



Types

Read: Scott, Chapters 7 and 8

Types and Type Systems

- A key concept in programming languages
- We saw some foundational type theory
 - Typed Lambda calculus (System F_1) in Lec 17
- Haskell's type system
- Today, a more pragmatic view of types

Lecture Outline

- Types
 - Type systems
 - Type checking
 - Type safety
 - Type equivalence
 - Types in C
-
- Primitive types
 - Composite types

What Is a type?

- A set of values and the valid operations on those values
 - Integers:
+, -, *, /, <, <=, ==, >=, >
 - Arrays:
lookUp(<array>,<index>)
assign(<array>,<index>,<value>)
initialize(<array>), setBounds(<array>)
 - User-defined types:
Java interfaces

What Is the Role of Types?

- What is the role of types in programming languages?
 - Semantic correctness
 - Data abstraction
 - Abstract Data Types (as we saw in Java)
 - Documentation (static types only)

3 Views of Types

- **Denotational** (or **set**) point of view:
 - A type is simply a **set** of values. A value has a given type if it belongs to the set. E.g.
 - `int` = { ...-1,0,1,2,... }
 - `char` = { 'a','b',... }
 - `bool` = { true, false }
- **Abstraction-based** point of view:
 - A type is an **interface** consisting of a set of operations with well-defined meaning

3 Views of Types

- **Constructive** point of view:
 - Primitive/simple types: e.g., `int`, `char`, `bool`
 - Composite/constructed types:
 - Constructed by applying **type constructors**
 - pointer e.g., `pointerTo(int)`
 - array e.g., `arrayOf(char)` or `arrayOf(char,20)` or ...
 - record/struct e.g., `record(age:int, name:arrayOf(char))`
 - union e.g. `union(int, pointerTo(char))`

CAN BE NESTED! `pointerTo(arrayOf(pointerTo(char)))`
- For most of us, types are a mixture of these 3 views

What Is a Type System?

- A mechanism to define types and associate them with programming language constructs
 - Deduce types of constructs
 - Deduce if a construct is “type correct” or “type incorrect”
- Additional rules for type equivalence, type compatibility
 - Important from pragmatic point of view

What Is a Type System?

What Is Type Checking?

- The process of ensuring that the program obeys the type rules of the language
- Type checking can be done statically
 - At compile-time, i.e., before execution
 - **Statically typed** (or **statically checked**) language
- Type checking can be done dynamically
 - At runtime, i.e., during execution
 - **Dynamically typed** (or **dynamically checked**) language

What Is Type Checking?

- **Statically typed** (better term: statically checked) languages
 - Typically require **type annotations** (e.g., `A a`, `List<A> list`)
 - Typically have a complex type system, and **most** of type checking is performed statically (at compile-time)
 - Ada, Pascal, Java, C++, Haskell, ML/OCaml
 - A form of early binding
- **Dynamically typed** (better term: dynamically checked) languages. Also known as **Duck typed**...
 - Typically require no **type annotations**!
 - All type checking is performed dynamically (at runtime)
 - Smalltalk, Lisp and Scheme, Python, JavaScript

What Is Type Checking?

- The process of ensuring that the program obeys the type rules of the language
- **Type safety**
 - Textbook defines term **prohibited application** (also known as **forbidden error**): intuitively, a prohibited application is an application of an operation on values of the wrong type
 - **Type safety** is the property that no operation ever applies to values of the wrong type at runtime. I.e., no prohibited application (forbidden error) ever occurs

Language Design Choices

- Design choice: what is the set of forbidden errors?
 - Obviously, we cannot forbid all possible semantic errors...
 - Define a set of forbidden errors
- Design choice: Once we've chosen the set of forbidden errors, how does the type system prevent them?
 - Static checks only? Dynamic checks only? A combination of both?
- Furthermore, are we going to absolutely disallow forbidden errors (be **type safe**), or are we going to allow for programs to circumvent the system and exhibit forbidden errors (i.e., be **type unsafe**)?

Forbidden Errors

- Example: indexing an array out of bounds
 - **$a[i]$, a is of size Bound , $i < 0$ or $\text{Bound} \leq i$**
 - In C, C++, this is not a forbidden error
 - **$0 \leq i$** and **$i < \text{Bound}$** is not checked (bounds are not part of type)
 - What are the tradeoffs here?
 - In Pascal, this is a forbidden error. Prevented with static checks
 - **$0 \leq i$** and **$i < \text{Bound}$** must be checked at compile time
 - What are the tradeoffs here?
 - In Java, this is a forbidden error. It is prevented with dynamic checks
 - **$0 \leq i$** and **$i < \text{Bound}$** must be checked at runtime
 - What are the tradeoffs here?

Type Safety

■ Java vs. C++:

- Java: `Duck q; ...; q.quack()` class `Duck` has `quack`
- C++: `Duck *q; ...; q->quack()` class `Duck` has `quack`

Can we write code that passes the type checker, and yet it calls `quack()` on an object that isn't a `Duck` at runtime?

- In Java?
- In C++?

- Java is said to be type safe while C++ is said to be type unsafe

C++ Is Type Unsafe

// #1

```
void* x = (void *) new A;  
B* q = (B*) x;    //a safe downcast?  
int case1 = q->foo() //what happens?
```

A virtual foo()
B virtual foo()
virtual foo(int)

// #2

```
void* x = (void *) new A;  
B* q = (B*) x;    //a safe downcast?  
int case2 = q->foo(66); //what happens?
```

q->foo(66) is a prohibited application (i.e., application of an operation on a value of the wrong type, i.e., forbidden error). Static type **B* q** “promises” the programmer that **q** will point to a **B** object. However, language does not “honor” this promise...

What Is Type Checking

	statically typed	not statically typed (i.e., dynamically typed)
type safe	ML/Ocaml, Haskell, Java*	Python, Scheme, R, JavaScript
type unsafe	C/C++	Assembly

What Is Type Checking?

- Static typing vs. dynamic typing
 - What are the advantages of static typing?
 - What are the advantages of dynamic typing?

Lecture Outline

- Types
- Type systems
 - Type checking
 - Type safety
- Type equivalence
- Types in C


- Primitive types
- Composite types

Type Equivalence and Type Compatibility

- We now move in the world of procedural von Neumann languages
 - E.g., Fortran, Algol, Pascal and C
 - Value model
 - Statically typed

Type Equivalence and Type Compatibility

■ Questions

e := expression  or 

Are **e** and **expression** of “same type”?

a + b  or 

Are **a** and **b** of “same type” and type supports +?

foo(arg1, arg2, ..., argN)  or 

Do the types of the arguments “match the types” of the formal parameters?

Type Equivalence

- Two ways of defining type equivalence
 - **Structural equivalence**: based on “shape”
 - Roughly, two types are the same if they consists of the same components, put together in the same way
 - **Name equivalence**: based on lexical occurrence of the type definition
 - Strict name equivalence
 - Loose name equivalence

T1 x; ...

T2 y;

x = y;

Structural Equivalence

- A type is structurally equivalent to itself
- Two types are structurally equivalent if they are formed by applying the same **type constructor** to structurally equivalent types (i.e., arguments are structurally equivalent)
- After **type declaration** **type** **n** = **T** or **typedef** **T** **n** in C, the type name **n** is structurally equivalent to **T**
 - Declaration makes **n** an **alias** of **T**. **n** and **T** are said to be **aliased types**

Structural Equivalence

- Example, Pascal-like language:

This is a type definition:
an application of the
array type constructor

```
type S = array [0..99] of char
type T = array [0..99] of char
```

- Example, C:

```
typedef struct {
    int j, int k, int *ptr
} cell;

typedef struct {
    int n, int m, int *p
} element;
```


Structural Equivalence

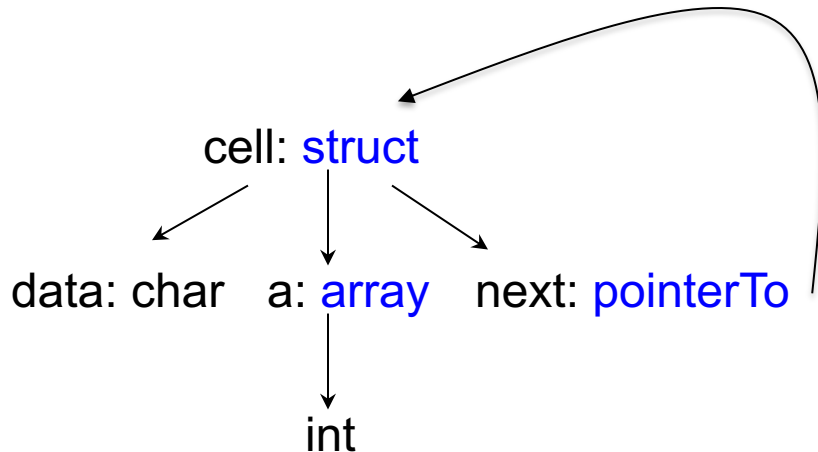
- Shown by isomorphism of corresponding type trees
 - Show the type trees of these constructed types
 - Are these types structurally equivalent?

<pre>struct cell { char data; int a[3]; struct cell *next; }</pre>	<pre>struct element { char c; int a[5]; struct element *ptr; }</pre>
--	--

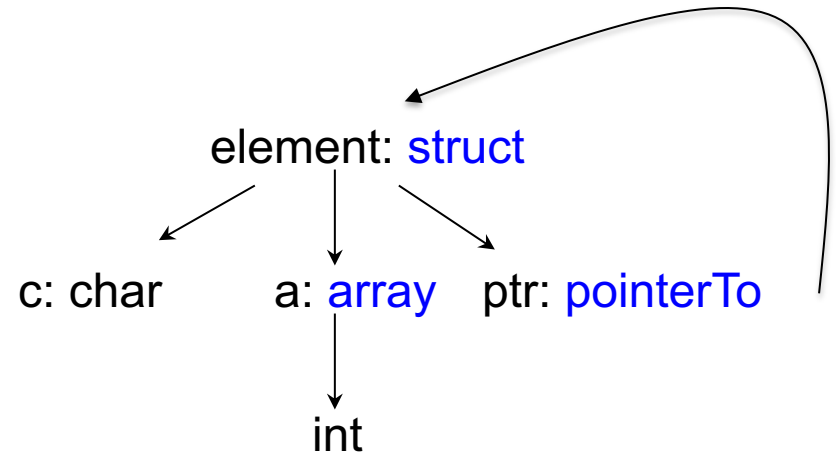
Equivalent types: are **field names** part of the **struct** constructed type?
are **array bounds** part of the **array** constructed type?

Structural Equivalence

```
struct cell
{
  char data;
  int a[3];
  struct cell *next;
}
```

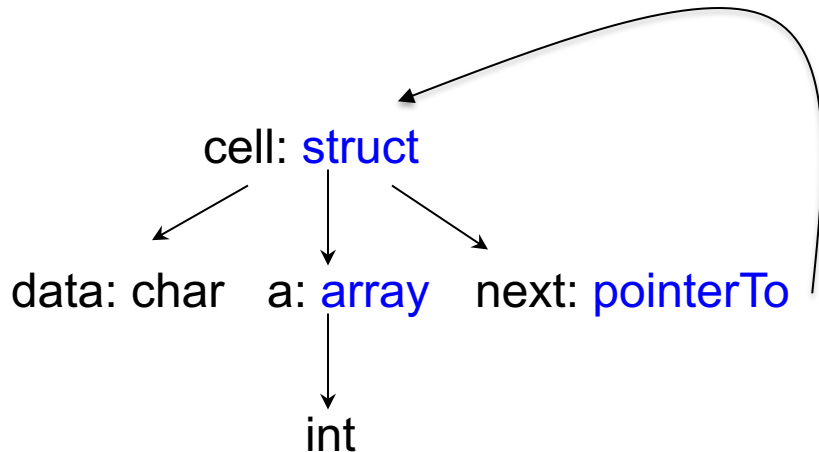


```
struct element
{
  char c;
  int a[5];
  struct element *ptr;
}
```

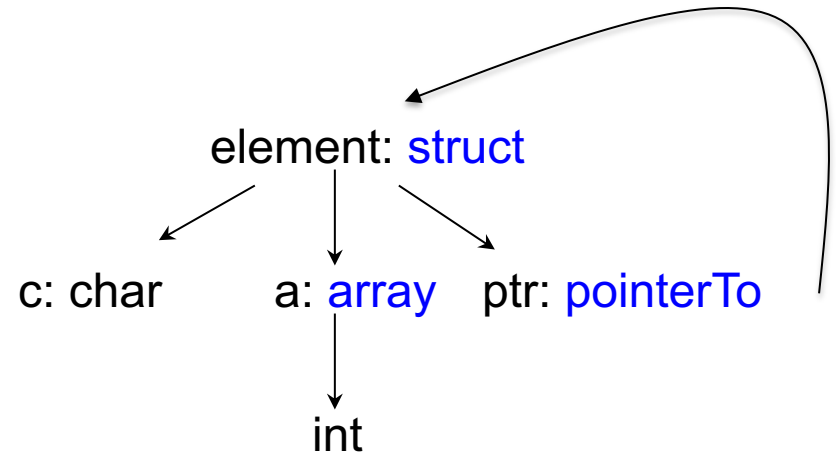


Structural Equivalence

```
struct cell
{
  char data;
  int a[3];
  struct cell *next;
}
```



```
struct element
{
  char c;
  int a[5];
  struct element *ptr;
}
```



Name Equivalence

Name equivalence

Roughly, based on lexical occurrence of **type definition**. An application of a **type constructor** is a **type definition**. E.g., the red **array[1..20]** ... is one type definition and the blue **array[1..20]** is a different type definition.

```
type T = array [1..20] of int;
```

```
x, y: array [1..20] of int;
```

```
w, z: T;
```

```
v: T;
```

x and **y** are of same type, **w**, **z**, **v** are of same type, but **x** and **w** are of different types!

Question

Name equivalence

```
w, z, v: array [1..20] of int;  
x, y: array [1..20] of int;
```

Are **x** and **w** of equivalent type according to name equivalence?

Answer: **x** and **w** are of distinct types.

Name Equivalence

- A subtlety arises with **aliased types** (e.g.,
`type n = T, typedef int Age` in C)
- **Strict name equivalence**
 - A language in which aliased types are considered distinct, is said to have strict name equivalence (e.g., `int` and `Age` above would be distinct types)
- **Loose name equivalence**
 - A language in which aliased types are considered equivalent, is said to have loose name equivalence (e.g., `int` and `Age` would be same)

Exercise

```
type cell = ... // record type
typealink = pointer to cell
type blink =alink
p,q : pointer to cell
r :alink
s :blink
t : pointer to cell
u :alink
```

Group p, q, r, s, t, u into equiv. classes, according to structural equiv., strict name equiv. and loose name equiv.

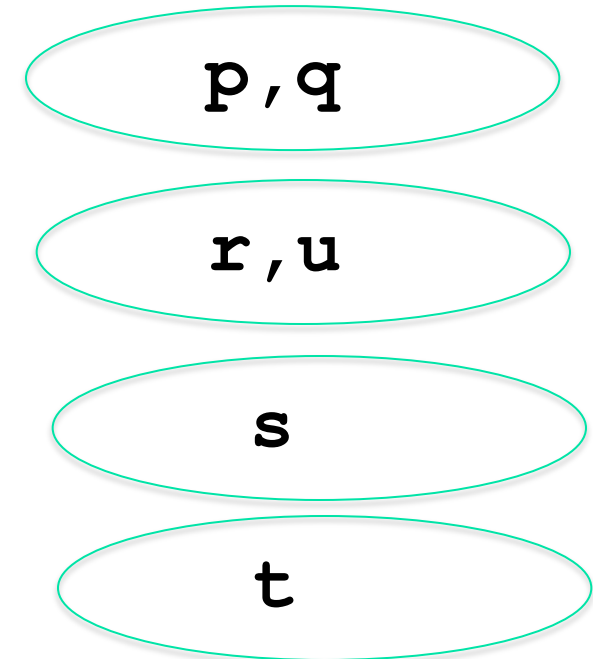
Exercise: Structural Equiv.

```
type cell = ... // record type
typealink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```

p,q,r,s,t,u

Exercise: Strict Name Equiv.

```
type cell = ... // record type
typealink = pointer to cell
typeblink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```



Exercise: Loose Name Equiv.

```
type cell = ... // record type
type alink = pointer to cell
type blink = alink
p,q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```



p,q

r,s,u

t

Example: Type Equivalence in C

- First, in the Algol family, **field names are part** of the record/struct constructed type. E.g., the record types below are NOT even structurally equivalent

```
type A = record
```

```
  x,y : real
```

```
end;
```

```
type B = record
```

```
  z,w : real
```

```
end;
```

Type Equivalence in C

- Anonymous types are differentiated by internal (compiler-generated) type names

This **struct** is of type anon1.

```
struct RecA
{ char x;
  int y;
} a;
```

```
typedef struct
{ char x;
  int y;
} RecB;
```

```
struct
{ char x;
  int y;
} c;
```

```
RecB b;
```

What variables are of **equivalent type** according to the rules in C?

Type Equivalence in C

- C uses structural equivalence for everything, except unions and structs, for which it uses loose name equivalence

```
struct A           struct B
{ char x;          { char x;
  int y;           int y;
}                   }
typedef struct A C;
typedef C *P;
typedef struct B *Q;
typedef struct A *R;
typedef int Age;
typedef int (*F) (int);
typedef Age (*G) (Age);
```

Type Equivalence in C

```
struct B { char x; int y; };  
typedef struct B A;  
struct { A a; A *next; } aa;  
struct { struct B a; struct B *next; } bb;  
struct { struct B a; struct B *next; } cc;
```

```
A a;  
struct B b;
```

```
a = b;  
aa = bb;  
bb = cc;
```

Which of the above assignments pass the type checker?

Question

- Structural equivalence for record types is considered a bad idea. Can you think of a reason why?

Type Equivalence and Type Compatibility

■ Questions:

e := **expression**

 or 

Are **e** and **expression** of “same type”?

■ **e** and **expression** may not be of equivalent types, but they may be of “compatible types”. It may be possible to convert the type of **expression** to the type of **e**

Type Conversion

- Implicit conversion – coercion
 - Conversion done implicitly by the compiler
 - In C, mixed mode numerical operations
 - In `e = expression` if `e` is a `double` and `expression` is an `int`, `expression` is implicitly coerced in to a `double`
 - `double d, e; ... e = d + 2; //2 coerced to 2.0`
 - `int` to `double`,
 - `float` to `double`
 - How about `float` to `int`?
 - No. May lose precision and thus, cannot be coerced!

Type Conversion

- Explicit conversion
 - Programmer must “acknowledge” conversion
 - In Pascal, **round** and **trunc** perform explicit conversion
 - **round(s)** real **to** int by rounding
 - **trunc(s)** real **to** int by truncating
 - In C, type **casting** performs explicit conversion
 - **freelist *s; ... (char *)s;** forces **s** to be considered as pointing to a char for the purposes of pointer arithmetic

Lecture Outline

- Types
- Type systems
 - Type checking
 - Type safety
- Type equivalence
- Types in C
- Primitive types
- Composite types

Pointers and Arrays in C

- Pointers and arrays are **interoperable**:

```
int n;  
int *a  
int b[10];
```

1. `a = b;`
2. `n = a[3];`
3. `n = *(a+3);`
4. `n = b[3];`
5. `n = *(b+3);`

Type Declaration in C

- What is the meaning of the following declaration in C? Draw the type trees.

1. **int *a[n]**

2. **int (*a)[n]**

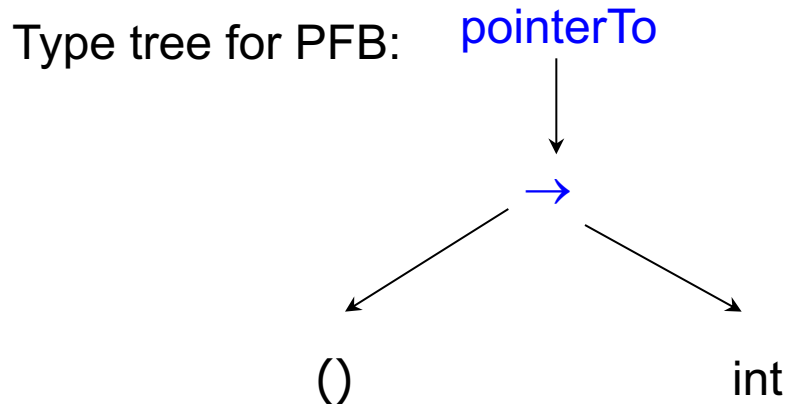
3. **int (*f)(int)**

Type Declaration in C

```
typedef int (*PFB)();           // Type variable PFB: what type?
struct parse_table {           // Type struct parse_table: what type?
    char *name;
    PFB func; };
int func1() { ... }            // Function func1: what type?
int func2() { ... }

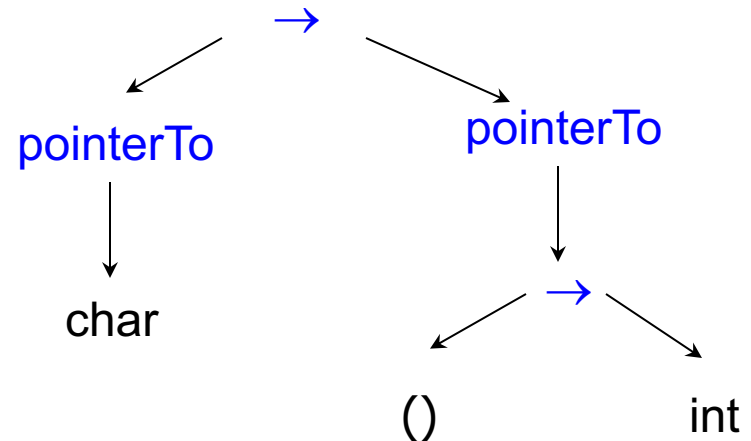
struct parse_table table[] = {  // Variable table: what type?
    {"name1", &func1},
    {"name2", &func2}
};
PFB find_p_func(char *s) {      // Function find_p_func: what type?
    for (i=0; i<num_func; i++)
        if (strcmp(table[i].name,s)==0) return table[i].func;
    return NULL; }
int main(int argc,char *argv[]) {
    ... }
```

Type Declarations in C



Type tree for type of `find_p_func`:

English: a function that takes a pointer to char as argument, and returns a pointer to a function that takes void as argument and returns int.



Exercise

```
struct _chunk {
    char name[10];
    int id; };
struct obstack {
    struct _chunk *chunk;
    struct _chunk *(*chunkfun)();
    void (*freefun) (); };

void chunk_fun(struct obstack *h, void *f) {
    h->chunkfun = (struct _chunk *(*)) f; }
void free_fun(struct obstack *h, void *f) {
    h->freefun = (void (*)(void)) f; }

int main() {
    struct obstack h;
    chunk_fun(&h,&xmalloc);
    free_fun(&h,&xfree); ... }
```

// Type struct_chunk: what type?

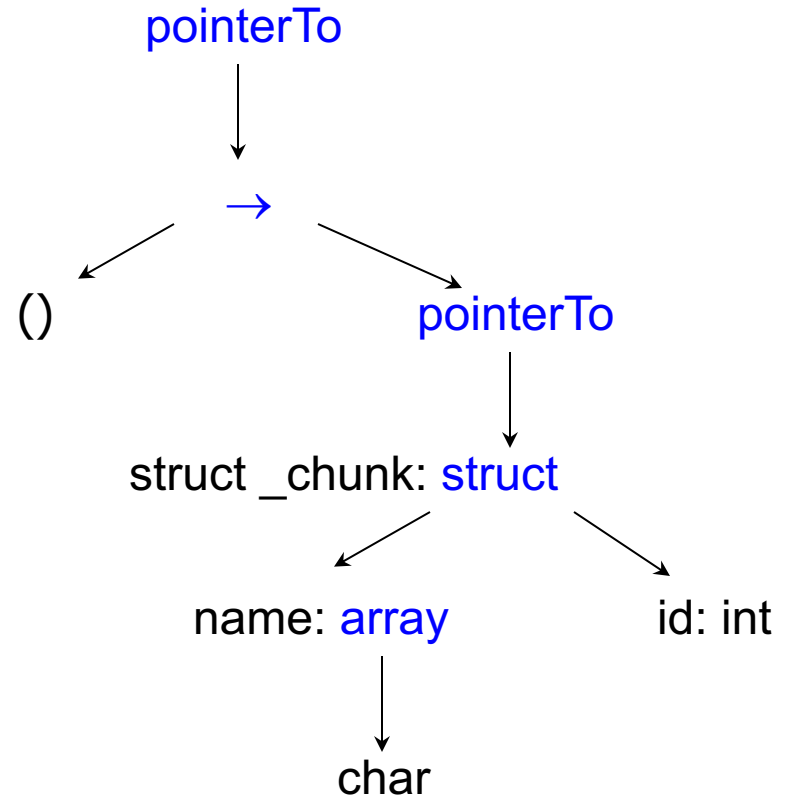
// Type struct obstack: what type?

// Function chunk_fun: what type?

// Function free_fun: what type?

Type Declarations in C

Type tree for type of field **chunkfun**:



Lecture Outline

- Types
- Type systems
 - Type checking
 - Type safety
- Type equivalence
- Types in C

- Primitive types
- Composite types

Primitive Types

- A small collection of built-in types
 - integer, float/real, etc.
- Design issues: e.g., boolean
 - Use integer 0/non-0 vs. true/false?
- Implementation issues: representation in the machine
 - Integer
 - Length fixed by standards or implementation (portability issues)
 - Multiple lengths (C: short, int, long)
 - Signs
 - Float/real
 - All issues of integers and more

Composite Types: Record (Struct)

- Collection of heterogeneous fields
- Operations
 - Selection through field names (**s.num**, **p->next**)
 - Assignment
 - Example: structures in C

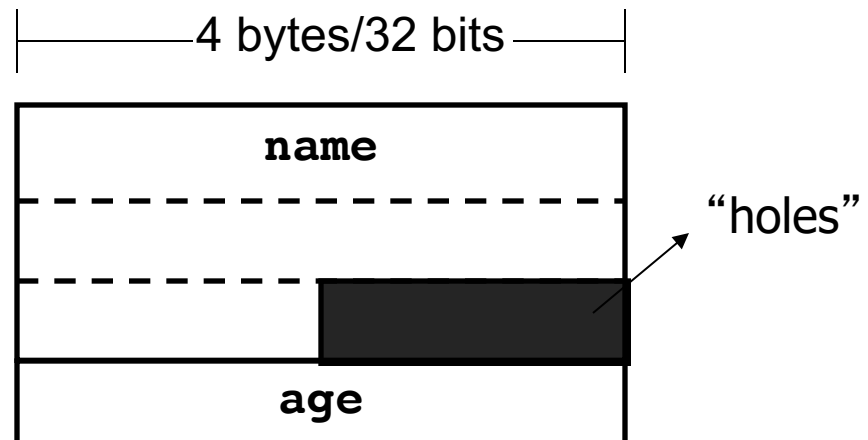
```
typedef struct cell listcell;  
struct cell {  
    int num;  
    listcell *next;  
} s, t;  
s.num = 0;  
s.next = 0;  
t = s;
```

Record (Struct)

- Definition of type. What is part of the type?
 - Order and type of fields (but not the name)
 - Name and type of fields
 - Order, name and type of fields
- Implementation issues: memory layout
 - Successive memory locations at offset from first byte. Usually, word-aligned, but sometimes **packed**

```
typedef struct {  
    char name[10];  
    int age;  
} Person;
```

```
Person p;
```



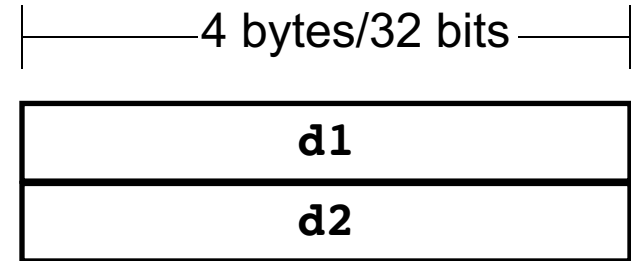
Variant (Union)

- Allow a collection of alternative fields; only one alternative is valid during execution
 - Fortran: equivalence
 - Algol68 and C: unions
 - Pascal: variant records
- Problem: how can we assure type-safety?
 - Pascal and C are not type-safe
 - Algol68 is type-safe! Uses run-time checks
- Usually alternatives use same storage
 - Mutually exclusive value access

Variants (Unions)

■ Example: unions in C

```
union data {  
    int k;  
    char c;  
} d1, d2;
```



■ Operations

- Selection through field names, Assignment:

```
d1.k = 3; d2 = d1; d2.c = 'b';
```

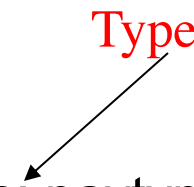
■ What about type safety?

```
if (n>0) d1.k = 5 else d1.c = 'a';
```

```
... d1.k << 2 ... // What is the problem?
```

Pascal's Variant Record

```
program main(input,output);
type paytype = (salaried, hourly);
var employee : record
    id : integer;
    dept: integer;
    age : integer;
    case payclass: paytype of
        salaried:
            (monthlyrate : real;
             startdate : integer);
        hourly:
            (rateperhour : real;
             reghours : integer;
             overtime : integer);
    end;
```



```
begin
employee.id:=001234;
employee.dept:=12;
employee.age:=38;
employee.payclass:=hourly;
employee.rateperhour:=2.75;
employee.reghours:=40;
employee.overtime:=3;
writeln(employee.rateperhour,
           employee.reghours,
           employee.overtime);
{this should bomb as there is no
 monthlyrate because
 payclass=hourly}
writeln(employee.monthlyrate);
```

Output:

2.750000E+00	40	3
2.750000E+00		

Pascal's Variant Record

```
type paytype = (salaried, hourly);
var employee : record
    id : integer;
    dept: integer;
    age : integer;
    case payclass: paytype of
        salaried: (
            monthlyrate : real;
            startdate : integer);
        hourly: (
            rateperhour : real;
            reghours : integer;
            overtime : integer);
    end;
```

```
employee.payclass:=salaried;
employee.monthlyrate:=575.0;
employee.startdate:=13085;
{this should bomb as there are no
 rateperhour, etc. because
 payclass=salaried}
writeln(employee.rateperhour,
employee.reghours
employee.overtime);
writeln(employee.monthlyrate);
end.
```

Output:

```
5.750000E+02    13085    3
5.750000E+02
```

Composite Types: Array

- Homogeneous, indexed collection of values
- Access to individual elements through subscript
- There are many design choices
 - Subscript syntax
 - Subscript type, element type
 - When to set bounds, compile time or run time?
 - How to initialize?
 - What built-in operations are allowed?

Array

- Definition of type. What is part of the type?
 - bounds/dimension/element type
 - Pascal
 - dimension/element type
 - C, FORTRAN, Algol68
- What is the lifetime of the array?
 - Global lifetime, static shape (in static memory)
 - Local lifetime (in stack memory)
 - Static shape (stored in fixed-length portion of stack frame)
 - Shape bound when control enters a scope
 - (e.g., Ada, Fortran allow definition of array bounds when function is entered; stored in variable-length portion of stack frame)
 - “Global” lifetime, dynamic shape (in heap memory)

Example: Algol68 Arrays

- Array type includes dimension and element type; it does not include bounds

```
[1:12] int month; [1:7] int day; row int
```

```
[0:10,0:10] real matrix;
```

```
[-4:10,6:9] real table; row row real
```

Note **table** and **matrix** are equivalent!

- Example - **[1:10] [1:5,1:5] int kinglear;**

- What is the type of **kinglear**?
- What is the type of **kinglear[j]**?
- What is the type of **kinglear[j][1,2]**?
- **kinglear[1,2,3]** ?

Array Addressing

- One dimensional array

- $\mathbf{x}[\mathbf{low}:\mathbf{high}]$ each element is E bytes
- Assuming that elements are stored into consecutive memory locations, starting at address $\mathbf{addr}(\mathbf{x}[\mathbf{low}])$, what is the address of $\mathbf{x}[\mathbf{j}]$?

$$\mathbf{addr}(\mathbf{x}[\mathbf{low}]) + (\mathbf{j} - \mathbf{low}) * E$$

- E.g, let $\mathbf{x}[\mathbf{0}:\mathbf{10}]$ be an array of reals (4 bytes)
 - $\mathbf{x}[\mathbf{3}]$? is $\mathbf{addr}(\mathbf{x}[\mathbf{0}]) + (3 - 0) * 4 = \mathbf{addr}(\mathbf{x}) + 12$
 - $\mathbf{x}[\mathbf{1}]$ is at address $\mathbf{addr}(\mathbf{x}[\mathbf{0}]) + 4$
 - $\mathbf{x}[\mathbf{2}]$ is at address $\mathbf{addr}(\mathbf{x}[\mathbf{0}]) + 8$, etc

Array Addressing

- Memory is a sequence of contiguous locations
- Two memory layouts for two-dimensional arrays:
 - Row-major order and column-major order
- Row-major order:
 - $y[0,0], y[0,1], y[0,2], \dots, y[0,n], y[1,*], y[2,*], \dots$
- $y[\text{low1}:\text{hi1}, \text{low2}:\text{hi2}]$ in Algol68, location $y[j,k]$ is

$$\text{addr}(y[\text{low1}, \text{low2}]) + (\text{hi2} - \text{low2} + 1) * E * (j - \text{low1}) + (k - \text{low2}) * E$$

#locs per row #rows in front # elements in row j in
 of row j front of element $[j,k]$

Array Addressing

Consider `y[0:2, 0:5] int matrix`.

Assume row-major order and find the address of `y[1, 3]`.

address of `y[1, 3]` = $\text{addr}(\text{y}[0, 0]) + (5-0+1)*4*(1-0) + (3-0)*4$

6 elements per row

1 row before row 1

3 elements in row 1 before 3

= $\text{addr}(\text{y}[0, 0]) + 24 + 12$

= $\text{addr}(\text{y}[0, 0]) + 36$

- Analogous formula holds for column-major order
- Row-major and column-major layouts generalize to n-dimensional arrays

Composite Types: Pointers

- A variable or field whose value is a reference to some memory location
 - In C: **int *p;**
- Operations
 - Allocation and deallocation of objects on heap
 - **p = malloc(sizeof(int));** **free(p);**
 - Assignment of one pointer into another
 - **int *q = p; int *p = &a;**
 - Dereferencing of pointer
 - ***q = 1;**
 - Pointer arithmetic
 - **p + 2**

Recursive Types

- A recursive type is a type whose objects may contain objects of the same type
 - Necessary to build linked structures such as linked lists
- Pointers are necessary to define recursive types in languages that use the value model for variables:

```
struct cell {  
    int num;  
    struct cell *next;  
}
```

Recursive Types

- Recursive types are defined naturally in languages that use the reference model for variables:

```
class Cell {  
    int num;  
    Cell next;  
  
    Cell() { ... }  
    ...  
}
```

The End
