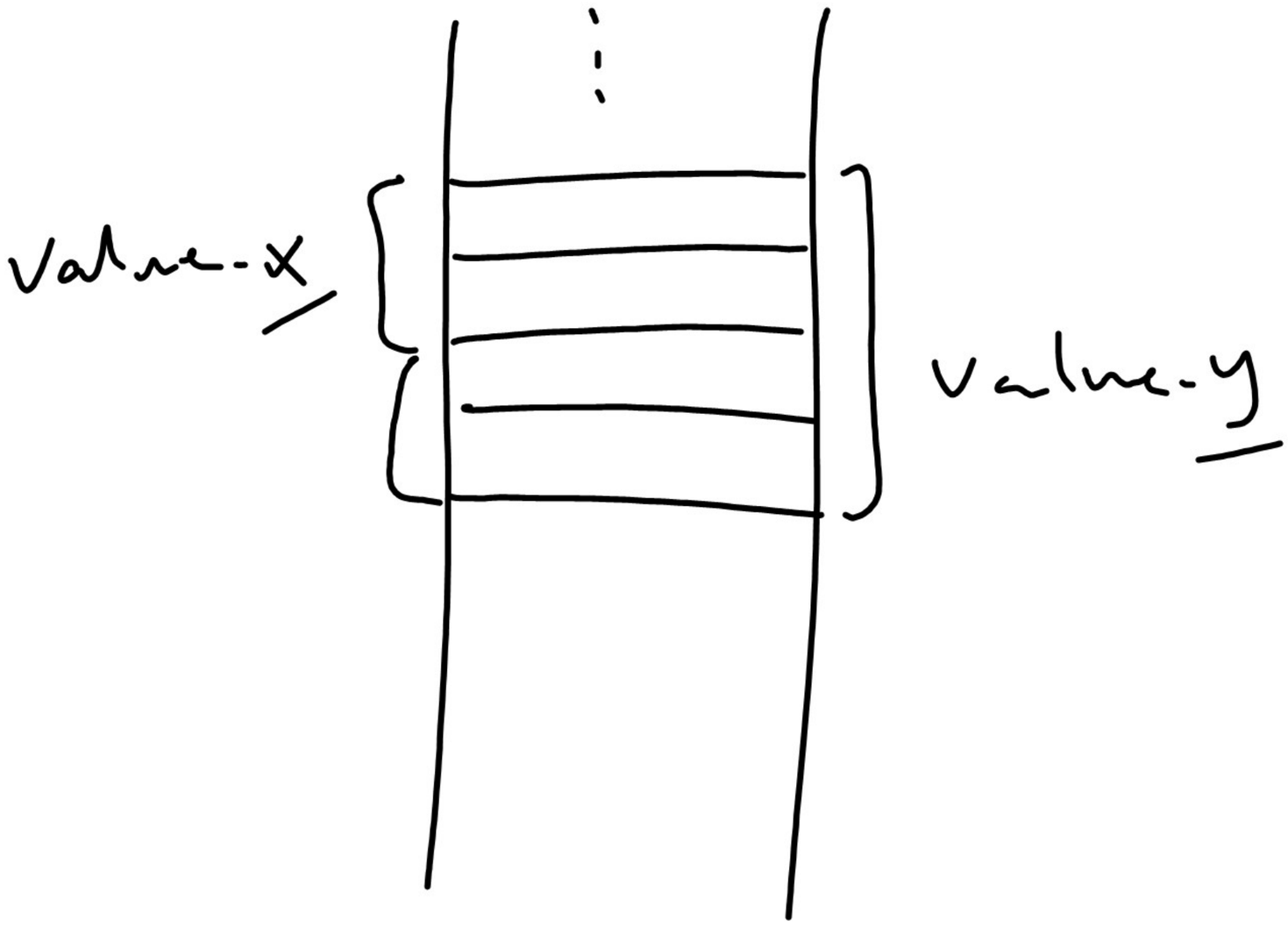
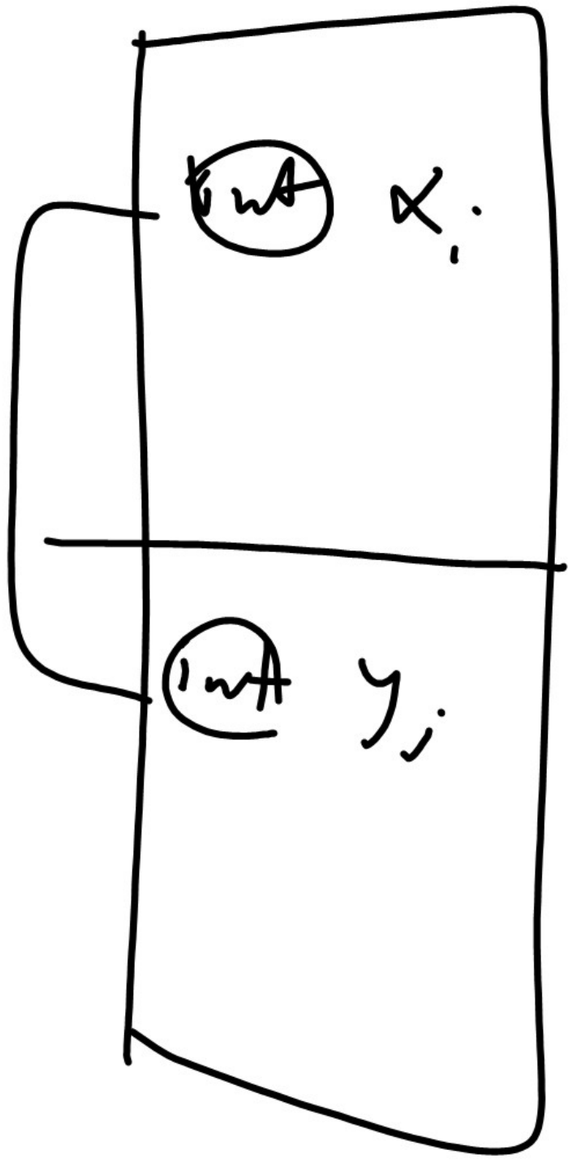
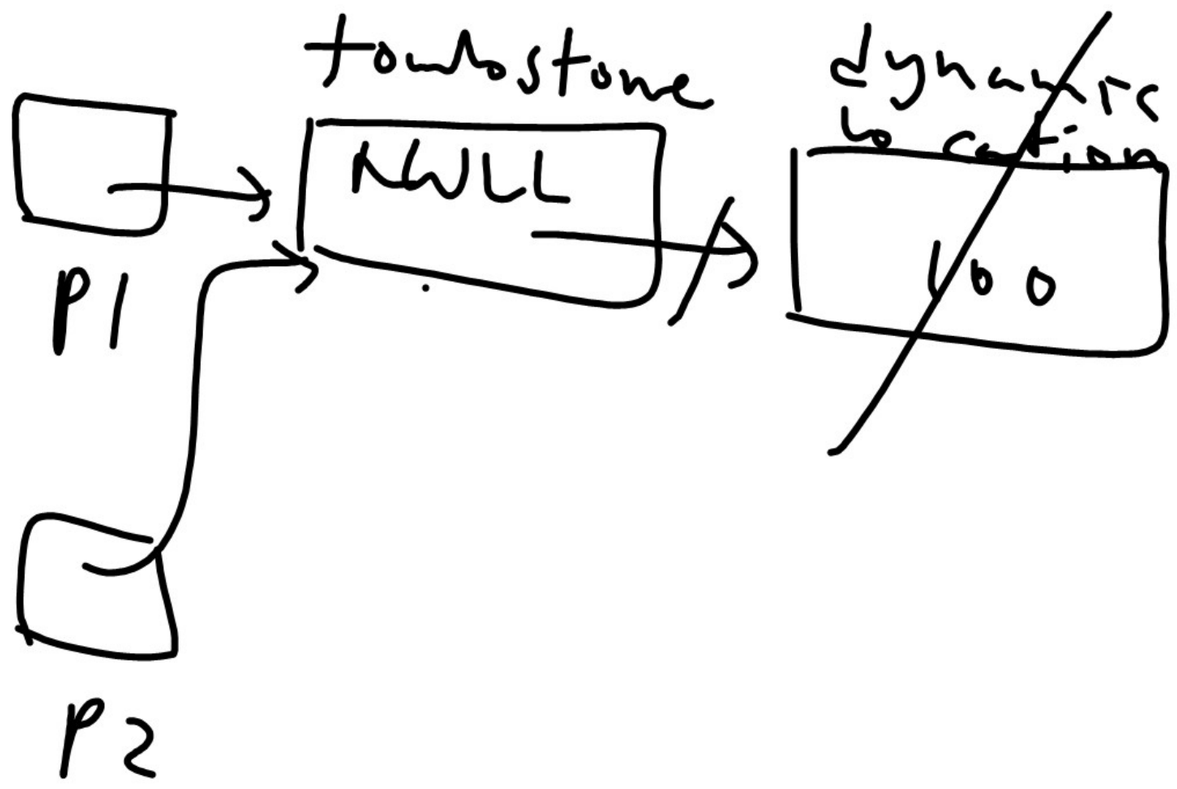
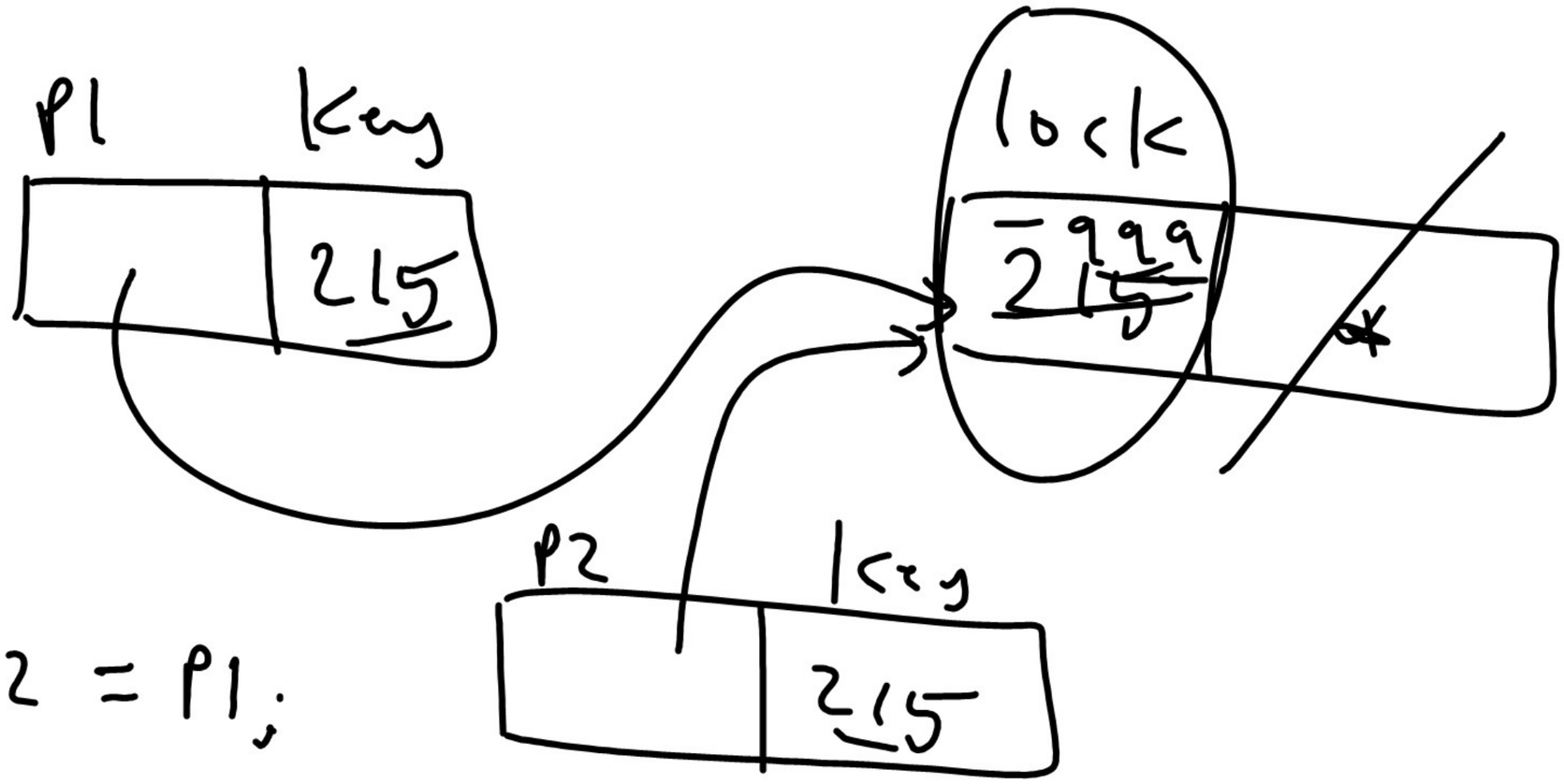

value-x
value-y


x_i
y_j

```
int *p1 = new int;
int *p2 = p1;
---
delete p1;
```



int * p1 = new int;



int * p2 = p1;

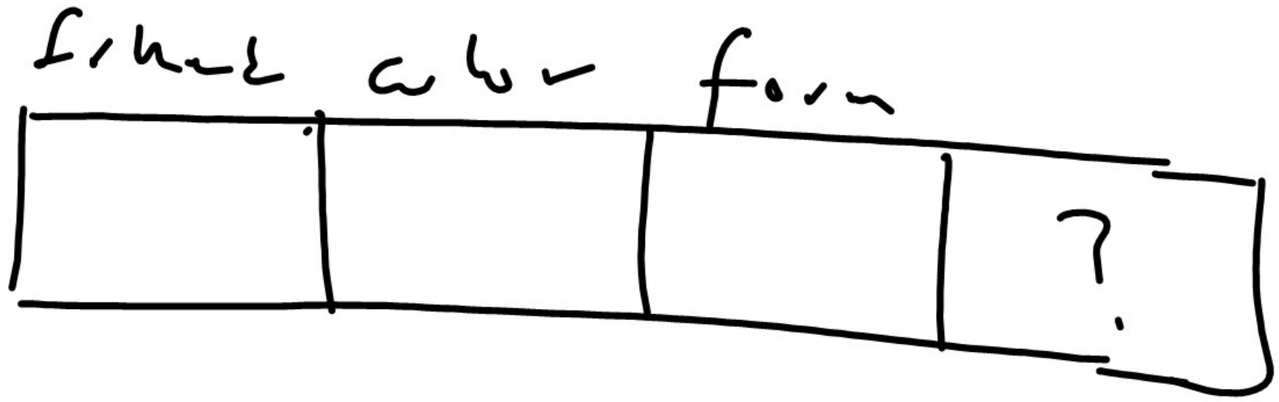
delete p1;

Record types

- A record is a heterogeneous aggregate of data elements in which individual elements are identified by names
- Introduced by Cobol in 60's

Ada Varying Records


```
type shape is (circle, triangle, rectangle);
type colors is (red, green, blue);
type figure (form: shape) is
    record
        filled: boolean;
        color: colors;
        case form is
                when circle =>
                        radius: float;
                when triangle=>
                        left-side: integer;
                        right-side: integer;
                        angle: float;
                when rectangle =>
                        side-1: integer;
                        side-2: integer;
        end case;
    end record;
    ...
Figure_1 : Figure;
Figure_1:=(filled=>true, color=>blue, form=>rectangle, side_1=>12, side_2=>3);
```



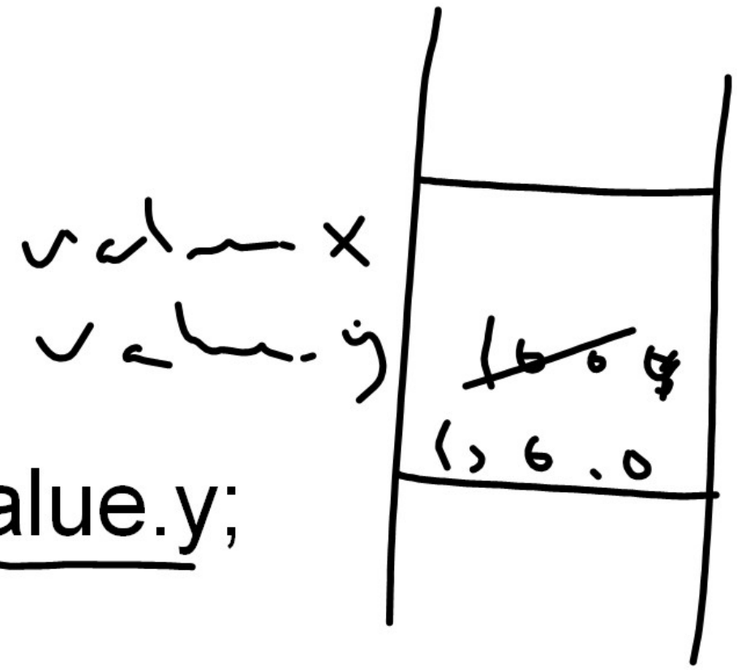
Union Types

- May store different type values at different times during program execution
- Fortran:

```
INTEGER X
REAL Y
EQUIVALENCE(X,Y)
```

C Union

```
union number{ int x; float y;} ;
number value;
void main(){
   value.x = 100;
   cout<<value.x<<"      "<<value.y;
   value.y = 100.0;
   cout<<endl<<value.x<<"      "<<value.y;
}
```

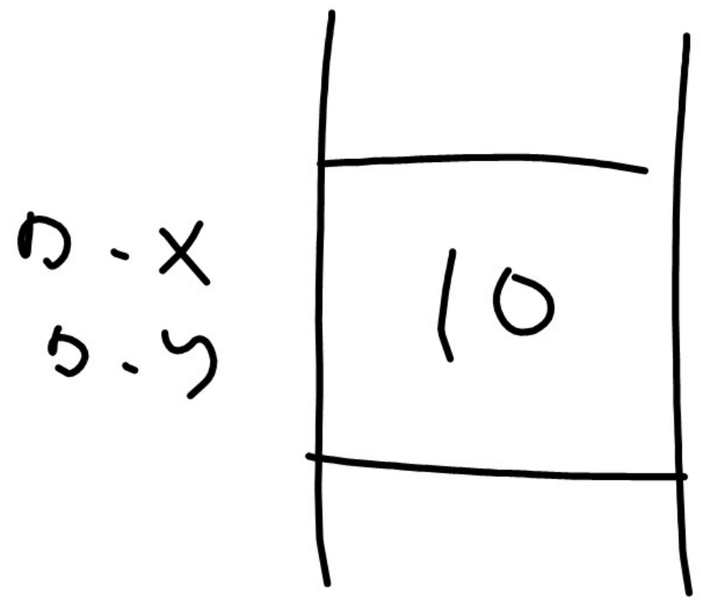


visual C++ output:

```
100                    1.4013e-043
1120403456             100
```

Union and Type Checking

```
union un{int x; float y;};
int a;
float b;
un D;
D.x = 10;
a = D.x; //valid
b = D.x; //valid after coercion
b = D.y; //meaningless value ends in b  (type error not detected)
// This makes the language less strongly types
```



Pointers

- Pointer variables have a range of values that consist of memory addresses and a special value, nill, indicating that the pointer cannot be used to reference any object
- Two basic operations on pointers
  - Referencing: produces the value of the pointer as **P + 2**
  - Dereferencing: produces the value in the location the pointer points to as in ***P + 2**
- C provides more operations on pointers

Pointers in PLI

- The first language to provide pointers
- A pointer can point to an object of any type

  POINTER P; INTEGER X; FLOAT Y;

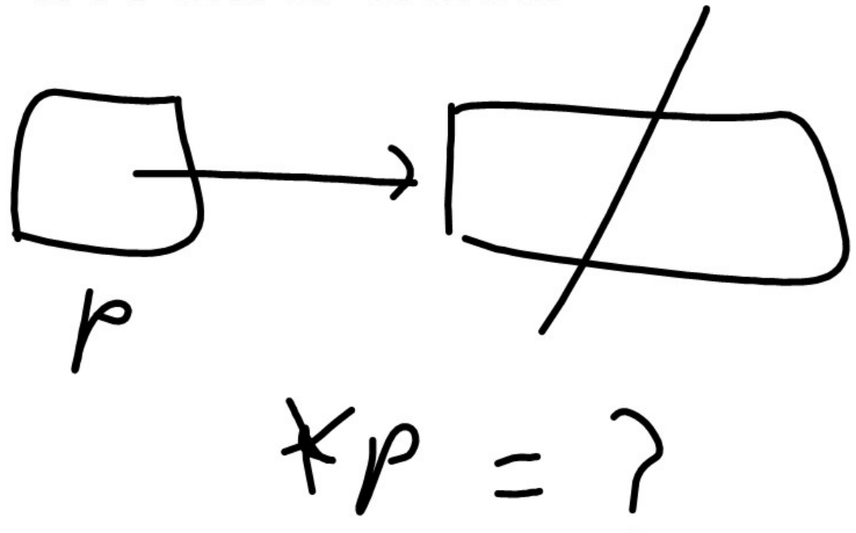
  P = addr(X);

  P = addr(Y);

- More flexible
- Makes static type checking of a pointer dereference impossible

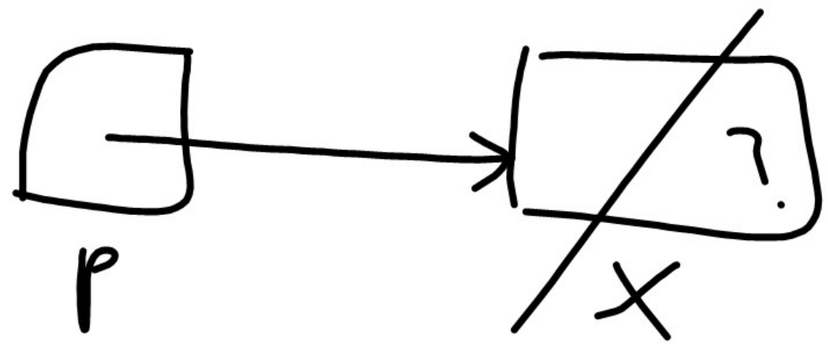
Dangling Pointer Problem

- A pointer that contains the address of a dynamic variable that has been deallocated
- Reduces reliability since accessing mistakingly the deallocated location through the pointer yields invalid data



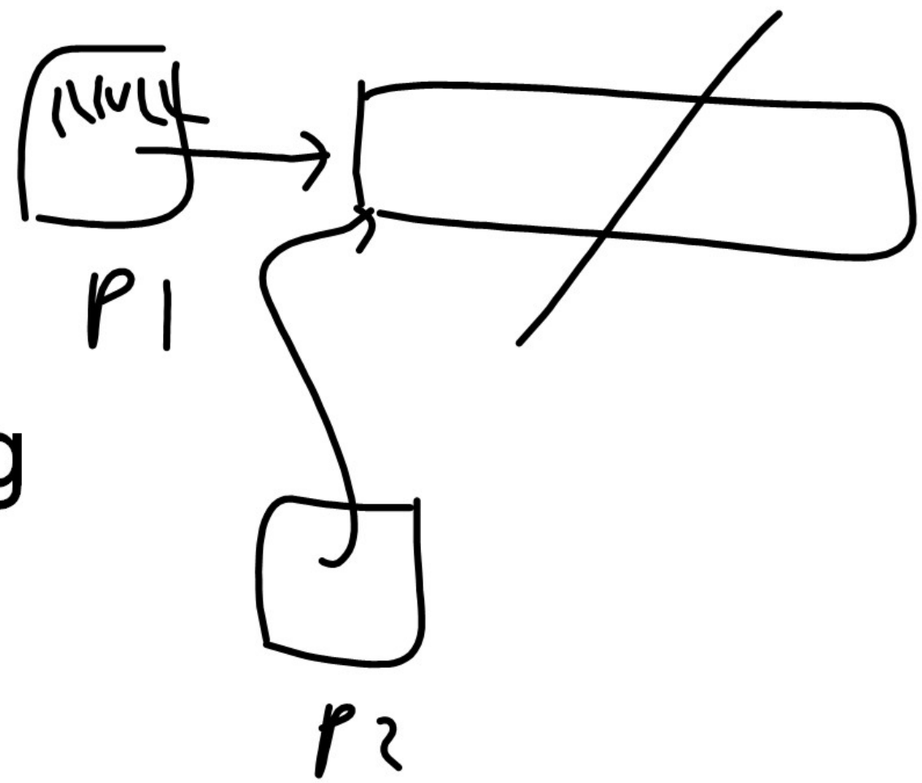
Dangling Pointer Example

```
If (…){
   int *p;
   …
   while(…){
        int x;
        p=&x;
        …
   }
   // p is dangling between end of while and end of if
   // accessing *p results in a logical error
   …
}
```



Dangling Pointer Example

```
int * p1=new int;
int * p2=p1;
delete p1;
// p2 becomes dangling
...
```



Tombstone Solution

- Use an inermediate pointer, called tombstone, for each dynamic variable.
- The pointer points to tombstone and the tobstone points to the dynamic variable.
- When the Dynamic variable is deallocated the intermediate pointer remains but it is assigned NULL value indicating that the dynamic vaiable no longer exists.
- Any reference to a pointer pointing to NULL tomstone is detected as an error
- Requires extra memory to store the tombstone and requires an extra memory access to reach the dynamic variable

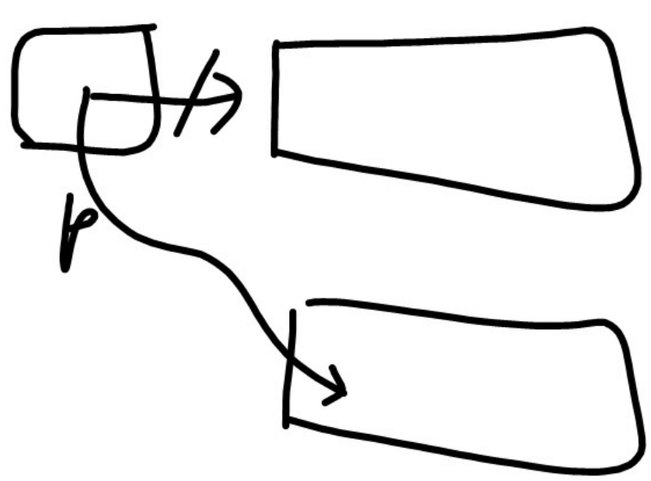
Key-lock Solution

- A key value (int) is added to the pointer and a lock value (int) is added to the dynamic variable
- In oder for the pointer to access a dynamic variable its key value must be equal to the lock value of the dynamic variable. A run-time error occurs if they do not match
- Locks must be maintained through out execution
- Requires extra time to compare key and lock values for each access to the dynamic variable

Lost Object Problem

- An allocated dynamic object that is no longer accessable by the program
- Also known as garbage or memory leak problem since the lost object is allocated memory and it is not being used
- Example

```
int *p; int sum=0;
while (1) {
   p=new int;
   cin>>*p;
   sum += *p;
   ...
}
```



Expressions

Side Effect Problem

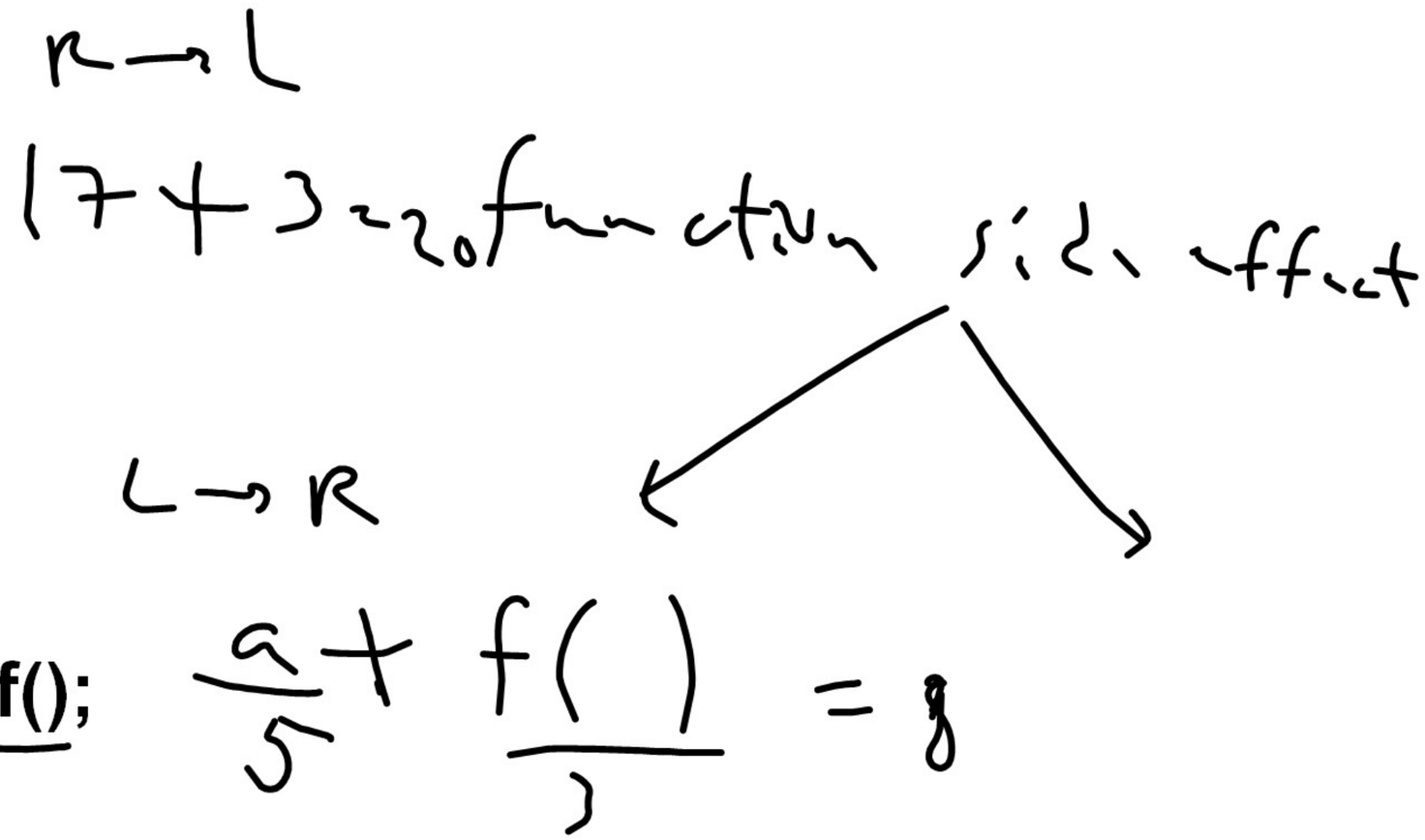
- A side effect of a function occurs when it changes the value of one of its parameters or changes the value of a global variable
- A side effect of an operator occurs when this operator changes the value of one of its operands
- Side effect is considered a problem since the value of an expression might be dependent on the order in which the operands are evaluated
- This might result in expressions that produce one value under one compiler and produce another value under another compiler

C language Examples

- +, *, /, %, >, <, … do not have side effects
- **A + B** returns the same value for
  L→R evaluation: load A, load B, add
  R→L evaluation: load B, load A, add
- ++, --, =, -=, +=, … do have side effects
- **A++ < ++A** returns true for L→R evaluation and false for R→L evaluation

Function Example

```
int a = 5;
int f(){
    a=17;
    return 3;
}//f
void main(){
    cout<<a+f();
}
```



The output will be either 8 or 20 depending on the order of evaluation of the operands of + operation

Suggested Solution1

- Language designer disallows functional side effects by stating rules such as:
  - No operator changes the value of its operand(S)
  - No function changes the value of one of its parameters or the value of a global variable
- Too restrictive

Suggested Solution 2

- Stating in language definition that operands are to be evaluated in a fixed order; left to right or right to left.
- Restricts the compiler ability to perform some optimizations. For example:

f(a) + x + f(a) = f(a) + f(a) + x

cannot be transformed into

2*f(a)+x