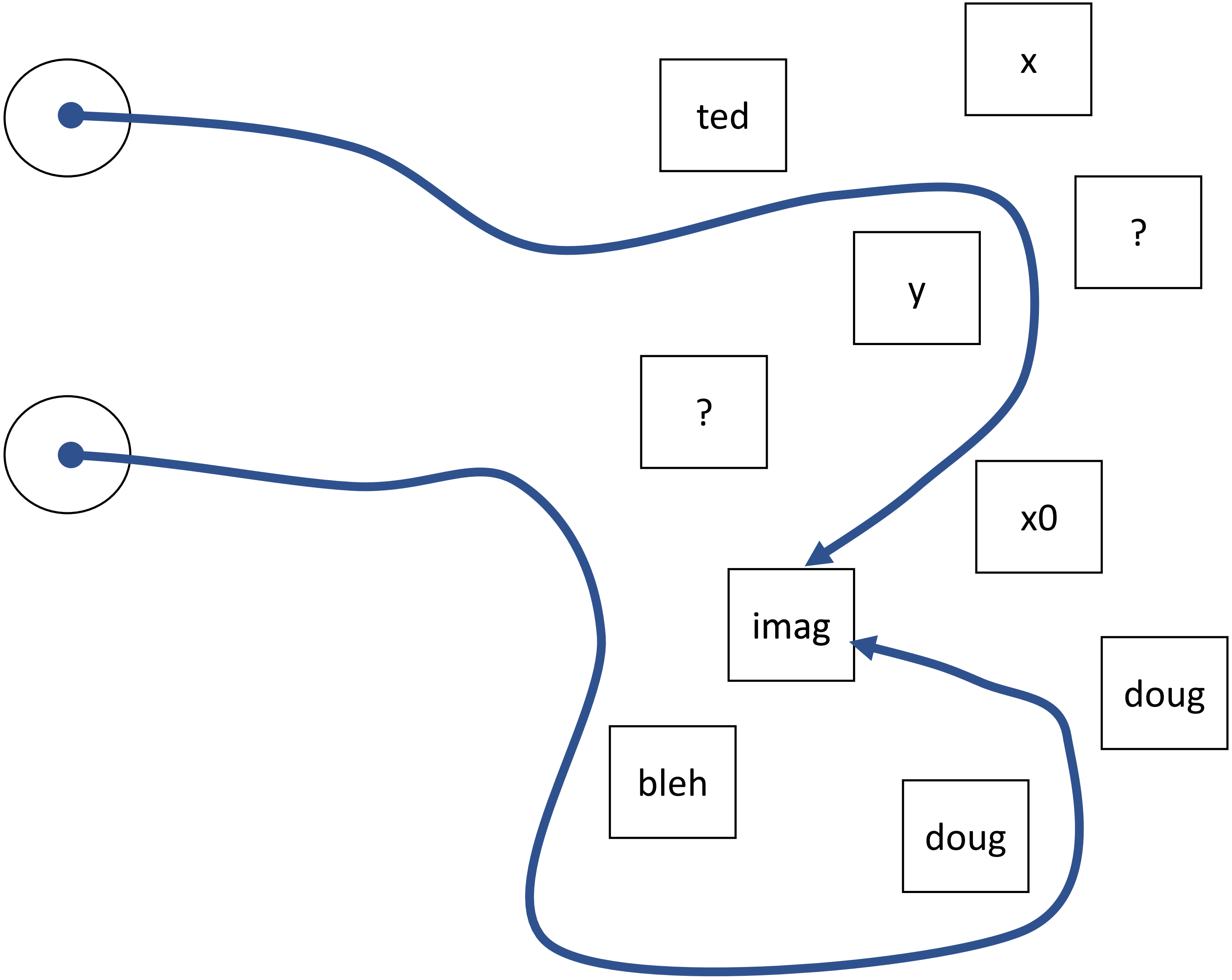
doug

bleh

doug


x
ted
?
y
?
x0
?
doug
bleh
doug


ted
x
?
y
?
imag
x0
?
doug
bleh
doug



x

ted

?

y

?

x0

imag

doug

bleh

doug

Pointers : the big idea

A “pointer” can do one of two things:

1. point at something
2. point at nothing

If you have a pointer to something
then you can easily access the thing it points to

The type of pointer types

Each pointer can only point to one type of thing

For any type T, we have the type “pointer to T”

- “Pointer to int” `int *`
- “Pointer to string” `string *`
- “Pointer to vector<int>” `vector<int> *`

A asterisk/star ‘*‘ converts a type to “pointer to type”

Creating pointer variables

Given

  a “pointer type” is a type

and

  we can create variables of any given type

then

  we can create variables with a “pointer type”

```
int *pi;          // pi can point at integers
float *pf;        // pf can point at floats
vector<int> *pvi; // pvi points at vectors of int
```

Creating pointer values

Given

a value of type “pointer to T” points at something of type T

and

we can only point at things that exist

then

1. the thing must exist before a value that points to the thing
2. the thing should exist for as long as the pointer value

Being a bit more precise

Type : *a set of possible values*

Value : *an element from a set*

Instance : *a location containing a value*

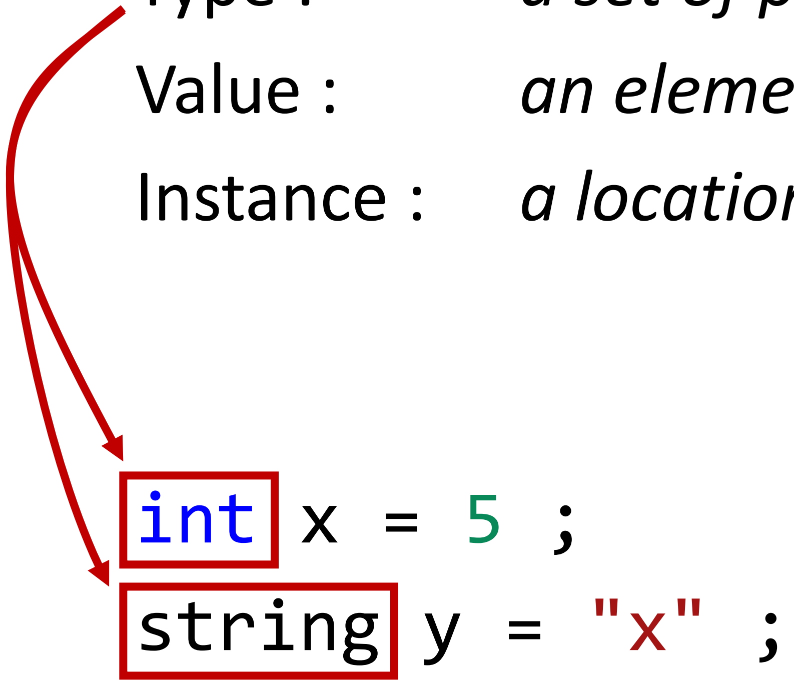
```
int x = 5 ;
string y = "x" ;
```

Being a bit more precise

Type : *a set of possible values*

Value : *an element from a set*

Instance : *a location containing a value*



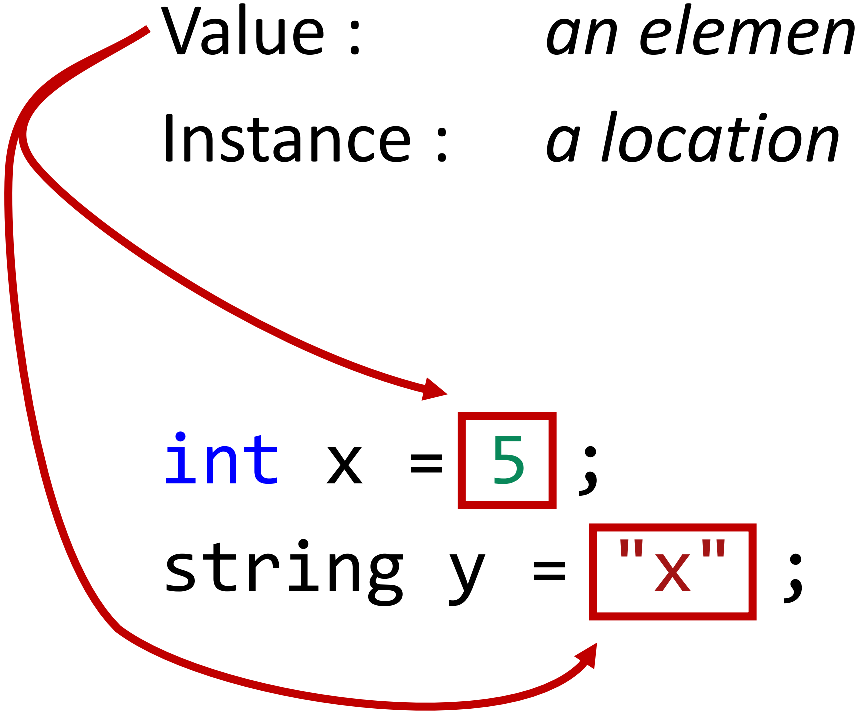
```
int x = 5 ;
string y = "x" ;
```

Being a bit more precise

Type : *a set of possible values*

Value : *an element from a set*

Instance : *a location containing a value*



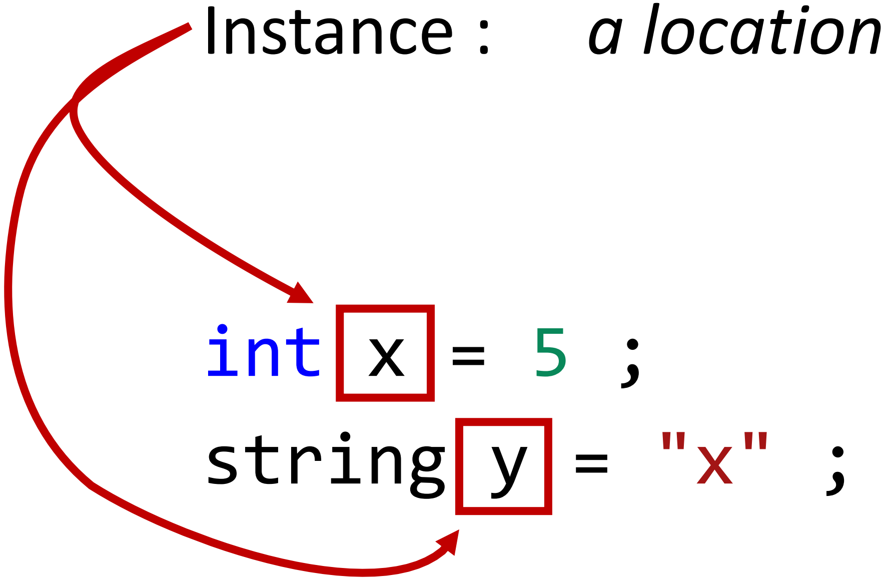
```
int x = 5 ;
string y = "x" ;
```

Being a bit more precise

Type : *a set of possible values*

Value : *an element from a set*

Instance : *a location containing a value*



```
int x = 5 ;
string y = "x" ;
```

Note: This is not *quite* true for "x" as we'll see in a bit

Creating pointer values

Given

a value of type “pointer to T” points at an in variable pe T

and

we can only point at variables of type T that already exist

then

1. the variable must exist before a value that points to the variable
2. the variable should exist for as long as the pointer value

We create pointer values using the address-of ’&‘ operator

```
int i;                    float f;
int *pi = &i;       float *pf = &f;
```

Getting back to the instance

If

  we have a pointer to an instance

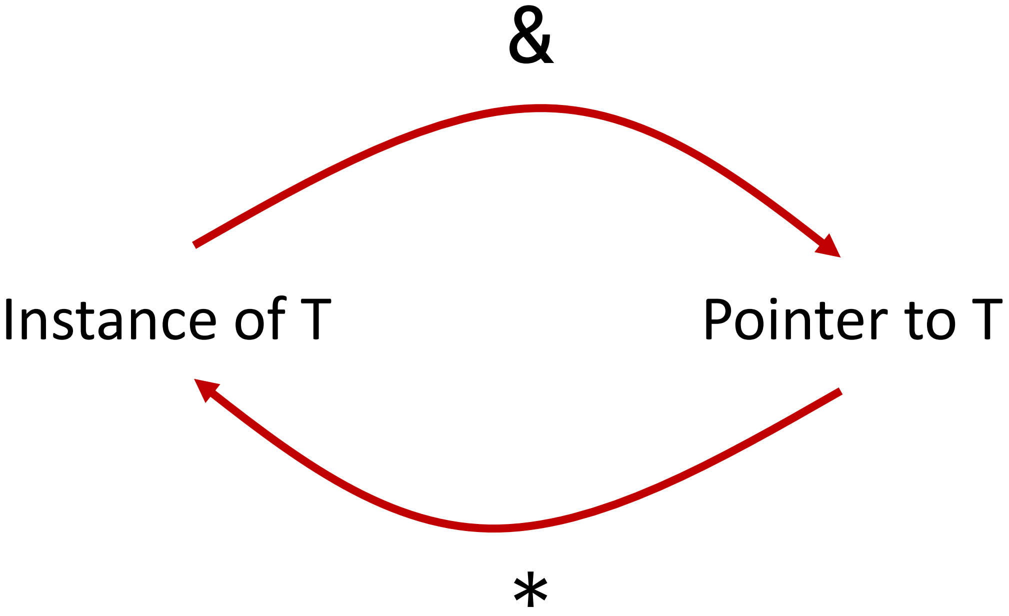
then

  we can read or write the instance

We can ***de-reference*** a pointer value using ‘*’

```
int i;             float f;
int *pi = &i ;     float *pf = &f ;
int z   = *pi ;    float g   = *pf ;
```

address-of (&) vs. de-reference (*)



&

Instance of T

Pointer to T

*

```
int i = 5;
cout << * ( &i );
```

```
int *p = &instance;
int *q = & ( *p ) ;
```

```
int a;
int *b  = &a;
int **c = &b;
cin >> * ( *c );
cout << a;
```

FAQ : What can I point to?

Q : What's the difference between values and instances?

A : An instance is something you can write to

or

if you can assign to it, you can create a pointer to it

```
int x;

x = 5;
~~5 = x;~~
```

```
int x;
int *p ;
p = &x;
p = ~~&5;~~
```

The technical term for an instance in C/C++ is an "lvalue" : a location value, or "value with a location"

Creating pointer parameters

Given

a “pointer type” is a type

and

we can create parameters of any given type

then

we can create parameters with a “pointer type”

```
void f(int *p)
{
  *p = 10;
}
```

```
void swap(int *pa, int *pb)
{
  int tmp = *pa;
  *pa     = *pb;
  *pb     = tmp;
}
```

The null value

A pointer value can do one of two things:

1. point at something
2. point at nothing

`nullptr` is “the value that points at nothing”

(the integer constant 0 is also allowed)

Never de-reference the null value!

Warning : the twilight zone

A pointer value can do one of two things:

1. point at something
2. point at nothing

What if it does *neither*?

Twilight 1 : Uninitialised pointers

What does a fresh pointer variable point to?

```
int main()
{
  int *bella;
  if( bella != nullptr){
    cout << *bella;
  }
}
```

```
int main()
{
  int ed;
  if( ed !=0 ){
    cout << ed;
  }
}
```

Twilight 2 : Dangling pointers

What happens when a variable no longer exists?

```
int *choose_team()
{
    int jacob = 3;
    int *p = &jacob;
    return p;
}
```

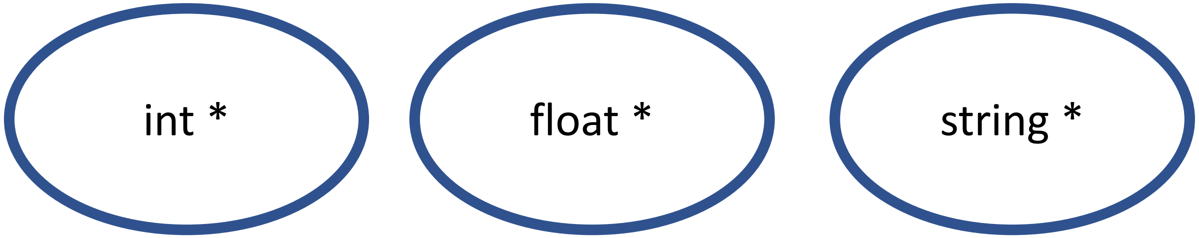
This was missing in the original version, so it was a different error than intended

Twilight 3 : Dereferencing nothing

```
*nullptr
```

A mathematical view of pointers

Pointers to different types are disjoint sets



… and that’s all we’ll say

We could do it, but:

- Mainly useful for EIE : *what is “information”*?
- The maths required is… different
- It would impede understanding, rather than help

Pointers in practise

```
void add1(int *p)
{
   *p = *p + 1;
}
```

```
int main()
{
   int x;
   cin >> x;
   int *r = &x;
   add1( r );
   cout << x;
}
```

```
int main()
{
   point p1 = {4,5};
   add1( &p1.x );
   cout << p1.x;
}
```

```
int main()
{
   vector<int> v = read();
   for(int i=0;i<v.size();i++){
     add1( &v[0] );
   }
   write(v);
}
```

Pointers : summary

- Pointers point at instances of things
  - You create a pointer to an instance with &
  - You get back to the instance using *
- Pointers are a bit dangerous in C++
  - You have to be a bit careful about what you write
  - If you can avoid pointers you should

- The true value of pointers comes next
  - How do we implement infinite sized types?
  - How can we create irregular types?