CSE 3302
Programming Languages

Data Types(cont.)

Chengkai Li, Weimin He
Spring 2008

Function Type in C

```
typedef int (*IntFunction)(int);

int square(int x) {return x*x;}

IntFunction f = square;

int evaluate(IntFunction g, int value){
    return g(value);
}
-
printf("%d\n",evaluate(f,3));
```

Function Type in ML

```
type IntFunction = int -> int;

fun square(x: int) = x * x;

val F = square;

Fun evaluate(g:IntFunction, value:int) = g value;
-
evaluate(F,3);
```

Vector, List

Functional languages:

- Vectors: like arrays, more flexibility, especially dynamic resizability.
- Lists: like vectors, can only be accessed by counting down from the first element.

Pointer

- A pointer type is a type in which the range of values consists of memory addresses and a special value, nil (or null)
- Advantages:
  - Addressing flexibility (address arithmetic, explicit dereferencing and address-of, domain type not fixed (void *))
  - Dynamic storage management
  - Recursive data structures
    - E.g., linked list

```
struct CharListNode
{ char data;
  struct CharListNode* next;
};
Typedef struct CharListNode* CharList;
```

Problems with Pointers

- Alias (with side-effect)

```
int *a, *b;
a=(int *) malloc(sizeof(int));
*a=2;
b=(int *) malloc(sizeof(int));
*b=3;
b=a;
*b=4;
printf("%d\n", *a);
```

Problems with Pointers

- Dangling pointers (dangerous)

```
int *a, *b;
a = (int *) malloc(sizeof(int));
*a = 1;
b = a;
free(a);
printf("%d\n", *b);
```


Problems with Pointers

- Garbages (waste of memory)

memory leakage

```
int *a;
a = (int *) malloc(sizeof(int));
*a=2;
a = (int *) malloc(sizeof(int));
```


Type System

- ***Type Constructors:***
  - Build new data types upon simple data types
- ***Type Checking:*** The translator checks if data types are used correctly.
  - ***Type Inference:*** Infer the type of an expression, whose data type is not given explicitly.
    e.g., x/y
  - ***Type Equivalence:*** Compare two types, decide if they are the same.
    e.g., x/y and z
  - ***Type Compatibility:*** Can we use a value of type A in a place that expects type B?

  Nontrivial with user-defined types and anonymous types

Strongly-Typed Languages

- Strongly-typed: (Ada, ML, Haskell, Java, Pascal)
- Most data type errors detected at translation time
- A few checked during execution and runtime error reported (e.g., subscript out of array bounds).

- Pros:
- No data-corrupting errors can occur during execution. (I.e., no unsafe program can cause data errors.)
- Efficiency (in translation and execution.)
- Security/reliability

- Cons:
- May reject safe programs (i.e., legal program is a subset of safe programs.)
- Burden on programmers, may often need to provide explicit type information.

Weakly-typed and untyped languages

- Weakly-typed: C/C++
  - e.g., interoperability of integers, pointers, arrays.
- Untyped (dynamically typed) languages: scheme, smalltalk, perl
  - Doesn't necessarily result in data errors.
  - All type checking performed at execution time.
  - May produce runtime errors too frequently.

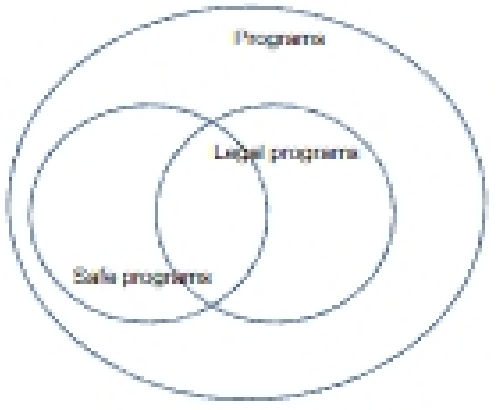
Security vs. flexibility

- Strongly-typed :
  - No data errors caused by unsafe programs.
  - Maximum restrictiveness, static type checking, illegal safe programs, large amount of type information supplied by programmers.
- Untyped:
  - Runtime errors, no data-corruptions. Legal unsafe programs.
  - reduce the amount of type information the programmer must supply.

Security vs. flexibility

- Strongly-typed :
- A type system tries to maximize both flexibility *and* security, where flexibility means: reduce the number of *safe* illegal programs & reduce the amount of type information the programmer must supply.
- Flexibility, no explicit typing or static type checking
  vs.
- Maximum restrictiveness, static type checking

Safe vs. Legal



Type Equivalence

- How to decide if two types are the same?
- Structural Equivalence
  - Types are sets of values
  - Two types are equivalent if they contain the same values.
- Name Equivalence

Structural Equivalence

```
struct RecA {
    char x;
    int  y;
}
struct RecB {
    char x;
    int  y;
}
struct RecC {
    char w;
    int  v;
}
struct RecD {
    int  y;
    char x;
}
```

Char I Int

Int I Char

But are they equivalent in these languages?

- In C:

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

b=a;
```

(Error: incompatible types in assignment)

But are they equivalent in these languages?

- In C:

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

b=&a;
```

(Warning: incompatible types in assignment)