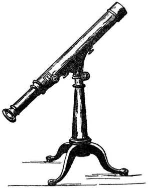
Names, Scope, and Types

Stephen A. Edwards

Columbia University

Fall 2018



Scope

Types

Types in C

Types of Type Systems

Overloading

Binding Time

What's Wrong With This?

```
a + f(b, c)
```

What's Wrong With This?

```
a + f(b, c)
```

Is `a` defined?

Is `f` defined?

Are `b` and `c` defined?

Is `f` a function of two arguments?

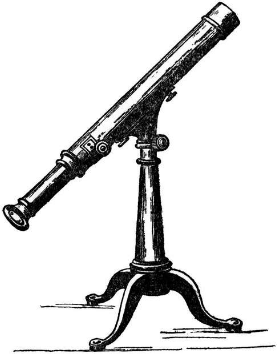
Can you add whatever `a` is to whatever `f` returns?

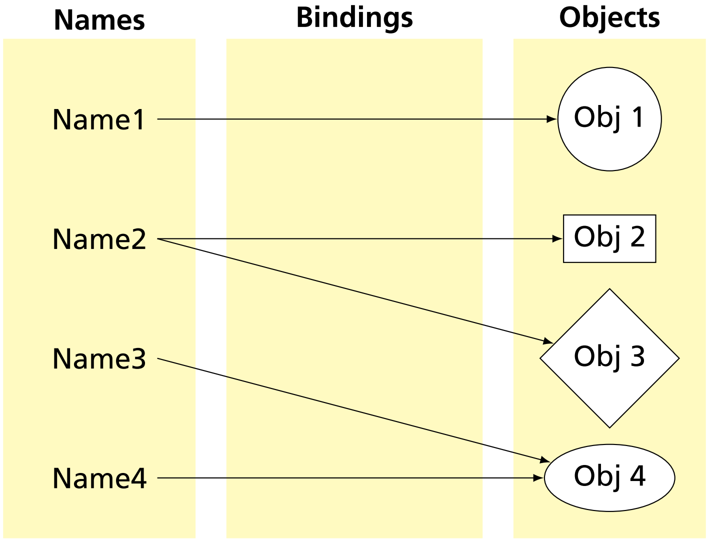
Does `f` accept whatever `b` and `c` are?

Scope questions   Type questions

Scope

What names are visible?




Names
Bindings
Objects
Name1
Name2
Name3
Name4
Obj 1
Obj 2
Obj 3
Obj 4

Scope

Scope: where/when a name is bound to an object

Useful for modularity: want to keep most things hidden

Scoping Policy	Visible Names Depend On
Static	Textual structure of program Names resolved by compile-time symbol tables Faster, more common
Dynamic	Run-time behavior of program Names resolved by run-time symbol tables, e.g., walk the stack looking for names Slower, more dynamic

Basic Static Scope in C, C++, Java, etc.

A name begins life where it is declared and ends at the end of its block.

From the CLRM, “The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.”

```
void foo()
{
    int x;



}
```

Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, “If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.”

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }


}
```

Static vs. Dynamic Scope

C

```c
int a = 0;

int foo() {
  return a + 1;
}

int bar() {
  int a = 10;

  return foo();
}
```

OCaml

```ocaml
let a = 0 in
let foo x = a + 1 in
let bar =
  let a = 10 in
  foo 0
```

Bash

```bash
a=0

foo ()
{
  a=`expr $a + 1`
}

bar ()
{
  local a=10
  foo
  echo $a
}

bar
```

Basic Static Scope in O'Caml

A name is bound after the "in" clause of a "let." If the name is re-bound, the binding takes effect *after* the "in."

```
let x = 8 in


let x = x + 1 in
```

Returns the pair (12, 8):

```
let x = 8 in
  (let x = x + 2 in
    x + 2),
x
```

Let Rec in O'Caml

```
let rec fib i =
  if i < 1 then 1 else
    fib (i-1) + fib (i-2)
in
  fib 5
```

The "rec" keyword makes a name visible to its definition. This only makes sense for functions.

```
(* Nonsensical *)
let rec x = x + 3 in
```

Let…and in O'Caml

Let…and lets you bind multiple names at once. Definitions are not mutually visible unless marked "rec."

```
let x = 8
and y = 9 in
```

```
let rec fac n =
      if n < 2 then
        1
      else
        n * fac1 n
and fac1 n = fac (n - 1)
in
fac 5
```

Forward Declarations

Languages such as C, C++, and Pascal require *forward declarations* for mutually-recursive references.

```
int foo(void);
int bar() { ... foo(); ... }
int foo() { ... bar(); ... }
```

Partial side-effect of compiler implementations. Allows single-pass compilation.

Dynamic Definitions in TEX

```
% \x, \y undefined
{
  % \x, \y undefined
  \def \x 1
  % \x defined, \y undefined

  \ifnum \a < 5
    \def \y 2
  \fi

  % \x defined, \y may be undefined
}
% \x, \y undefined
```

Static vs. Dynamic Scope

Most modern languages use static scoping.

Easier to understand, harder to break programs.

Advantage of dynamic scoping: ability to change environment.

A way to surreptitiously pass additional parameters.

Application of Dynamic Scoping

```
program messages;
var message : string;

  procedure complain;
  begin
    writeln(message);
  end

  procedure problem1;
  var message : string;
  begin
    message := 'Out of memory';
    complain
  end

  procedure problem2;
  var message : string;
  begin
    message := 'Out of time';
    complain
  end
```

Open vs. Closed Scopes

An *open scope* begins life including the symbols in its outer scope.

Example: blocks in Java

```
{
  int x;
  for (;;) {
    /* x visible here */
  }
}
```

A *closed scope* begins life devoid of symbols.

Example: structures in C.

```
struct foo {
  int x;
  float y;
}
```

Types

What operations are allowed?

Types

A restriction on the possible interpretations of a segment of memory or other program construct.

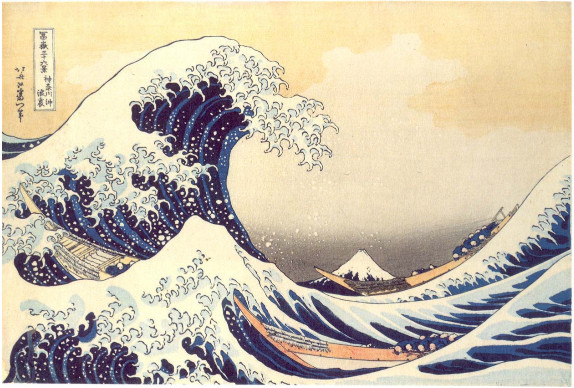
Two uses:

Safety: avoids data being treated as something it isn't

Optimization: eliminates certain runtime decisions

Types in C

What types are processors best at?



The C/C++ Machine Model

Arithemtic and other operators map to machine instructions

`+ % -> [] *`

Aggregate objects are composed by simple concatenation

Arrays, structs, C++ classes

Memory is a set of sequences of objects; pointers are machine addresses

(After Stroustrup, due to Ritchie)

Basic C Types

C was designed for efficiency: basic types are whatever is most efficient for the target processor.

On an (32-bit) ARM processor,

```
char c;                /* 8-bit binary */

short d;               /* 16-bit two's-complement binary */
unsigned short d;      /* 16-bit binary */

int a;                 /* 32-bit two's-complement binary */
unsigned int b;        /* 32-bit binary */

float f;               /* 32-bit IEEE 754 floating-point */
double g;              /* 64-bit IEEE 754 floating-point */
```

Number Behavior

Basic number axioms:

$a + x$	$=$	a if and only if $x = 0$	Additive identity
$(a+b)+c$	$=$	$a+(b+c)$	Associative
$a(b+c)$	$=$	$ab+ac$	Distributive

Misbehaving Floating-Point Numbers

`1e20 + 1e-20 = 1e20`

`1e-20` $\ll$ `1e20`

`(1 + 9e-7) + 9e-7` $\neq$ `1 + (9e-7 + 9e-7)`

`9e-7` $\ll$ `1`, so it is discarded, however, `1.8e-6` is large enough

$1.00001(1.000001 - 1) \neq 1.00001 \cdot 1.000001 - 1.00001 \cdot 1$

$1.00001 \cdot 1.000001 = 1.00001100001$ requires too much intermediate precision.

What's Going On?

Floating-point numbers are represented using an exponent/significand format:

$$1 \quad \underbrace{10000001}_{\text{8-bit exponent}} \quad \underbrace{01100000000000000000000}_{\text{23-bit significand}}$$

$$= \quad -1.011_2 \times 2^{129-127} = -1.375 \times 4 = -5.5.$$

What to remember:

$$\underbrace{\color{red}{1363.4568}}_{\text{represented}}\underbrace{46353963456293}_{\text{rounded}}$$

What's Going On?

Results are often rounded:

$$\begin{array}{r} 1.00001000000 \\ \times 1.00000100000 \\ \hline 1.000011\underbrace{00001}_{\text{rounded}} \end{array}$$

When $b \approx -c$, $b + c$ is small, so $ab + ac \neq a(b + c)$ because precision is lost when ab is calculated.

Moral: Be aware of floating-point number properties when writing complex expressions.

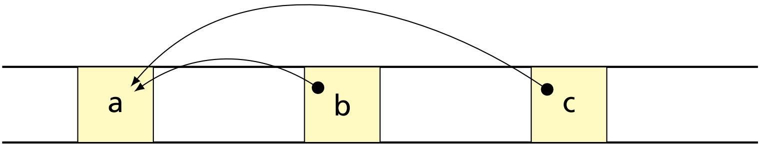
Pointers and Arrays

A pointer contains a memory address.

Arrays in C are implemented with arithmetic on pointers.

A pointer can create an *alias* to a variable:

```
int a;
int *b = &a;  /* "pointer to integer b is the address of a" */
int *c = &a;  /* c also points to a */

*b = 5;       /* sets a to 5 */
*c = 42;      /* sets a to 42 */

printf("%d %d %d\n", a, *b, *c); /* prints 42 42 42 */
```



Pointers Enable Pass-by-Reference

```c
void swap(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

Does this work?

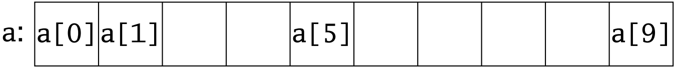
Pointers Enable Pass-by-Reference

```
void swap(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

Does this work?
Nope.

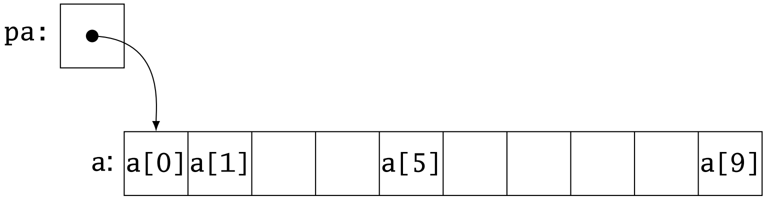
```
void swap(int *px, int *py)
{
  int temp;

  temp = *px; /* get data at px */
  *px = *py;  /* get data at py */
  *py = temp; /* write data at py */
}

void main()
{
  int a = 1, b = 2;

  /* Pass addresses of a and b */
  swap(&a, &b);

  /* a = 2 and b = 1 */
}
```

Arrays and Pointers



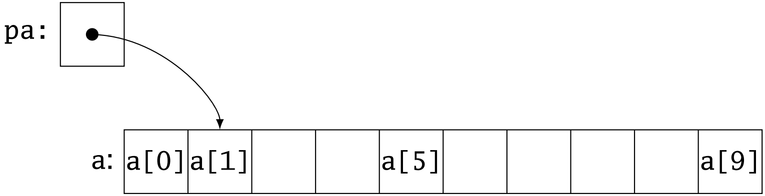

```
int a[10];
```

Arrays and Pointers



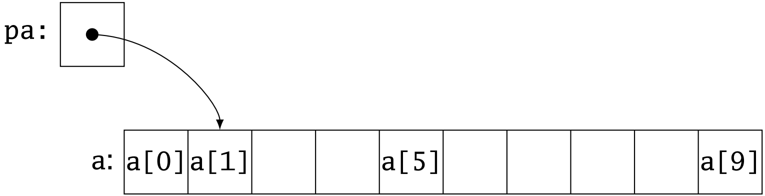

```
int a[10];
int *pa = &a[0];
```

Arrays and Pointers



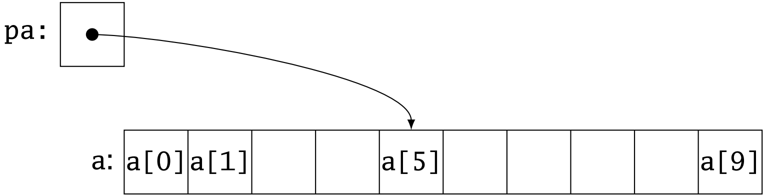

```
int a[10];
int *pa = &a[0];
pa = pa + 1;
```

Arrays and Pointers




```
int a[10];
int *pa = &a[0];
pa = pa + 1;
pa = &a[1];
```

Arrays and Pointers




```
int a[10];
int *pa = &a[0];
pa = pa + 1;
pa = &a[1];
pa = a + 5;
```

`a[i]` is equivalent to `*(a + i)`

Multi-Dimensional Arrays

```
int monthdays[2][12] = {
  { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
  { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } };
```

`monthdays[i][j]` is at address `monthdays + 12 * i + j`

Structures

Structures: each field has own storage

```
struct box {
  int x, y, h, w;
  char *name;
};
```

Unions: fields share same memory

```
union token {
  int i;
  double d;
  char *s;
};
```

Structs

Structs can be used like the objects of C++, Java, et al.

Group and restrict what can be stored in an object, but not what operations they permit.

```
struct poly { ... };

struct poly *poly_create();
void        poly_destroy(struct poly *p);
void        poly_draw(struct poly *p);
void        poly_move(struct poly *p, int x, int y);
int         poly_area(struct poly *p);
```

Unions: Variant Records

A `struct` holds all of its fields at once. A `union` holds only one of its fields at any time (the last written).

```
union token {
  int i;
  float f;
  char *string;
};

union token t;
t.i = 10;
t.f = 3.14159;        /* overwrite t.i */
char *s = t.string;   /* return gibberish */
```

Kind of like a bathroom on an airplane

Applications of Variant Records

A primitive form of polymorphism:

```
struct poly {
  int type;
  int x, y;
  union { int radius;
          int size;
          float angle; } d;
};

void draw(struct poly *shape)
{
  switch (shape->type) {
  case CIRCLE:  /* use shape->d.radius */

  case SQUARE:  /* use shape->d.size */

  case LINE:    /* use shape->d.angle */

  }

}
```

Name vs. Structural Equivalence

```
struct f {
  int x, y;
} foo = { 0, 1 };

struct b {
  int x, y;
} bar;

bar = foo;
```

Is this legal in C? Should it be?

C's Declarations and Declarators

Declaration: list of specifiers followed by a comma-separated list of declarators.

$$\underbrace{\texttt{static unsigned }\overbrace{\texttt{int}}^{\text{basic type}}}_{\text{specifiers}}\underbrace{\texttt{(*f[10])(int, char*);}}_{\text{declarator}}$$

Declarator's notation matches that of an expression: use it to return the basic type.

Largely regarded as the worst syntactic aspect of C: both pre- (pointers) and post-fix operators (arrays, functions).

Types of Type Systems

What kinds of type systems do languages have?

Strongly-typed Languages

Strongly-typed: no run-time type clashes (detected or not).

C is definitely not strongly-typed:

```
float g;
union { float f; int i } u;
u.i = 3;
g = u.f + 3.14159; /* u.f is meaningless */
```

Is Java strongly-typed?

Statically-Typed Languages

Statically-typed: compiler can determine types.

Dynamically-typed: types determined at run time.

Is Java statically-typed?

```
class Foo {
   public void x() { ... }
}

class Bar extends Foo {
   public void x() { ... }
}

void baz(Foo f) {
  f.x();
}
```

Implementing Dynamic Typing

Each variable contains both raw data and information about its type: how to interpret the raw data.

E.g., in Python, every object is derived from PyObject:

```
typdef struct _object {
  Py_ssize_t ob_refcnt;          /* Reference count for GC */
  struct _typeobject *ob_type; /* Information about actual type */
} PyObject;
```

E.g., integers have a PyObject header and payload:

```
typedef struct {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
  long ob_ival;                  /* Actual integer value */
} PyIntObject;
```

In Tcl, Everything Is A String

Each object in Tcl can be a string, a raw value, or both.
Recomputed lazily; updating one invalidates the other.

```
typedef struct Tcl_Obj {
  int refCount;          /* Reference count for GC */
  char *bytes;           /* String representation */
  int length;            /* Length of string */
  Tcl_ObjType *typePtr; /* Information about type */
  union {
    long longValue;
    double doubleValue;
    VOID *otherValuePtr;
    struct { VOID *ptr1, *ptr2; } twoPtrValue;
  } internalRep;  /* raw value */
} Tcl_Obj;

typedef struct Tcl_ObjType {
  char *name;
  Tcl_FreeInternalRepProc *freeIntRepProc; /* free obj */
  Tcl_DupInternalRepProc *dupIntRepProc;   /* copy obj */
  Tcl_UpdateStringProc *updateStringProc;  /* to string */
  Tcl_SetFromAnyProc *setFromAnyProc;      /* from string */
} Tcl_ObjType;
```

Polymorphism

Say you write a sort routine:

```
void sort(int a[], int n)
{
  int i, j;
  for ( i = 0 ; i < n-1 ; i++ )
    for ( j = i + 1 ; j < n ; j++ )
      if (a[j] < a[i]) {
        int tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
```

Polymorphism

To sort doubles, only need to change two types:

```
void sort(double a[], int n)
{
  int i, j;
  for ( i = 0 ; i < n-1 ; i++ )
    for ( j = i + 1 ; j < n ; j++ )
      if (a[j] < a[i]) {
        double tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
```

C++ Templates

```
template <class T> void sort(T a[], int n)
{
  int i, j;
  for ( i = 0 ; i < n-1 ; i++ )
    for ( j = i + 1 ; j < n ; j++ )
      if (a[j] < a[i]) {
        T tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}

int a[10];

sort<int>(a, 10);
```

C++ Templates

C++ templates are essentially language-aware macros. Each instance generates a different refinement of the same code.

```
sort<int>(a, 10);

sort<double>(b, 30);

sort<char *>(c, 20);
```

Fast code, but lots of it.

Faking Polymorphism with Objects

```
class Sortable {
  bool lessthan(Sortable s) = 0;
}

void sort(Sortable a[], int n) {
  int i, j;
  for ( i = 0 ; i < n-1 ; i++ )
    for ( j = i + 1 ; j < n ; j++ )
      if ( a[j].lessthan(a[i]) ) {
        Sortable tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
}
```

Faking Polymorphism with Objects

This `sort` works with any array of objects derived from `Sortable`.

Same code is used for every type of object.

Types resolved at run-time (dynamic method dispatch).

Does not run as quickly as the C++ template version.

Parametric Polymorphism

In C++,

```
template <typename T>
T max(T x, T y)
{
  return x > y ? x : y;
}

struct foo {int a;} f1, f2, f3;

int main()
{
  int a = max<int>(3, 4); /* OK */
  f3 = max<struct foo>(f1, f2);  /* No match for operator> */
}
```

The `max` function only operates with types for which the > operator is defined.

Parametric Polymorphism

In OCaml,

```
let max x y = if x - y > 0 then x else y

max : int -> int -> int
```

Only `int` arguments are allowed because in OCaml, – only operates on integers.

However,

```
let rec map f = function [] -> [] | x::xs -> f x :: map f xs

map : ('a -> 'b) -> 'a list -> 'b list
```

Here, `'a` and `'b` may each be *any type*.

OCaml uses parametric polymorphism: type variables may be of any type.

C++'s template-based polymorphism is ad hoc: there are implicit constraints on type parameters.

Overloading

What if there is more than one object for a name?

Overloading versus Aliases

Overloading: two objects, one name

Alias: one object, two names

In C++,

```
int foo(int x) { ... }
int foo(float x) { ... } // foo overloaded

void bar()
{
  int x, *y;
  y = &x;  // Two names for x: x and *y
}
```

Examples of Overloading

Most languages overload arithmetic operators:

```
1 + 2            // Integer operation
3.1415 + 3e-4    // Floating-point operation
```

Resolved by checking the *type* of the operands.

Context must provide enough hints to resolve the ambiguity.

Function Name Overloading

C++ and Java allow functions/methods to be overloaded.

```
int   foo();
int   foo(int a);   // OK: different # of args
float foo();        // Error: only return type
int   foo(float a); // OK: different arg types
```

Useful when doing the same thing many different ways:

```
int add(int a, int b);
float add(float a, float b);

void print(int a);
void print(float a);
void print(char *s);
```

Function Overloading in C++

Complex rules because of *promotions*:

```
int i;
long int l;
l + i
```

Integer promoted to long integer to do addition.

```
3.14159 + 2
```

Integer is promoted to double; addition is done as double.

Function Overloading in C++

1. Match trying trivial conversions
   `int a[]` to `int *a`, *T* to *const T*, etc.
2. Match trying promotions
   `bool` to `int`, `float` to `double`, etc.
3. Match using standard conversions
   `int` to `double`, `double` to `int`
4. Match using user-defined conversions
   `operator int() const { return v; }`
5. Match using the elipsis `...`

Two matches at the same (lowest) level is ambiguous.

Binding Time

When are bindings created and destroyed?

Binding Time

When a name is connected to an object.

Bound when	**Examples**
language designed	`if` `else`
language implemented	data widths
Program written	`foo` `bar`
compiled	static addresses, code
linked	relative addresses
loaded	shared objects
run	heap-allocated objects

Binding Time and Efficiency

Earlier binding time ⇒ more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```
switch (statement) {

case add:
   r = a + b;
   break;

case sub:
   r = a - b;
   break;

   /* ... */
}
```

```
add %o1, %o2, %o3
```

Binding Time and Efficiency

Dynamic method dispatch in OO languages:

```
class Box : Shape {
  public void draw() { ... }
}

class Circle : Shape {
  public void draw() { ... }
}

Shape s;
s.draw();  /* Bound at run time */
```

Binding Time and Efficiency

Interpreters better if language has the ability to create new programs on-the-fly.

Example: Ousterhout's Tcl language.

Scripting language originally interpreted, later byte-compiled.

Everything's a string.

```
set a 1
set b 2
puts "$a + $b = [expr $a + $b]"
```

Binding Time and Efficiency

Tcl's `eval` runs its argument as a command.

Can be used to build new control structures.

```
proc ifforall {list pred ifstmt} {
  foreach i $list {
    if [expr $pred] { eval $ifstmt }
  }
}

ifforall {0 1 2} {$i % 2 == 0} {
  puts "$i even"
}

0 even
2 even
```