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?
Scope

Scope: where/when a name is bound to an object
Useful for modularity: want to keep most things hidden

<table>
<thead>
<tr>
<th>Scoping Policy</th>
<th>Visible Names Depend On</th>
</tr>
</thead>
</table>
| 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 |
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.”
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.”

```c
void foo()
{
    int x;
    while ( a < 10 ) {
        int x;
    }
}
```
### Static vs. Dynamic Scope

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
</table>
| **C** | `int a = 0;`  
  `int foo() {`  
  `  return a + 1;`  
  `}`  
  `int bar() {`  
  `  int a = 10;`  
  `  return foo();`  
  `}` |
| **OCaml** | `let a = 0 in`  
  `let foo x = a + 1 in`  
  `let bar =`  
  `  let a = 10 in`  
  `  foo 0` |
| **Bash** | `a=0`  
  `foo () {`  
  `  a=`expr $a + 1`  
  `}`  
  `bar () {`  
  `  local a=10`  
  `  foo`  
  `  echo $a`  
  `}`  
  `bar` |
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

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

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

```ocaml
let rec x = x + 3 in
```

(* Nonsensical *)
Let...and in O’Caml

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

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

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

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

  \ifnum \texttt{a} < 5
    \def \y 2
  \fi

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

```pascal
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

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

A *closed scope* begins life devoid of symbols.

Example: structures in C.

```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

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

On an (32-bit) ARM processor,

```c
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:

\[ a + x = a \text{ if and only if } x = 0 \] 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 ≪ 1e20

(1 + 9e-7) + 9e-7 ≠ 1 + (9e-7 + 9e-7)

9e-7 ≪ 1, so it is discarded, however, 1.8e-6 is large enough

1.00001(1.000001 − 1) ≠ 1.00001 · 1.000001 − 1.00001 · 1

1.00001 · 1.000001 = 1.00001100001 requires too much intermediate precision.
What’s Going On?

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

\[\begin{array}{c}
1 & \begin{array}{c}
10000001
\end{array} & \begin{array}{c}
01100000000000000000000000000
\end{array} \\
\end{array}\]

8-bit exponent \quad 23-bit significand

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

What to remember:

\[\overbrace{1363.456846353963456293}^{\text{represented}} \quad \overbrace{46353963456293}^{\text{rounded}}\]
What’s Going On?

Results are often rounded:

\[
\begin{array}{c}
1.0000100000 \\
\times 1.00000100000 \\
\hline
1.00001100001 \\
\end{array}
\]

\(1.00001100001\)

rounded

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:

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

Does this work?
Pointers Enable Pass-by-Reference

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

Does this work?
Nope.

```c
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 */
}
```

```c
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

```c
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

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

- **Unions:** fields share same memory

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

```c
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:

```c
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 */
    }
}
```
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.

\[
\text{basic type} \quad \text{specifiers} \quad \text{declarator}
\]

\[
\text{static unsigned int(*f[10])(int, char*)};
\]

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:

```c
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?

```java
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:

```c
typedef 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:

```c
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:

```c
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;
            }
}
```
To sort doubles, only need to change two types:

```c
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**

```cpp
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 are essentially language-aware macros. Each instance generates a different refinement of the same code.

```cpp
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.
In C++,

```cpp
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,

```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,

```ocaml
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++,

```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:

\[
\begin{align*}
1 + 2 & \quad // \text{Integer operation} \\
3.1415 + 3e-4 & \quad // \text{Floating-point operation}
\end{align*}
\]

Resolved by checking the type of the operands.

Context must provide enough hints to resolve the ambiguity.
C++ and Java allow functions/methods to be overloaded.

```c++
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:

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

void print(int a);
void print(float a);
void print(char *s);
```
Complex rules because of *promotions*:

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

Integer promoted to long integer to do addition.

```cpp
3.14159 + 2
```

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

<table>
<thead>
<tr>
<th>Bound when</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>language designed</td>
<td>if else</td>
</tr>
<tr>
<td>language implemented</td>
<td>data widths</td>
</tr>
<tr>
<td>Program written</td>
<td>foo bar</td>
</tr>
<tr>
<td>compiled</td>
<td>static addresses, code</td>
</tr>
<tr>
<td>linked</td>
<td>relative addresses</td>
</tr>
<tr>
<td>loaded</td>
<td>shared objects</td>
</tr>
<tr>
<td>run</td>
<td>heap-allocated objects</td>
</tr>
</tbody>
</table>
Earlier binding time ⇒ more efficiency, less flexibility

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

```java
switch (statement) {
    case add:
        r = a + b;
        break;
    case sub:
        r = a - b;
        break;
    /* ... */
}
```

add %o1, %o2, %o3
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 */
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.

```tcl
set a 1
set b 2
puts "$a + $b = [expr $a + $b]"
```
Tcl’s `eval` runs its argument as a command.

Can be used to build new control structures.

```tcl
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
```