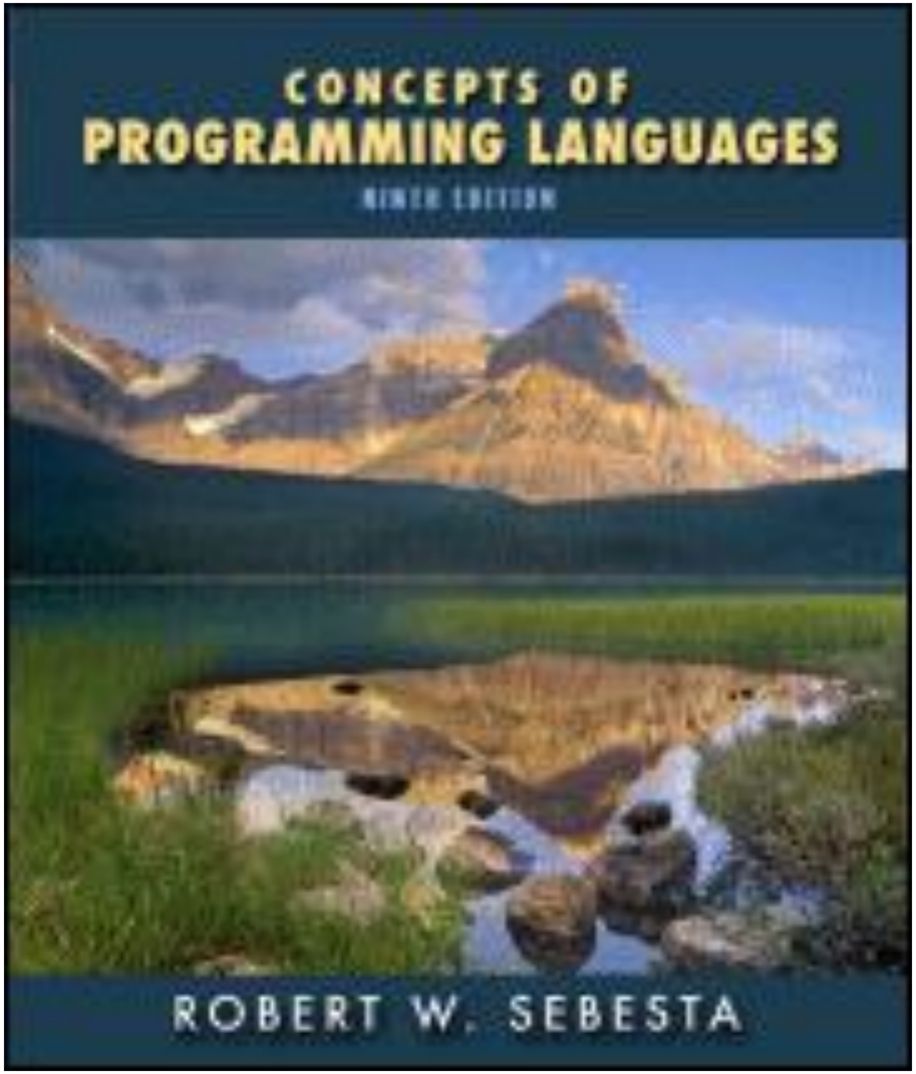
Chapter 6

Data Types



Ch06 – Data Types

6.1 Introduction*
6.2 Primitive Data Types*
6.3 Character String Types*
6.4 User-defined Ordinal Types*
6.5 Array Types*
6.6 Associative Arrays*
6.7 Record Types*
6.8 Union Types*
6.9 Pointer and Reference Types
6.10 Type Checking
6.11 Strong Typing
6.12 Type Equivalence
6.13 Theory and Data Types

6.9 Pointer and Reference Types

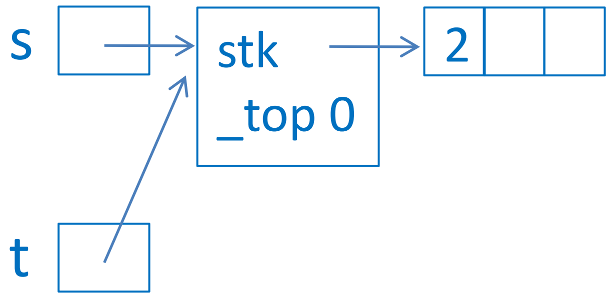
- Pointer type
  - Usually explicit dereferencing
  - Example – C/C++

```
class stack {
public:   stack(int n) : stk(new int[n]), _top(-1) {}
          ~stack() { delete [] stk; }
          void push(int x) { stk[++_top] = x; }
          void pop() { _top--; }
          int& top()   { return stk[_top]; }   // return by ref.
          const int& top() const   { return stk[_top]; }
          bool empty() const { return _top==-1; }
private: int *stk,_top;
};
```

6.9 Pointer and Reference Types

- Example (Cont'd)

```
int main()
{
    stack* s = new stack(3);
    p(s);
    (*s).top()++;
    delete s;
}
void p(stack* t) { (*t).push(2); }        // call by value
```



6.9 Pointer and Reference Types

- Reference type
  - Usually implicit dereferencing
- Reference type in C++
  - for call and return by reference
  - Property
    - A reference variable must be initialized.
    - The binding can't be altered.
  - Example (Cont'd)

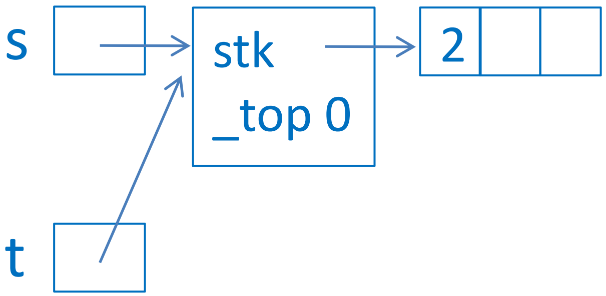
    stack& s;   // ill-formed

    stack& s = *new stack(3);

6.9 Pointer and Reference Types

- Example (Cont'd)

```
int main()
{
    stack& s = *new stack(3);
    p(s);
    s.top()++;
    delete &s;
}
void p(stack& t) { t.push(2); }   // call by reference
```



- Comment

  Parameter passing and function value returning are treated as initialization.

6.9 Pointer and Reference Types

- Reference type in Java
  - Primitive types are value types whose variables store values
    Class and array types are reference types whose variables store references.
  - Example

    int x = 2;

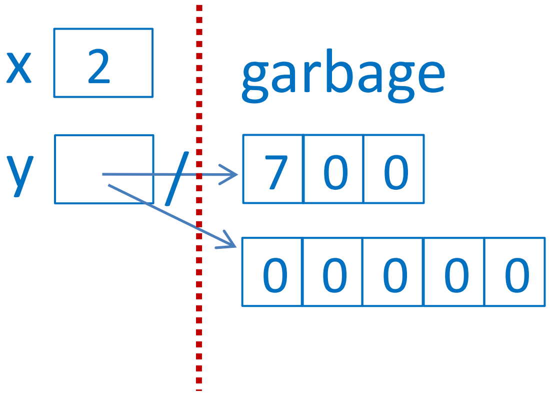
    int[] y; // y is a reference to an array

    y = new int[3];

    y[0] = 7;

    y = new int[5];

    N.B. Garbage is recycled by garbage collector.



6.9 Pointer and Reference Types

- Property

  Similar to C/C++ pointer types, rather than C++ reference types

  - A reference variable may or may not be initialized.
  - The binding may be altered.

- Comment

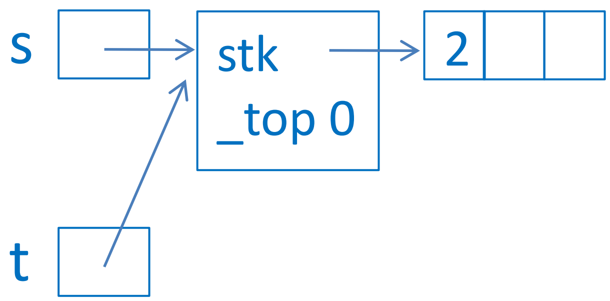
  Java uses call and return by value

6.9 Pointer and Reference Types

- Example

```
class demo {
    public static void main(String[] args)
    {
        stack s = new stack(3);  // s is a reference to a stack
        p(s);
        // s.top()++;   // NO
    }   // the stack becomes garbage

    static void p(stack t) { t.push(2); }    // call by value
}
```



6.9 Pointer and Reference Types

- Example (Cont'd)

```
class stack {
    public stack(int n) { stk=new int[n]; _top=-1; }
    public void push(int x) { stk[++_top]=x; }
    public void pop() { _top--; }
    public int top() { return stk[_top]; }  // return by value
    public boolean empty() { return _top==-1; }
    private int[] stk;
    private int _top;
}
```

N.B. There is no destructor. Java uses garbage collection.

6.9 Pointer and Reference Types

- Reference type in perl
  - Example

```
$a = 7;
$b = \$a;          # \ is similar to C/C++'s &
print $$b;         # 7      same as $a
@c = (1,3,5);
$b = \@c;
print @$b;         # 135
print $$b[0];      # 1      same as $c[0]
print $b->[0];     # 1
```

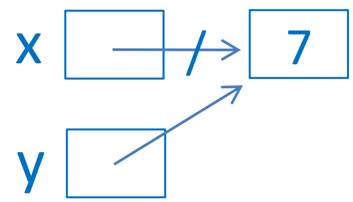
  - Property

    Similar to C/C++ pointer types, rather than C++ reference types

6.9 Pointer and Reference Types

- Dangling pointer
  - A pointer that contains the address of a heap-dynamic variable that has been deallocated.

```
int *x = new int(7);
int *y = x;
cout << *y;        // ok
delete x;
cout << *y;        // dangling pointer, caused by deallocator
```

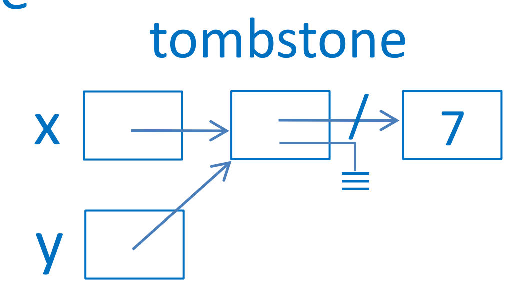


- How to solve dangling pointer? --- Java, Perl, SML, Scheme
  - Approach 1 – Provide no deallocator; use garbage collector
  - Approach 2 – Detect dangling pointer

6.9 Pointer and Reference Types

- Detecting dangling pointer
  - Tombstone

```
int *x=new int(7);  // obtain storage; create a tombstone
int *y=x;           // y points to the tombstone
cout << *y;         // ok, non-nil tomstone
delete x;           // reclaim storage; set tombstone to null
cout << *y;         // error, nil tombstone
```



Drawbacks

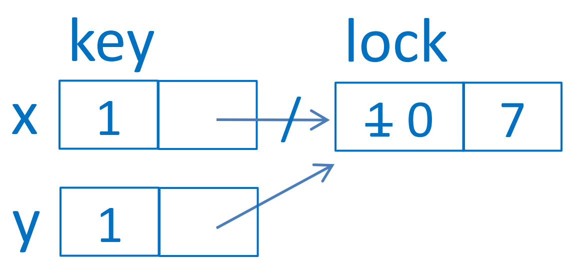
1 Tombstones are never deallocated.

2 Require one more level of indirection

6.9 Pointer and Reference Types

- Locks-and-keys

```
int *x = new int(7);    // obtain storage and create a lock;
                        // copy the lock to x's key
int *y = x;             // copy x's (key, pointer) pair to y
cout << *y;             // ok, y's key = lock
delete x;               // reclaim storage and clear the lock
                        // to an illegal value (say, 0)
cout << *y;             // error, y's key ≠ lock
```



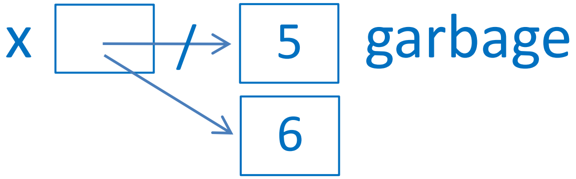
- Either method is time- and space-consuming.

6.9 Pointer and Reference Types

- Garbage (memory leak)
  - A heap-dynamic variable that is no longer accessible.
  - Example

    Integer x=new Integer(5);

    x=new Integer(6);



  - Garbage is usually recycled by reference counting or garbage collector.
- Reference counting
  - Each object has a count of the number of references to it.

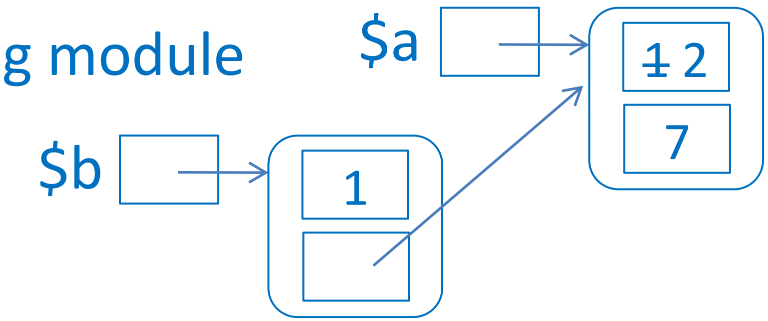
    The count is incremented/decremented when a reference to the object is created/destroyed.

    If the count reaches 0, the storage is reclaimed.

6.9 Pointer and Reference Types

- Example (Perl)

```
use Devel::Peek;  # debugging module
$a = 7;
Dump $a;          # (1)
$b = \$a;
Dump $a;
```




```
SV = IV(0x61c988) at 0x600ec8
    REFCNT = 1
    FLAGS = (IOK,pIOK)
    IV = 7
```

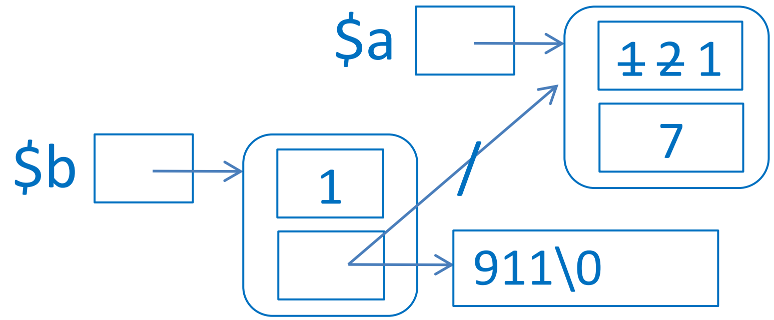
```
SV = IV(0x61c988) at 0x600ec8
    REFCNT = 2
    FLAGS = (IOK,pIOK)
    IV = 7
```

SV	Scalar Value
IV	Integer Value
RV	Reference Value
PV	Pointer Value

6.9 Pointer and Reference Types

- Example (Cont'd)

```
Dump $b;
$b = "911";
Dump $a;      # same as (1)
Dump $b;
```




```
SV = PV(0x603898) at 0x600f68
   REFCNT = 1
   FLAGS = (POK,pPOK)
   PV = 0x604900 "911"\0
   CUR = 3
   LEN = 8
```

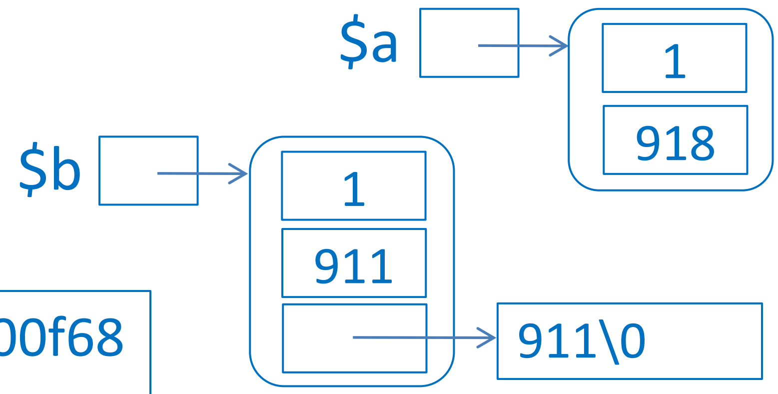
```
SV = RV(0x62f060) at 0x600f68
   REFCNT = 1
   FLAGS = (ROK)
   RV = 0x600ec8
   SV = IV(0x61c988) at 0x600ec8
      REFCNT = 2
      FLAGS = (IOK,pIOK)
      IV = 7
```

6.9 Pointer and Reference Types

- Example (Cont'd)

  $a += $b;

  Dump $b;



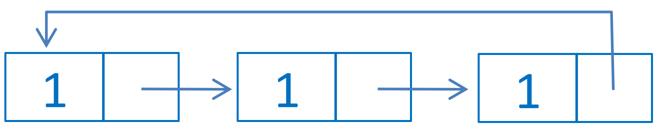

```
SV = PVIV(0x6020c8) at 0x600f68
   REFCNT = 1
   FLAGS = (IOK,POK,pIOK,pPOK)
   IV = 911
   PV = 0x604900 "911"\0
   CUR = 3
   LEN = 8
```

Perl trades memory for processing speed. Instead of doing a lot of conversions, Perl does a lot of look up.

6.9 Pointer and Reference Types

- Property

  1 Eager and deterministic

  Once an object becomes inaccessible, it is collected.

  2 Can’t collect a cycle

  Need specific cycle-detecting algorithms, e.g. Devel::Cycle

- Garbage collector
  - Each heap cell contains a garbage collection bit.
  - Naïve mark-and-sweep

    0 Clear all garbage collection bits

    1 Marking phase

    Starting with a root set (e.g. runtime stack, global data), mark all reachable heap cells.

6.9 Pointer and Reference Types

- Naïve mark-and-sweep (Cont'd)
  2 Sweeping phase
  Reclaim all heap cells that have not been marked
- Property
  1 Lazy and nondeterministic
  When will inaccessible objects be collected are unpredictable.
  2 Freeze programs periodically and unpredictably
- Some languages allow user to invoke the garbage collector
  e.g. In Java,
  System.gc();

6.10 Type Checking

- Type checking
  - Type checking

    Check if the operands of an operator are of compatible types
  - Compatible type

    A compatible type is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type
    - This automatic conversion is called a coercion.
  - Static type checking: Compile-time type checking
  - Dynamic type checking: Run-time type checking

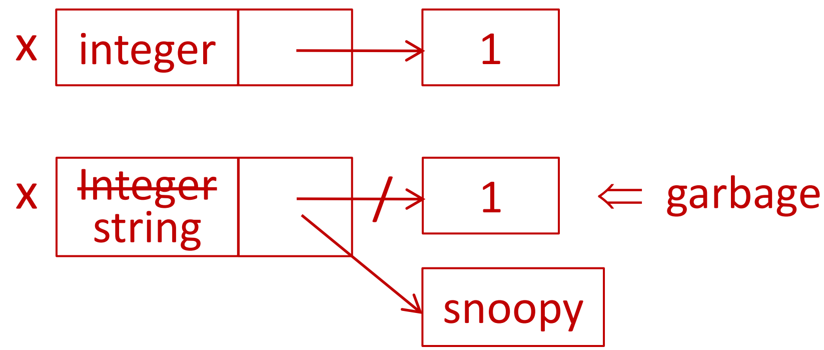
6.10 Type Checking

- Dynamic type checking in Scheme
  - Internal representation of variable (storage binding)

    > (define x 1)

    > (set! x "snoopy")

    > (define x "snoopy")



    Note: Garbage is reclaimed by garbage collector.

  - N.B. Scheme naturally supports unlimited integers.

    (define f (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))))

    (f 100) ⇒ 9332621544394415268169923885626670049071596826
    43816214685929638952175999932299156089414639761565182862
    536979208272237582511852109168640000000000000000000000000

6.10 Type Checking

- Compiled code

> (+ x 1)

The compiled code reads as

(if (not (number? x))

runtime type error

(cond ((complex? x) $(+^{complex}$ x 1+0i))

((real? x) $(+^{real}$ x 1.0))

((rational? x) $(+^{rational}$ x 1/1))

(else $(+^{integer}$ x 1)))) ; i.e. (integer? x)

Numerical tower in Scheme

number
complex
real
rational
integer

6.10 Type Checking

- Dynamic type checking in JavaScript
  - A JavaScript program doesn't yield runtime type errors.
  - It converts values into suitable types according to their use.

```
var a = [7,"2","a"]     // mixed-type array
function f() {}
a[0]+a[1]    ⇒ "7"+"2" = "72"
a[0]–a[1]    ⇒ 7–2 = 5
a[0]<a[2]    ⇒ 7<NaN = false
alert(a+f)   ⇒ "7,2,a"+"function f() {}" = 7,2,afunction f() {}
alert(a*f)   ⇒ NaN*NaN = NaN
```

6.10 Type Checking

- Compiled code

  alert(x+1)

  is compiled to

  if (typeof x=="number") alert(x+^{real}1)   // only 64-bit reals

  else alert(String(x)+^{string} String(1))

  alert(x*1)

  is compiled to

  alert(Number(x)*1)

  where Number(x) converts x to a number, if possible; otherwise, it returns an NaN.

6.11 Strong Typing

- Strong typing
  - The term "strongly typed" has no commonly agreed-upon definition.
  - Book's definition

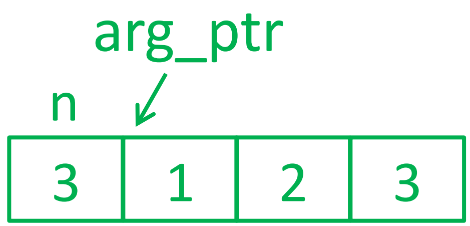
    A programming language is *strongly typed* if type errors are always detected (at compile- and/or run-time).
- C/C++ aren't strongly typed in this sense.
  - Unions are not type-checked

    ```
    union t { int x; double y; } a;
    a.x = 2;
    cout << a.y;   // reinterpret a 4-byte int as an 8-byte double
    ```

6.11 Strong Typing

- Parameter type checking can be avoided.

```
#include <cstdarg>
int sum (int n,...)                    // variable argument list
{
    va_list arg_ptr;                   // typedef char* va_list;
    int s = 0;                         // sum(3,1,2,3)
    va_start(arg_ptr,n);
    for (int i=1;i<=n;i++)
        s += va_arg(arg_ptr,int);
    va_end(arg_ptr);                   // arg_ptr=NULL
    return s;
}
```



6.11 Strong Typing

- Parameter type checking can be avoided. (Cont'd)

  ```
  sum(3,1,2,3)        // correct call
  sum(4,1,2,3,4)      // correct call
  sum(2,1.0,2.0)      // incorrect call
  sum(2)              // incorrect call
  ```

  But all of these calls are not type-checked.

- Coercions affect strong typing.

  ```
  int f(int n) { return n==0? 1: n*f(n-1); }
  int main() { cout << f(5.6); }
  ```

  The erroneous call f(5.6) can't be detected, since C/C++ allow coercions.

6.11 Strong Typing

- ML is strongly typed in either sense.
  - Coercions are disallowed in ML.

    – 2 + 3.4;   (* Error *)
  - Unions are type checked.

    – datatype t = I of int | R of real;   a $\boxed{\text{I}}\boxed{2}$

    – val a = I 2;

    – val R y = a;   (* Error: nonexhaustive binding failure *)

    Thus, one can't reinterpret a 4-byte int as an 8-byte real

    Note: ML's unions are discriminated unions; C/C++'s unions are free unions. (See 6.8)

    – datatype t = L of int | R of int;   vs   union t {int x; int y; }

    t = { L x | x ∈ int } U { R x | x ∈ int }   t = int U int

6.11 Strong Typing

- Typing strength is a continuum
  - ML is more strongly typed than C++, which in turn is more strongly typed than C.
  - ```
    void p() {}                 // unknown parameter list in C
    int main()                  // parameterless in C++
    {
        p("Snoopy ");
    }
    ```
    In C++, this causes a compile-time type-checking error.
    But in C, there is no type checking error.
    To detect the error in C, do this:
    ```
    void p(void) {}             // parameterless in C (and C++)
    ```
    ```

6.12 Type Equivalence

- Type equivalence and type compatibility
  - Type compatibility = Type equivalence + Coercion

    int a,b; float c;

    a = b;   // type equivalent and compatible

    a = c;   // type compatible
  - They are interchangeable, especially when considering structured types (∵ coercion of structured types are rare).
  - Two approaches of type equivalence

    struct X { int m; int n; } a, b;

    struct Y { int m; int n; } c;

    a = b;   // name equivalence (C/C++) ⇒ structure equiv.

    a = c;   // structure equivalence

6.12 Type Equivalence

- Name (type) equivalence
  - Two types are equivalent if they have the same name.
  - The compiler has to assign an internal name for each unnamed type.

```
struct  { int m; int n; } a;  ⇒  struct T1 { int m; int n; } a;
struct  { int m; int n; } b;  ⇒  struct T2 { int m; int n; } b;
a = b;      // No, T1 and T2 are distinct names.

struct  { int m; int n; } a, b;
a = b;      // Yes (C/C++)  ⇒  struct T { int m; int n; } a, b;
            // No (Ada)     ⇒  struct  { int m; int n; } a;
                               struct  { int m; int n; } b;
```

6.12 Type Equivalence

- Pro: Easy to implement
- Con: More restrictive (inflexible)

```
void p(int (&a)[5]);
int a[5];
p(a);        // OK, C++ uses structure equivalence for arrays
```

It is illegal with name equivalence and has to be written as

```
typedef int T[5];      // Assume that T is a new type
void p(T& a);
T a;  p(a);
```

N.B. typedef doesn’t introduce new types in C/C++.

```
typedef int INT;
int x; INT y; x = y;      // OK, still name equivalence
```

6.12 Type Equivalence

- Structure (type) equivalence
  - Two types are equivalent if they have the same structure.
  - Pro: More flexible
  - Con: Harder to implement

    struct X { int m; int n; } a;

    struct Y { int s; int t; } b;

    Are X and Y structure equivalence, i.e. are field names part of the structure?

    N.B. Field names are considered in SML records.

    – {m = 2, n = 3} = {s = 2, t = 3}; (* error: different types *)

    ↑ ↑

    {m : int, n : int} {s : int, t : int}

6.12 Type Equivalence

- Con: Can’t differentiate between types with the same structure, e.g.

  type celsius = float;

  fahrenheit = float;

  It is unreasonable to treat celsius and fahrenheit as equivalent types.