Announcements

- HW9 due today

- HW10 coming up, will post after class
  - Team assignment
  - Data abstraction (types) and control abstraction (parameter passing)
  - Due on Tuesday, November 27th

Last Class

- Types
- Type systems
  - Type checking
  - Type safety
- Type equivalence

- Remember, we are now in the world of von Neumann languages using value model for variables!

Lecture Outline

- Type equivalence
- Types in C
  - Primitive types
  - Composite types
    - Records (Structures in C)
    - Variants (Unions in C)
    - Arrays
    - Pointers

Types

Read: Scott, Chapters 7 and 8

Type Equivalence

- Two ways of defining type equivalence
  - Structural equivalence: based on “shape”
    - Roughly, two types are the same if they consist 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

\[
\begin{align*}
T_1 & \ x; \ \ldots \\
T_2 & \ y; \\
x & = y;
\end{align*}
\]

Structural Equivalence

- A type name 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 \texttt{type \ n = T} or \texttt{typedef T \ n} in C, the type name \texttt{n} is structurally equivalent to \texttt{T}
  - Declaration makes \texttt{n} an alias of \texttt{T}. \texttt{n} and \texttt{T} are said to be aliased types
Structural Equivalence

- Example, Pascal-like language:
  ```
  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;
  ```

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
 type alink = 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 into equiv. classes, according to structural equiv., strict name equiv. and loose name equiv.
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:

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

```
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 to a double
    - double d,e; e = d + 2; // e coerced to 2.0
  - int to double,
  - float to double
  - How about float to int?
    - No. May lose precision and thus, cannot be coerced!

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

- Type equivalence
- Types in C
  - Primitive types
  - Composite types
    - Records (Structures in C)
    - Variants (Unions in C)
  - Arrays
  - Pointers

Pointers: 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());
  - struct parse_table{
    - char *name;
    - PFB func;
  };
  - int func1() {...} // Function func1: what type?
  - int func2() {...} // Function func2: what type?
  - struct parse_table table[] = {
    - (*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 PFB:

```
pointerTo

() → int
```

Type tree for type of find_p_func:

```
pointerTo

pointerTo

char → int
```

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

```
// Type struct_chunk: what type?
struct_chunk {
    char name[10];
    int id;
};

// Type struct_obstack: what type?
struct obstack {
    struct _chunk *chunk;
    struct _chunk *(*chunkfun)();
    void (*freefun)();
};

// Function chunk_fun: what type?
void chunk_fun(struct obstack *h, void *f) {
    h->chunkfun = (struct _chunk * (*)(())) f;
}

// Function free_fun: what type?
void free_fun(struct obstack *h, void *f) {
    h->freefun = (void (*)()) f;
}

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

Lecture Outline

- Type equivalence
- Types in C
- Primitive types
- Composite types
  - Records (Structures in C)
  - Variants (Unions in C)
  - Arrays
  - Pointers

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

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

```c
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
  ```c
  union data {
    int k;
    char c;
  } d1, d2;
  ```
- Operations
  - Selection through field names, Assignment:
    ```c
d1.k = 3; d2 = d1; d2.c = 'b';
    ```
- What about type safety?
  ```c
  if (n>0) d1.k=5 else d1.c='a';
  ...
  d1.k << 2 ...
  ```

Pascal's Variant Record

```pascal
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 :=salaried;
  employee.monthlyrate :=576.0;
  employee.startdate :=13085;
  if random <.5 then employee :=salaried
    else employee :=hourly;
  writeln(employee.monthlyrate);
  writeln(employee.startdate);
end.
```

Output:

```
2.750000E+00       40           3
2.750000E+00
```

AlgL68 Discriminated Unions

```algol
union(int, real, bool) kitchensink;
kitchensink := 3;
kitchensink := 3.14;
kitchensink := true;
if random <.5 then kitchensink := 1
  else kitchensink := 2.78
fi;
case kitchensink in
  (int j): print ("integer",j));
  (real e): print ("real",e));
  (bool b): print ("bool",b));
esac;
```

Output:

```
2.750000E+02       13085           3
2.750000E+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, Algol

- 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
- \(X[low:high]\) each element is \(E\) bytes
- Assuming that elements are stored into consecutive memory locations, starting at address \(addr(X[low])\), what is the address of \(X[j]\)?
  \(addr(X[low]) + (j-low)*E\)
- E.g., let \(X[0:10]\) be an array of reals (4 bytes)
  - \(X[3]\) is \(addr(X[0]) + (3 - 0)*4 = addr(X) + 12\)
  - \(X[1]\) is at address \(addr(X[0]) + 4\)
  - \(X[2]\) is at address \(addr(X[0]) + 8\), etc

Example: Algol68 Arrays

- Trimming: yields some cross section of an original Algol68 array (slicing an array into subarrays)
- Subscripting: limiting 1 dimension to a single index value

Example:
  - \([1:10]\) int a,b; \([1:20]\) real x; \([1:20,1:20]\) real xx;
    - assigns first 4 elements of \(a\) into first 4 elements of \(b\)
  - \(b := a - ?\)
  - \(xx[4,1:20] := x - ?\)
  - \(xx[8:9,7] := x[1:2]\)
    - assigns \(x[1]\) into \(xx[8,7]\) and \(x[2]\) into \(xx[9,7]\)

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:
  - \(y[0,0], y[0,1], y[0,2], \ldots, y[0,n], y[1,*], y[2,*], \ldots\)
- \(y[low1:hi1,low2:hi2]\) in Algol68, location \(y[j,k]\) is
  \(addr(y[low1,low2]) + (hi1-low1+1)*E* (j-low1) + (k-low2)*E\)
  - \#cols per row \#rows in front \# elements in row \# elements in row \#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 \( y[1,3] = \text{addr}(y[0,0]) + 5(5-0) + 4(1-0) + 3(0) + 4 \)
6 elements per row
1 row before row 1
3 elements in row 1 before 3
\[= \text{addr}(y[0,0]) + 24 + 12 = \text{addr}(y[0,0]) + 36\]
- 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: \( \text{int } *p; \)
- Operations
  - Allocation and deallocation of objects on heap
    - \( p = \text{malloc(sizeof(int))}; \text{free}(p); \)
  - Assignment of one pointer into another
    - \( \text{int } *q = p; \text{int } *p = \&a; \)
  - Dereferencing of pointer
    - \( *q = 1; \)
  - Pointer arithmetic
    - \( p + 2 \)

Points: 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:

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

Points: Recursive Types

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

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