

Announcements

- **Prolog programming due May 3**
- **Some C assignments are being regraded**
 - specifically, those that had “seg faults”
- **Review session Friday May 7, 12:00-3:00 PM, SEC 218 (with Prof Borgida and Prof Ryder)**
 - a practice final is on the web site
- **Final exam: May 10th, 4:00-7:00 pm in the College Ave Gym Annex**
 - let me know about any conflicts by Thursday (tomorrow!)
- **Office hours next week will be announced on web**

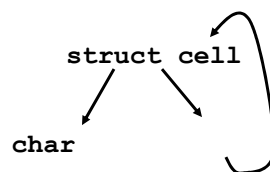
1

Recursive Types

A recursive (self-referential) type:

```
struct cell {  
    char data;  
    struct cell c;  
}
```

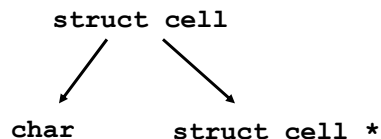
(not valid C)



Tree representation of type

In C use a pointer to know the size of the struct

```
struct cell {  
    char data;  
    struct cell *ptr;  
}
```



2

More on type equivalence

- **Structural equivalence**
 - Intuitively means ‘same shape’
 - Shown by isomorphism of corresponding type trees (with or without labels)
 - Are these equivalent types?

```
struct cell          struct element
{
    char data;      char c;
    int a[3];       int a[5];
    struct cell *next;  struct element *ptr;
}
```

3

Type Equivalence in C

- **Anonymous types are actually differentiated by internal (to the C compiler) type names**
 - Louden Ch 6.5

```
struct RecA          typedef struct          struct
{
    char x;          char x;          {
    int y;          int y;          char x;
} a;                } RecB;          int y;
                    } c;
typedef struct RecA RecA;
RecB b;
```

Which variables (of a, b, c) are of equivalent type?

4

Name equivalence

- Intuitively, 2 types are equivalent if they have the same name

```
struct RecA a;          struct RecA
RecA b;                {
struct RecA c;          char x;
                        int y;
                        }
struct {                typedef struct RecA RecA;
  char x;
  int y;
}d;
```

- *Which variables are of types that are name equivalent?*
- `typedef A B` does not make variables declared A to be name equivalent to variables declared B (however, it does make their types structurally equivalent).

5

Type equivalence in C

- Use structural equivalence for everything except unions and structs, which use 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;
int (*F) (int);
Age (*G) (Age);
```

6

More C examples

```
int main()
{
    struct B { char x; int y;};

    // A and B equiv
    typedef struct B A;

    struct { A a; A *next;} aa;

    // Not equiv to previous one
    struct { struct B a; struct B *next;} bb;
    struct { struct B a; struct B *next;} cc;

    A a;    struct B b;
    a = b;
}

```

- **aa** and **bb**: types are incompatible, even though they seem to be constructed from equivalent types
- **bb** and **cc**: types are incompatible, even though they seem to consist of the same named types

7