Announcements

- HW 7 is due on Thursday, April 21th
  - Questions?

- Rainbow grades
  - HW 1-5. HW 6 still not available
  - Exam 1-2
  - Quiz 1-6
  - Any questions/concerns, let us know ASAP

Last Class

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

Lecture Outline

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

Type Equivalence

- Structural equivalence
  - Two types are equivalent if they have "same shape"
- Name equivalence
  - Each type definition forms a new type
  - Strong name equivalence
    - Alias types are distinct. E.g., T and n in typedef T n are distinct types
  - Loose name equivalence
    - Alias types are equivalent. E.g., T and n are equivalent

Type Equivalence and Type Compatibility

Questions:

- e := expression
  - 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

Read: Scott, Chapter 7.1 – 7.4
Type Conversion

- **Implicit conversion** — coercion
  - Conversion done implicitly by the compiler
  - In C, mixed mode numerical operations
    - In expression if e is a double and expression is an int, expression is implicitly coerced 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`?

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

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 Declarations in C

- Type tree for PFB:
  - `typedef int (*PFB)();` // Type variable PFB: what type?
  - `struct parse_table {
      char *name;
      PFB func;
      int (*func)();
      int func2() {...} // Function func1: what type?
      struct parse_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 tree for type of `find_p_func`:
  - `PFB` is 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.
Group Exercise

```c
struct _chunk {  // Type struct_chunk: what type?
    char name[10];  // id: int;
    struct obstack {  // Type struct obstack: what type?
        struct _chunk *chunk;  // struct_chunk *chunkfun;
        void (*chunkfun)(int);  // Function chunk_fun: what type?
    } chunkfun;  // Function free_fun: what type?
    struct obstack *freefun;  // void (*)();
};

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

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

int main() {  // Function free_fun: what type?
    struct obstack h;
    chunk_fun(&h,&xmalloc);
    free_fun(&h,&xfree); ...
```

Type Declarations in C

```c
Type tree for type of field chunkfun:

```

Lecture Outline

- Type conversion
- 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 (Structure)

- Collection of heterogeneous fields
- Operations
  - Selection through field names (s.num, p->next)
  - Assignment
    - Example: structures in C
typedef struct cell listcell;
struct cell {  // struct_chunk:
    int num;
    listcell *next;
} s,t;
s.num = 0;
s.next = 0;
t = s;

Record (Structure)

- Definition of type. What is part of the type?
  - order and type of fields
  - 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]:
    char name[10];  // 4 bytes/32 bits
    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
  ```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';
  ```

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

Pascal variant record

Program main(input,output);
type paytype = (salaried,hourly);
var employee : record
  id : integer;
depth : 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);
  if (employee.payclass=salaried) then
    writeln(employee.monthlyrate);
  else
    writeln(employee.rateperhour,
            employee.reghours,
            employee.overtime);
end.
```

Output:
```
2.750000E+00          40           3
2.750000E+00
```

Pascal discriminant unions

```c
union(int, real, bool) kitchensink;
kitchensink := 3;
kitchensink := 3.14;
kitchensink := true;
if random < .5 then kitchensink := 1
  else kitchensink := 2.78
fi;
```
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
d  - C, FORTRAN, Algol68
  - Choice has ramifications on kind of type checking needed
- 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] \text{int month}; [1:7] \text{int day}; \text{row int}
  - \[[0:10,0:10] \text{real matrix}; \text{row real}
  - \[-4:10,6:9] \text{real table}; \text{row row real}

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

Example - \([1:10] \{1:5,1:5\} \text{ int kinglear;}

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

Array Addressing

- One dimensional array
  - \(X[\text{low:high}]\) each element is \(E\) bytes
  - Assuming that elements are stored into consecutive memory locations, starting at address \(\text{addr}(X[\text{low}])\), what is the address of \(X[j]\)?
  - \(\text{addr}(X[\text{low}]) + (j-\text{low}) \times E\)

    - E.g. let \(X[0:10]\) be an array of reals (4 bytes)
      - \(X[3]\) is \(\text{addr}(X[0]) + (3-0) \times 4 = \text{addr}(X) + 12\)
      - \(X[1]\) is at address \(\text{addr}(X[0]) + 4\)
      - \(X[2]\) is at address \(\text{addr}(X[0]) + 8\), etc
Array Addressing

Consider $y[0:2, 0:5]$ int matrix. Assume row-major order and find the address of $y[1,3]$.

address of $y[1,3] = \text{addr}(y[0,0]) + ?$

Analogous formula holds for column-major order.

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`

Pointers: 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;
  }
  ```

Pointers: Recursive Types

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

  ```
  class Cell {
    int num;
    Cell next;
  }
  ```

  ```
  Cell() { ... }
  ```