Safe Memory-Leak Fixing for C Programs

Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang,
Weikun Yang, Zhaoping Zhou, Bing Xie, Hong Mei

Institute of Software, Peking Unversity

内存管理

- 安全攸关软件的开发必然涉及内存管理问题
- 软件工程经典问题，数千篇论文
- 垃圾回收
  - 广泛用于Java, Go等大量新语言
  - 通过动态扫描内存发现需要回收的内存

垃圾回收 vs 安全攸关软件

- 大量系统资源无法通过垃圾回收管理
  - 文件句柄、线程锁等
- 在安全攸关软件中，内存对象常常也无法通过垃圾回收管理
  - 实时嵌入式系统，运行资源有限
  - 大数据处理系统，垃圾收集耗时过长
    - 处理数据量达到10G时，垃圾收集运行时间占程序运行总时间一半以上

内存泄露的例子

内存分配

内存释放

内存使用

泄露

泄露

```c
#include <stdlib.h>
#include <stdio.h>

void f(int *p, int **q){
  *q = p;
}
void g(int *p){
  free(p);
}
int h(int size, int num, int sum){
  int *p = (int*)malloc(sizeof(int)*size);
  int **q = (int**)malloc(sizeof(int*));
  if (size == 0)
    g(p);
  else
    for (int i = 0; i < size; ++i)
      if (p[i] != num){
        f(p, q);
        sum += (*q)[i];
      }
      else
        return i;
  printf("%d", sum);
  return sum;
}
```

内存泄露的例子


```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void f(int *p, int **q){
5    *q = p;
6  }
7  void g(int *p){
8    free(p);
9  }
10 int h(int size, int num, int sum){
11   int *p = (int*)malloc(sizeof(int)*size);
12   int **q = (int**)malloc(sizeof(int*));
13   if (size == 0)
14     g(p);
15   else
16     for (int i = 0; i < size; ++i)
17       if (p[i] != num){
18         f(p, q);
19         sum += (*q)[i];
20       }
21       else
22         return i;
23   printf("%d", sum);
24   return sum;
25 }
```

free(q);

free(p);

free(q);

free(p);

free(q);

修复内存泄露仍是难题

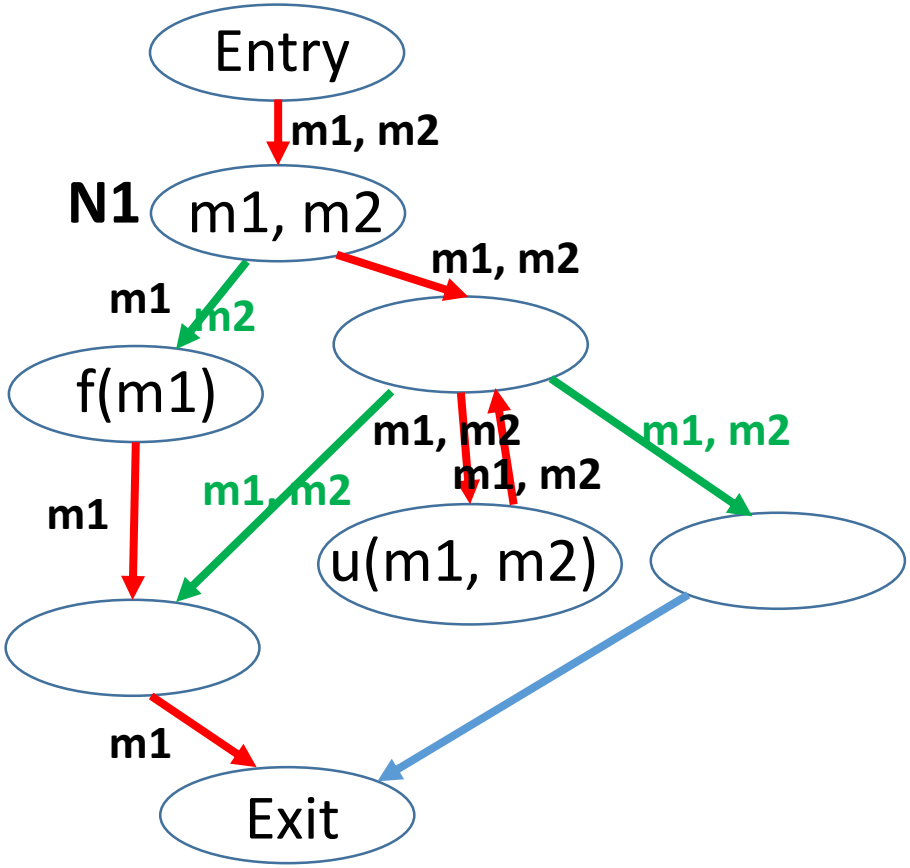
- 需要考虑多个条件
  - 内存释放前必须已分配
  - 内存释放时必须有能从栈上访问的路径
  - 在任意路径上，内存不能被释放两次
  - 在任意路径上，内存使用前不能被释放
- 漏掉任何一条都将导致致命错误
- 实践中，常常出现发现内存泄露但不敢修改的情况

研究成果：自动内存修复技术

- 通过对C程序代码的分析，自动查找内存泄露并修复
- 保证修复的绝对正确性
  - 对于任意插入的free语句和任意执行路径
    - 释放前分配：在执行到free之前所指的内存已经分配
    - 无双重释放：在该路径上没有任何其他free语句释放同一块内存
    - 无释放后使用：在该free之后所释放内存不能再被使用
- 能在较短时间内完成对大型程序的分析工作
- 论文已经被软件工程顶级会议ICSE’15接收

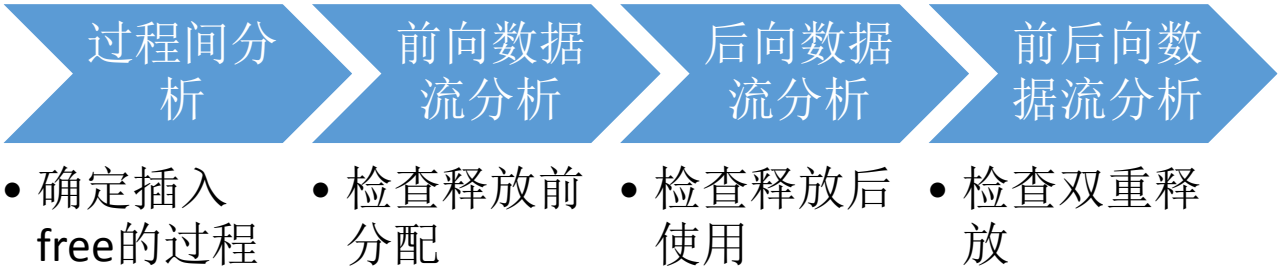
背景：数据流分析

- 标准编译技术
- 通过遍历控制流图，给结点附上分析结果
- 保证分析的安全性
  - 分析结果一定是真实结果的子集或超集



技术路线和创新

- 反复使用数据流分析



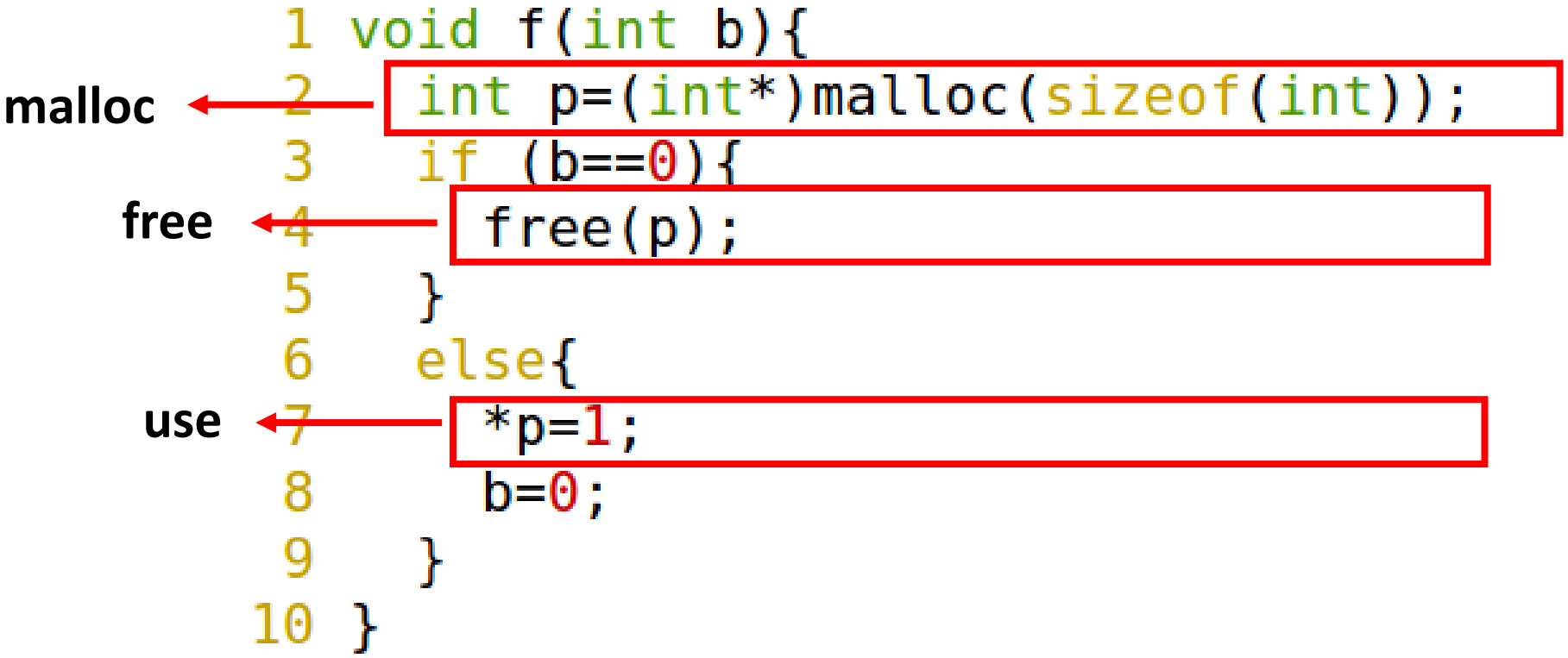
- 处理各种复杂情况
  - 循环、全局变量、多重分配、空指针判断等问题

- 在一定程度上，用数据流分析的效率达到了路径敏感分析的效果

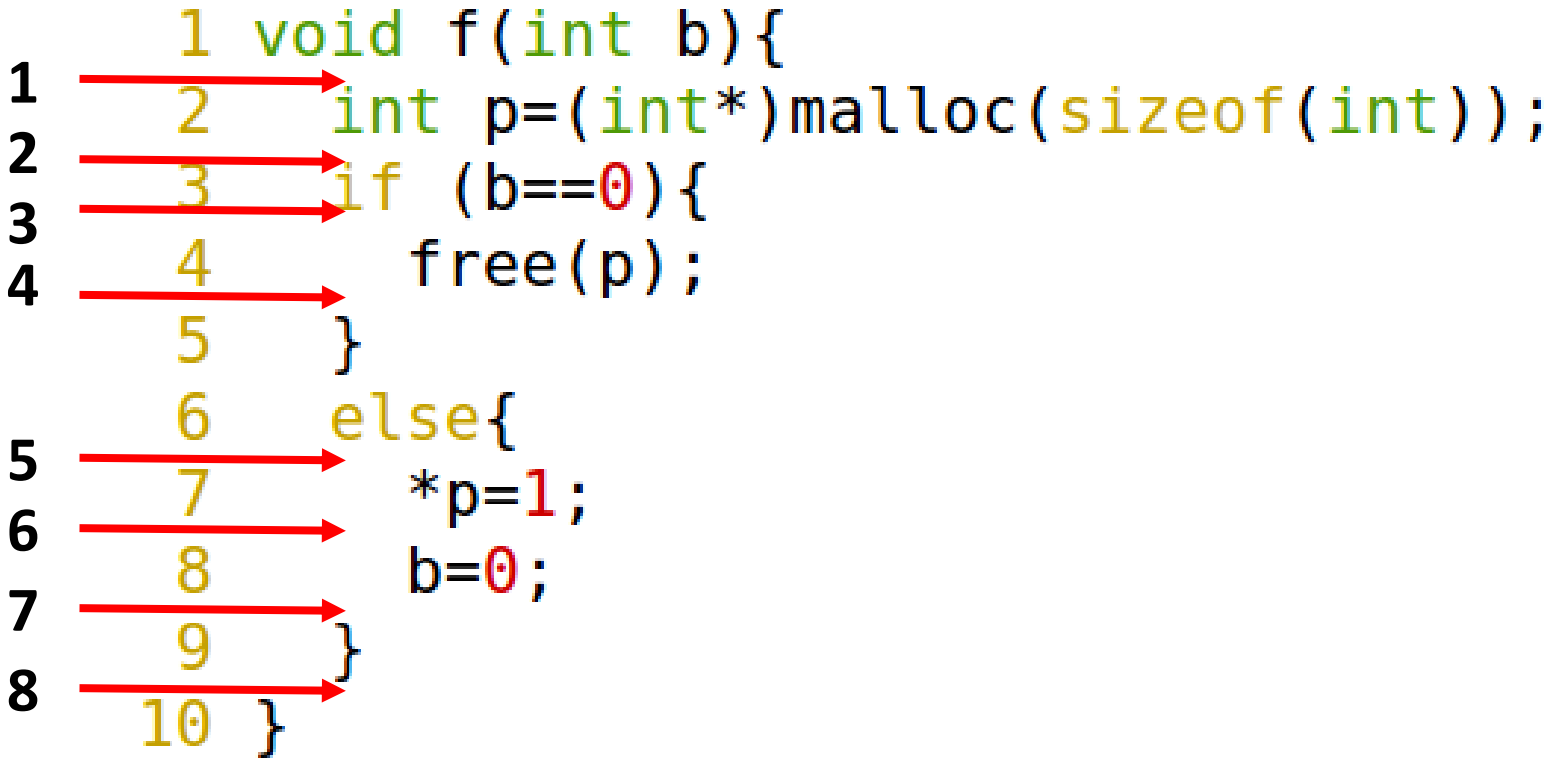
Defining a safe fix

- A fix consists of two components
  - A location to insert a free invocation
  - The expression to be used in the free invocation

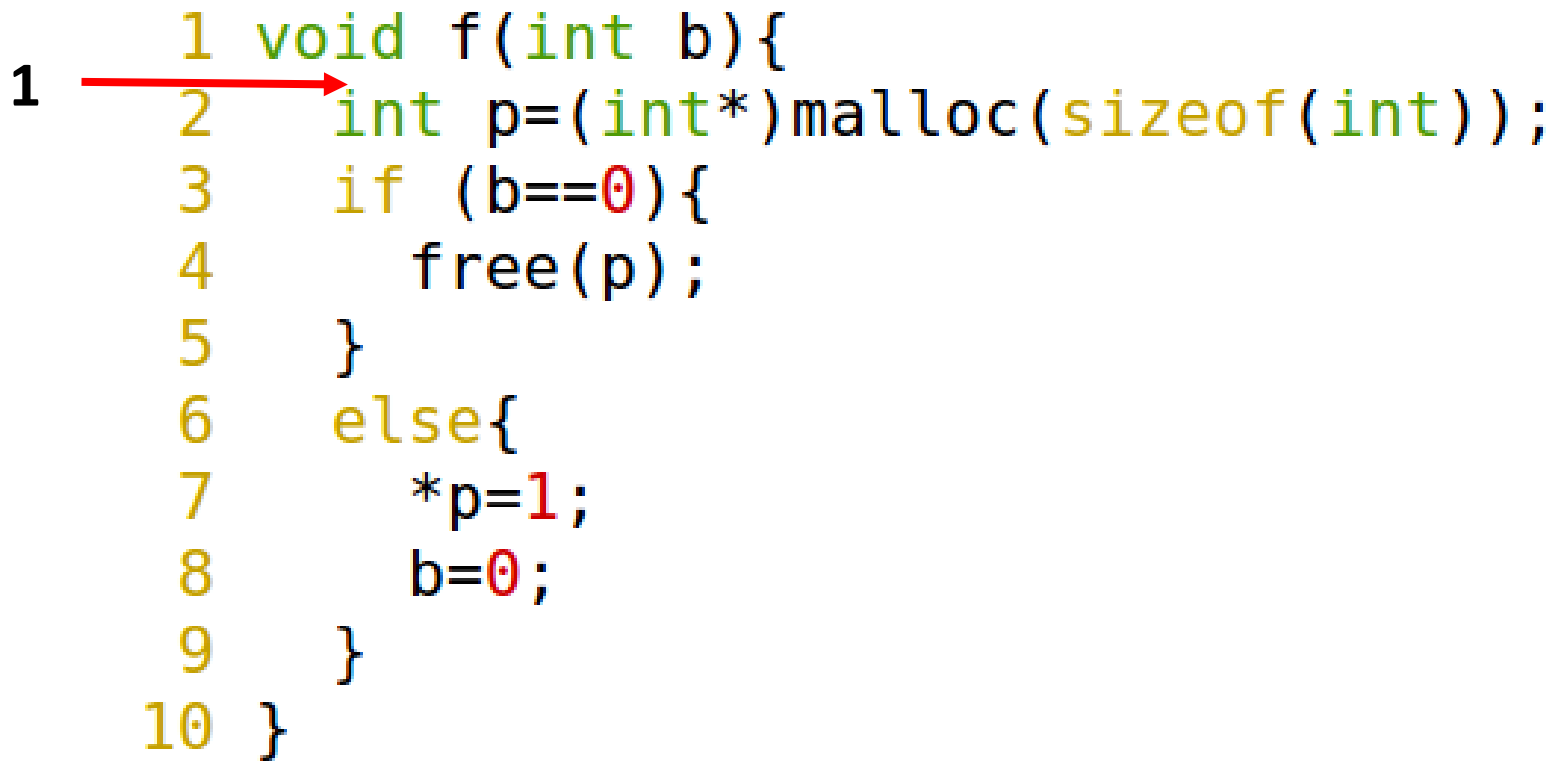
An Example


malloc
free
use
1 void f(int b){
2 int p=(int*)malloc(sizeof(int));
3 if (b==0){
4 free(p);
5 }
6 else{
7 *p=1;
8 b=0;
9 }
10 }

An Example

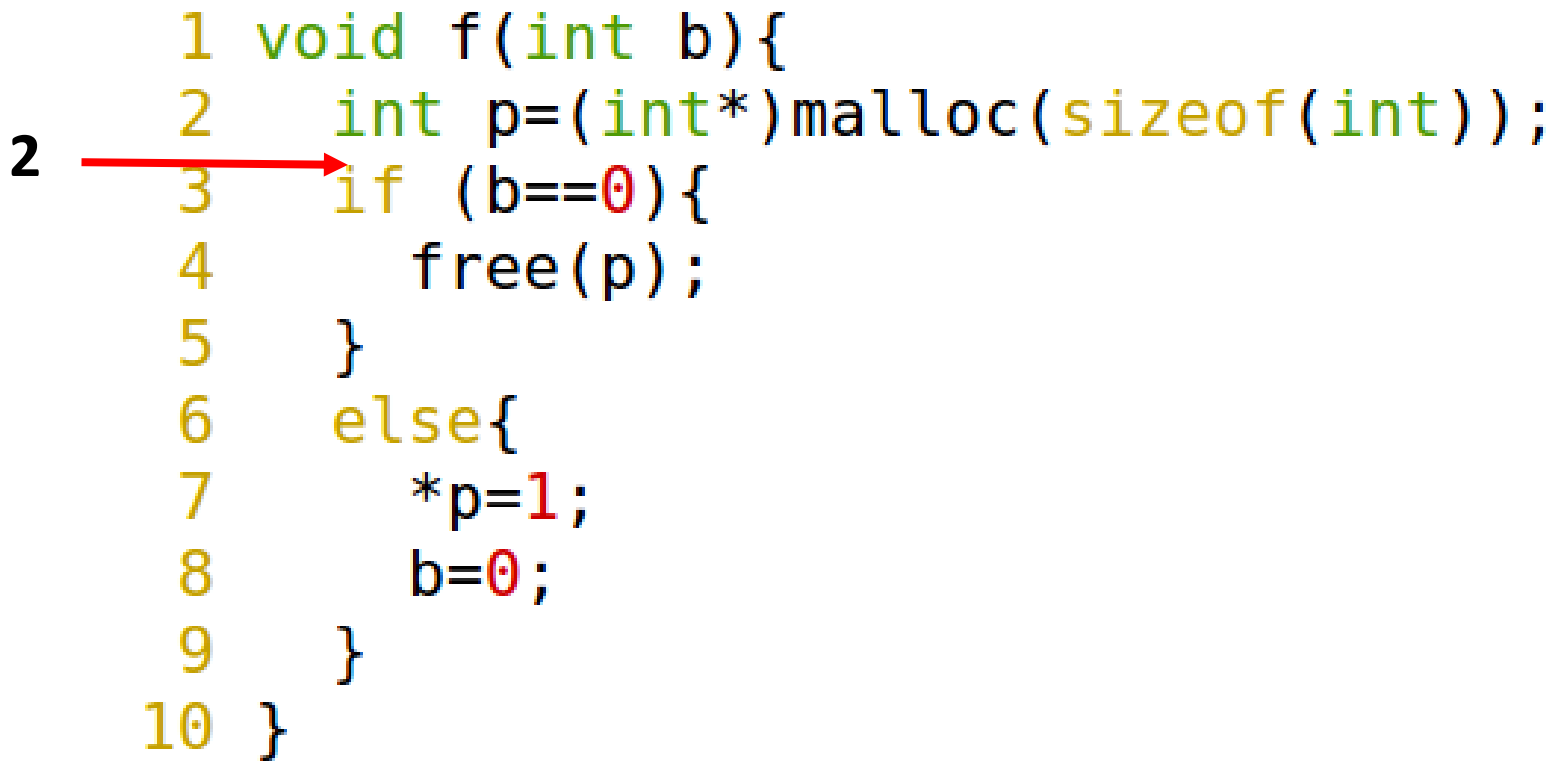

1
2
3
4
5
6
7
8
1 void f(int b){
2 int p=(int*)malloc(sizeof(int));
3 if (b==0){
4 free(p);
5 }
6 else{
7 *p=1;
8 b=0;
9 }
10 }

An Example



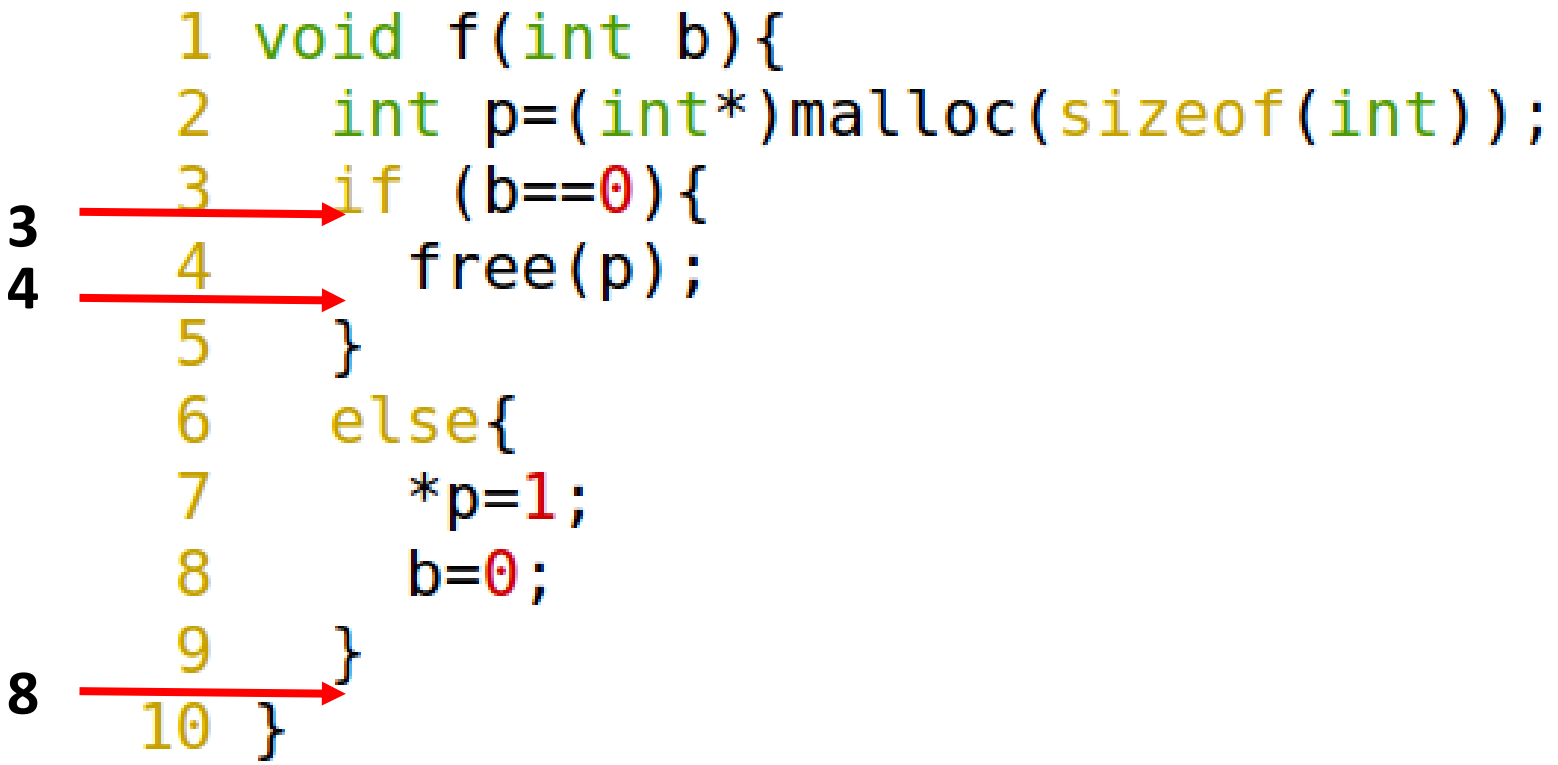
1: no allocation, no expression to use in *free()*

An Example



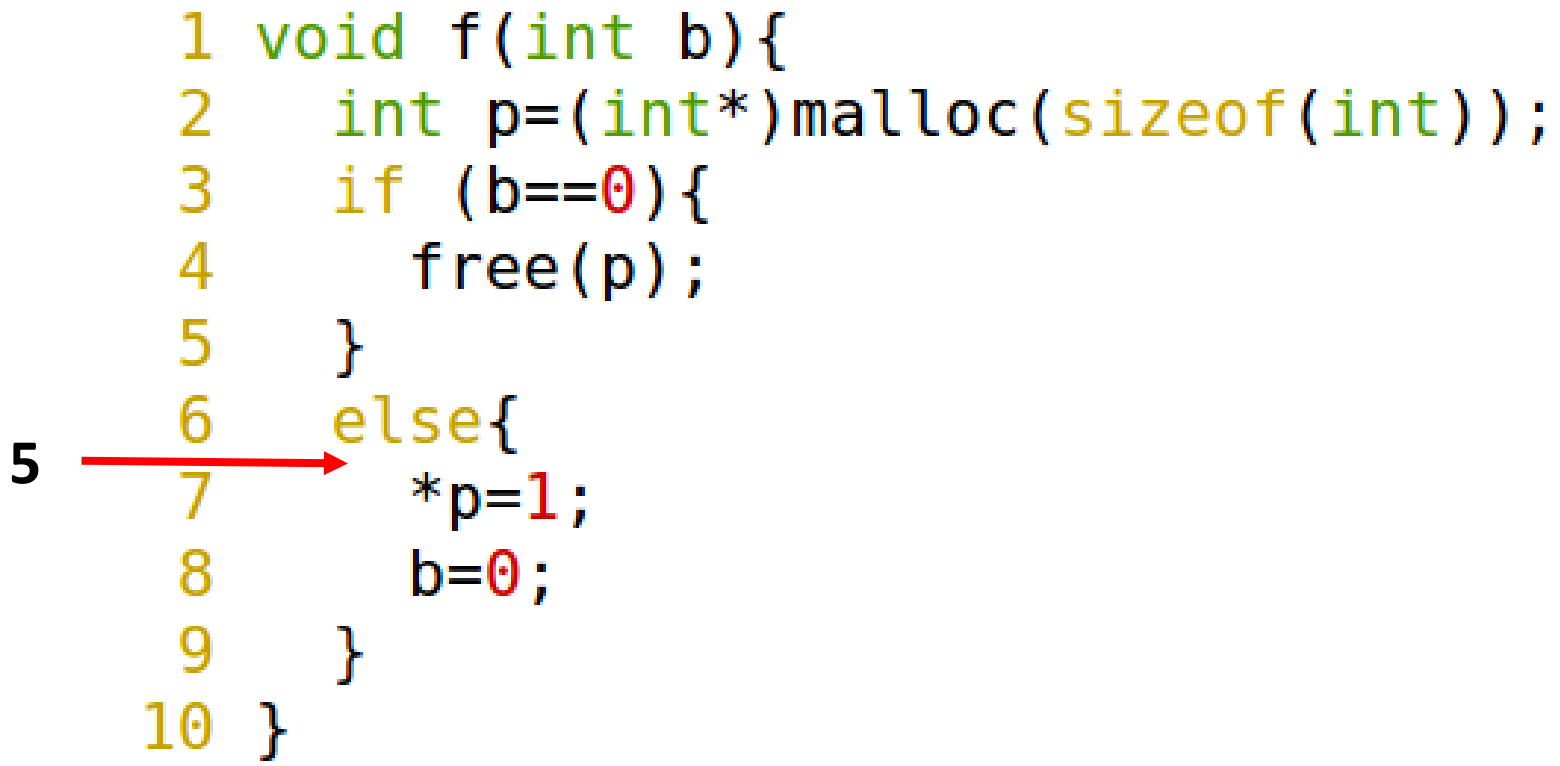
2: double free, use after free

An Example



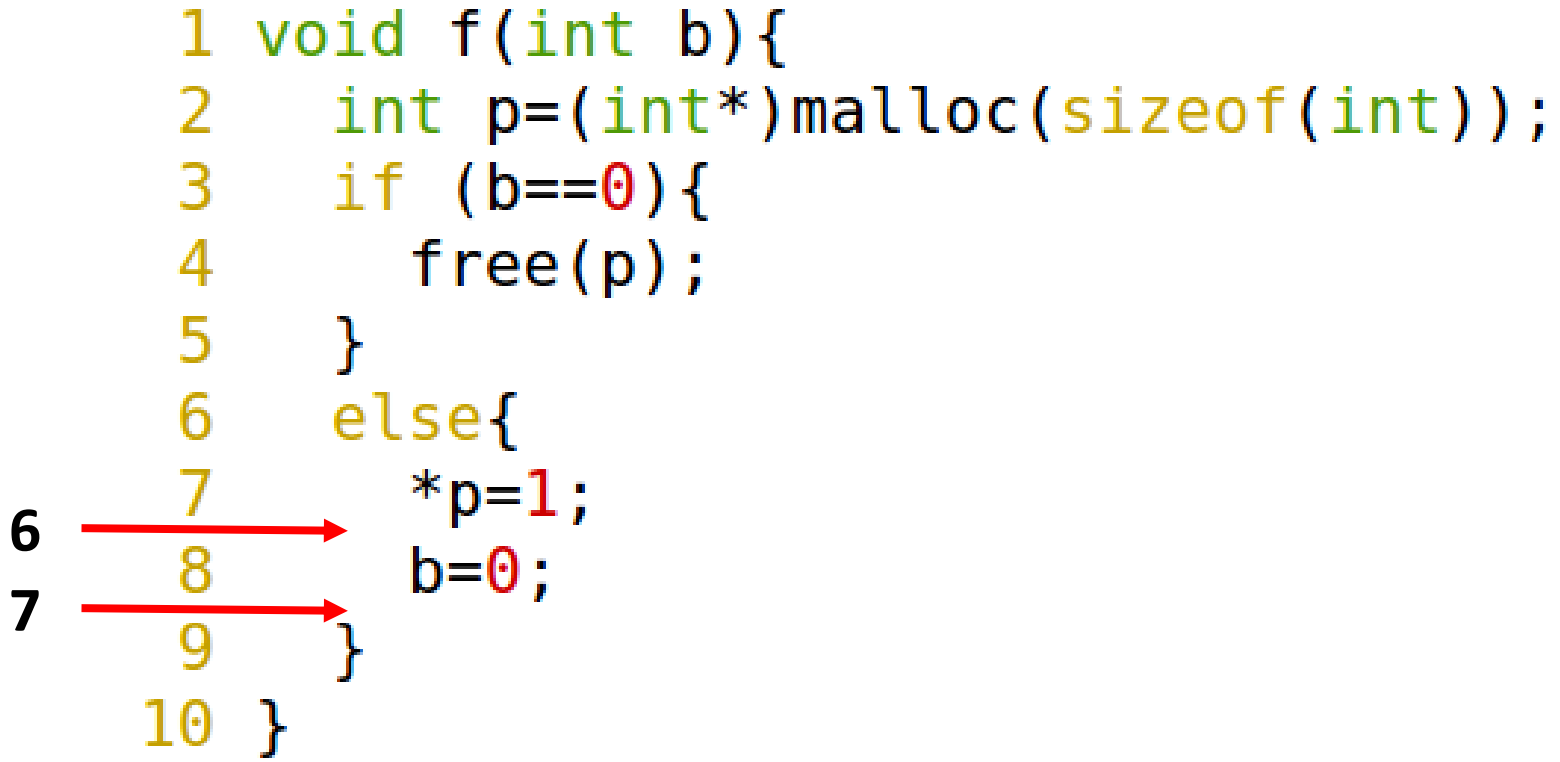
3,4,8: double free

An Example



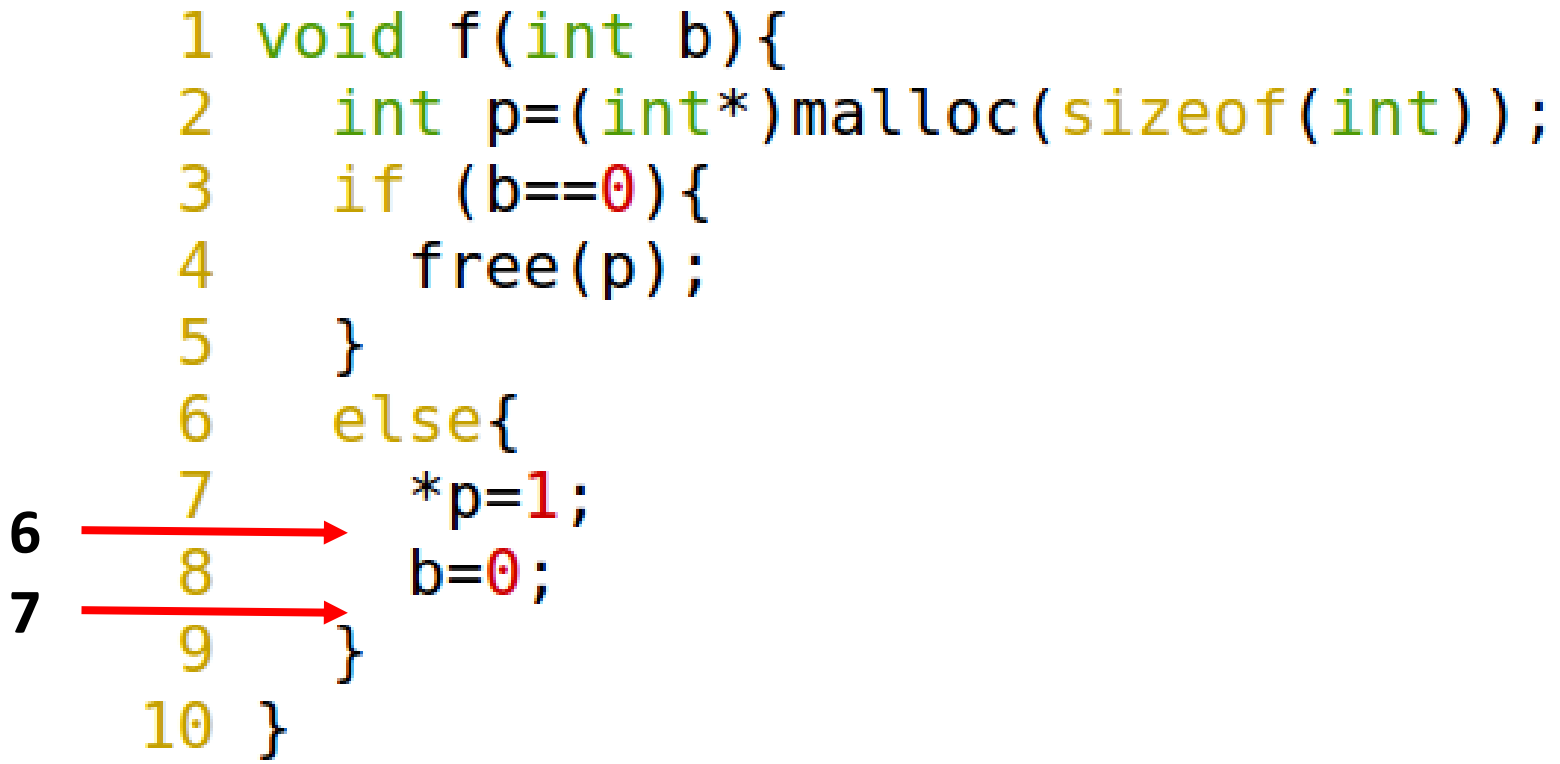
5: use after free

An Example



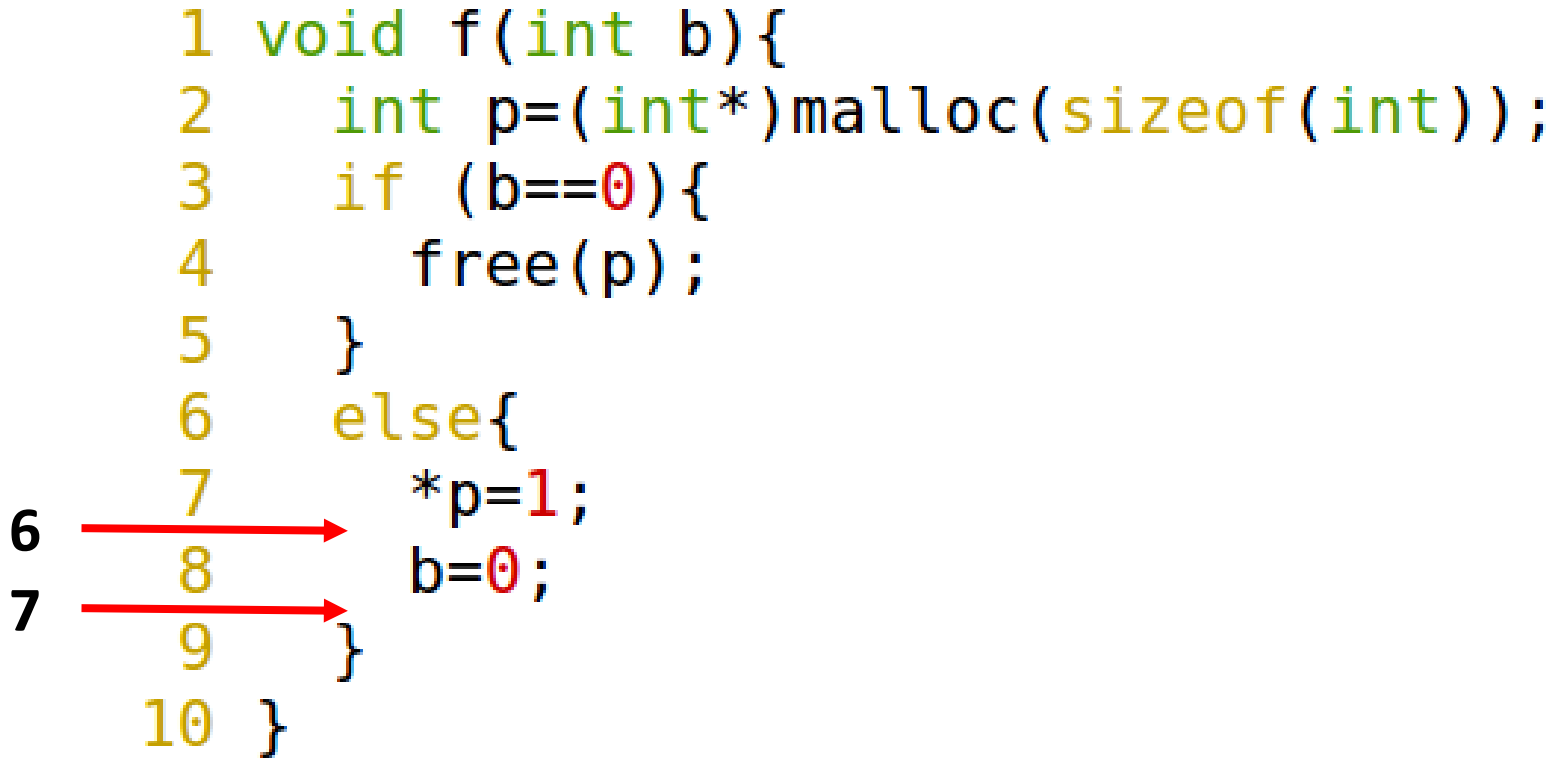
6,7: safe fix locations

An Example



6,7: both have pointers to the allocated memory object

An Example



6 is earlier than 7

An Example

- Fix statement at line 6: free(p);

```
1  void f(int b){
2    int p=(int*)malloc(sizeof(int));
3    if (b==0){
4      free(p);
5    }
6    else{
7      *p=1;
8      b=0;
9    }
10 }
```

6 → 7

6 is earlier than 7

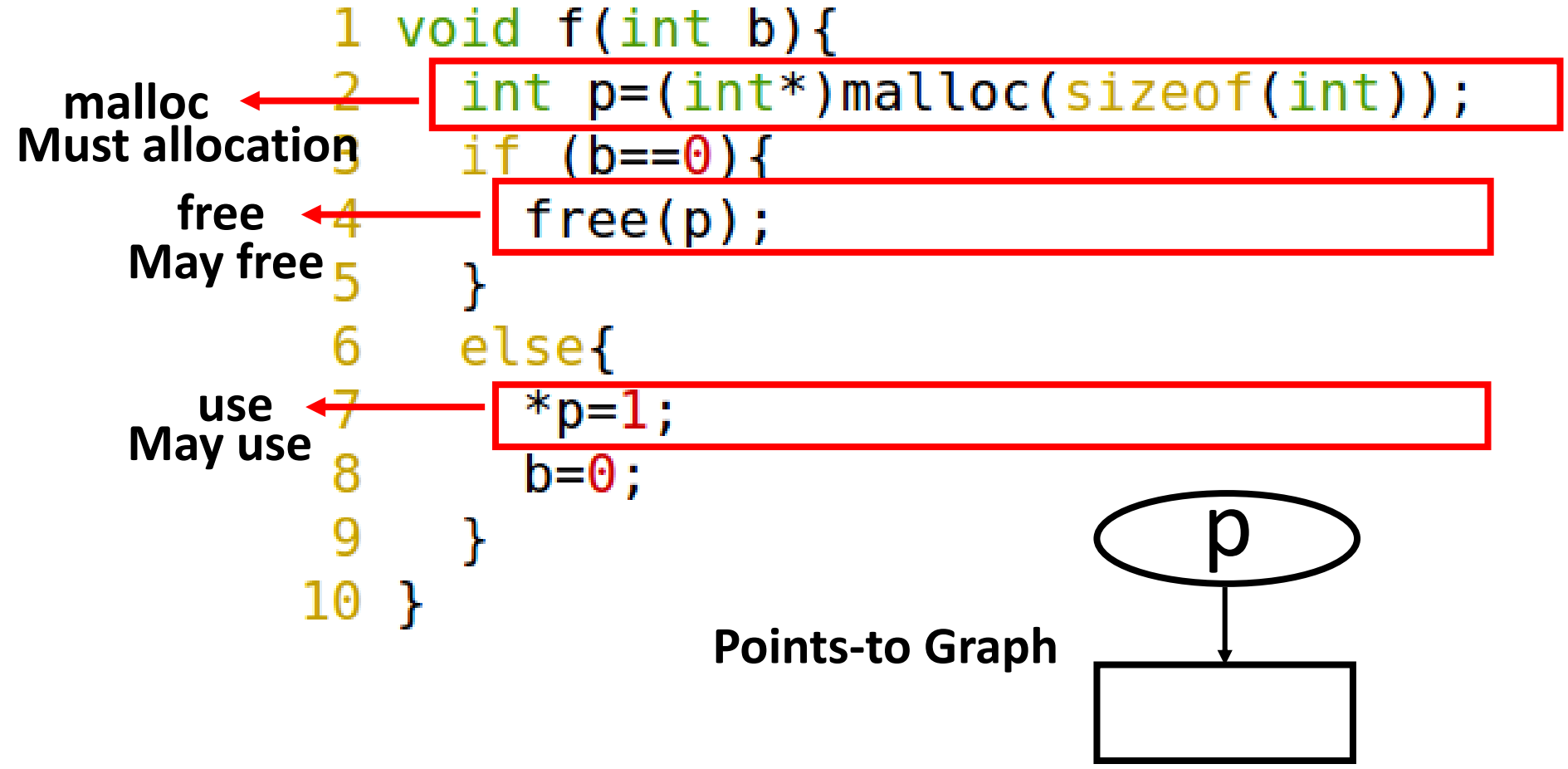
A safe fix

- The fix is after an allocation
- There is no double free
- There is no use after free
- There is an expression that always points to the allocation

A safe fix

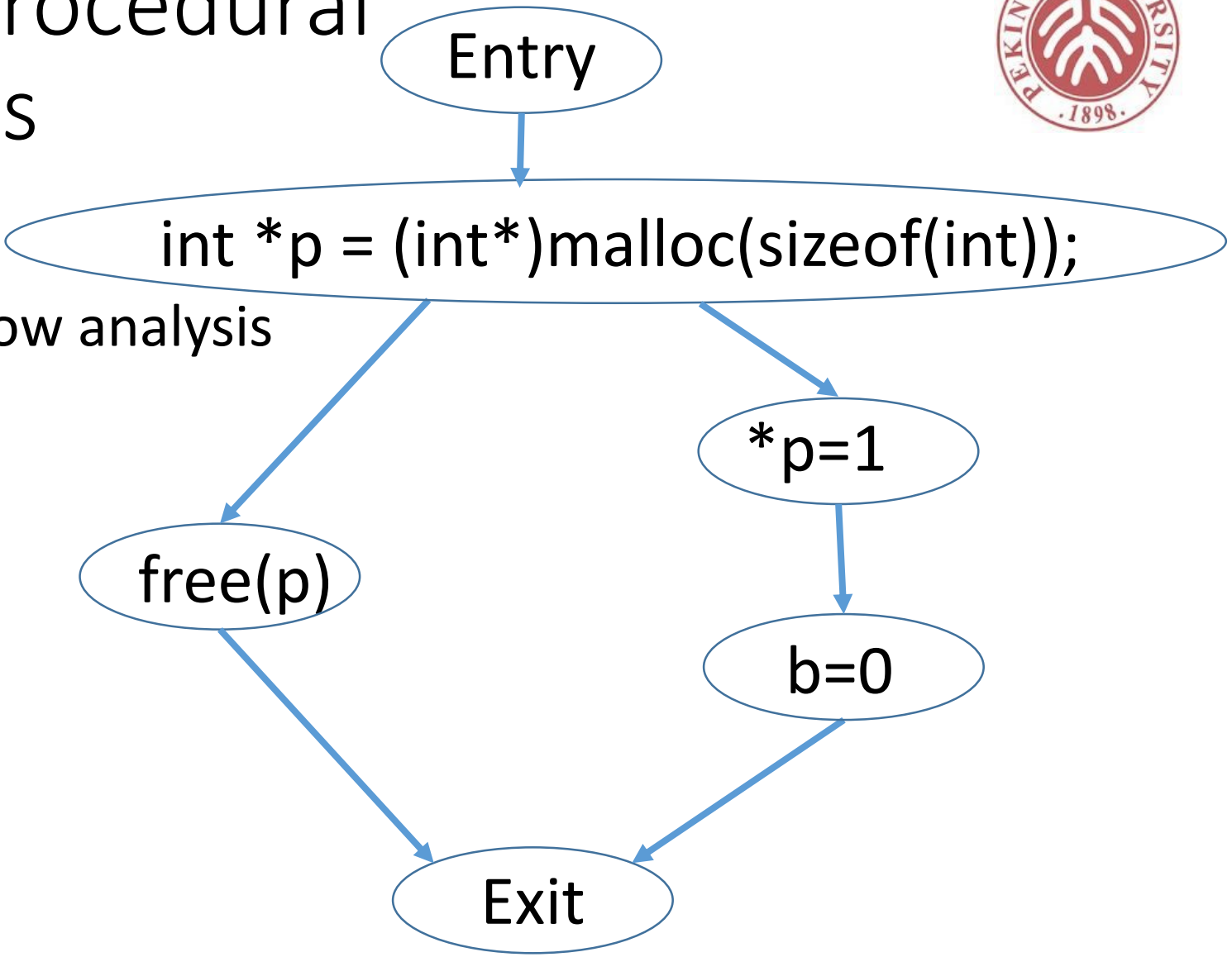
- The fix is after an allocation
- There is no double free
- There is no use after free
- There is an expression that always points to the allocation

Memory-related elements



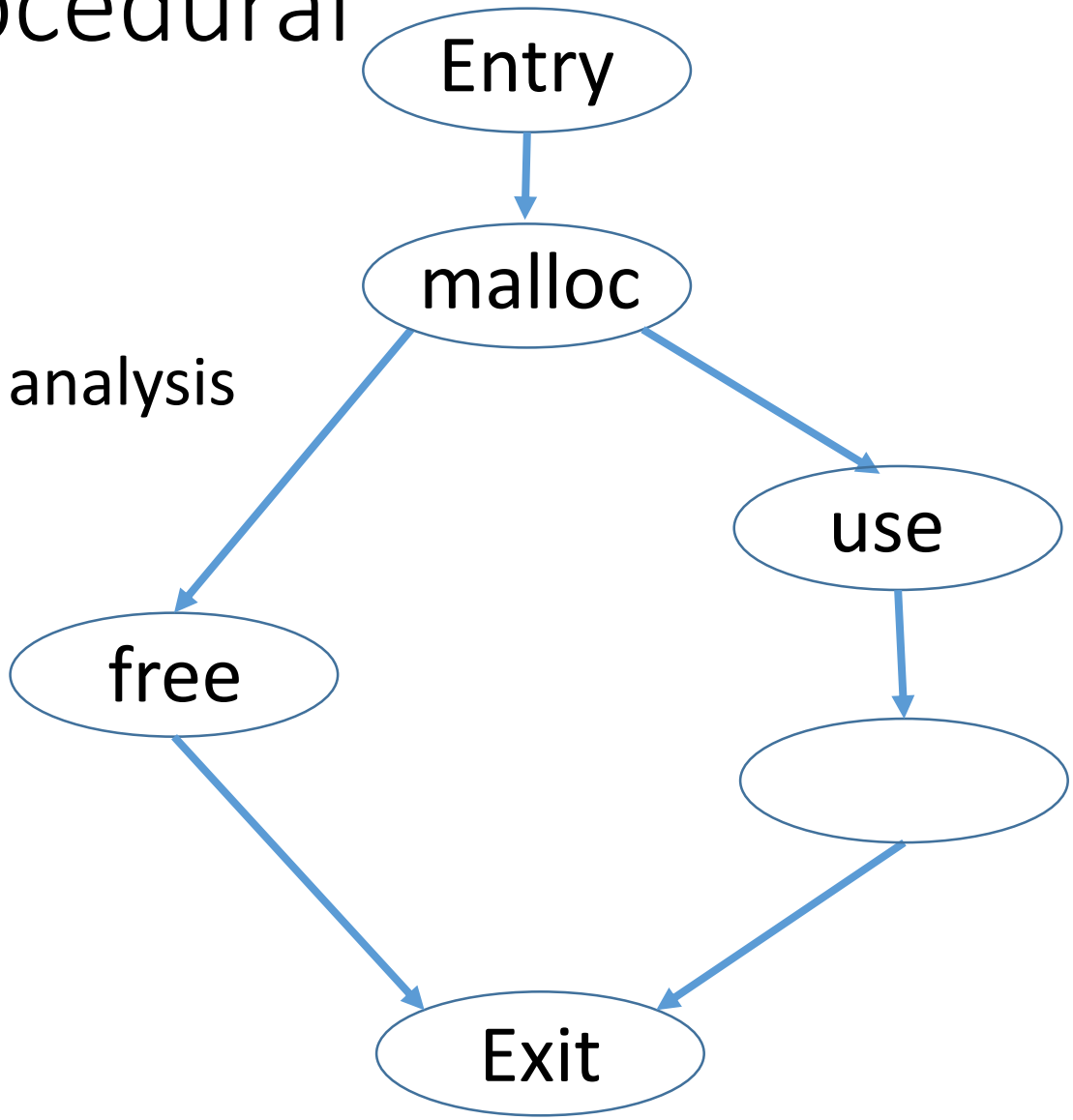
Intra-procedural analysis

- Data-flow analysis



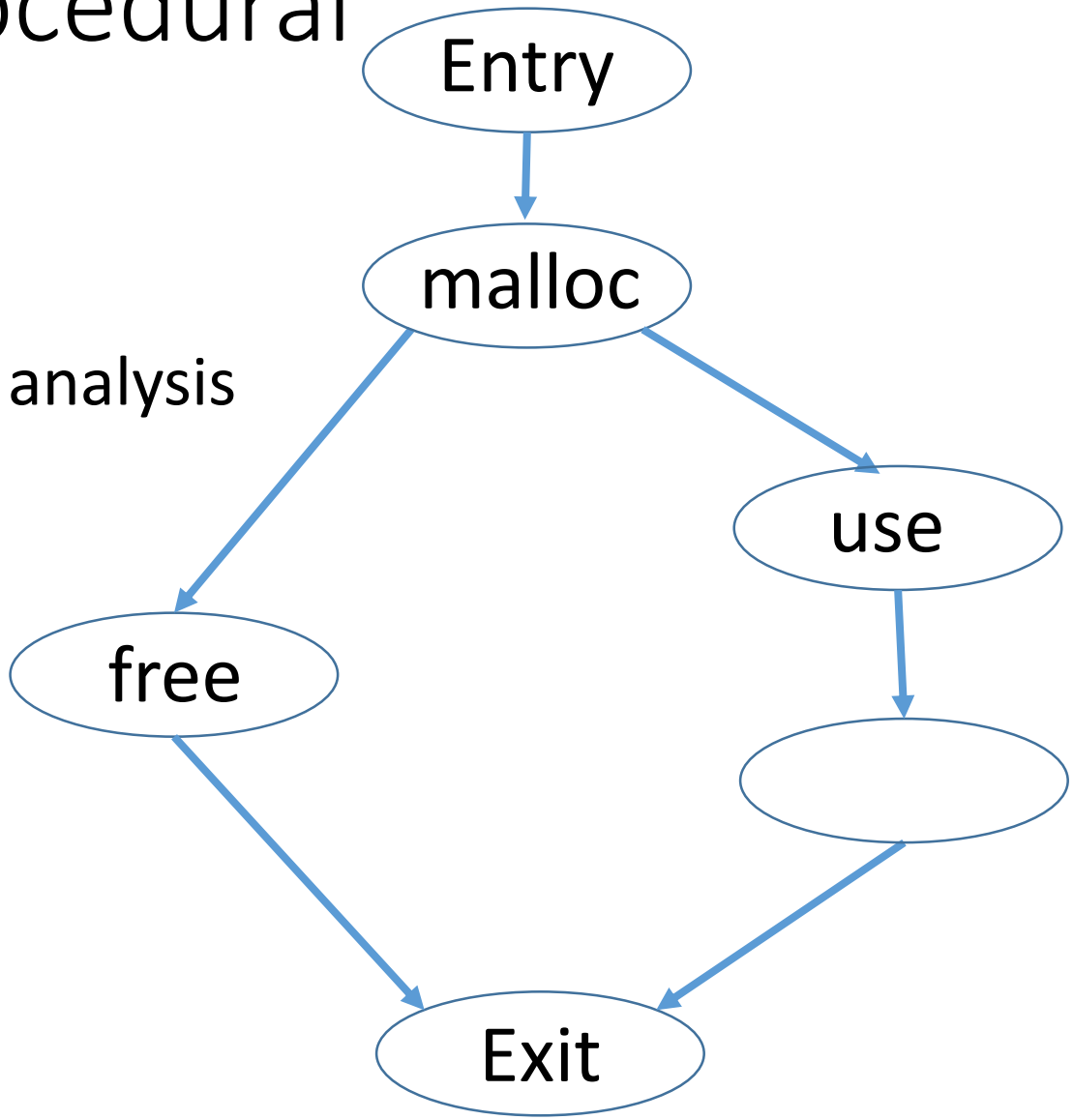
Intra-procedural analysis

- Data-flow analysis



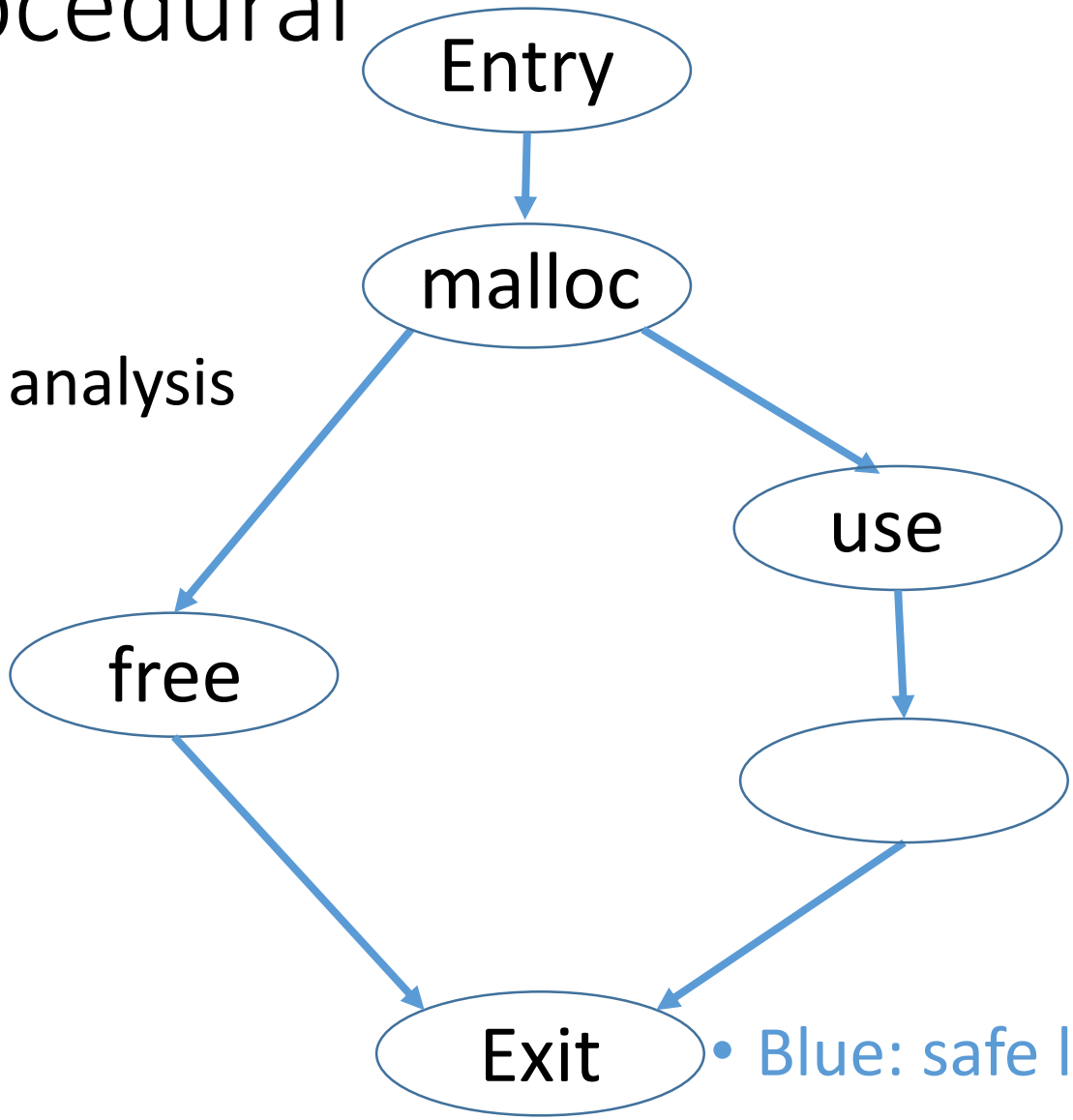
Intra-procedural analysis

- Data-flow analysis
- 4 steps



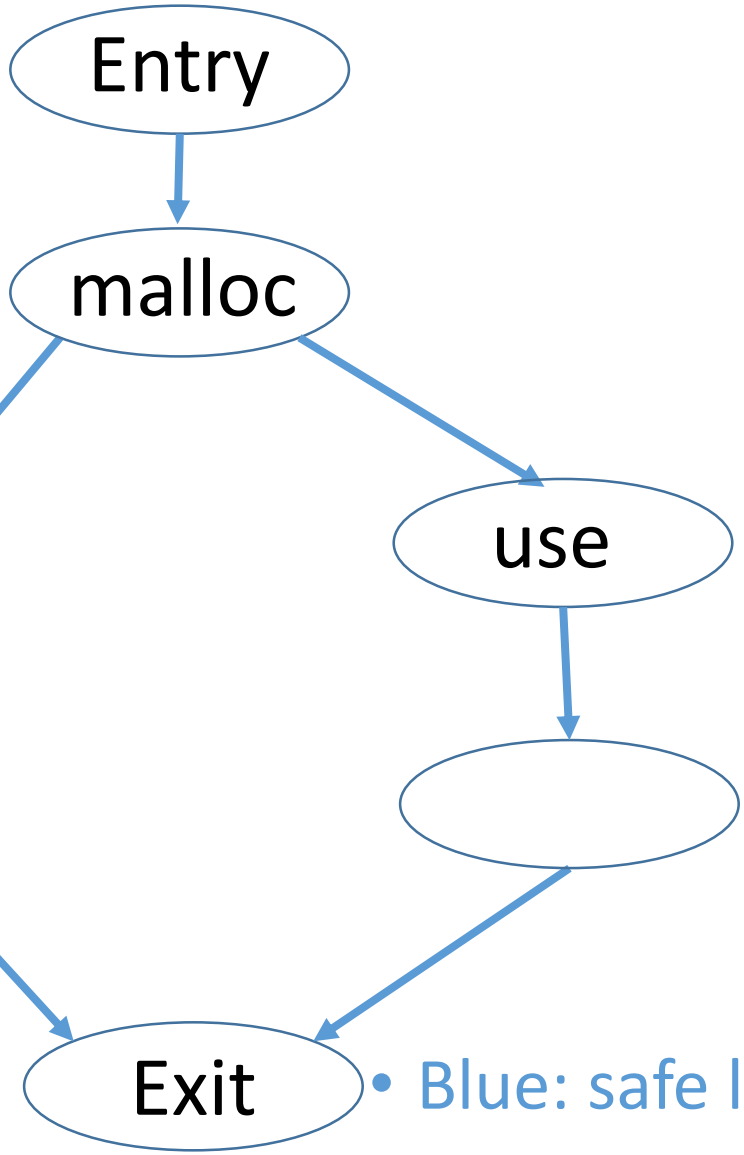
Intra-procedural analysis

- Data-flow analysis
- 4 steps



- Blue: safe location
- Red: unsafe location

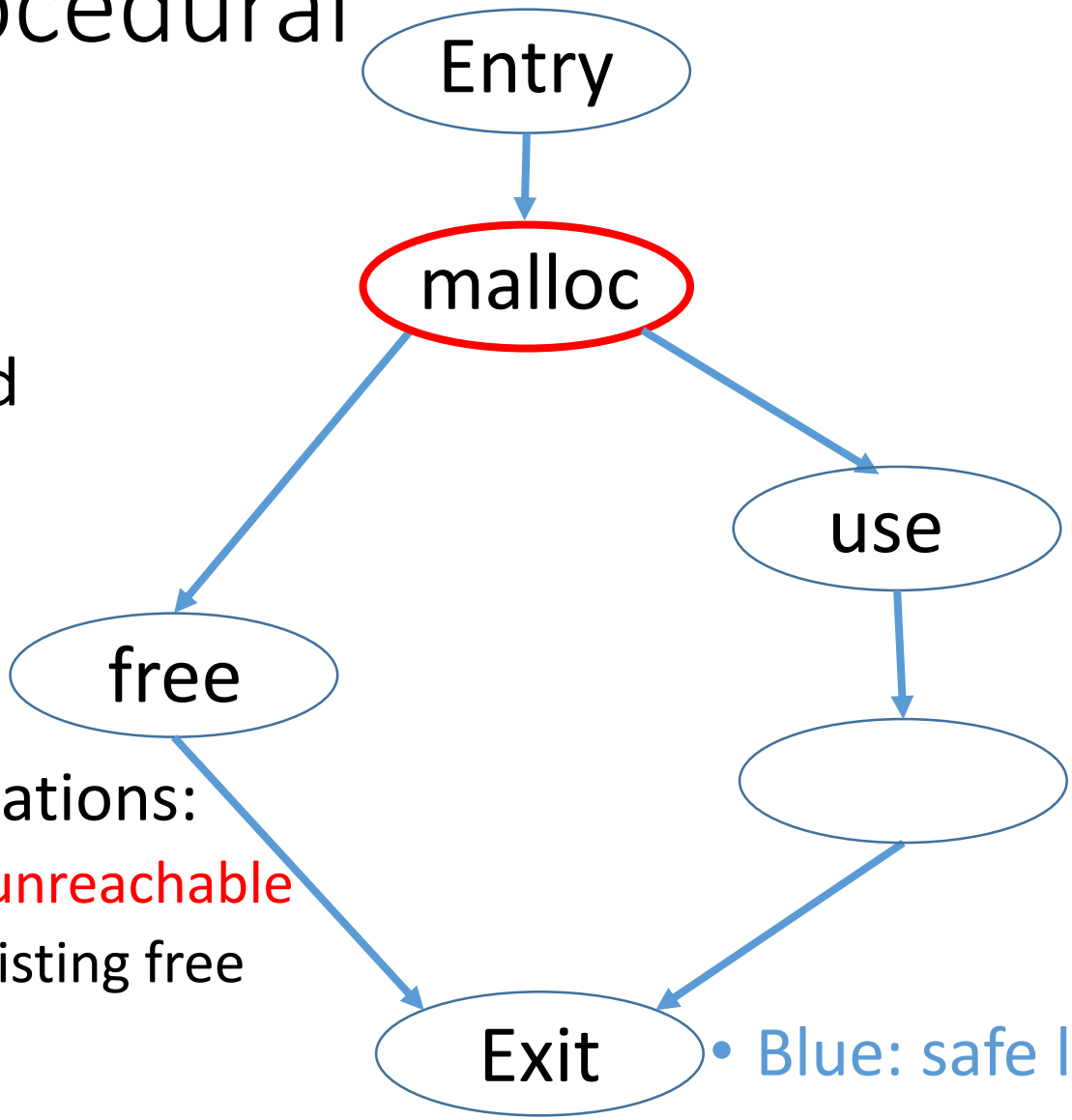
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free

- Blue: safe location
- Red: unsafe location

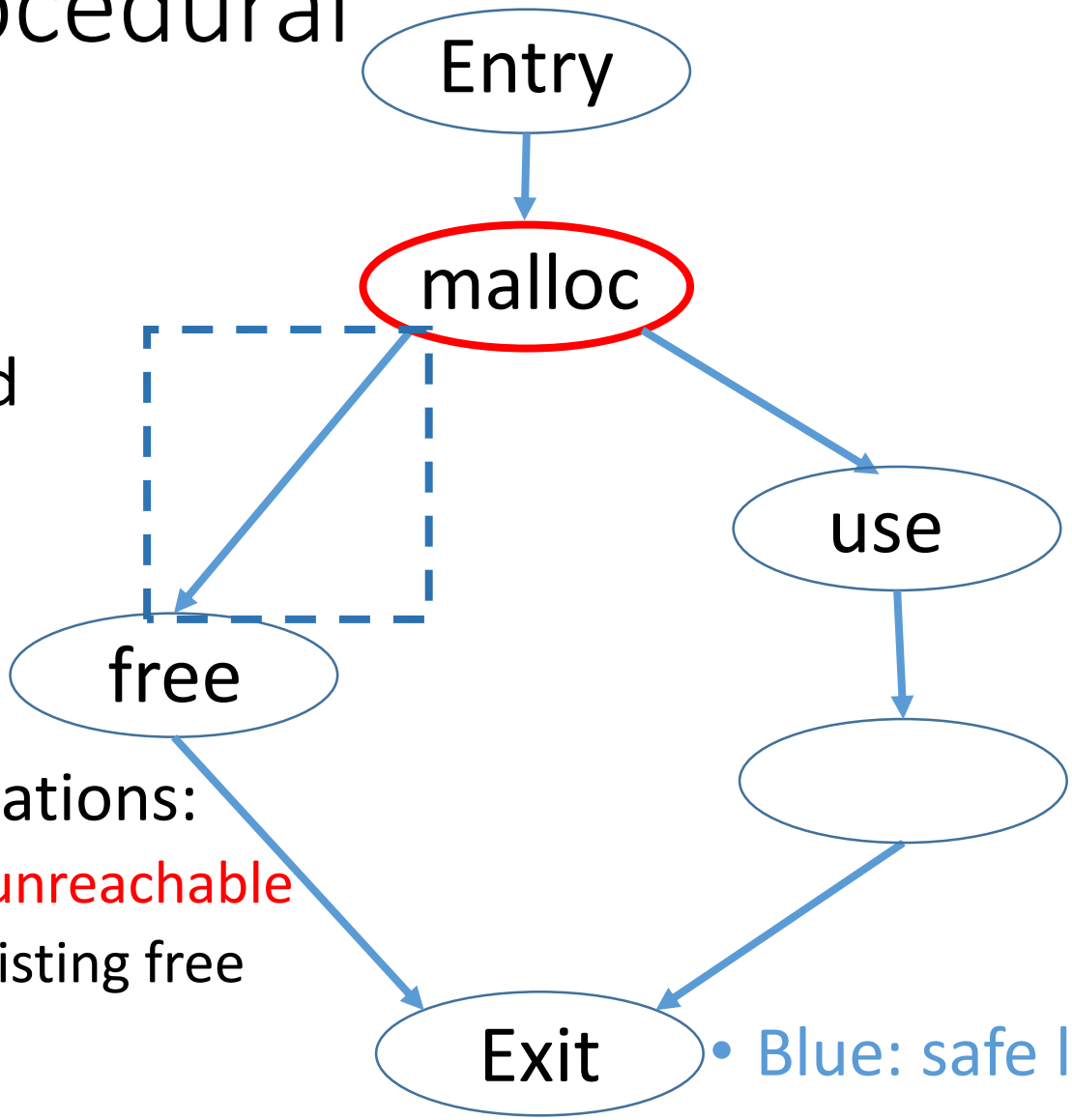
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

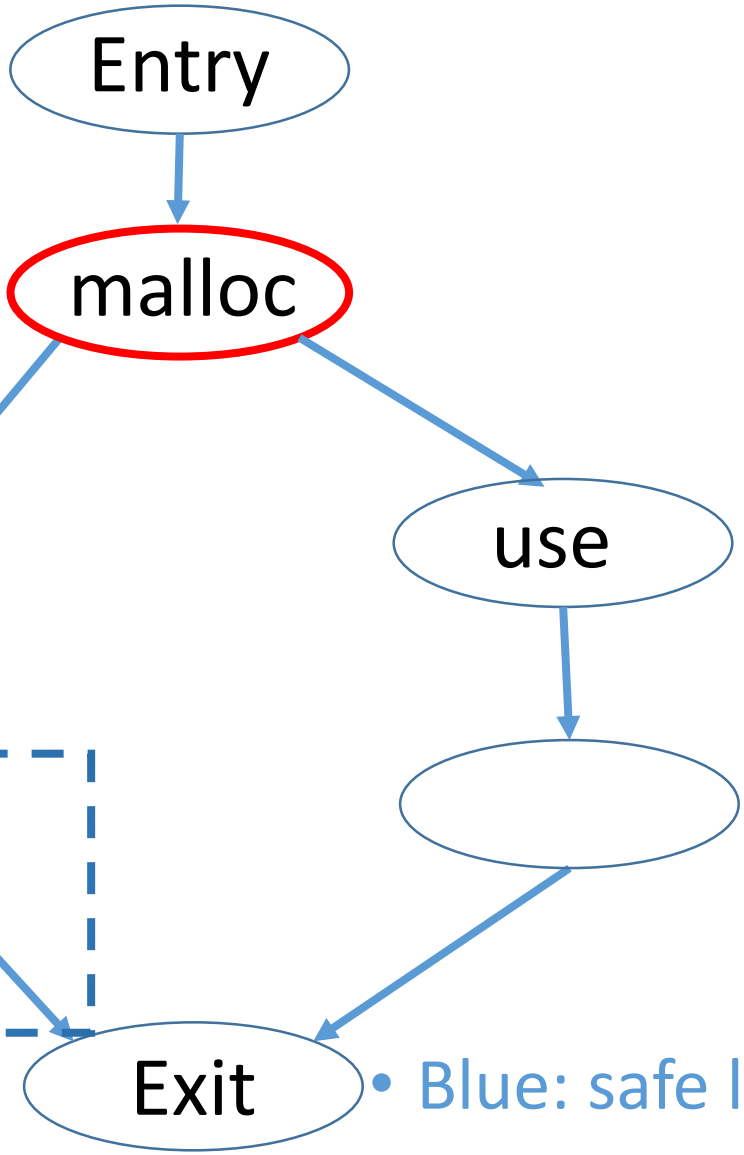
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

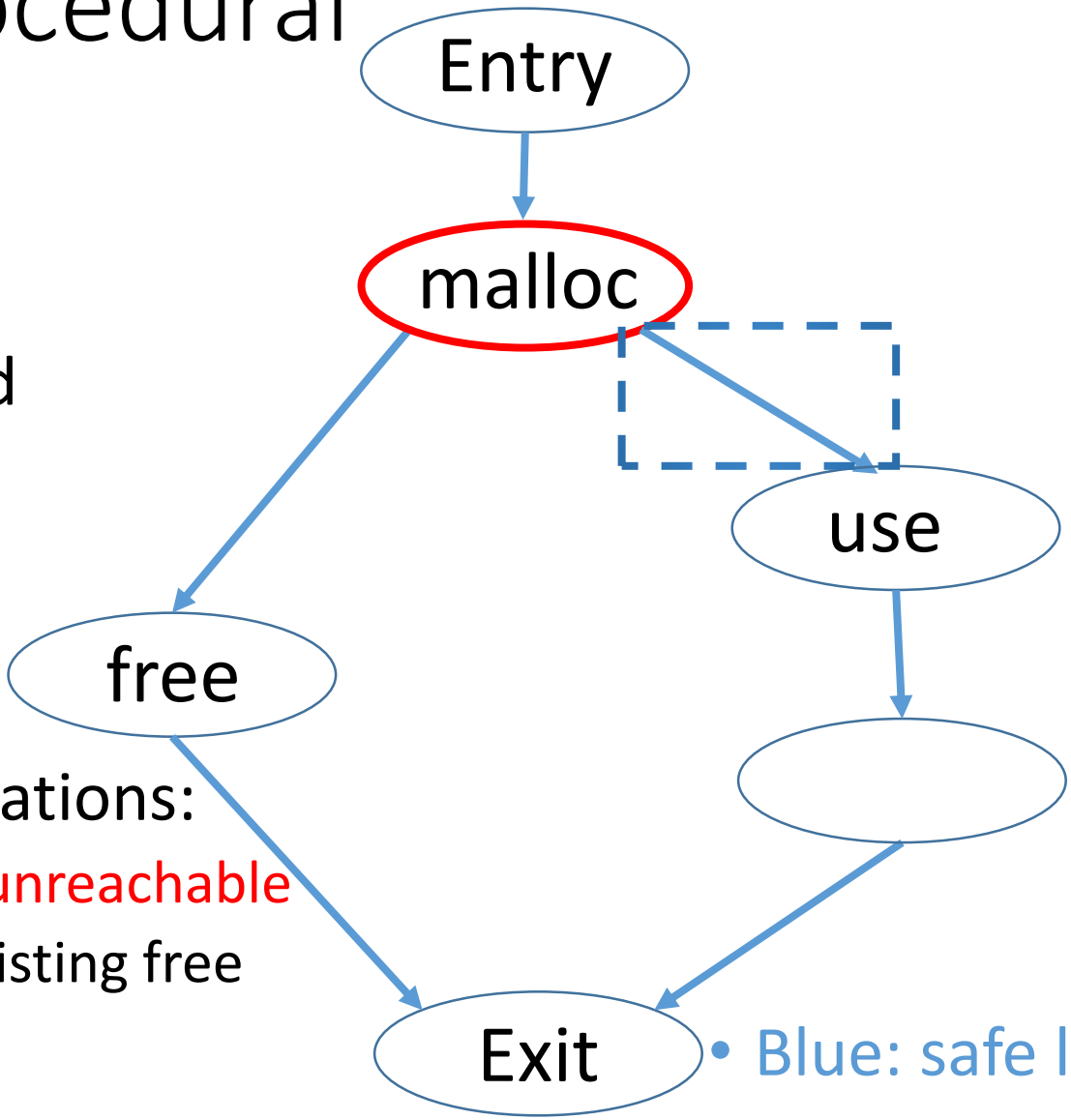
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

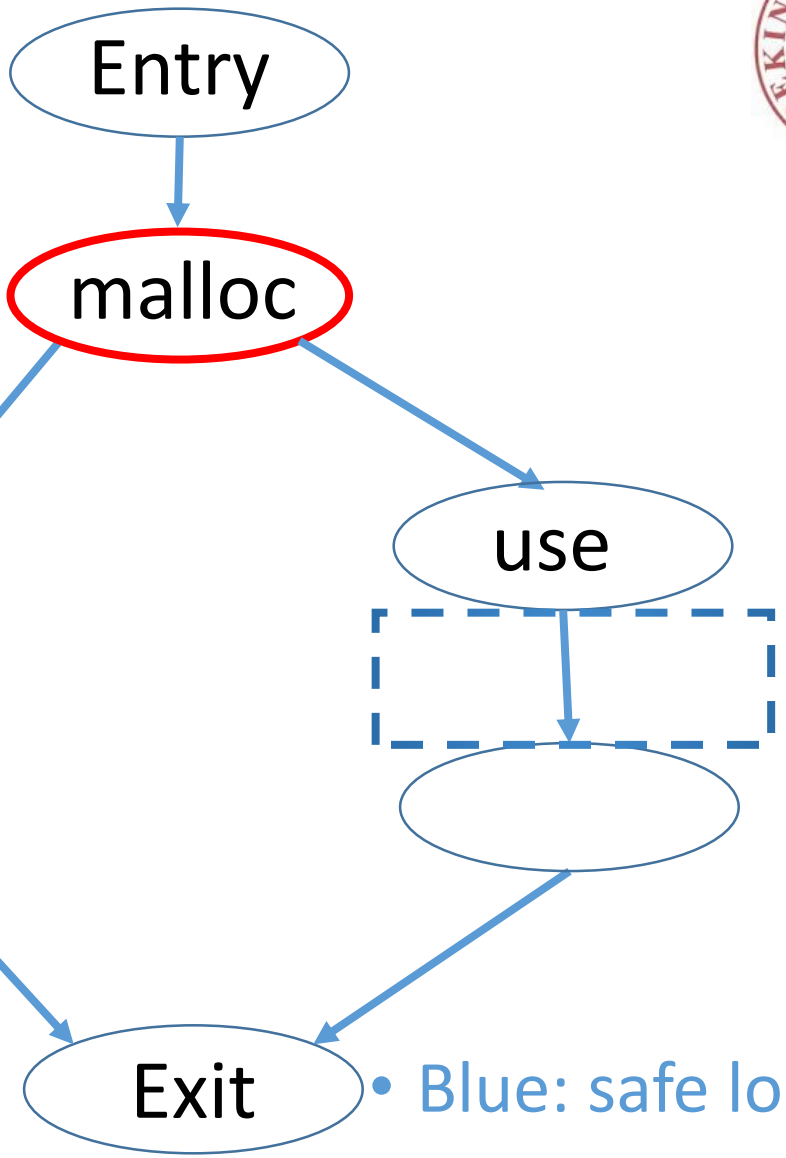
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

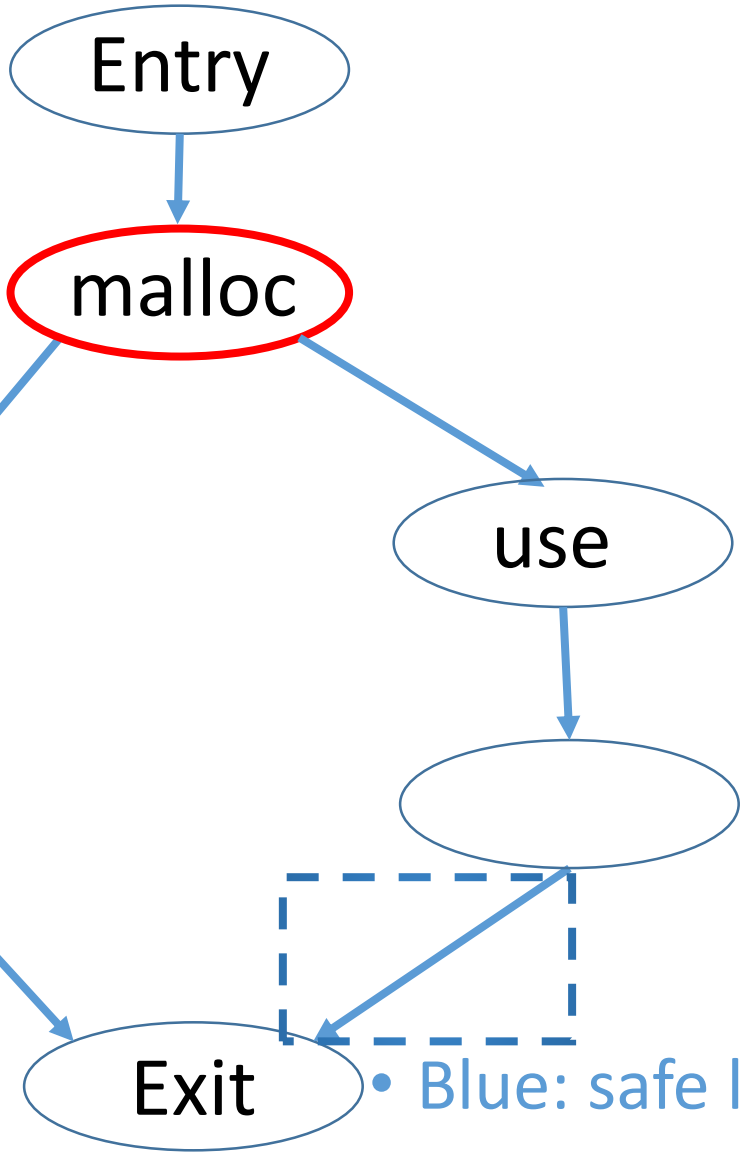
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

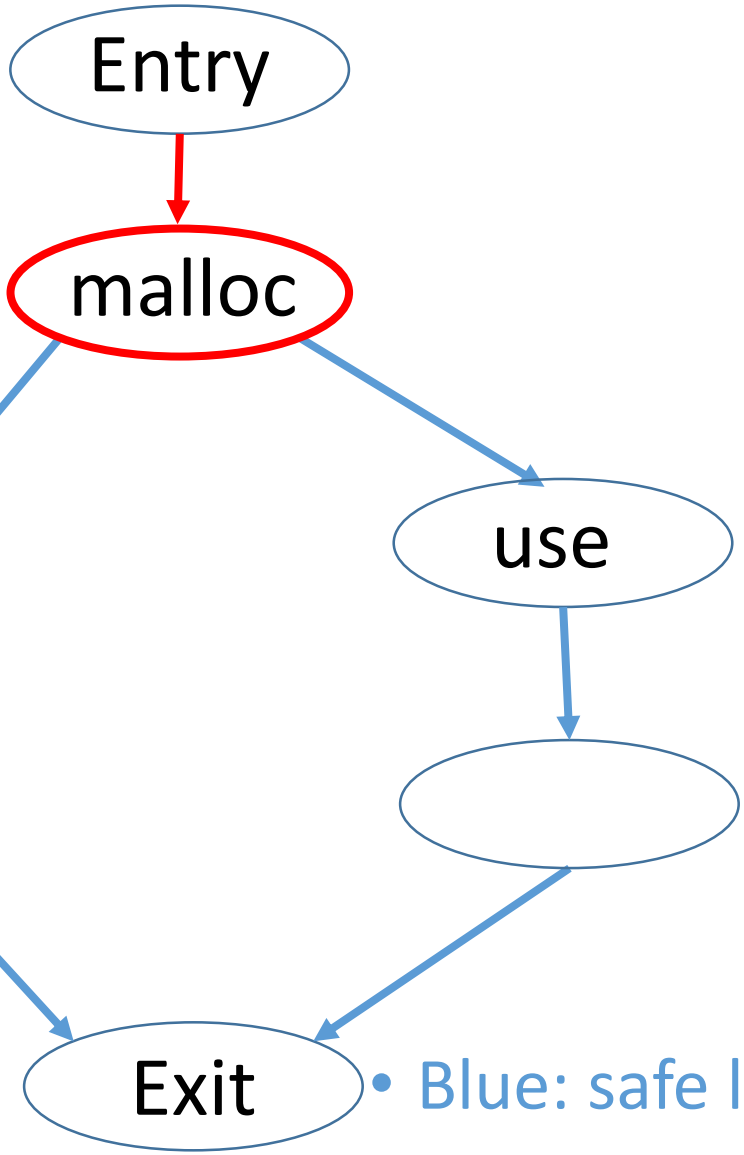
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

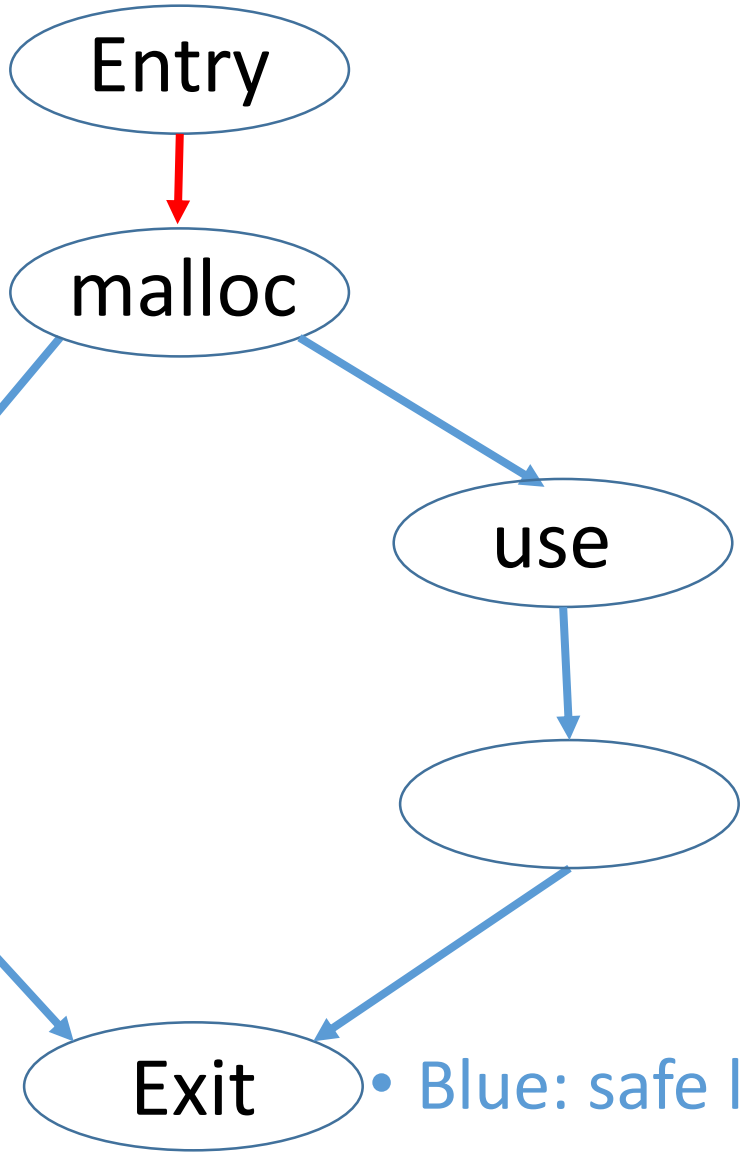
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

The fix is after an allocation

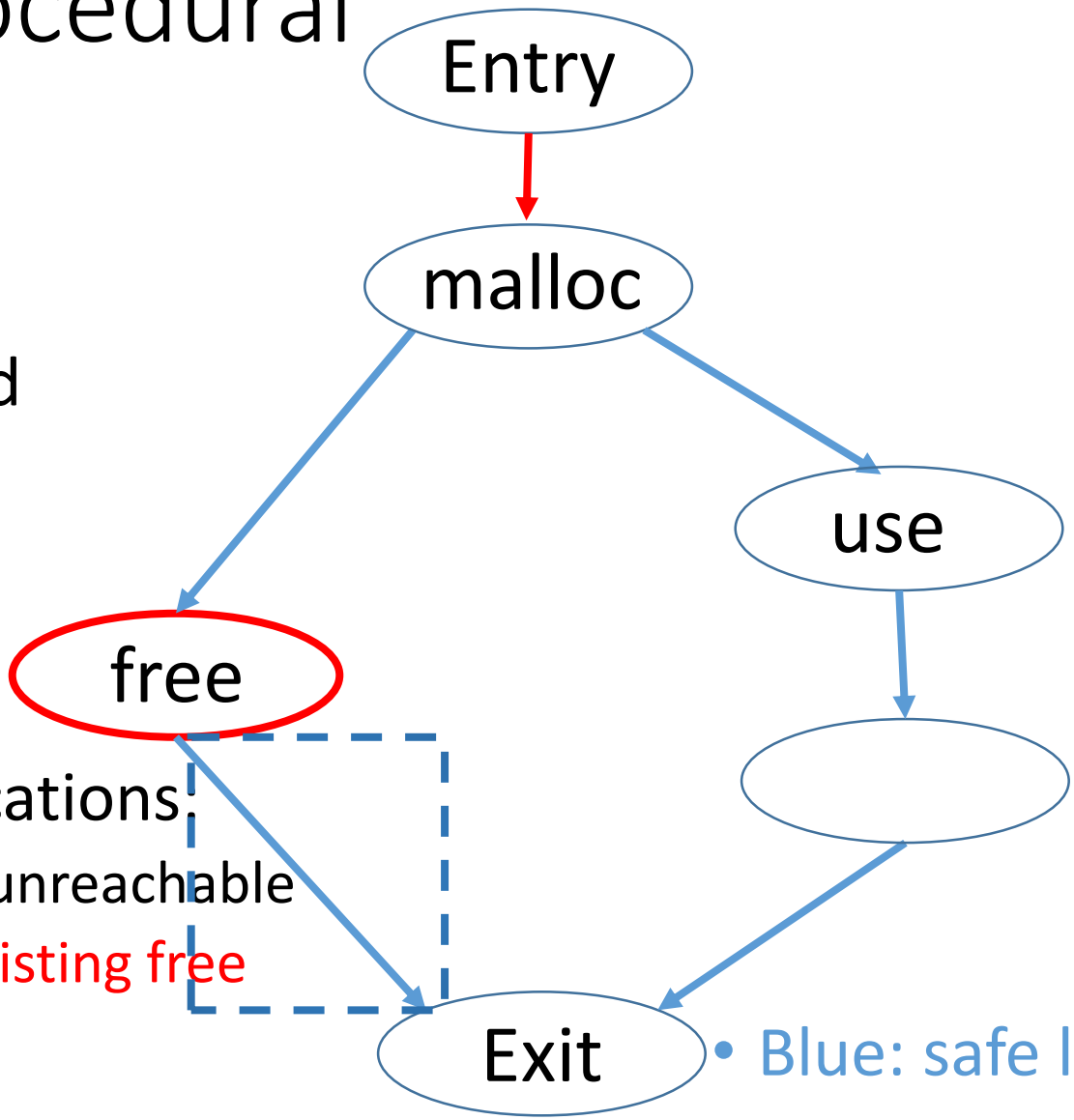
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

There is no double free (forward)

Intra-procedural analysis

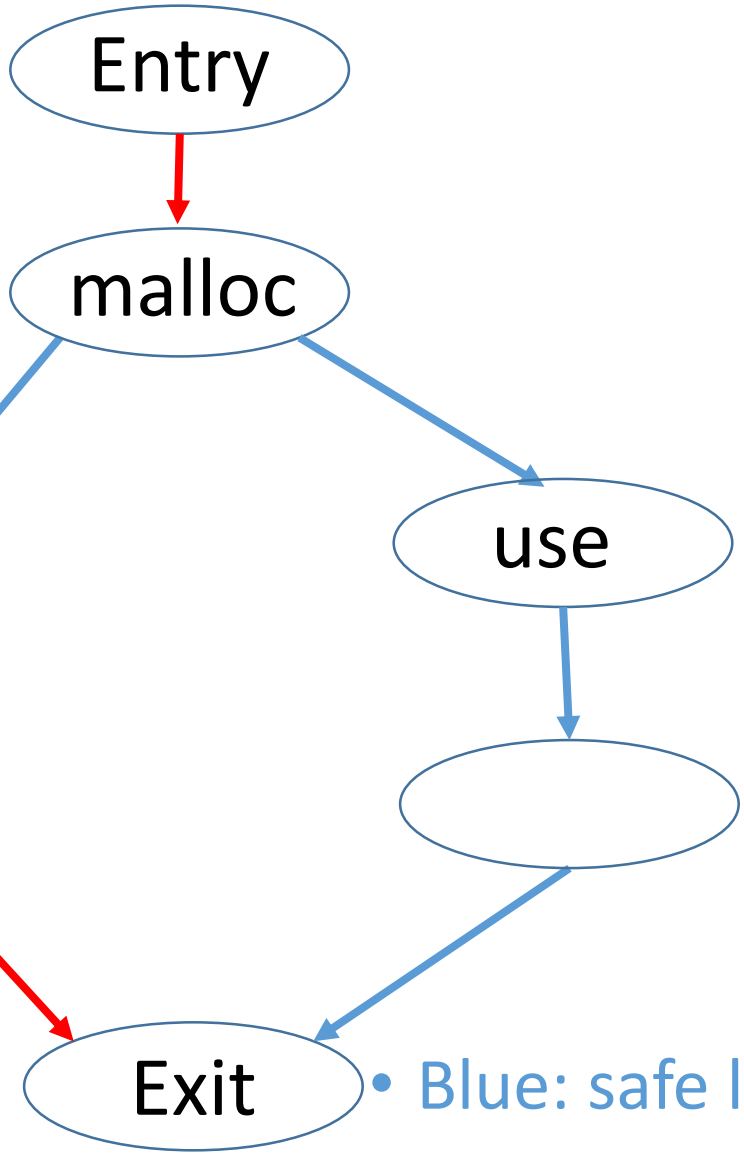


- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free

- Blue: safe location
- Red: unsafe location

There is no double free (forward)

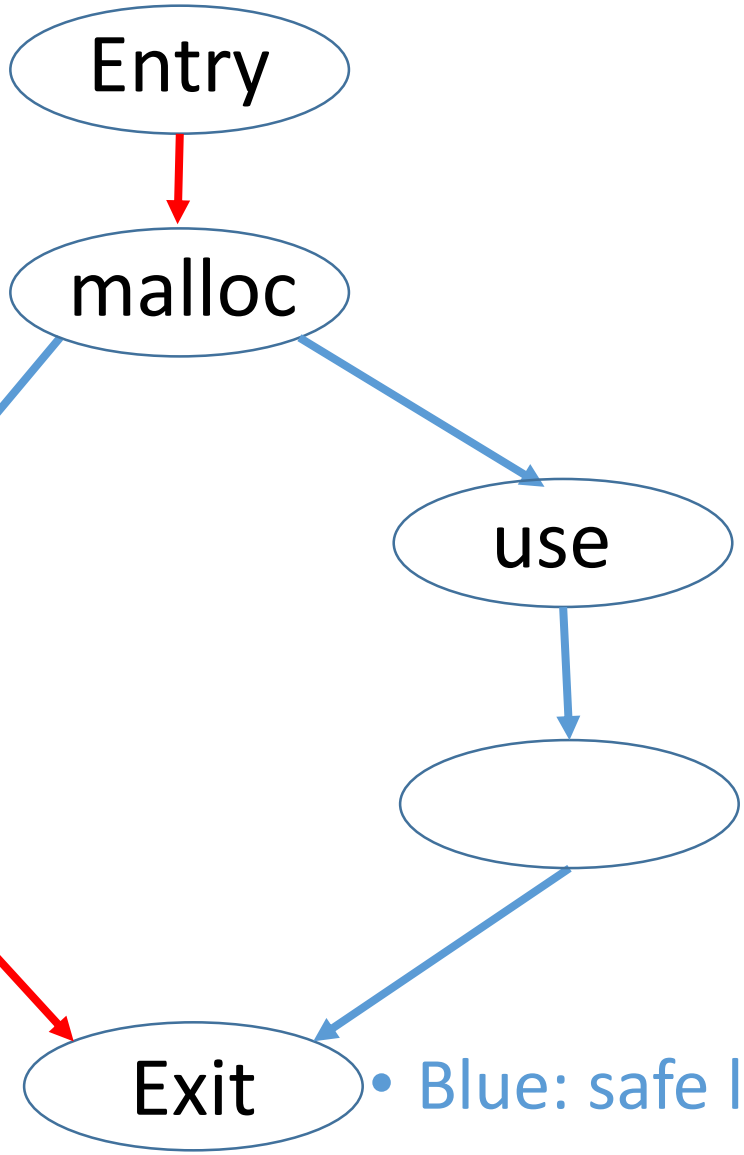
Intra-procedural analysis



- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free
- Blue: safe location
- Red: unsafe location

There is no double free (forward)

Intra-procedural analysis



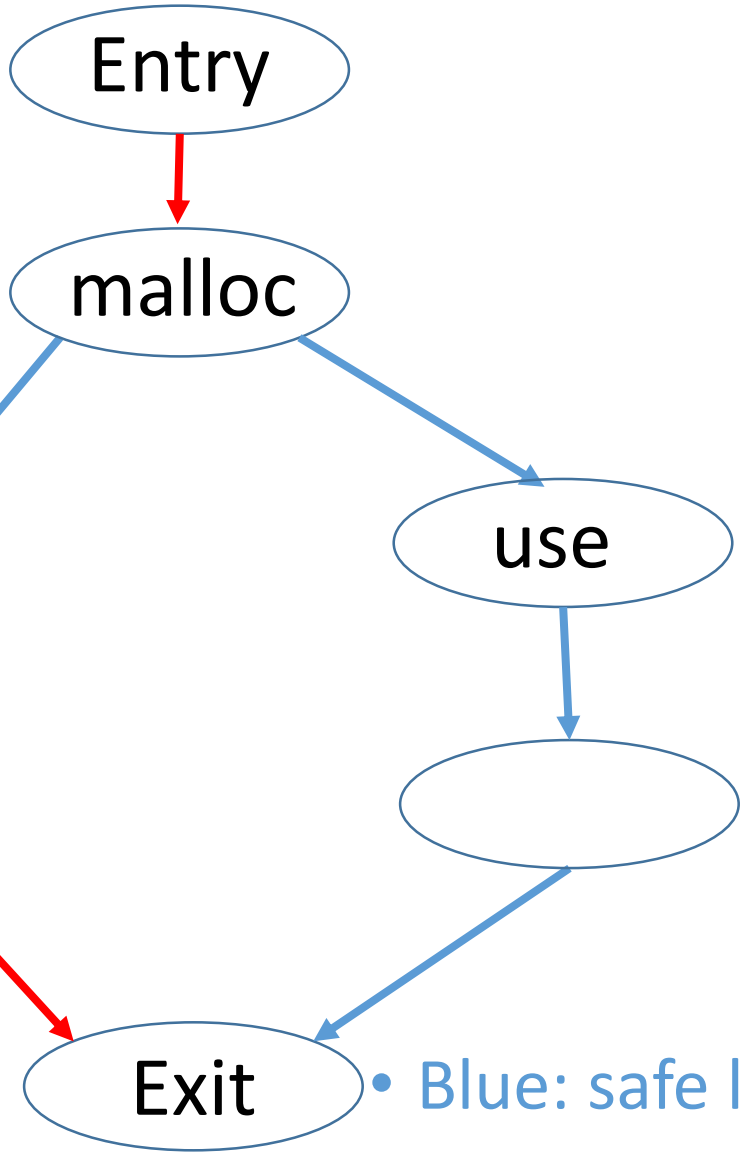
- 1. Forward
- unsafe locations:
  - Malloc unreachable
  - After existing free

The fix is after an allocation

There is no double free (forward)

- Blue: safe location
- Red: unsafe location

Intra-procedural analysis



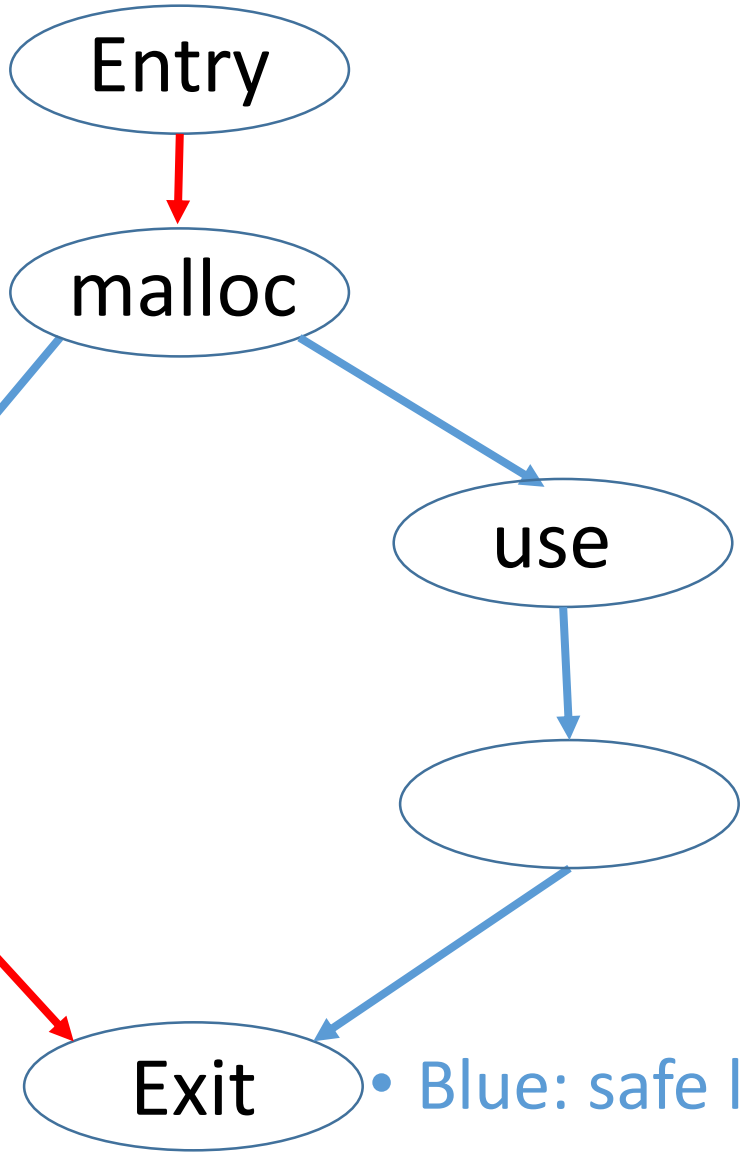
- 2. Backward

unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

Intra-procedural analysis



- 2. Backward

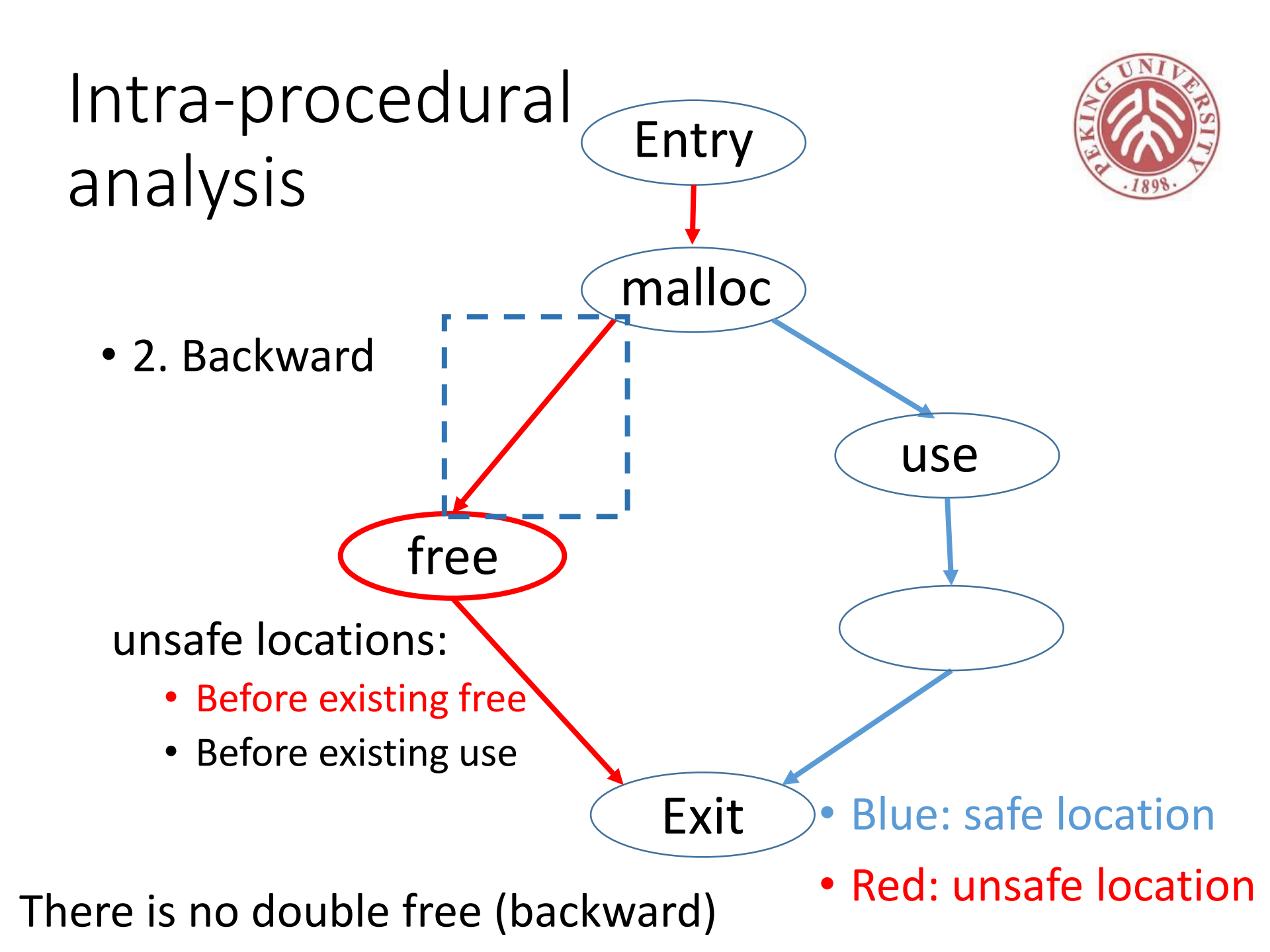
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no double free (backward)

Intra-procedural analysis



- 2. Backward

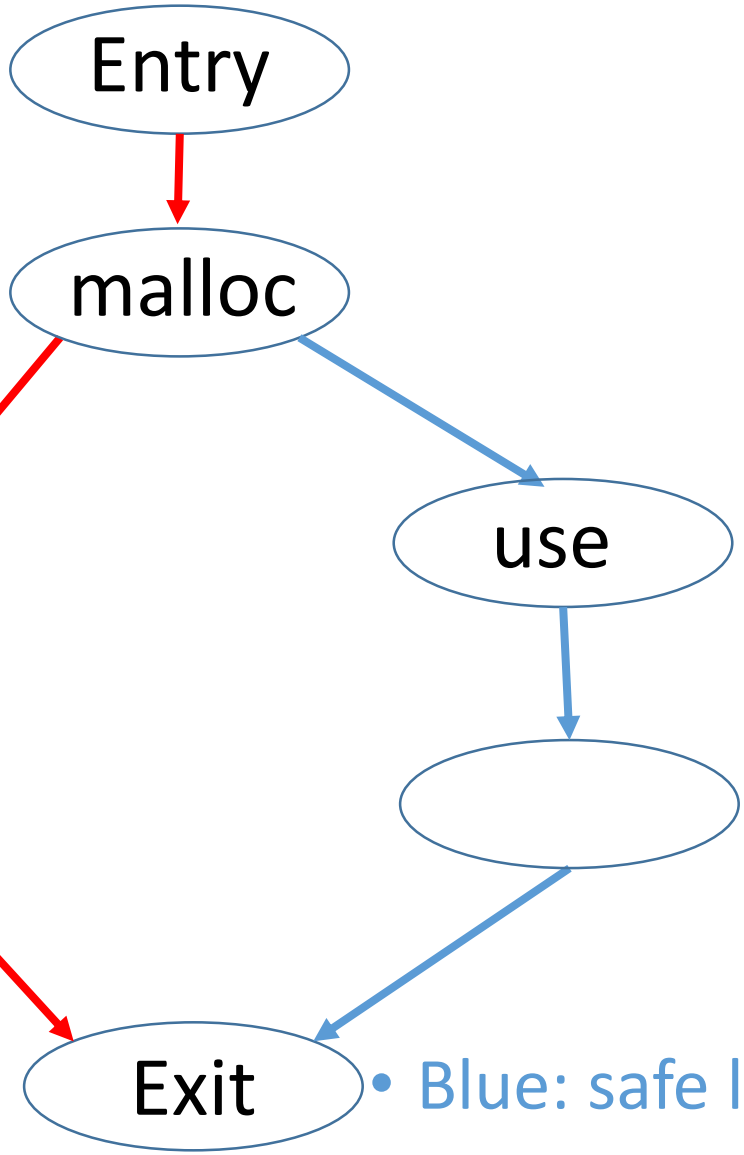
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no double free (backward)

Intra-procedural analysis



- 2. Backward

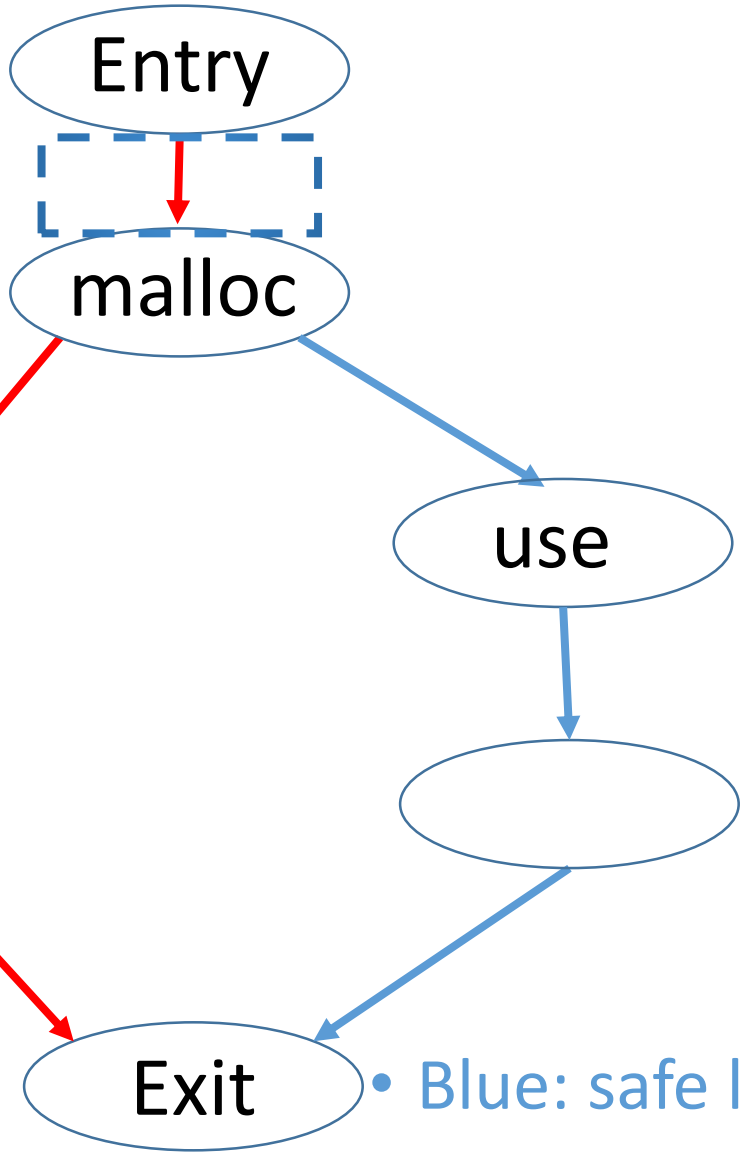
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no double free (backward)

Intra-procedural analysis



- 2. Backward

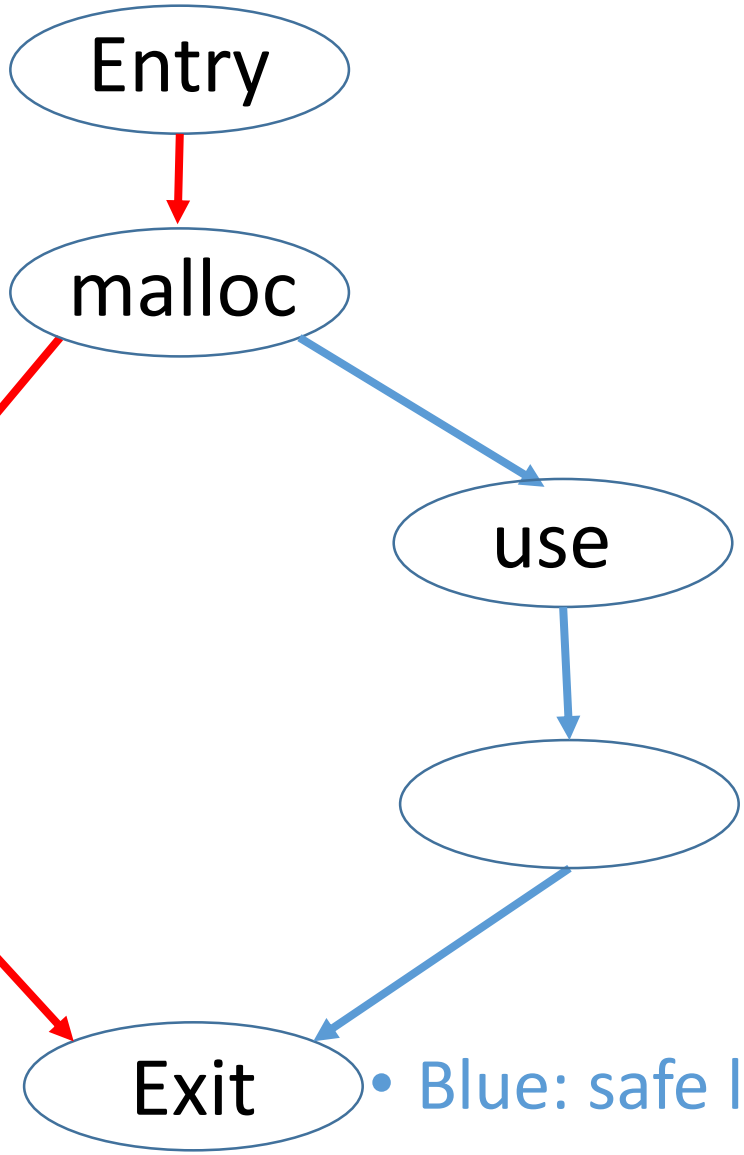
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no double free (backward)

Intra-procedural analysis



- 2. Backward

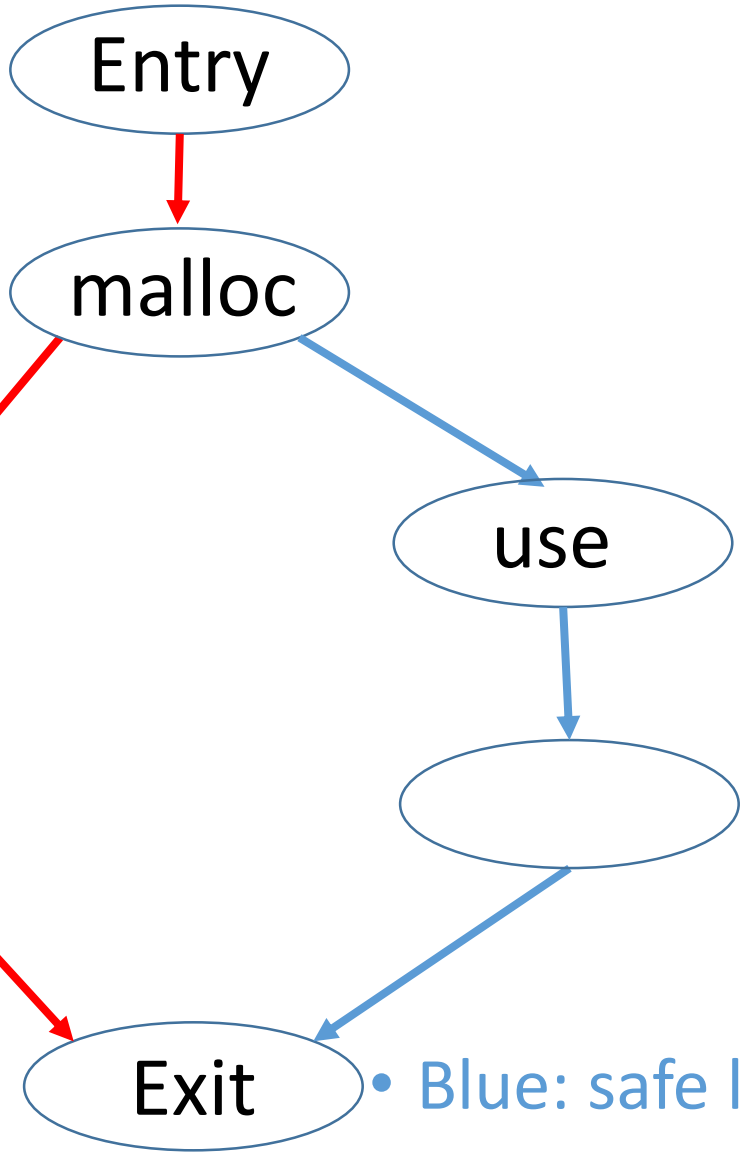
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no double free (backward)

Intra-procedural analysis



- 2. Backward

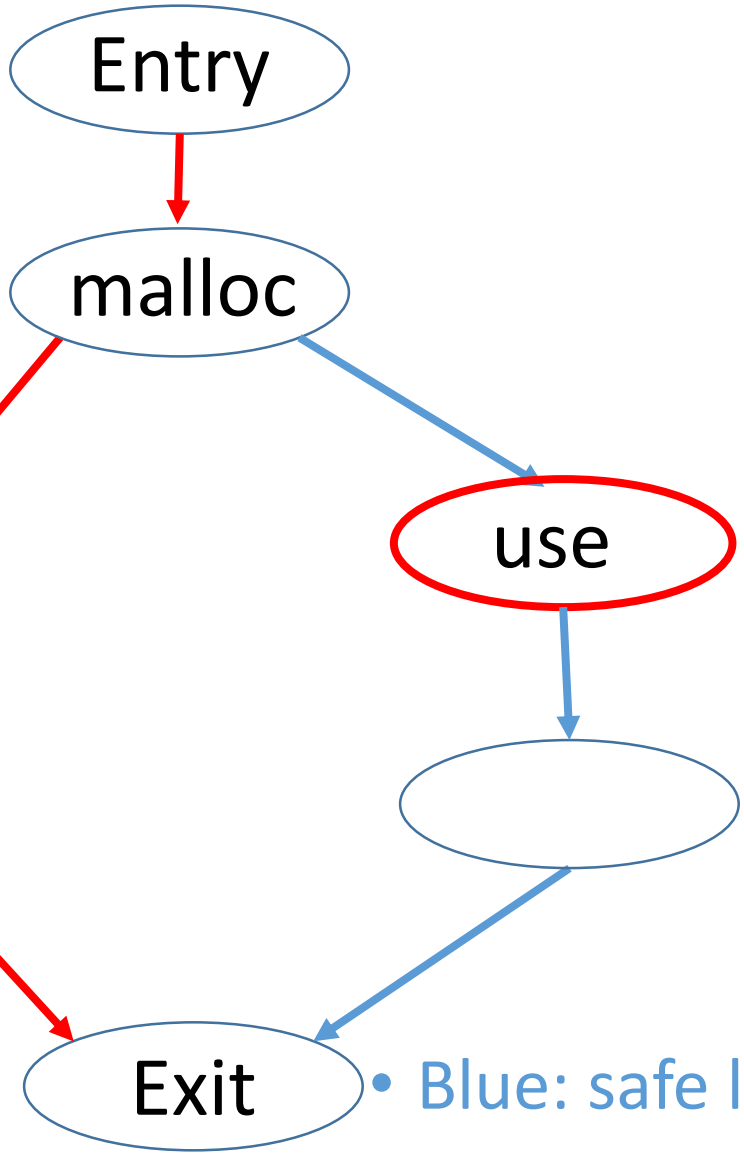
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no use after free

Intra-procedural analysis



- 2. Backward

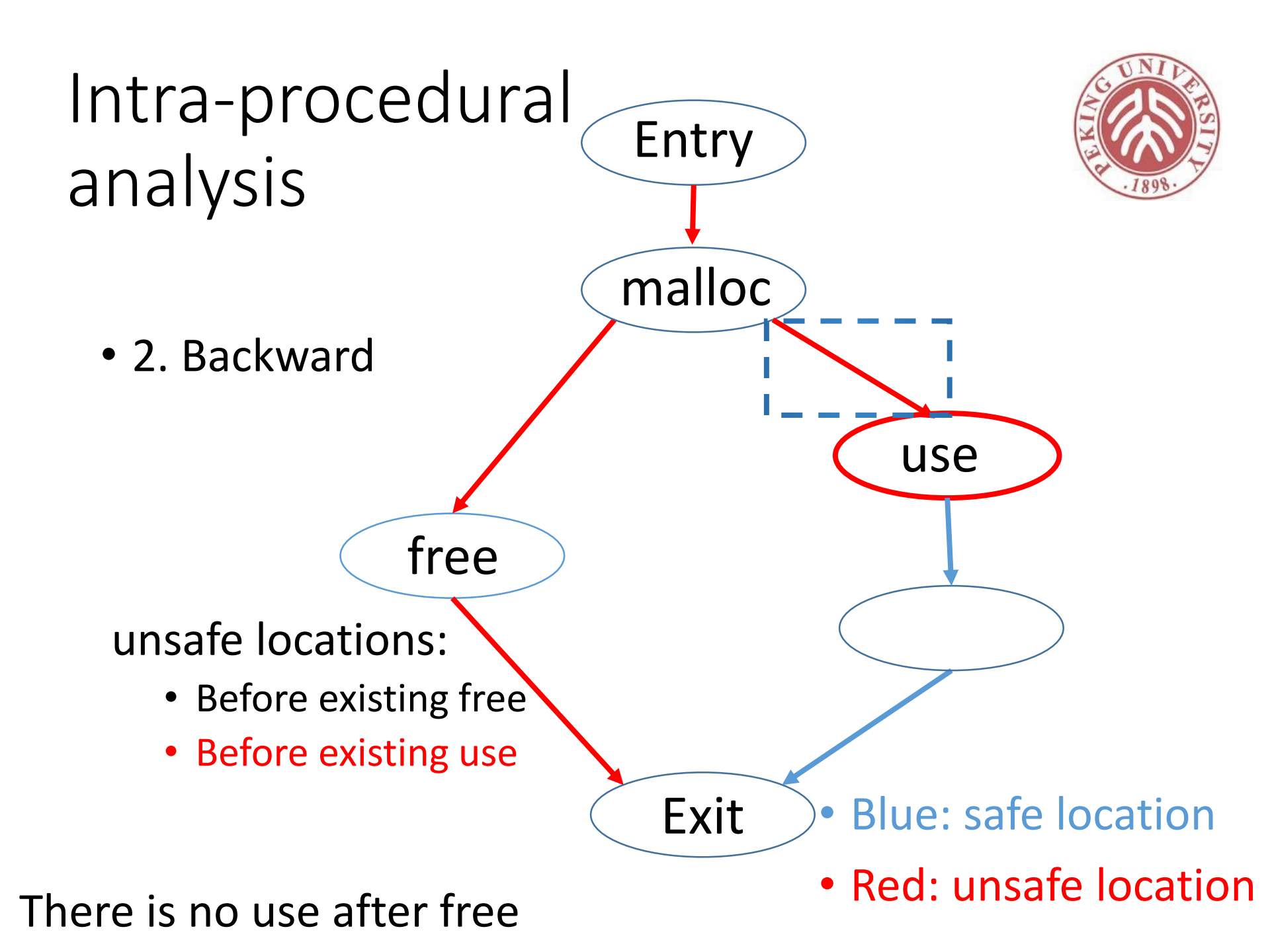
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no use after free

Intra-procedural analysis



- 2. Backward

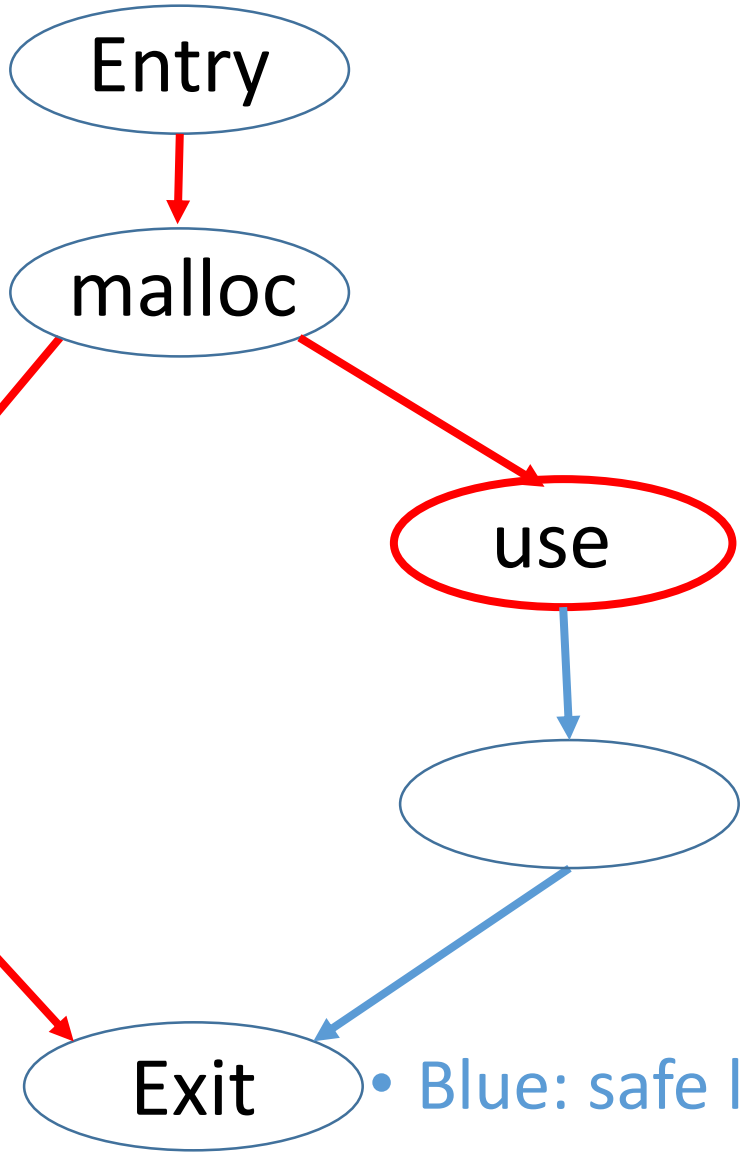
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no use after free

Intra-procedural analysis



- 2. Backward

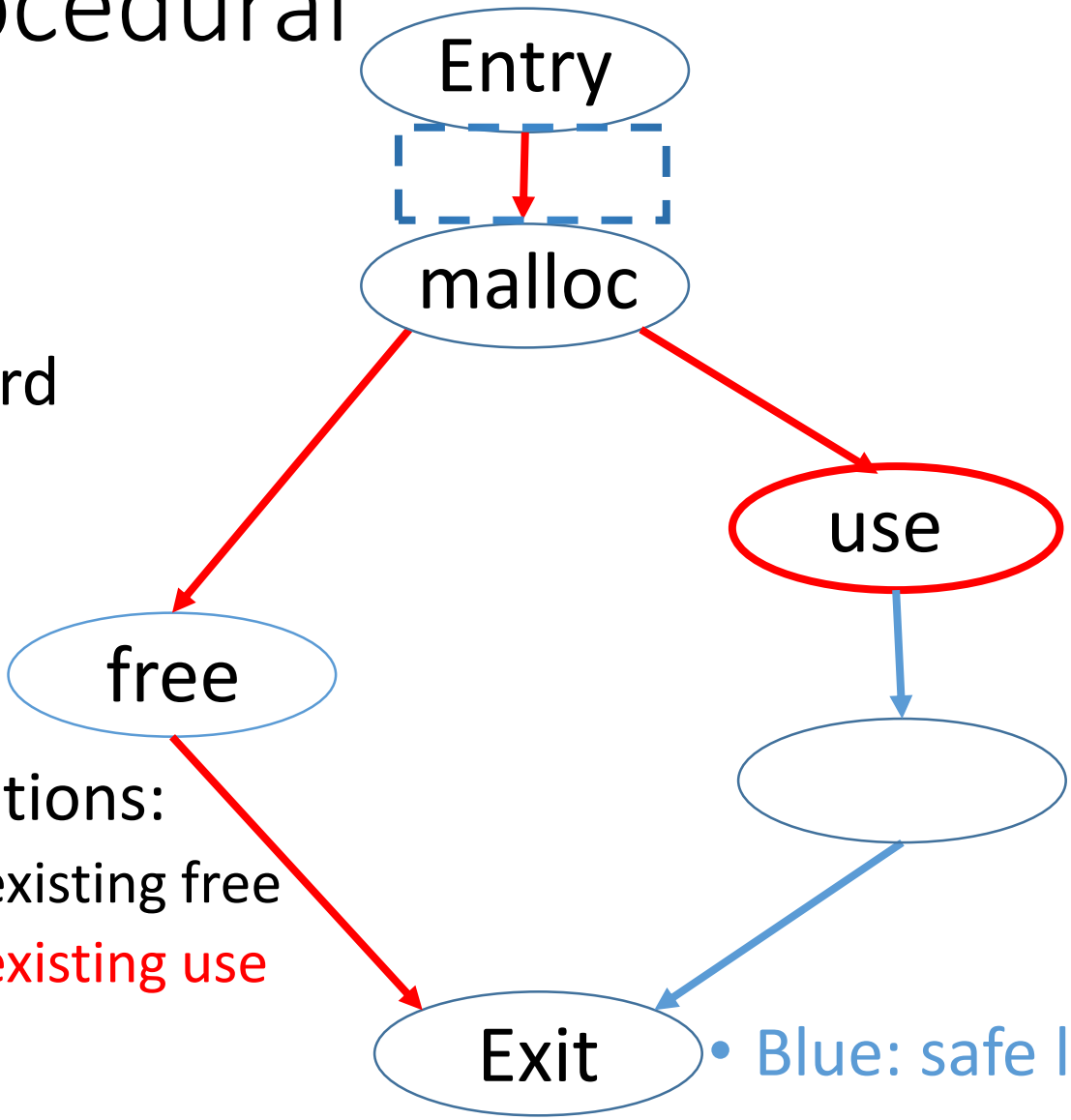
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no use after free

Intra-procedural analysis



- 2. Backward

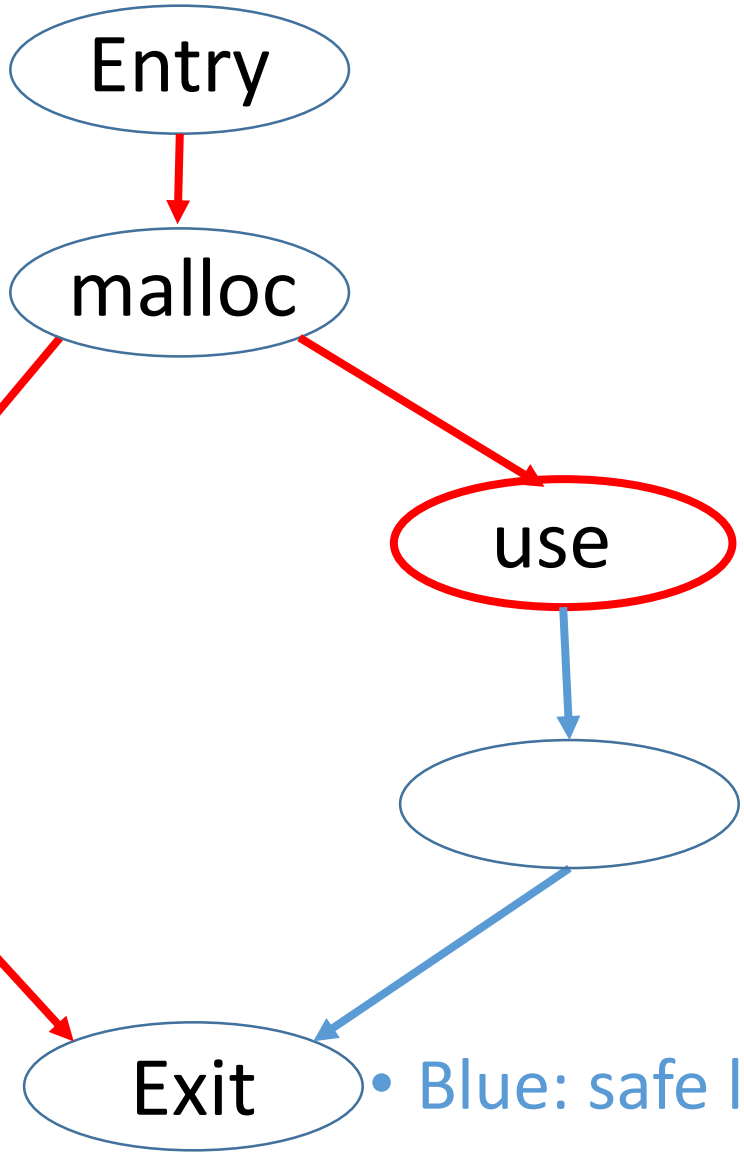
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no use after free

Intra-procedural analysis



- 2. Backward

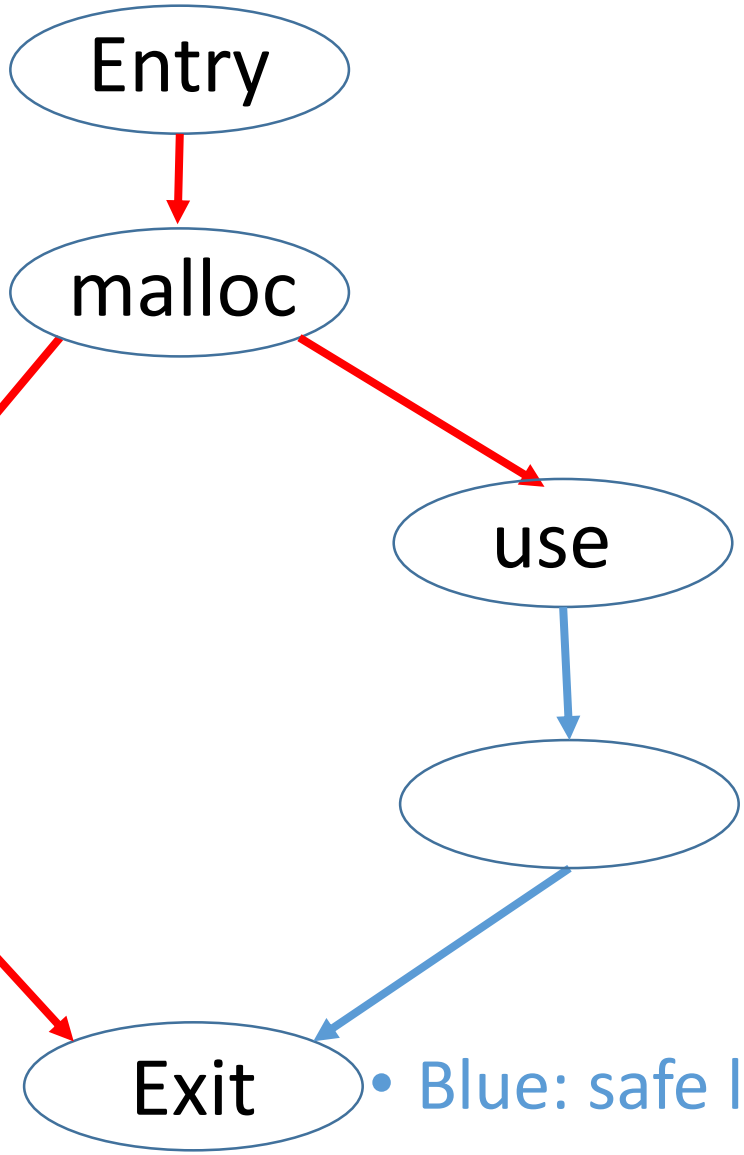
unsafe locations:

- Before existing free
- Before existing use

- Blue: safe location
- Red: unsafe location

There is no use after free

Intra-procedural analysis



- 2. Backward

unsafe locations:

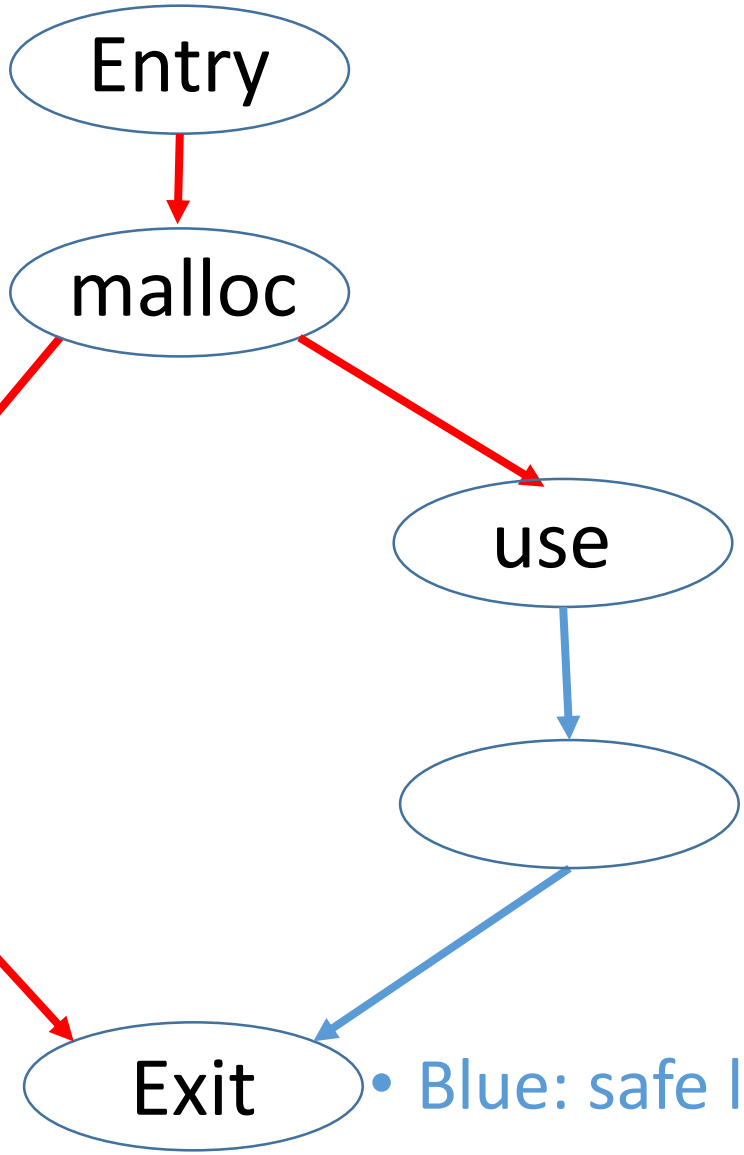
- Before existing free
- Before existing free

There is no use after free

There is no double free (backward)

- Blue: safe location
- Red: unsafe location

Intra-procedural analysis



- Now we have:
  - Safe free locations
- Next:
  - Safe expressions

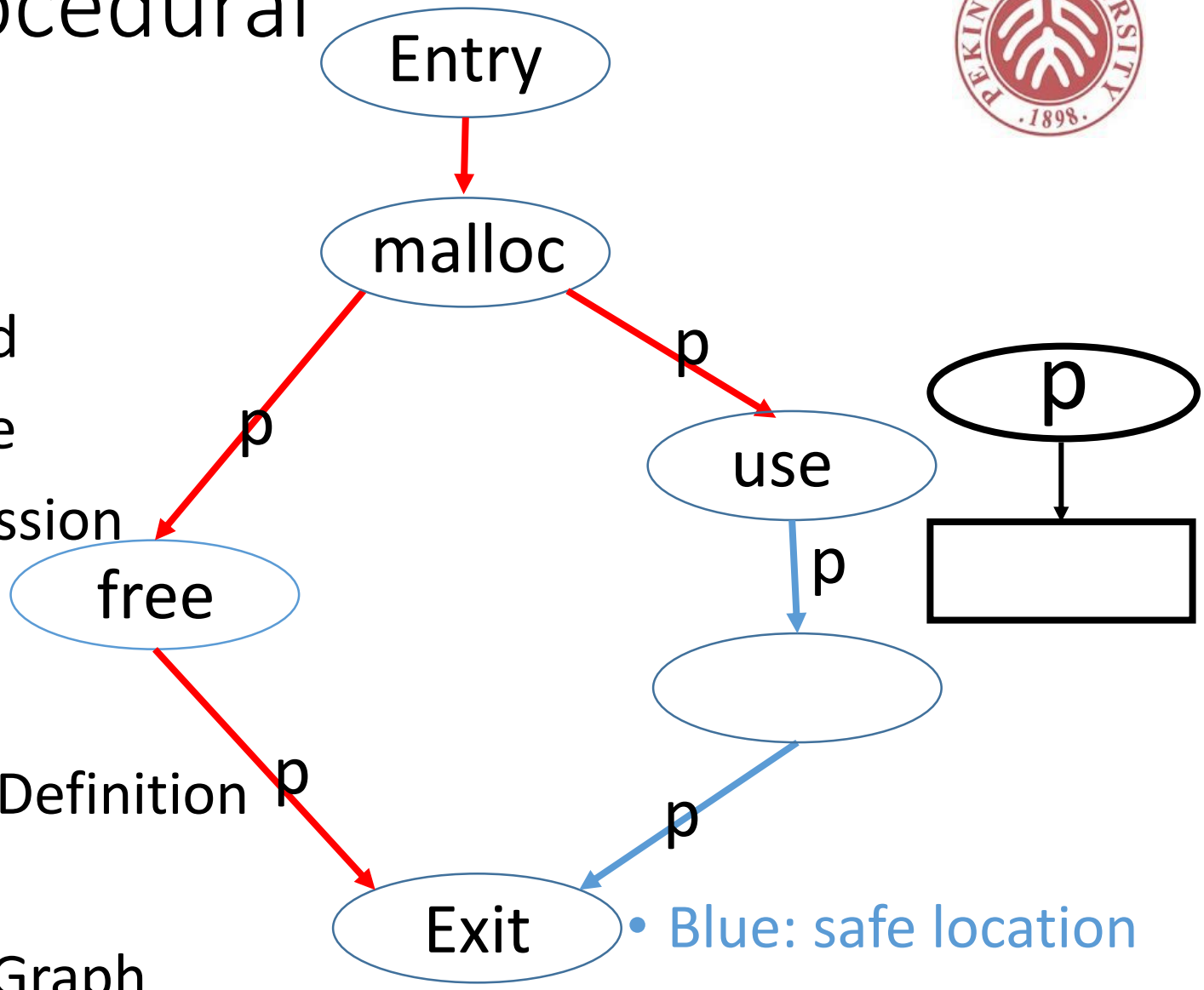
The fix is after an allocation

There is no double free

There is no use after free

- Blue: safe location
- Red: unsafe location

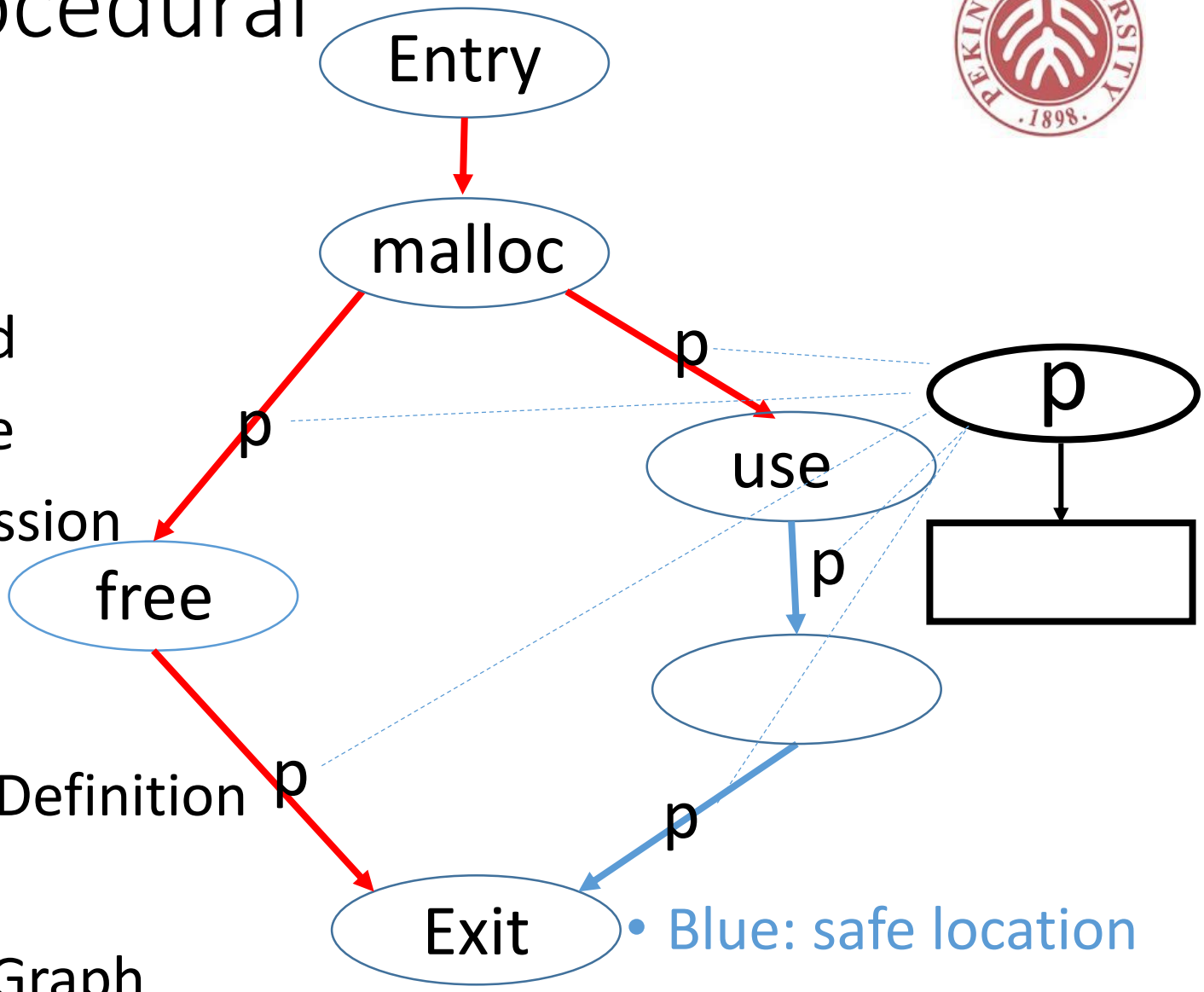
Intra-procedural analysis



- 3. Forward
- Determine safe expression
- Reaching-Definition Analysis
- Points-to Graph

- Blue: safe location
- Red: unsafe location

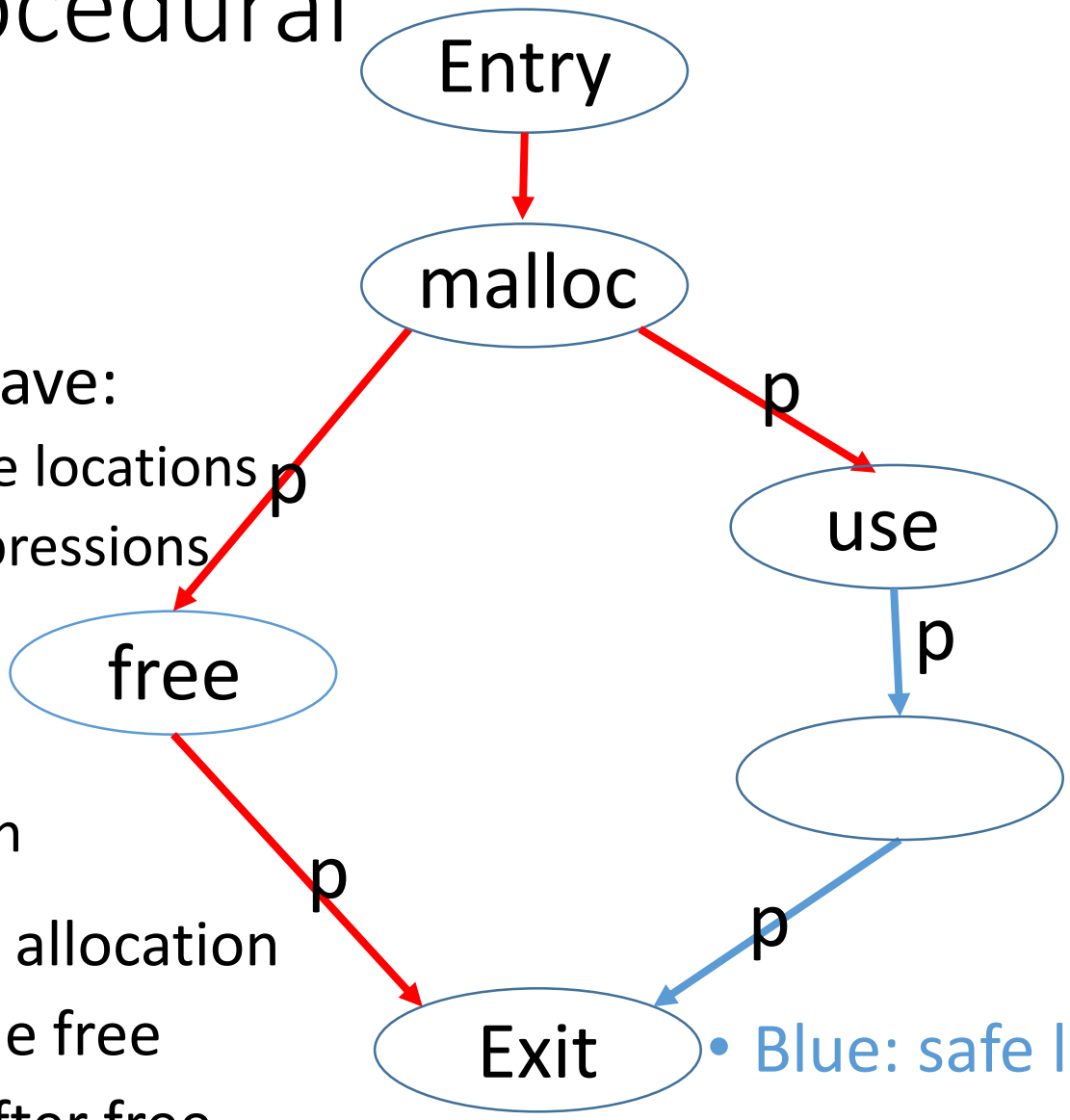
Intra-procedural analysis



- 3. Forward
- Determine safe expression
- Reaching-Definition Analysis
- Points-to Graph

- Blue: safe location
- Red: unsafe location

Intra-procedural analysis



- Now we have:
  - Safe free locations
  - Safe expressions
- Next:
  - Insertion

The fix is after an allocation

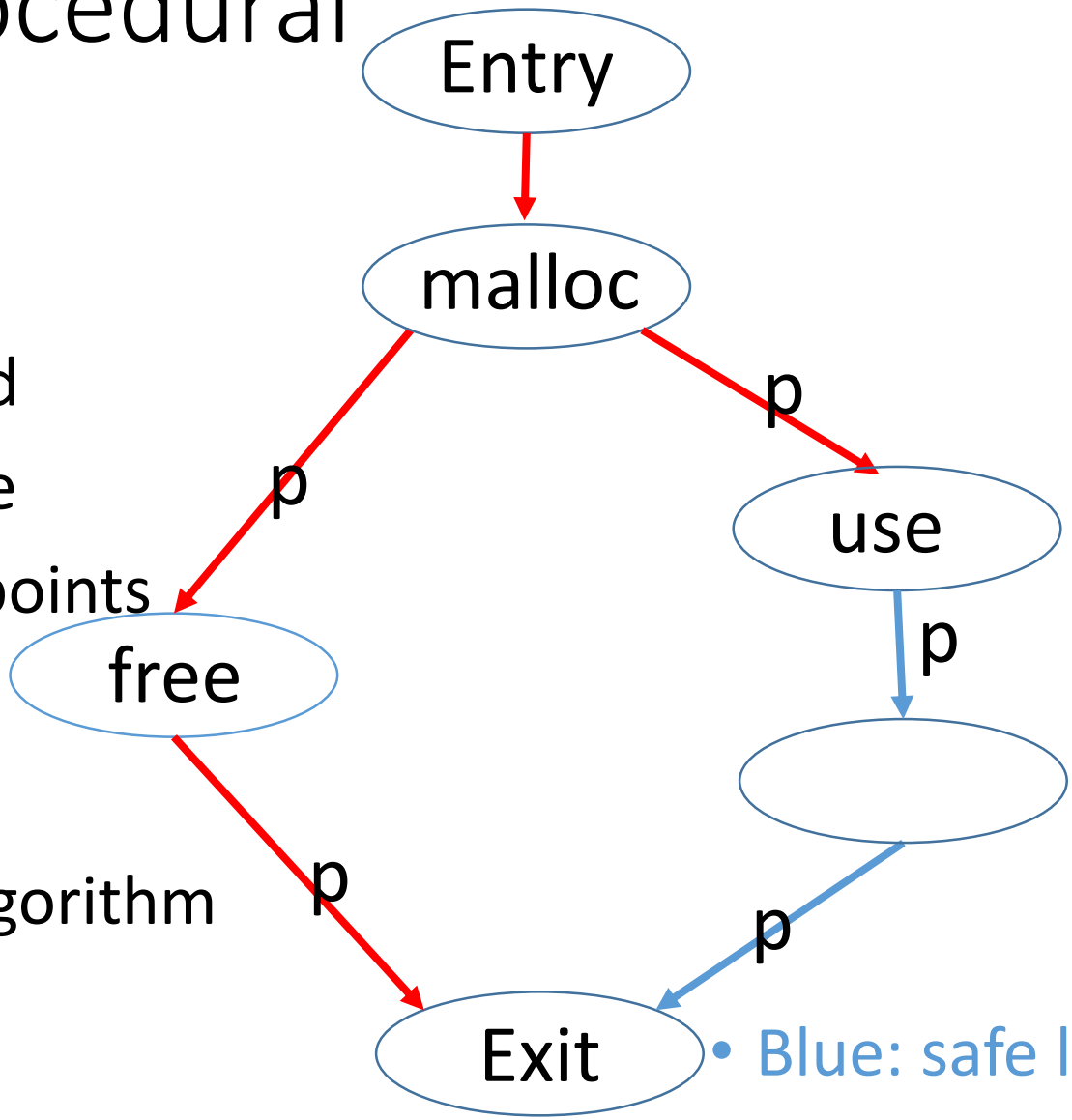
There is no double free

There is no use after free

There is an expression

- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

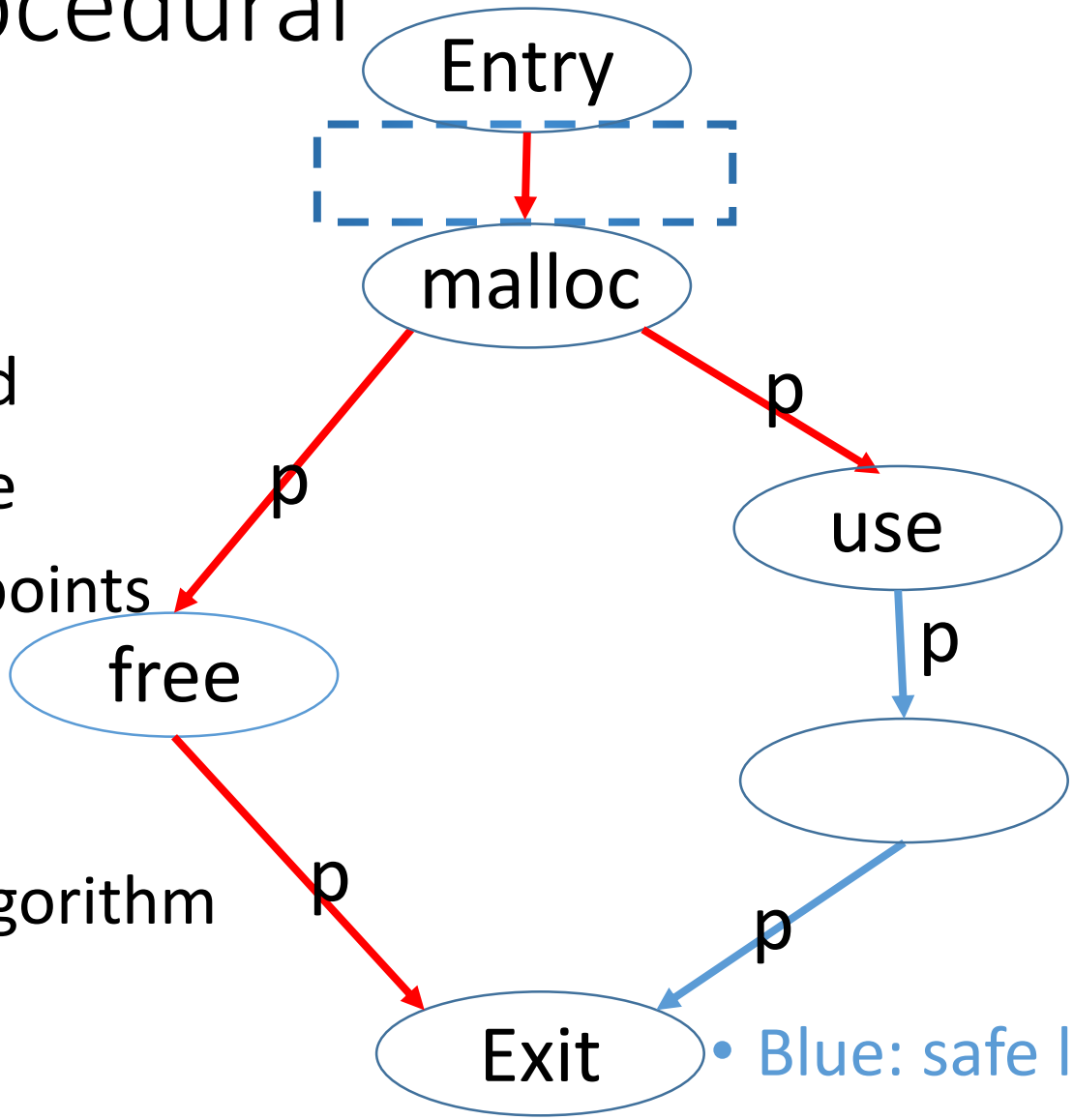


- 4. Forward
- Determine early free points
- Greedy algorithm

- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

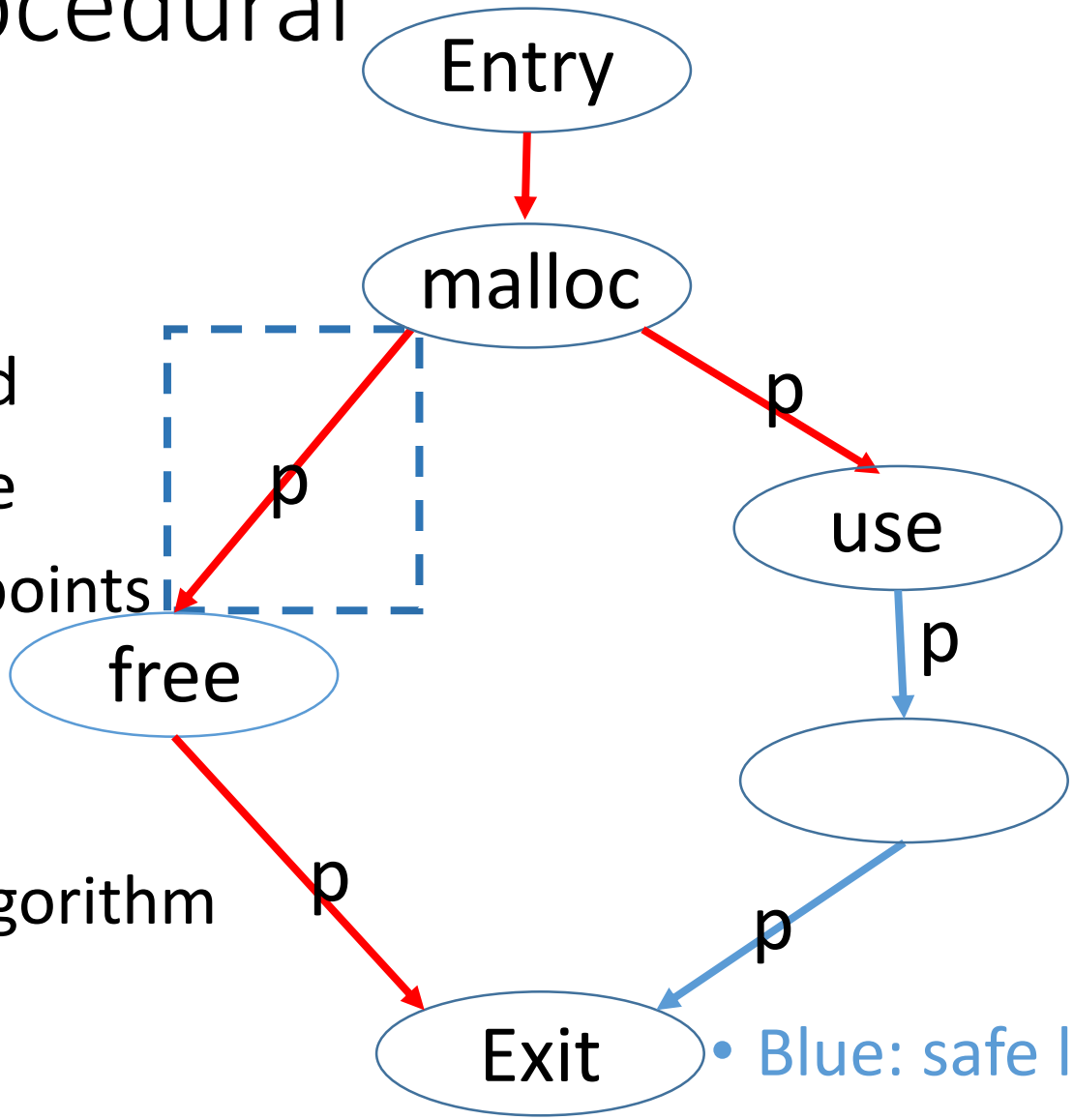
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

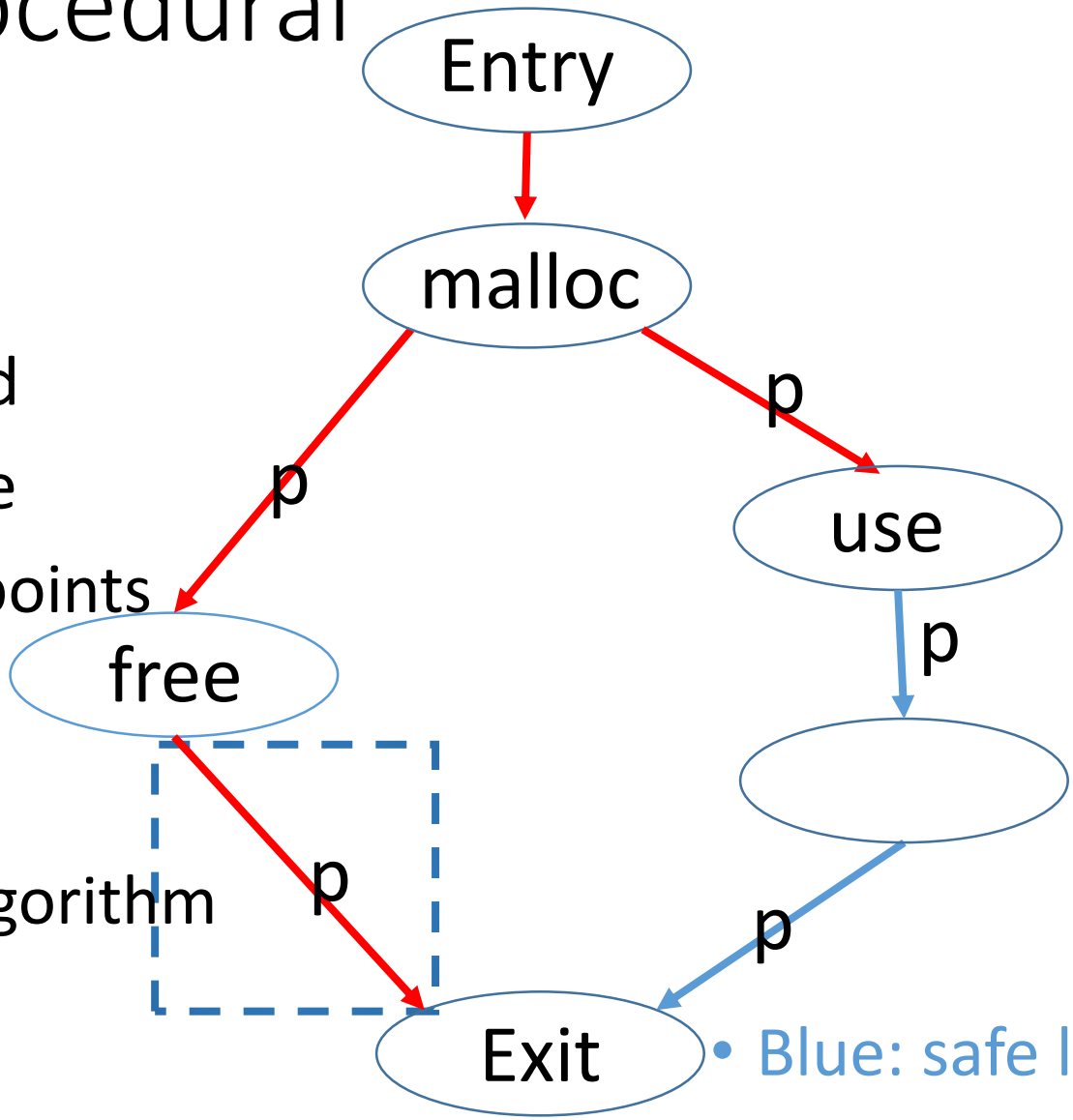
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

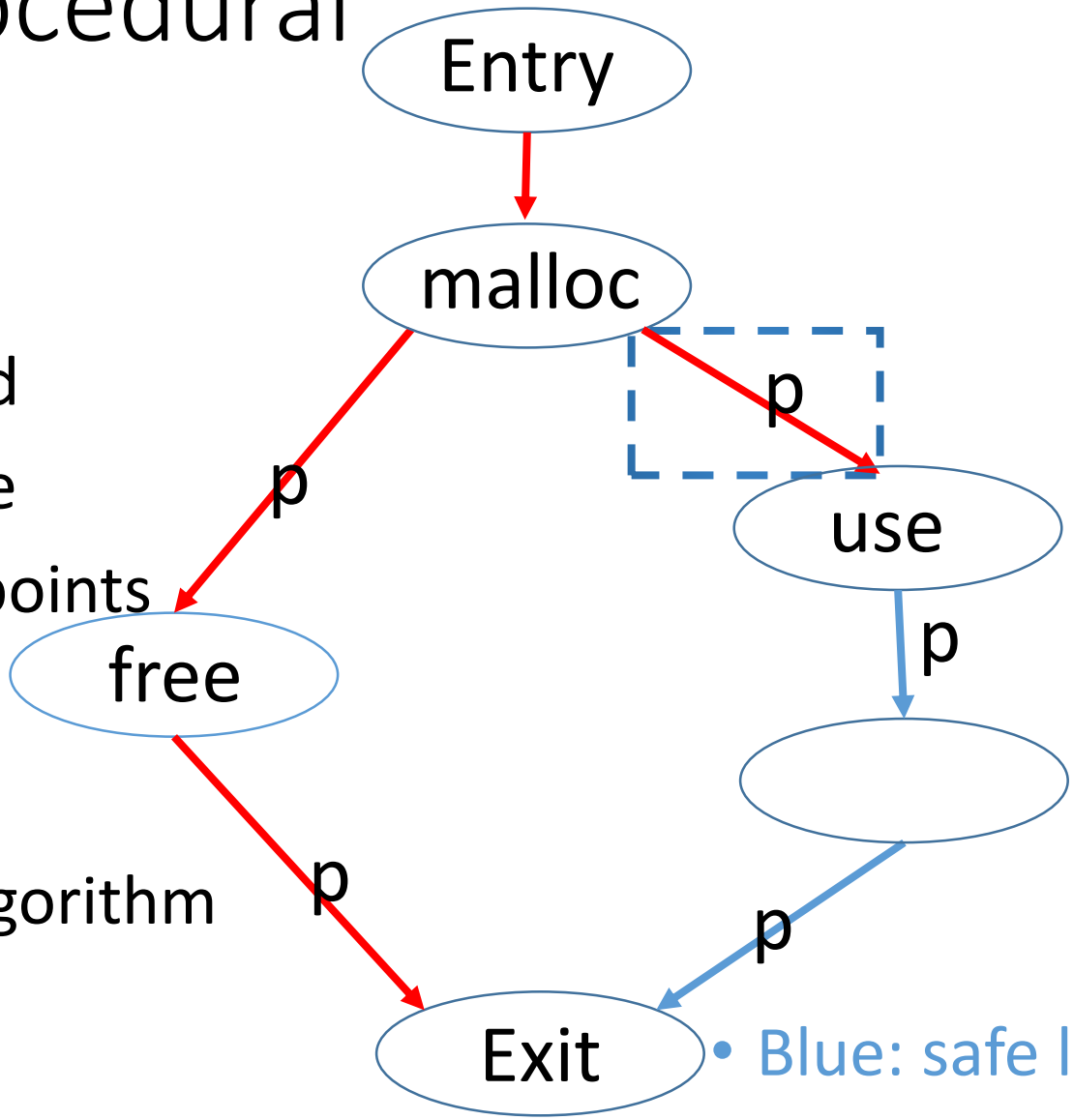
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

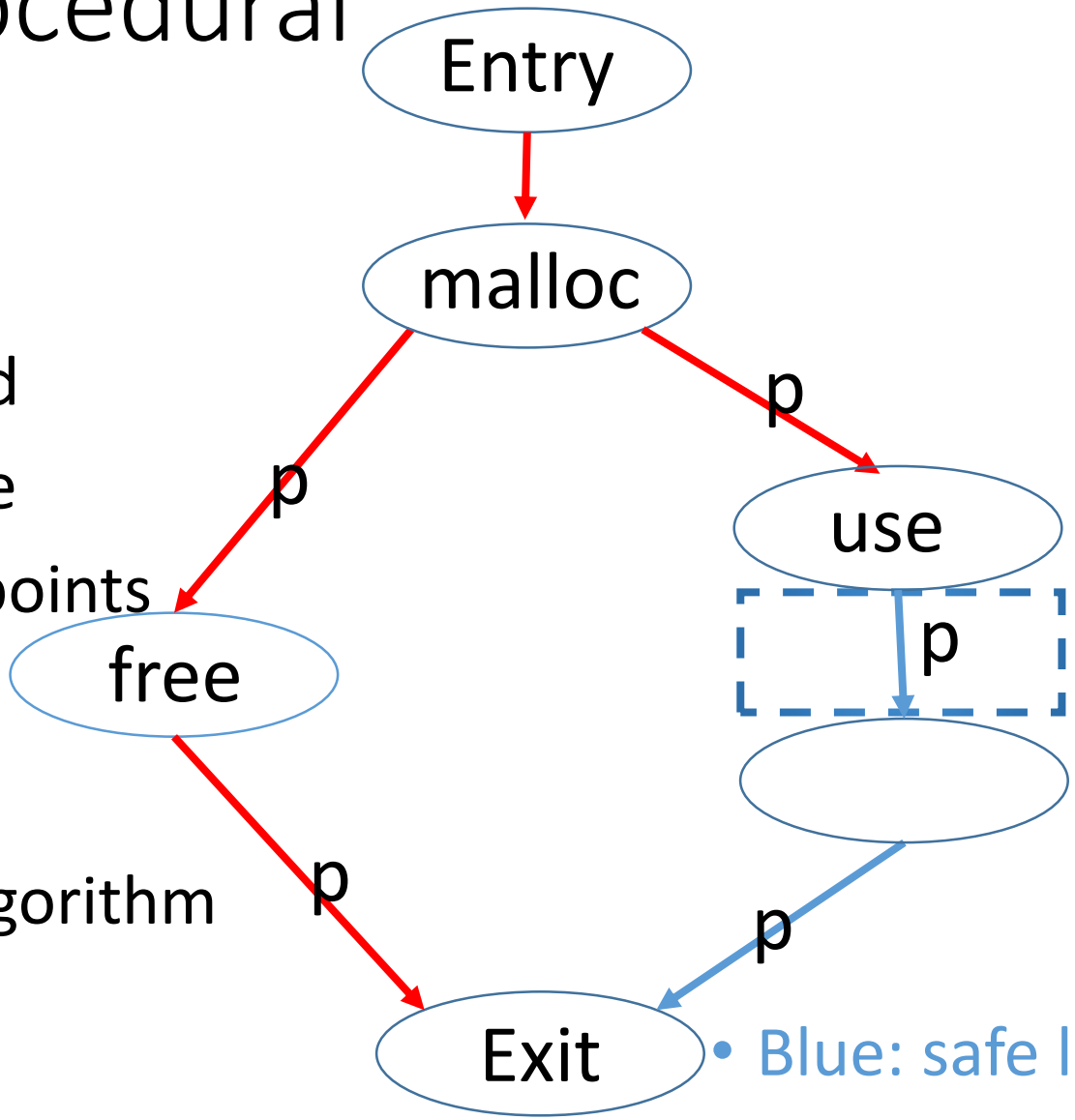
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

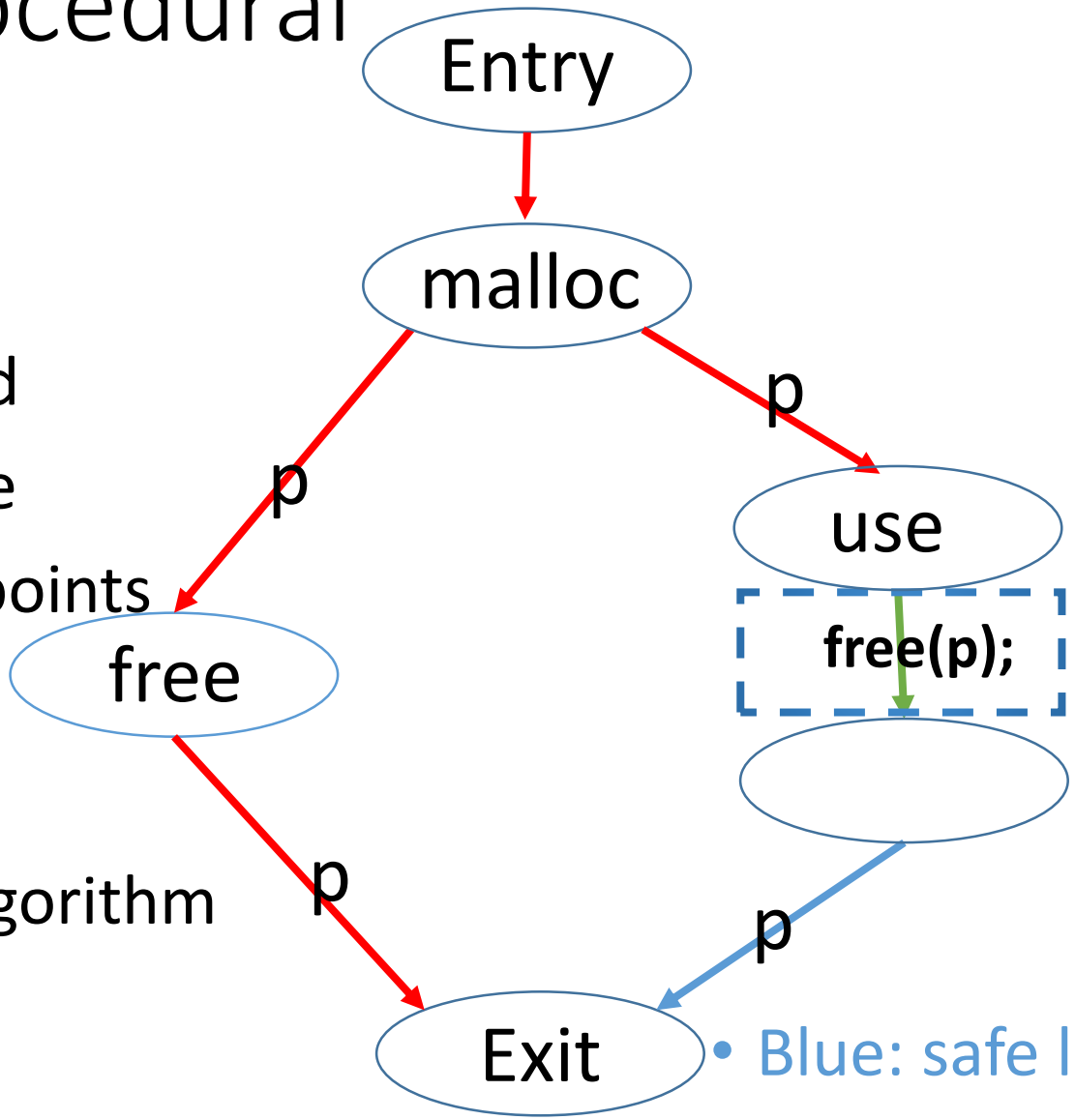
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

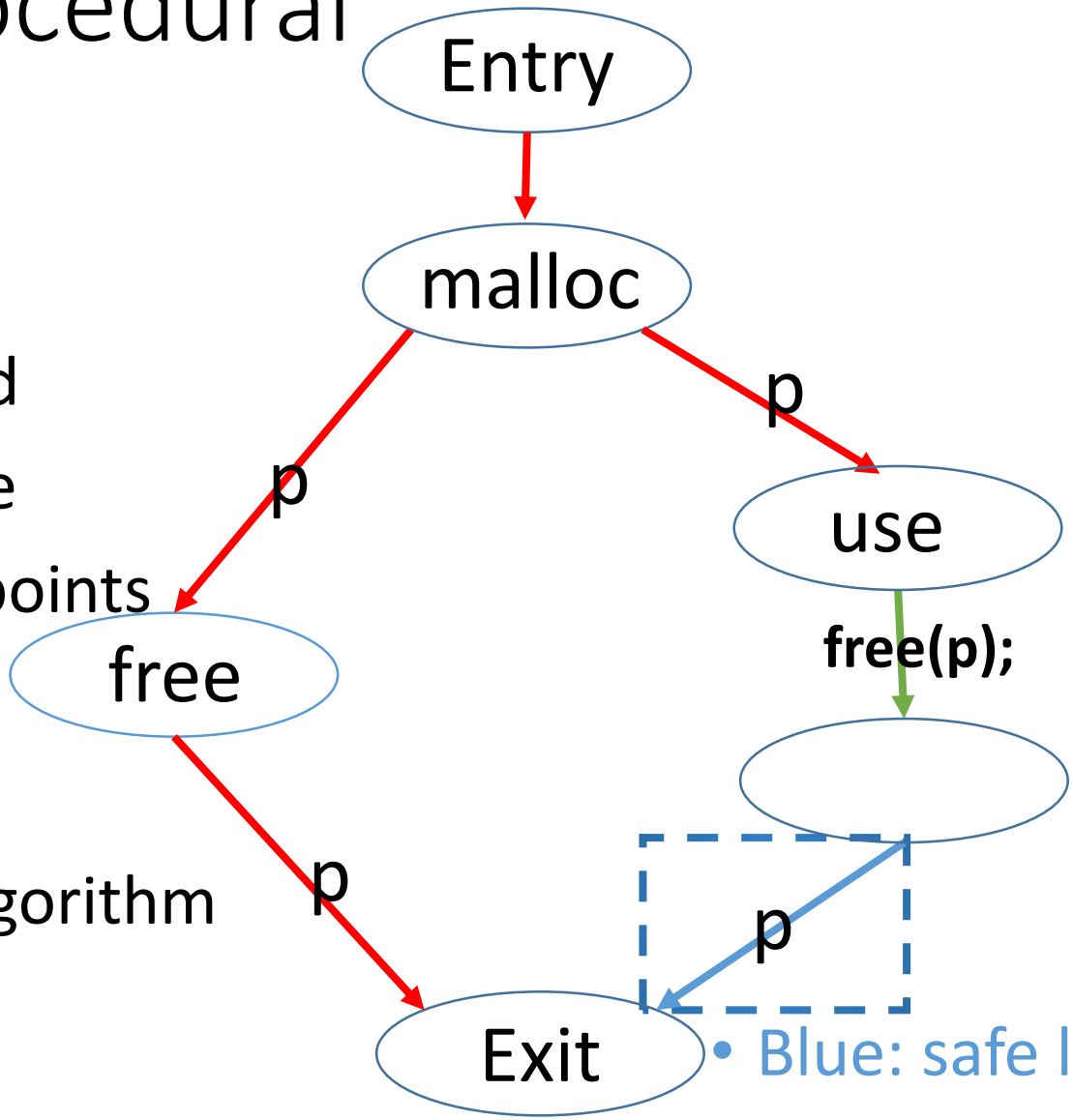
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

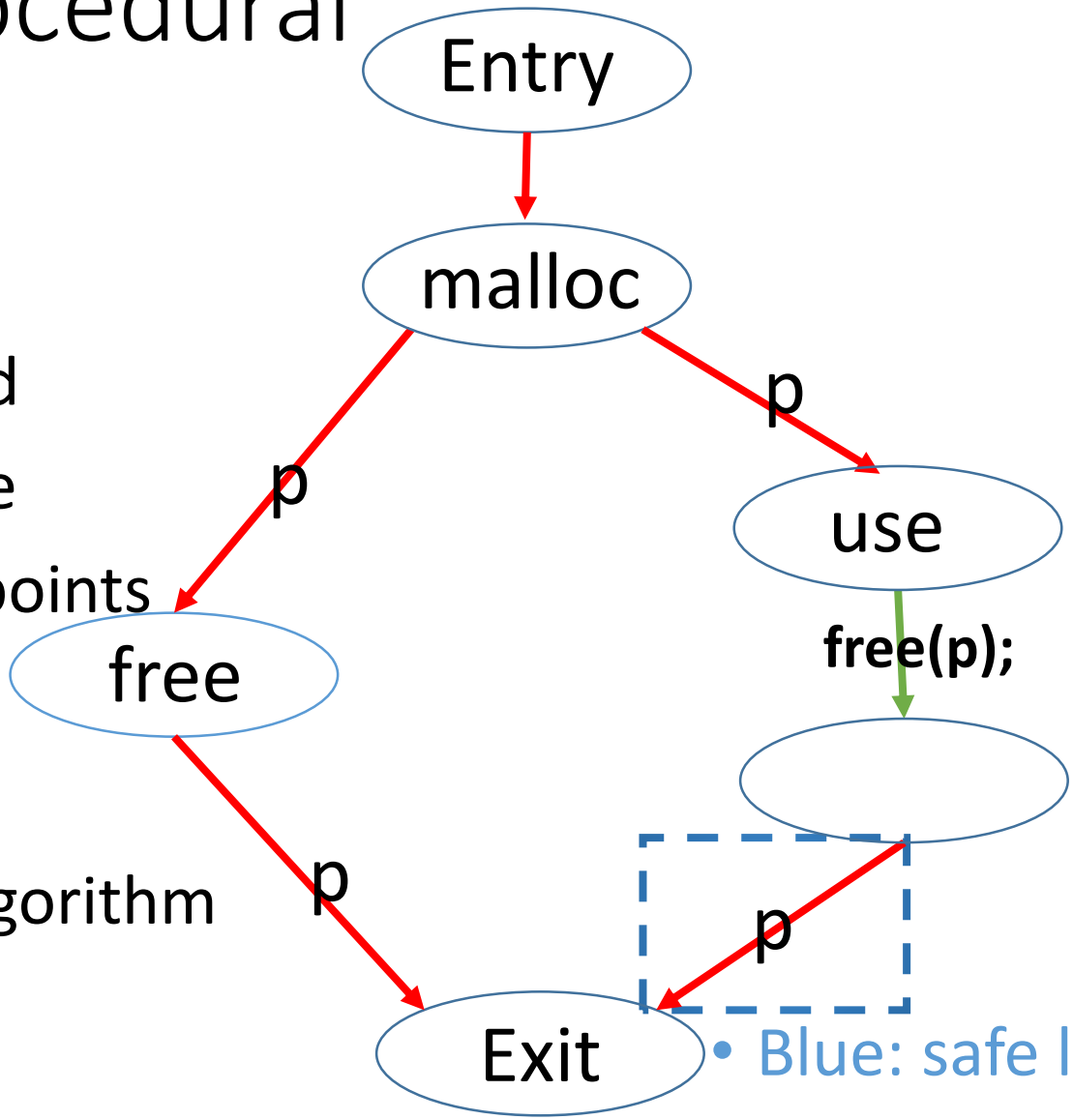
- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

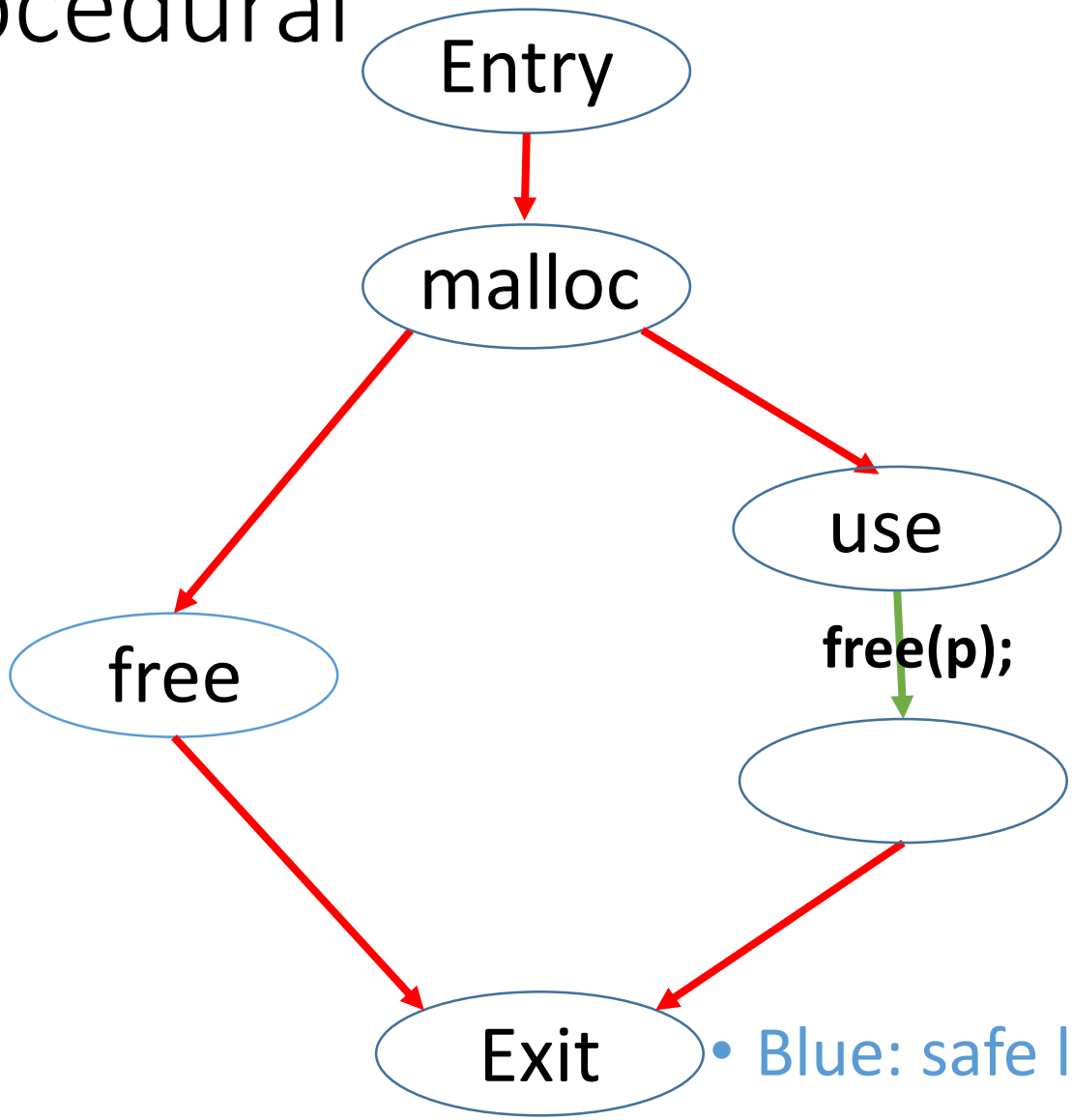
Intra-procedural analysis

- 4. Forward
- Determine early free points
- Greedy algorithm



- Blue: safe location
- Red: unsafe location

Intra-procedural analysis

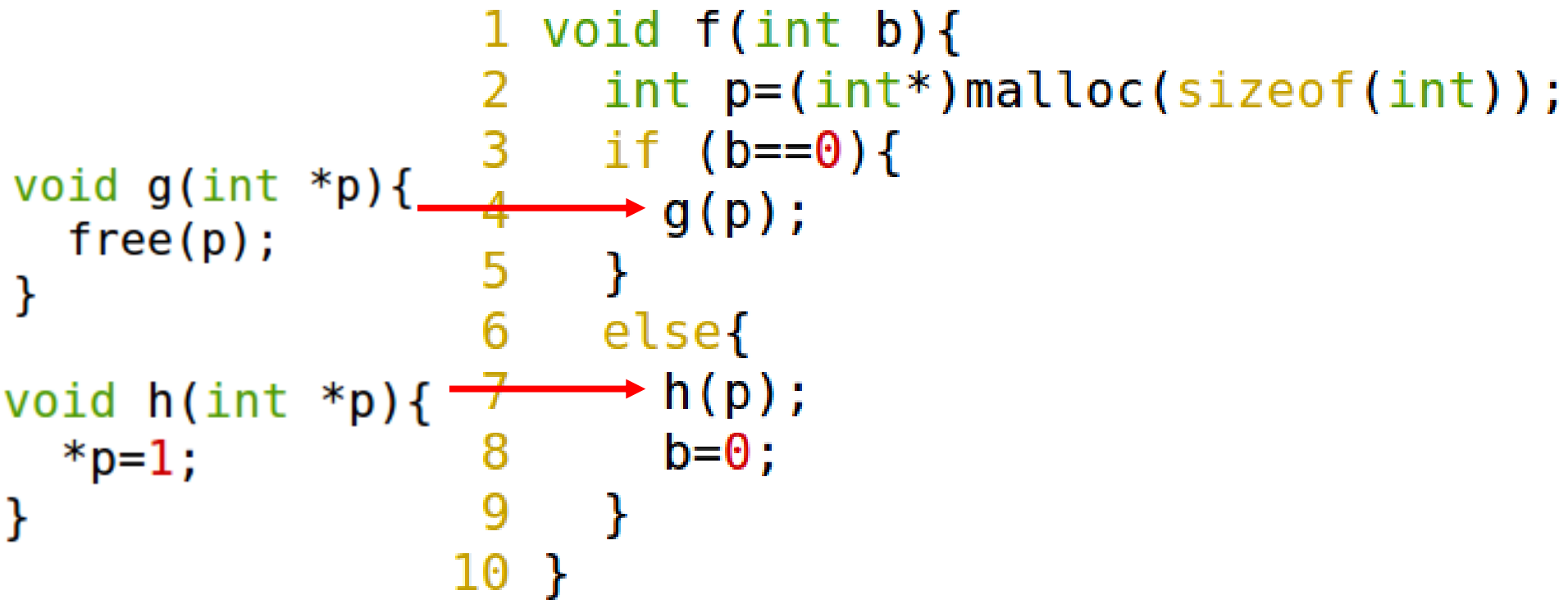


- Blue: safe location
- Red: unsafe location

More Complicated Cases

Inter-procedure

- Malloc, free, use elements are method calls
- Solution: Build summary for each method



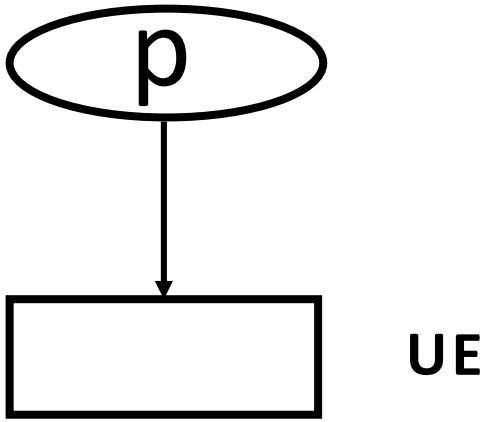
Inter-procedure

- Build summaries for each function
  - Summarize the allocation, use, free, escape information
  - Iteratively update the callers'
- Analyze Intra-procedurally

Building summaries

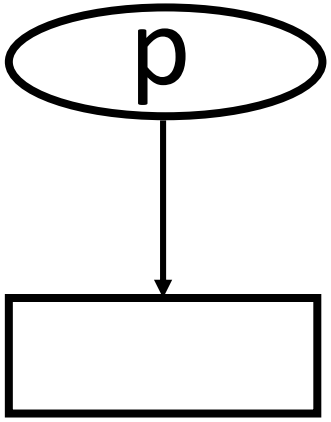
- Method type for each memory chunk
  - Allocation（A）
  - Free（F）
  - Use（U）
  - Escape (E)

```
void h(int *p){
  *p=1;
}
```



UE

```
void g(int *p){
  free(p);
}
```



F E

Inter-procedure

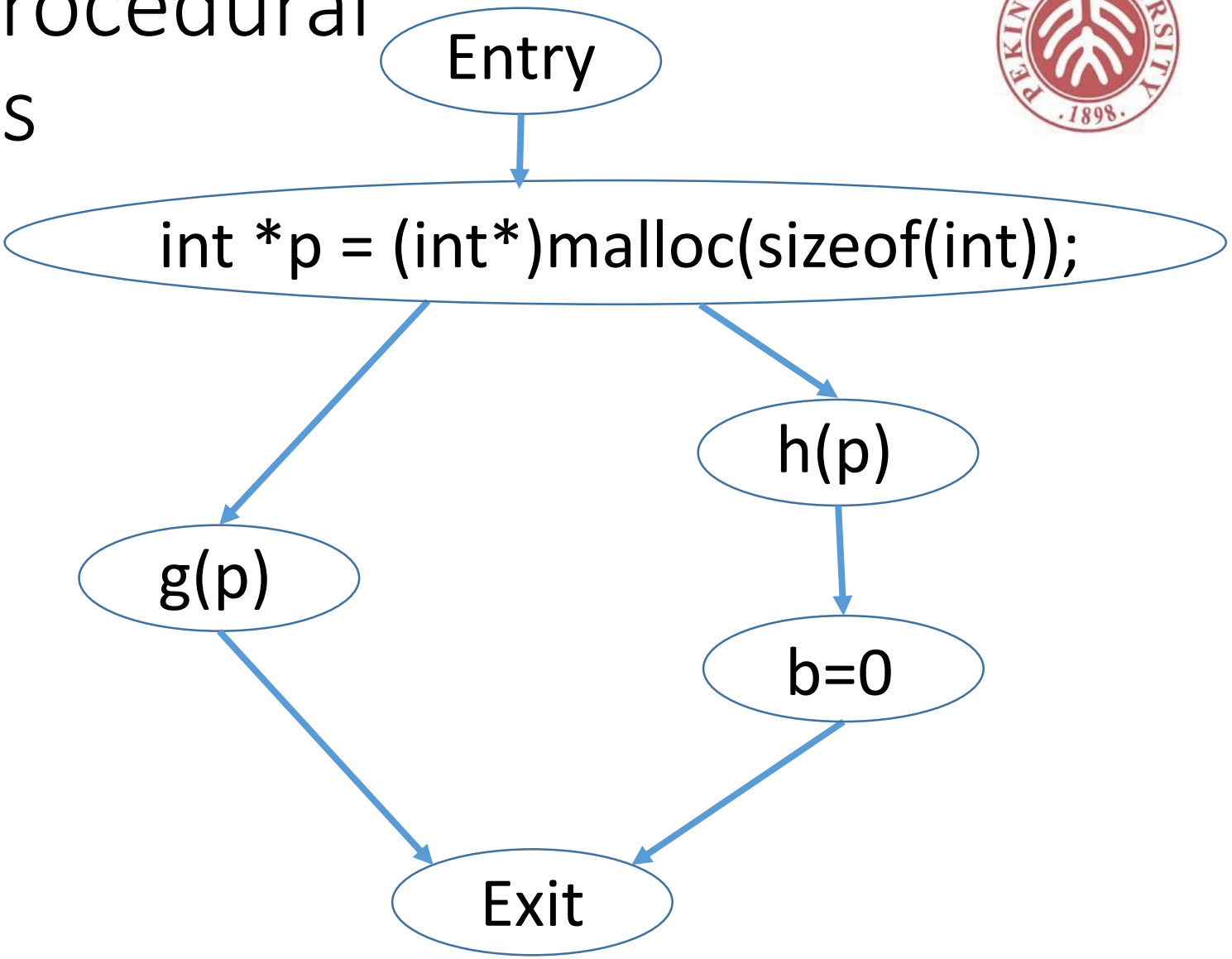
- Scan each statement
- Consult the summary

```
1  void f(int b){
2    int p=(int*)malloc(sizeof(int));
3    if (b==0){
4      g(p);
5    }
6    else{
7      h(p);
8      b=0;
9    }
10 }
```

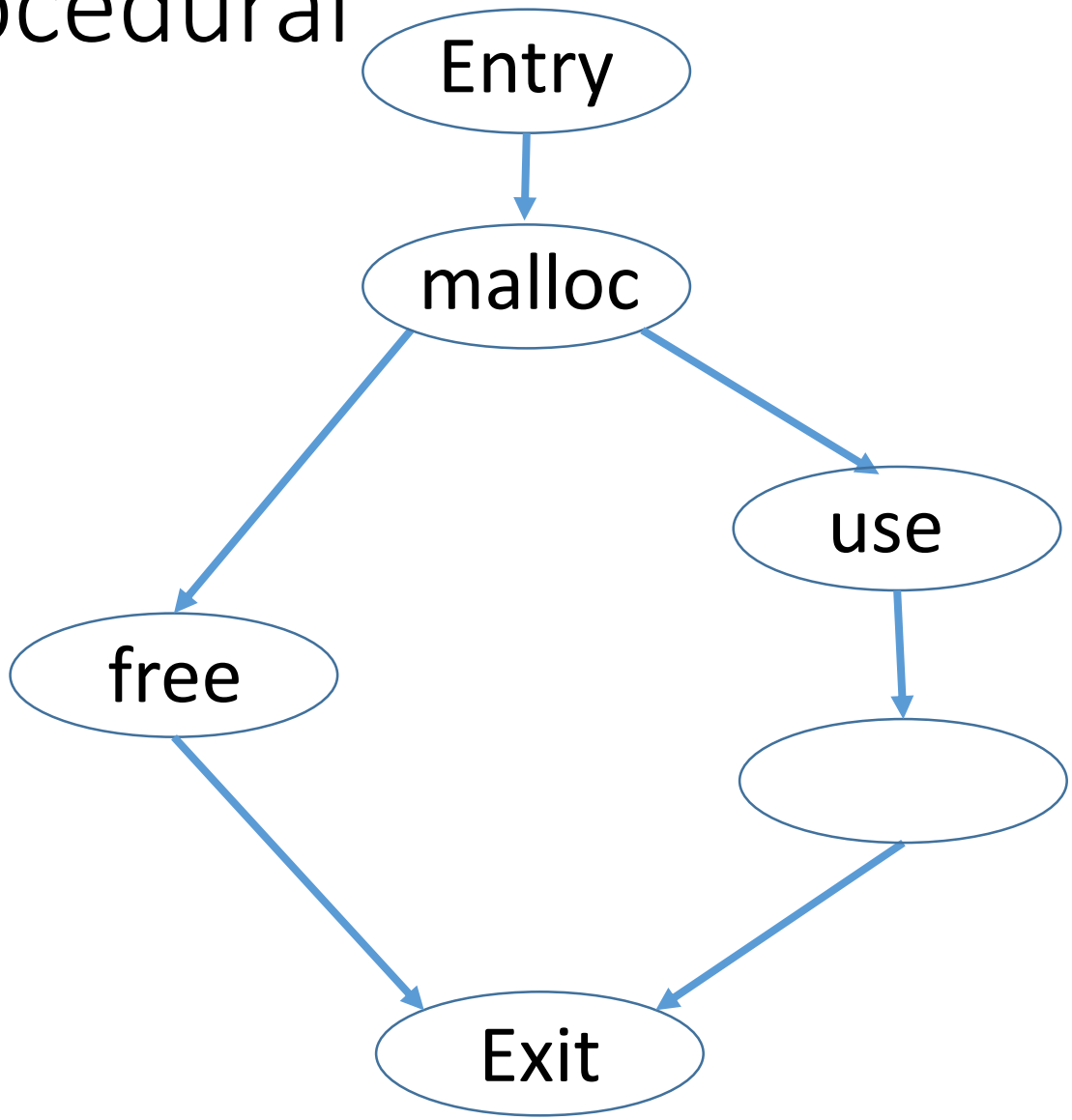
```
void g(int *p){
  free(p);
}

void h(int *p){
  *p=1;
}
```

Intra-procedural analysis

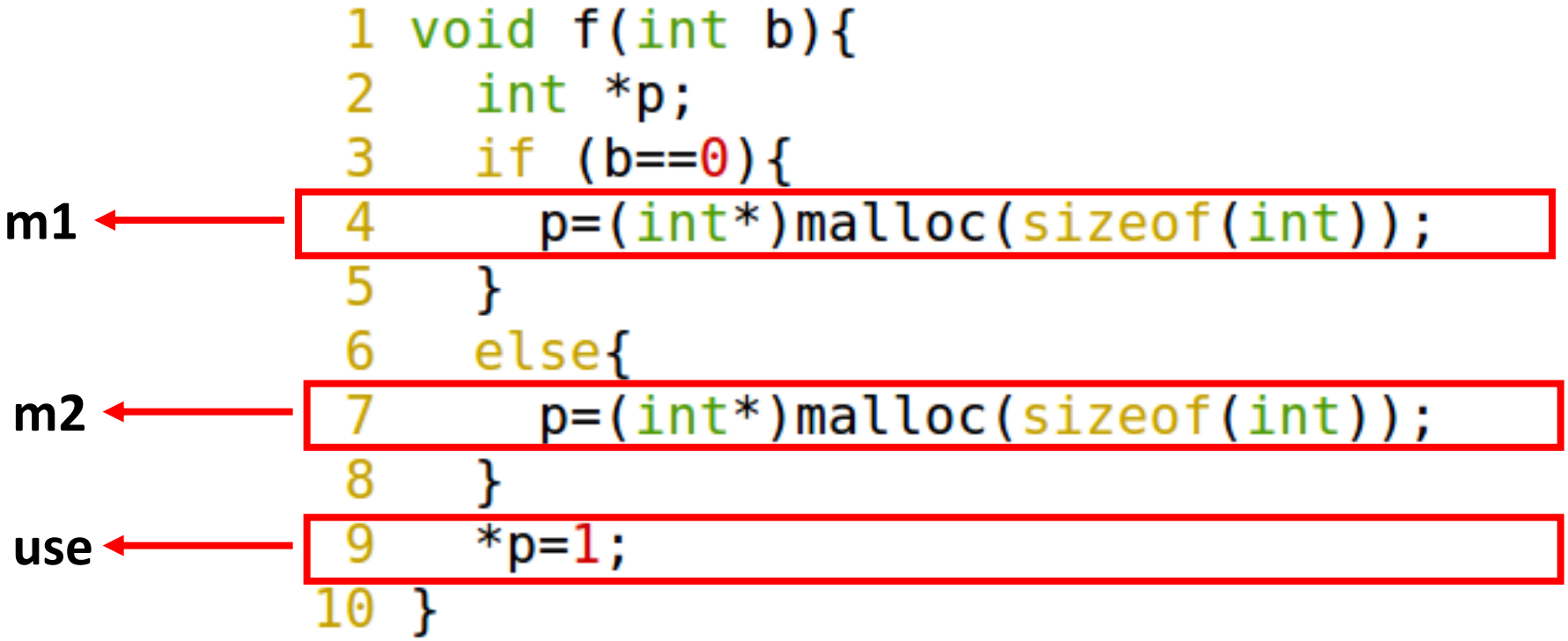


Intra-procedural analysis

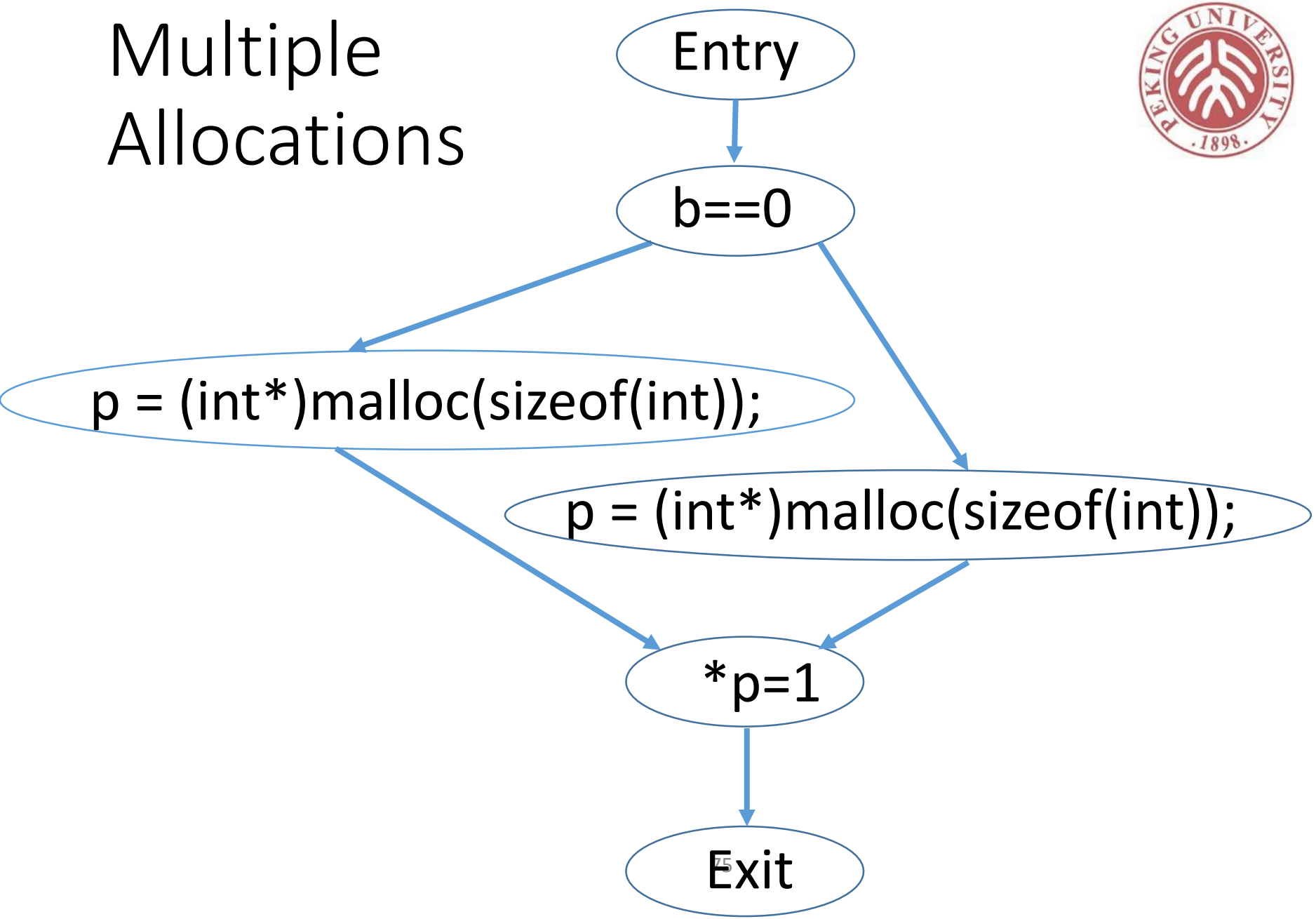


Multiple Allocations

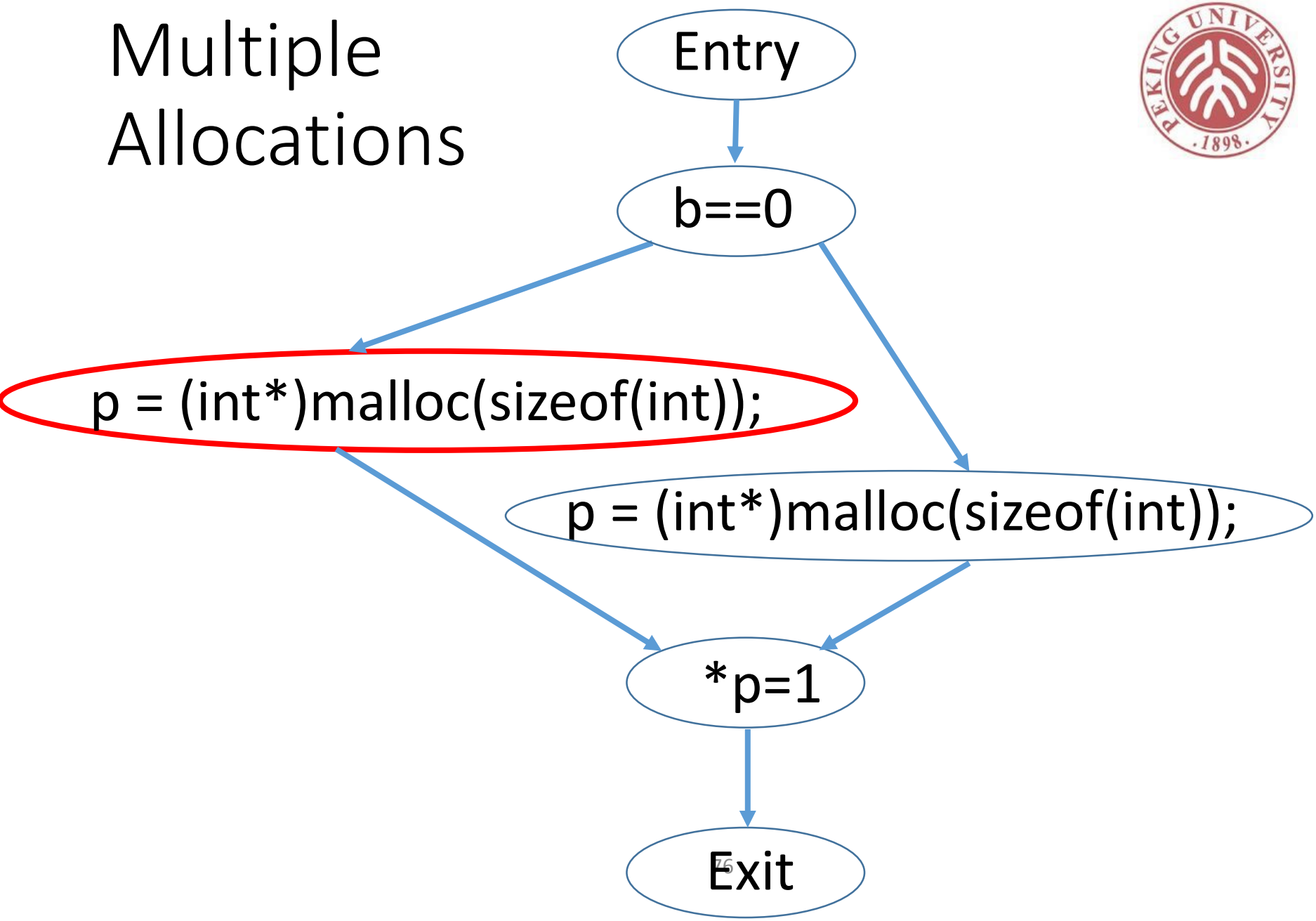
- Multiple allocations are used as one allocation
- Solution: set-based edge condition on CFG edges, instead of blue-red binary



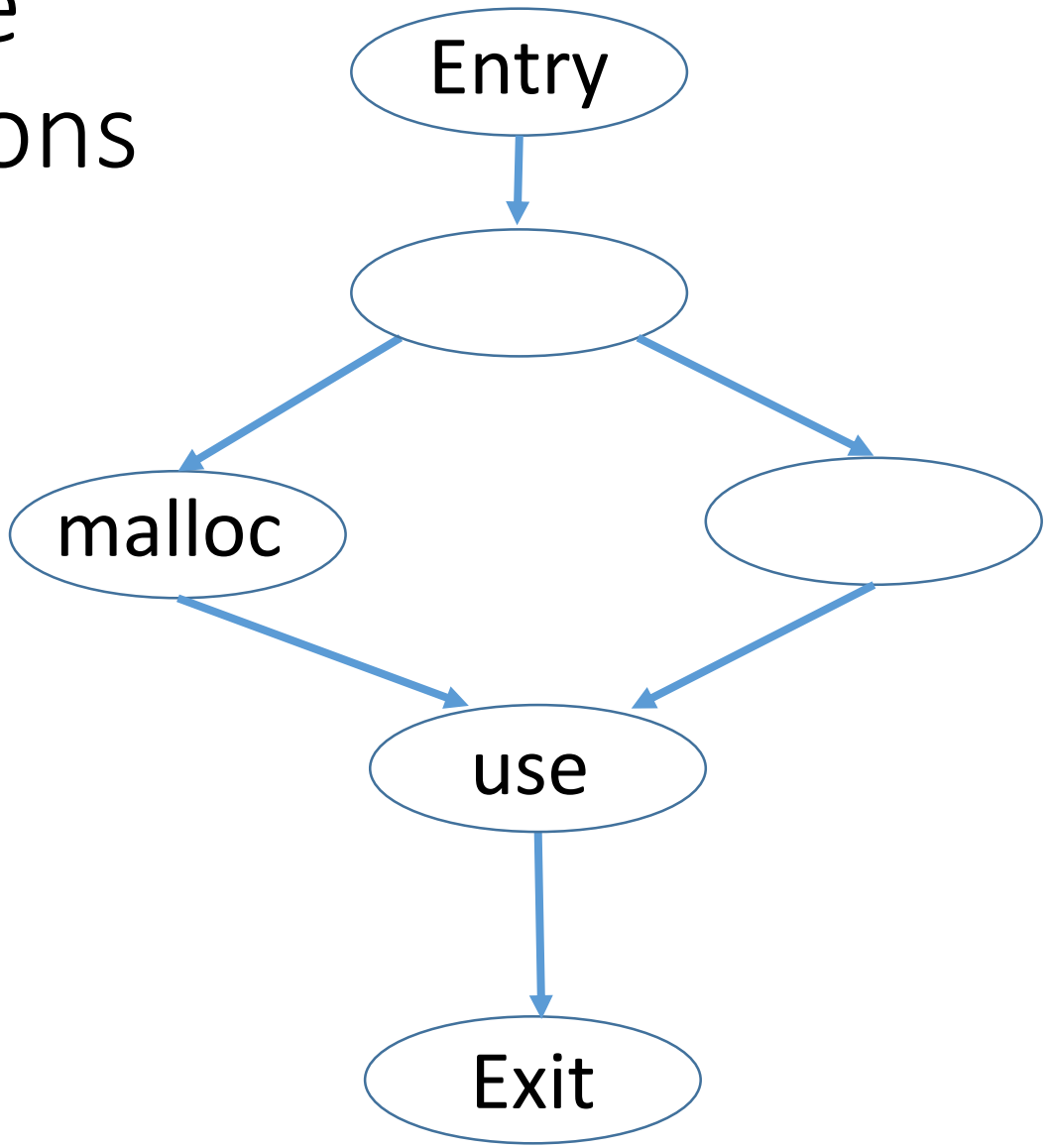
Multiple Allocations



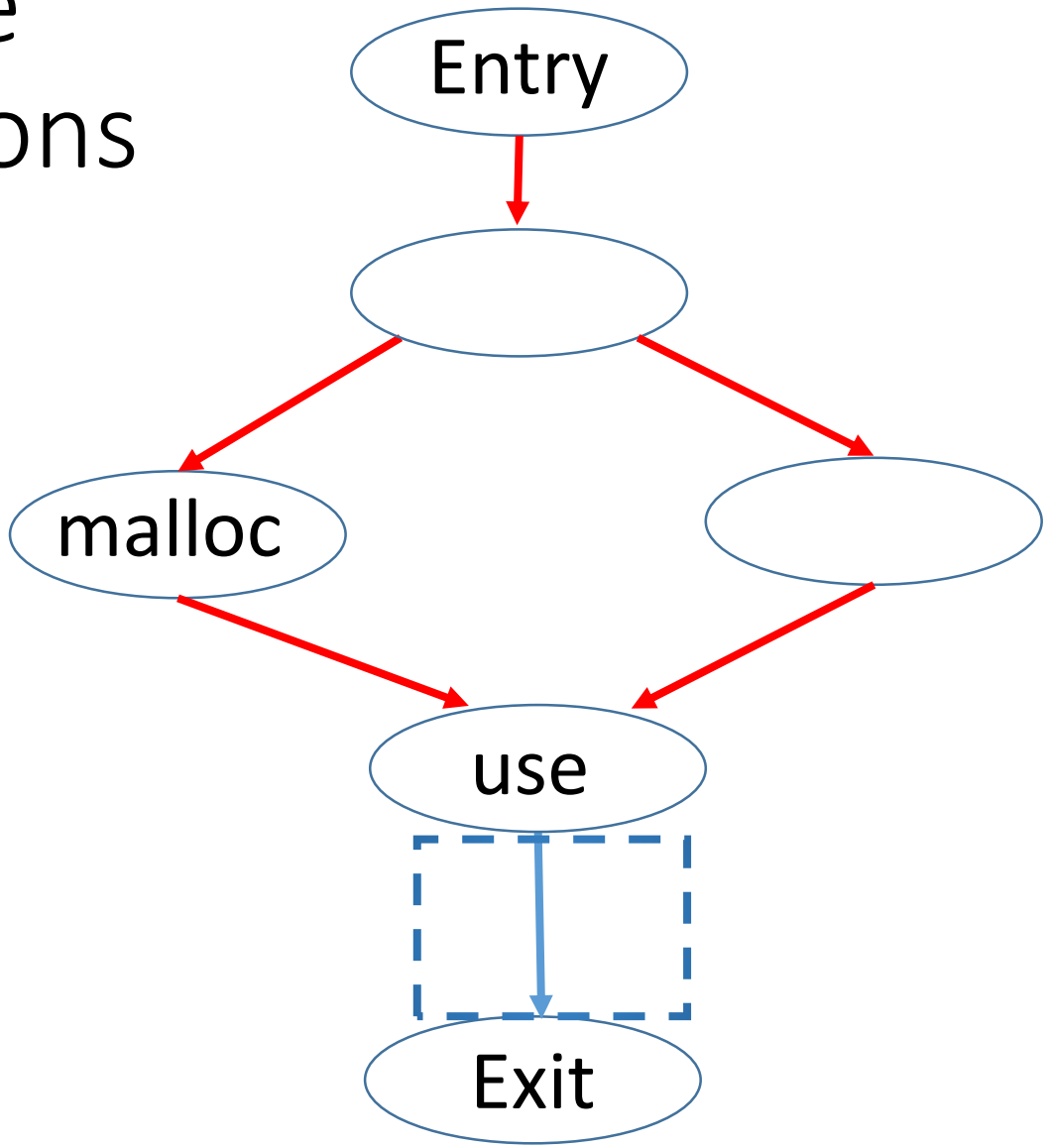
Multiple Allocations



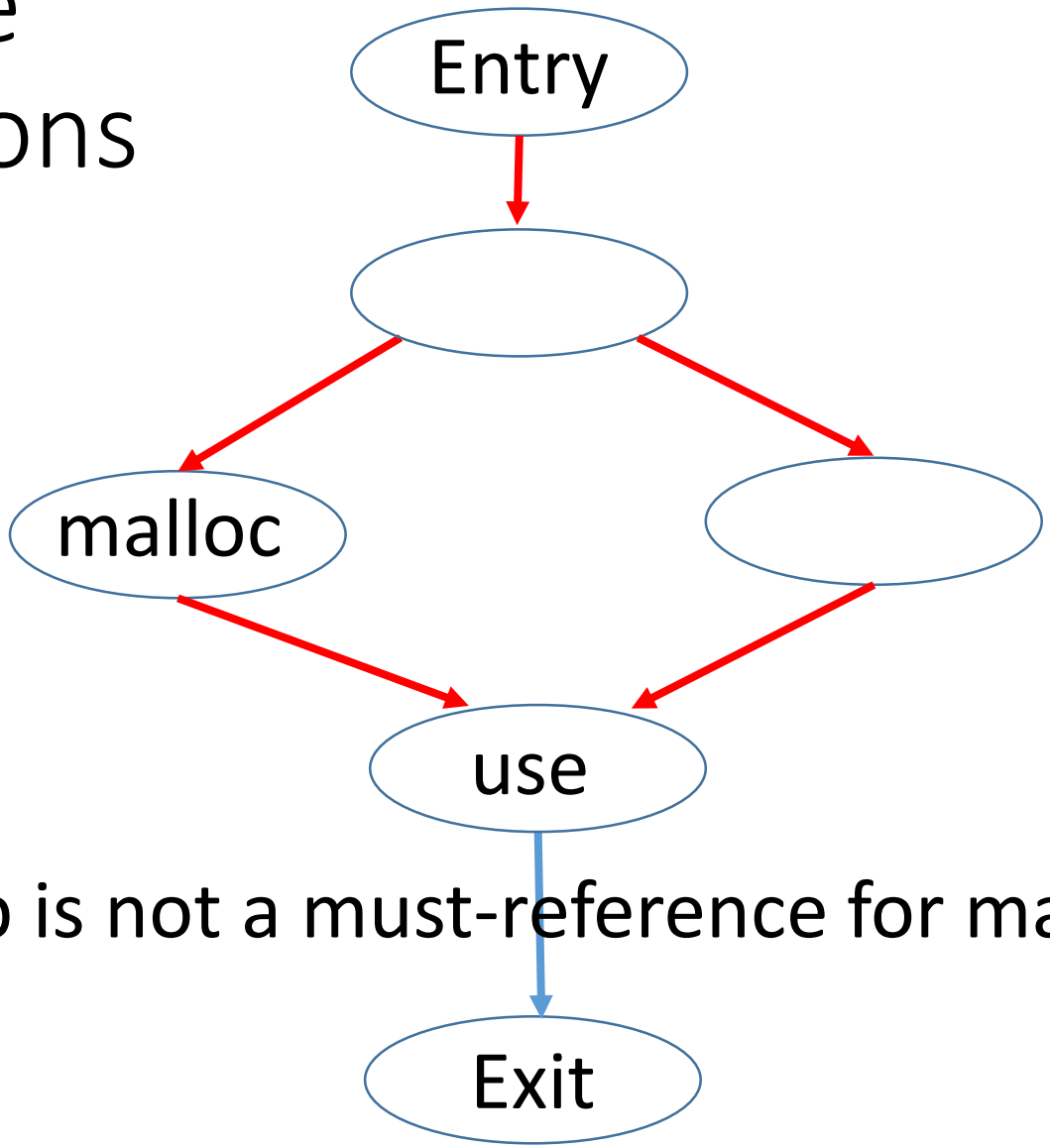
Multiple Allocations



Multiple Allocations

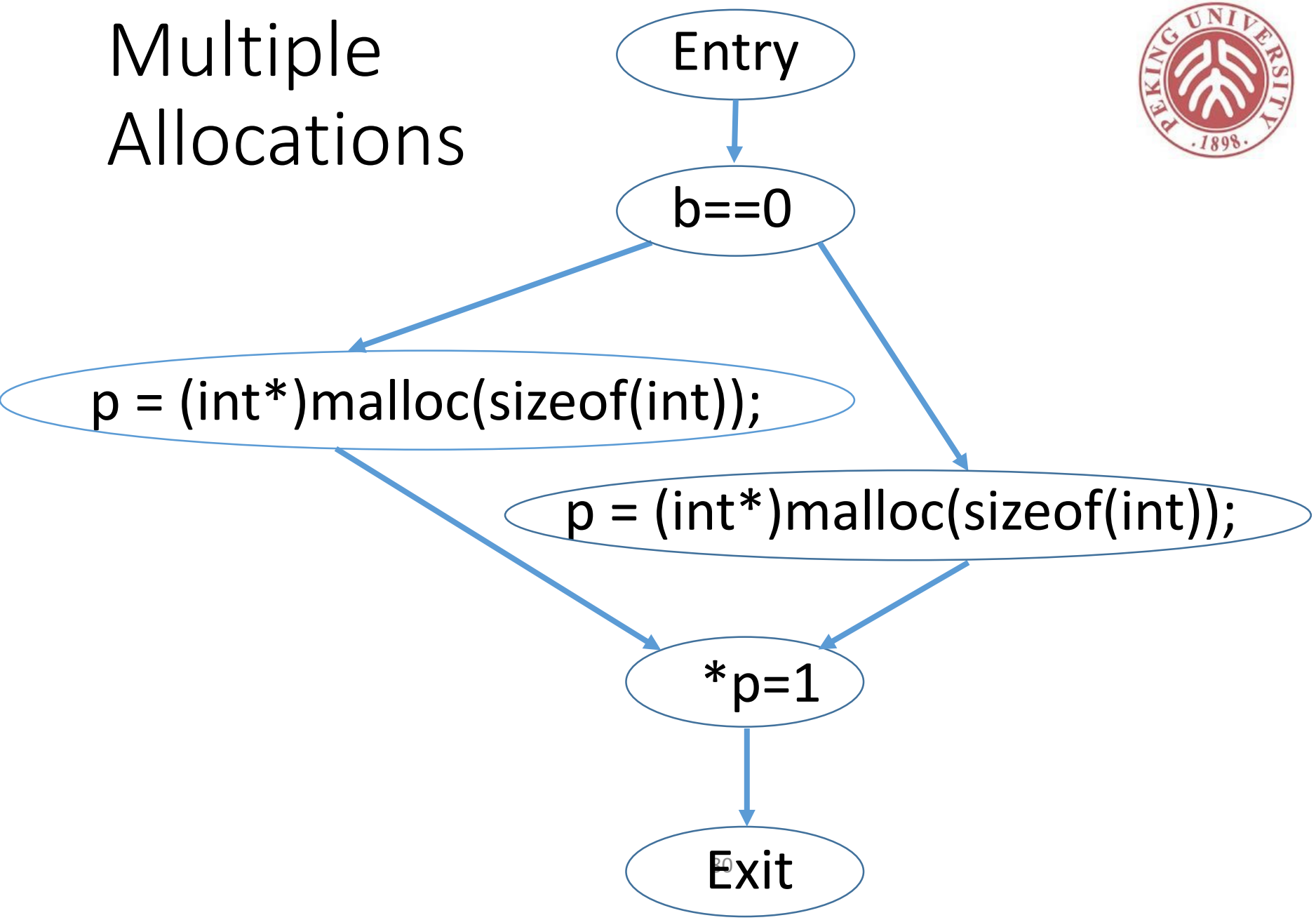


Multiple Allocations

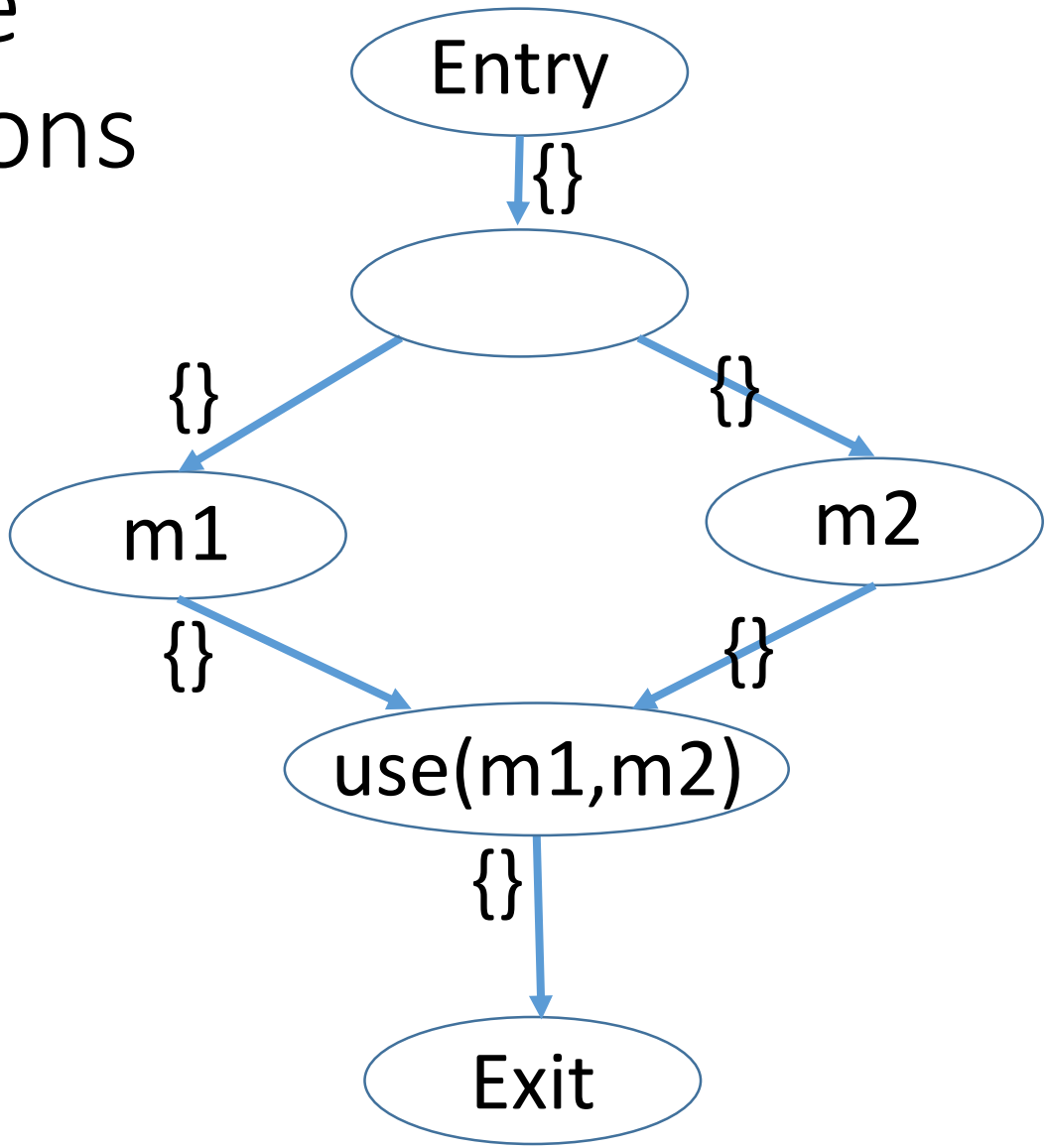


p is not a must-reference for malloc

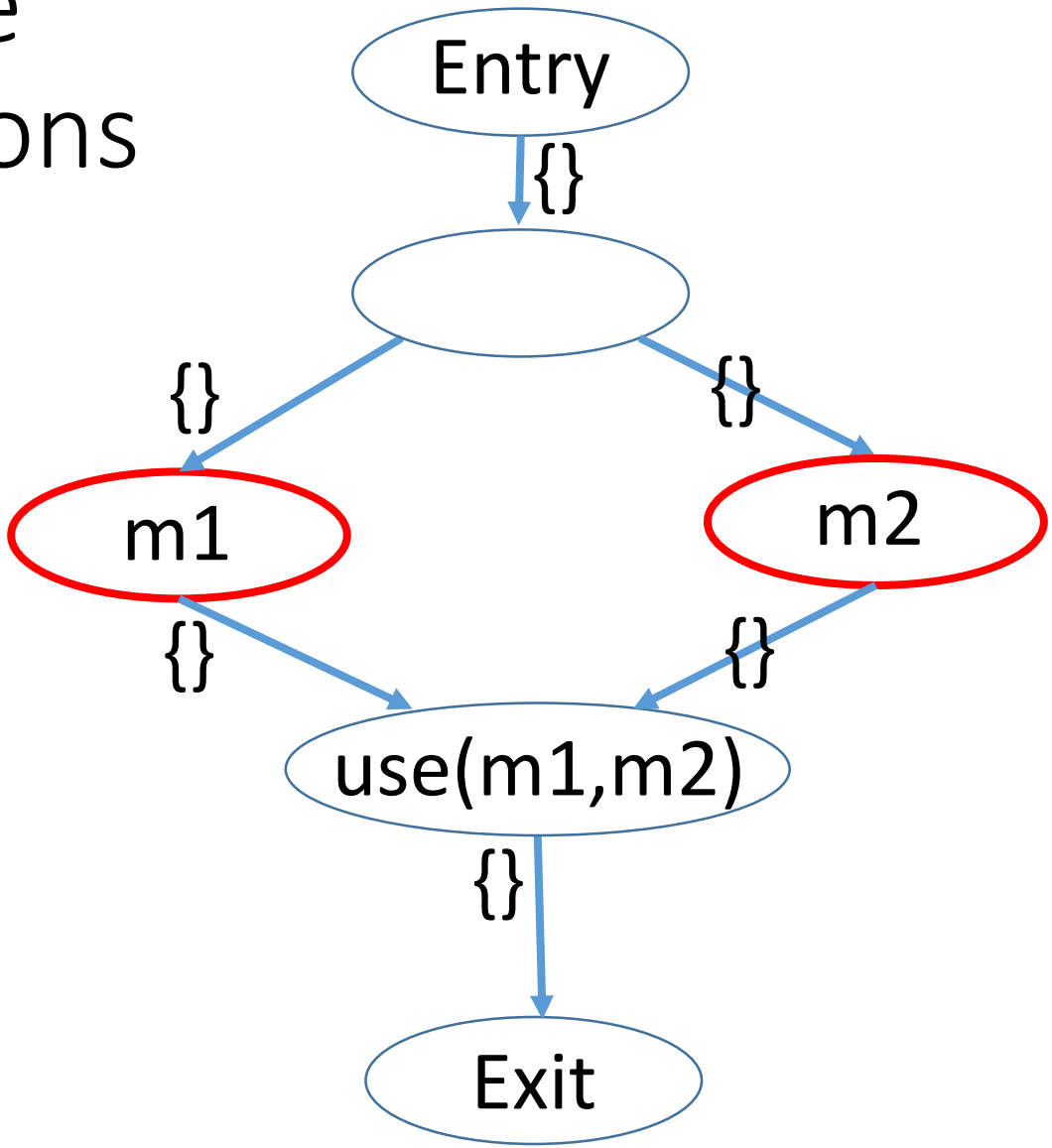
Multiple Allocations



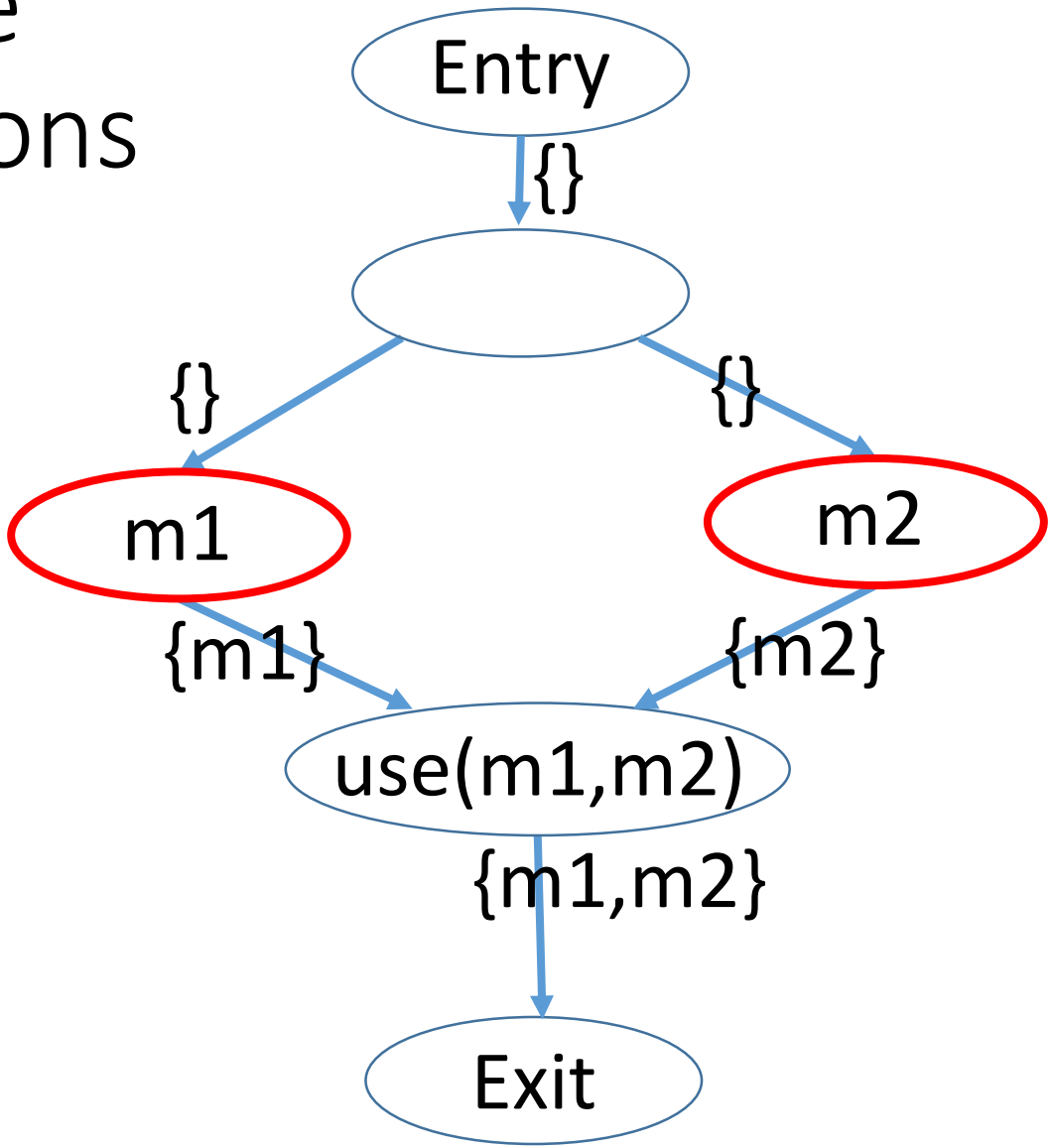
Multiple Allocations



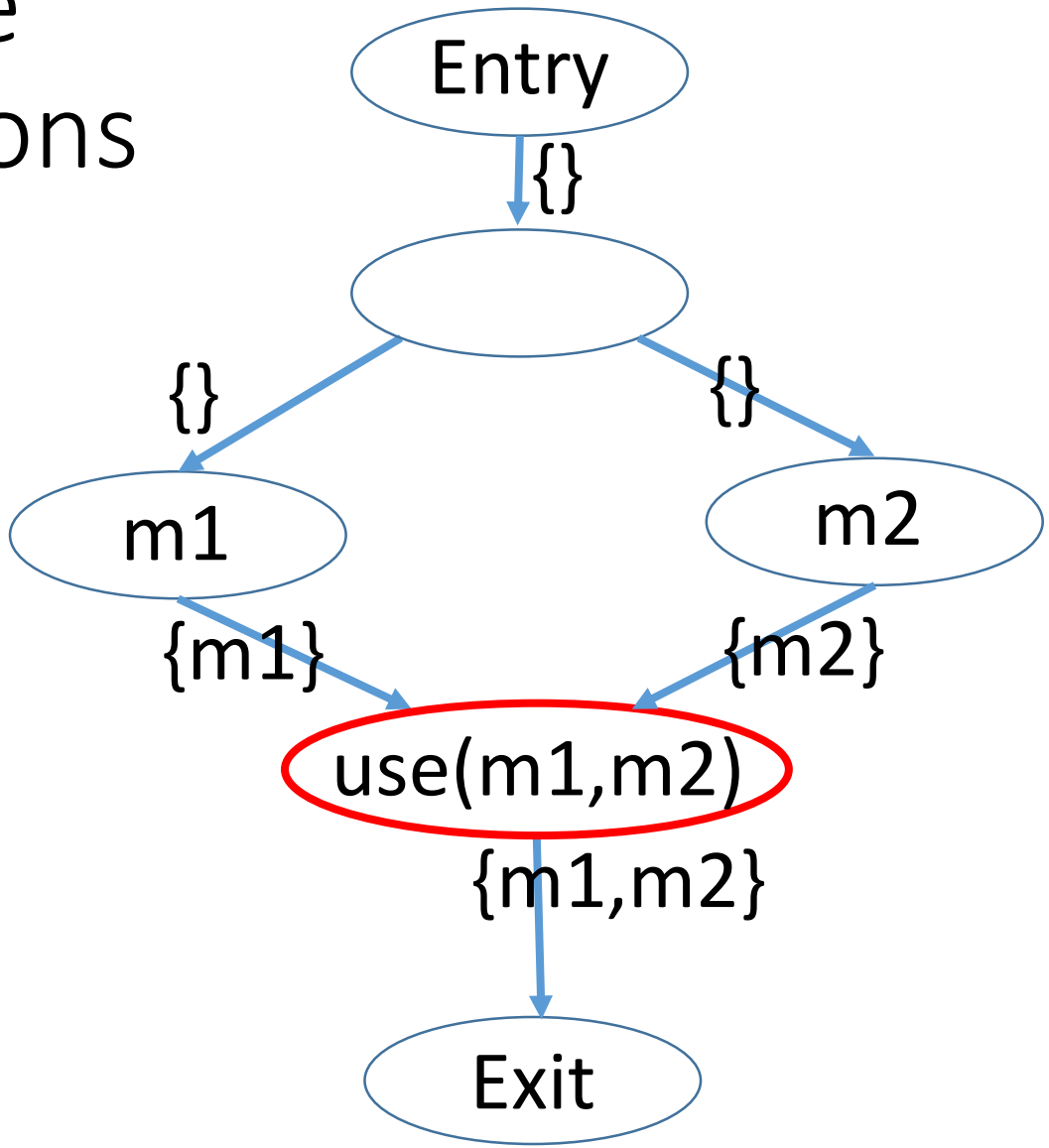
Multiple Allocations



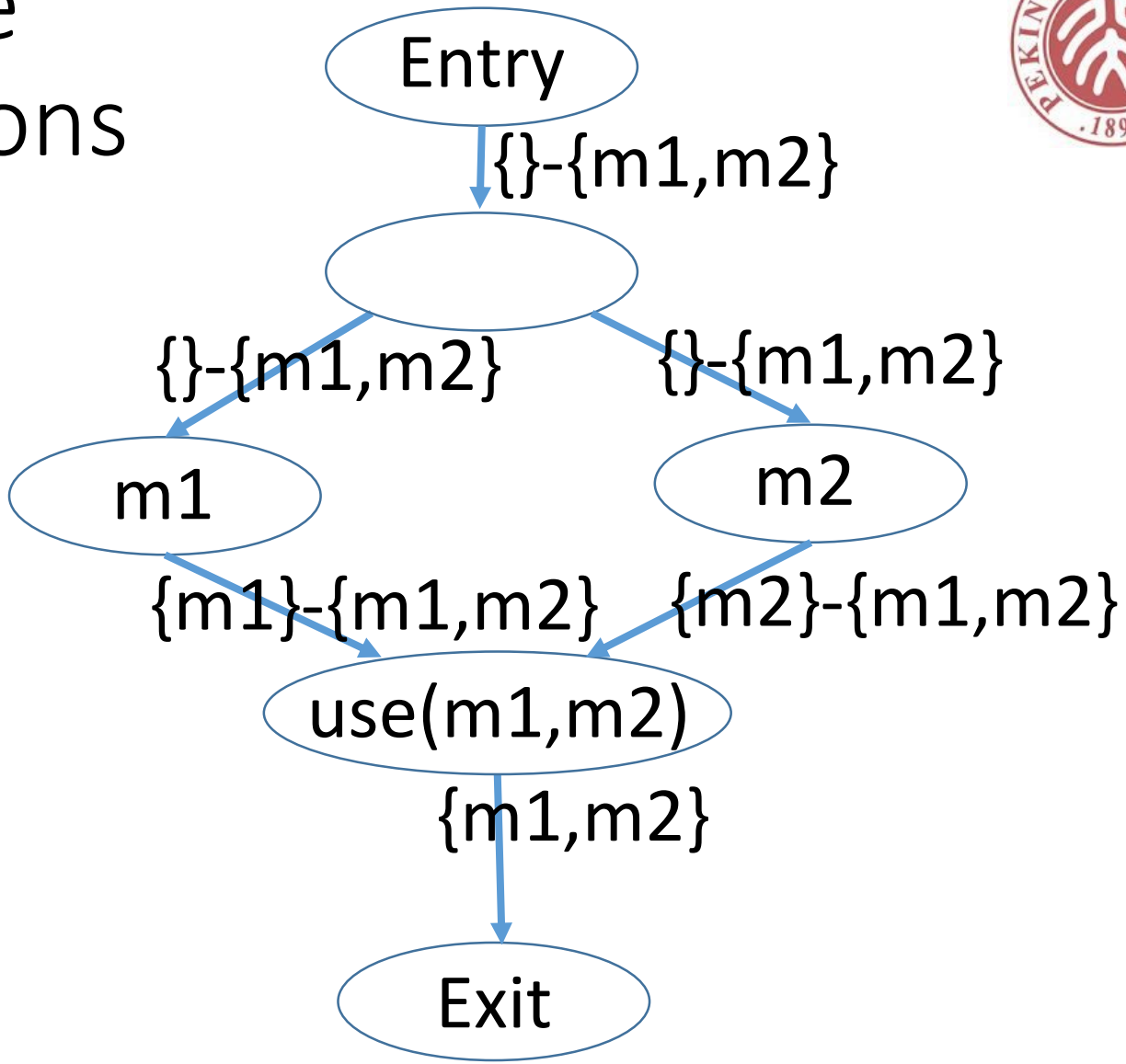
Multiple Allocations



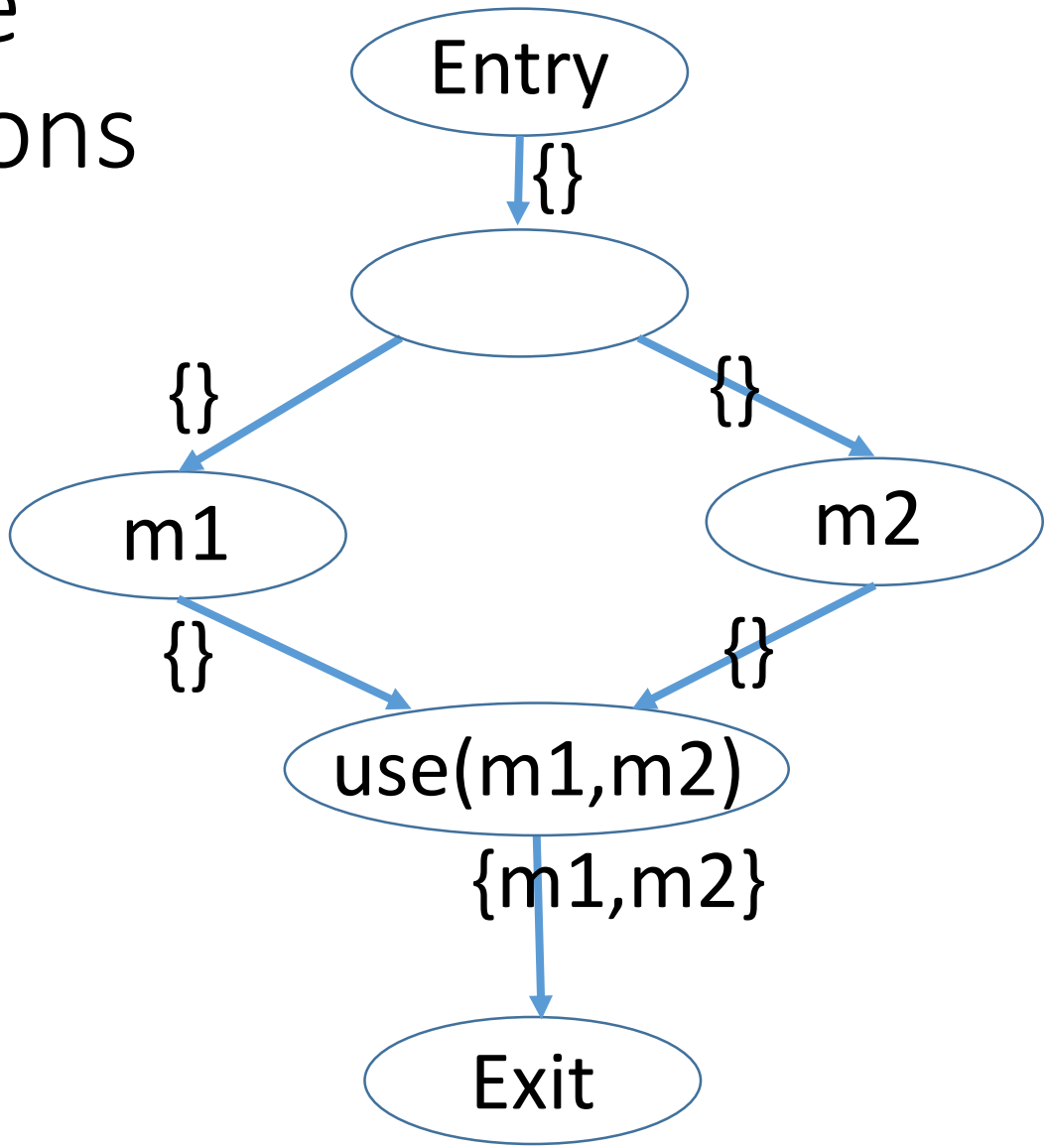
Multiple Allocations



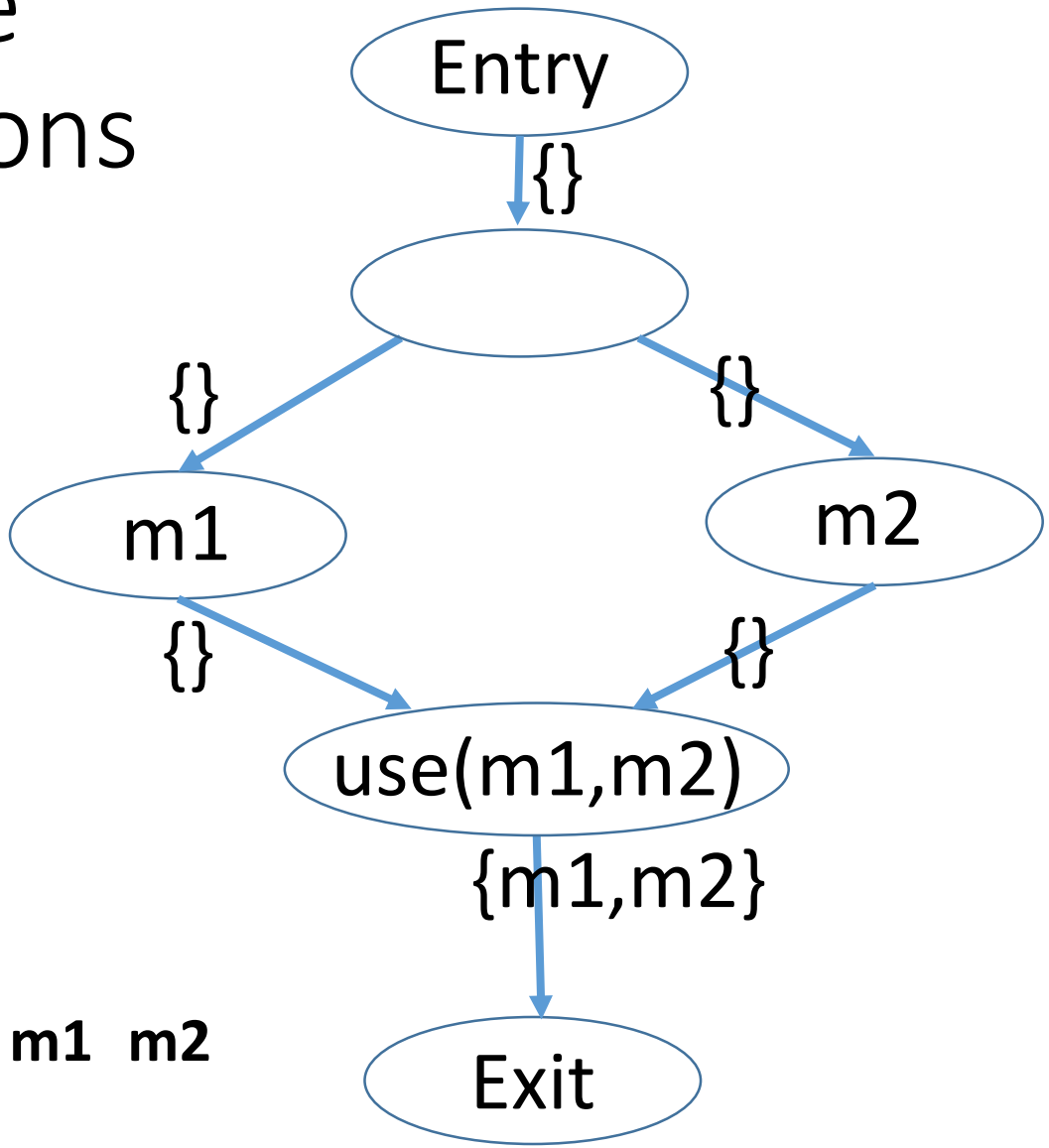
Multiple Allocations

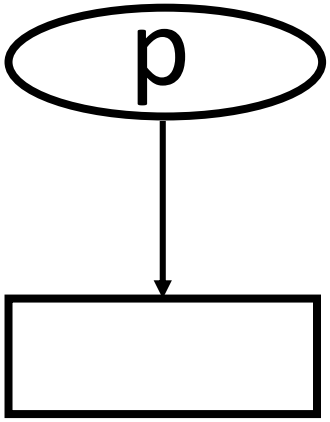


Multiple Allocations



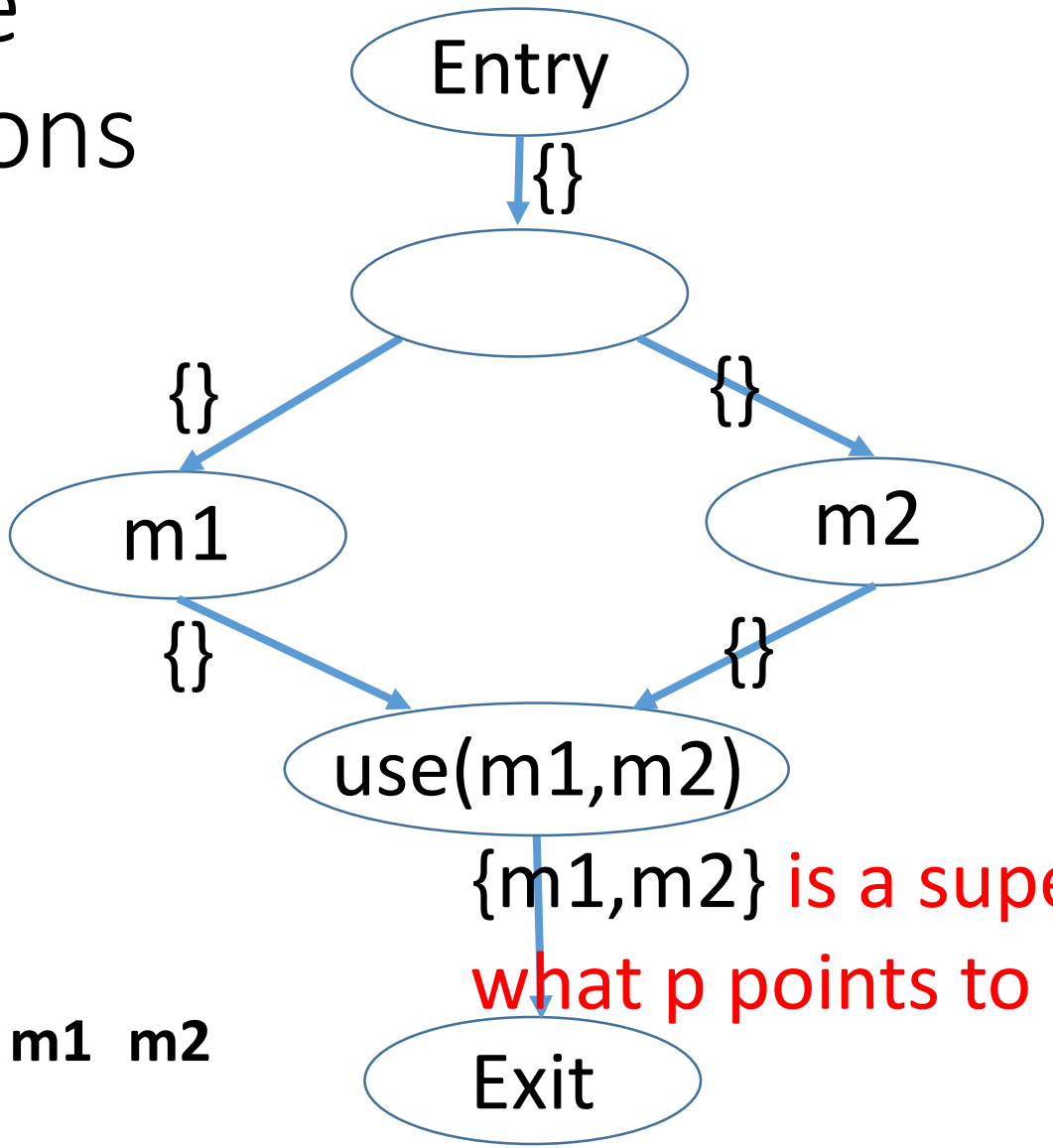
Multiple Allocations

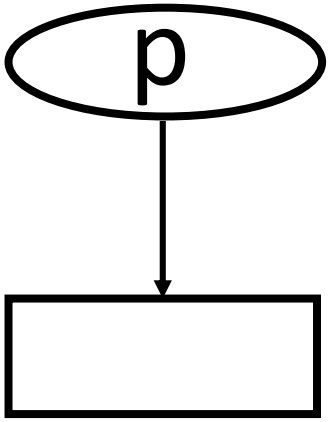




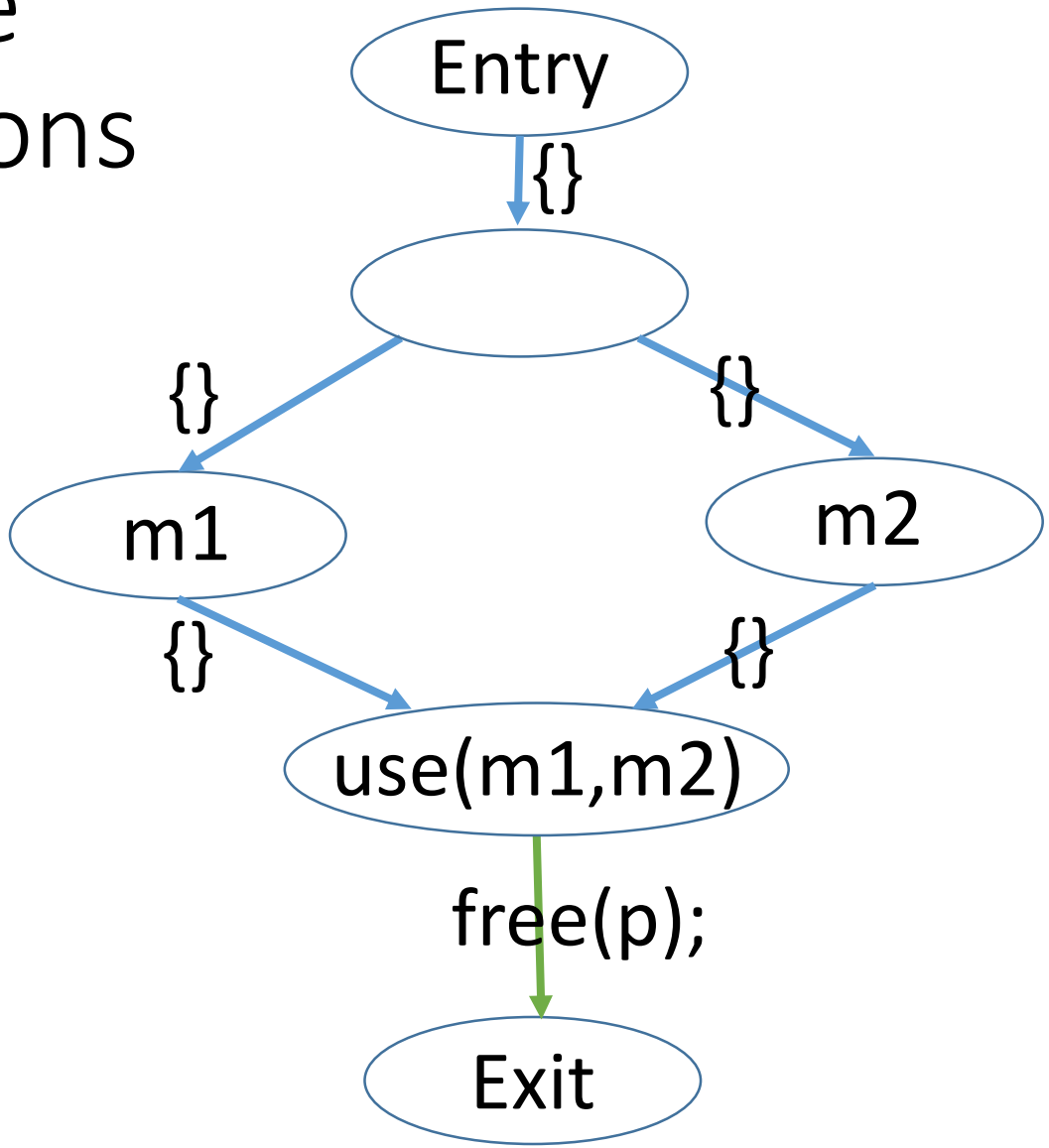
m1 m2

Multiple Allocations

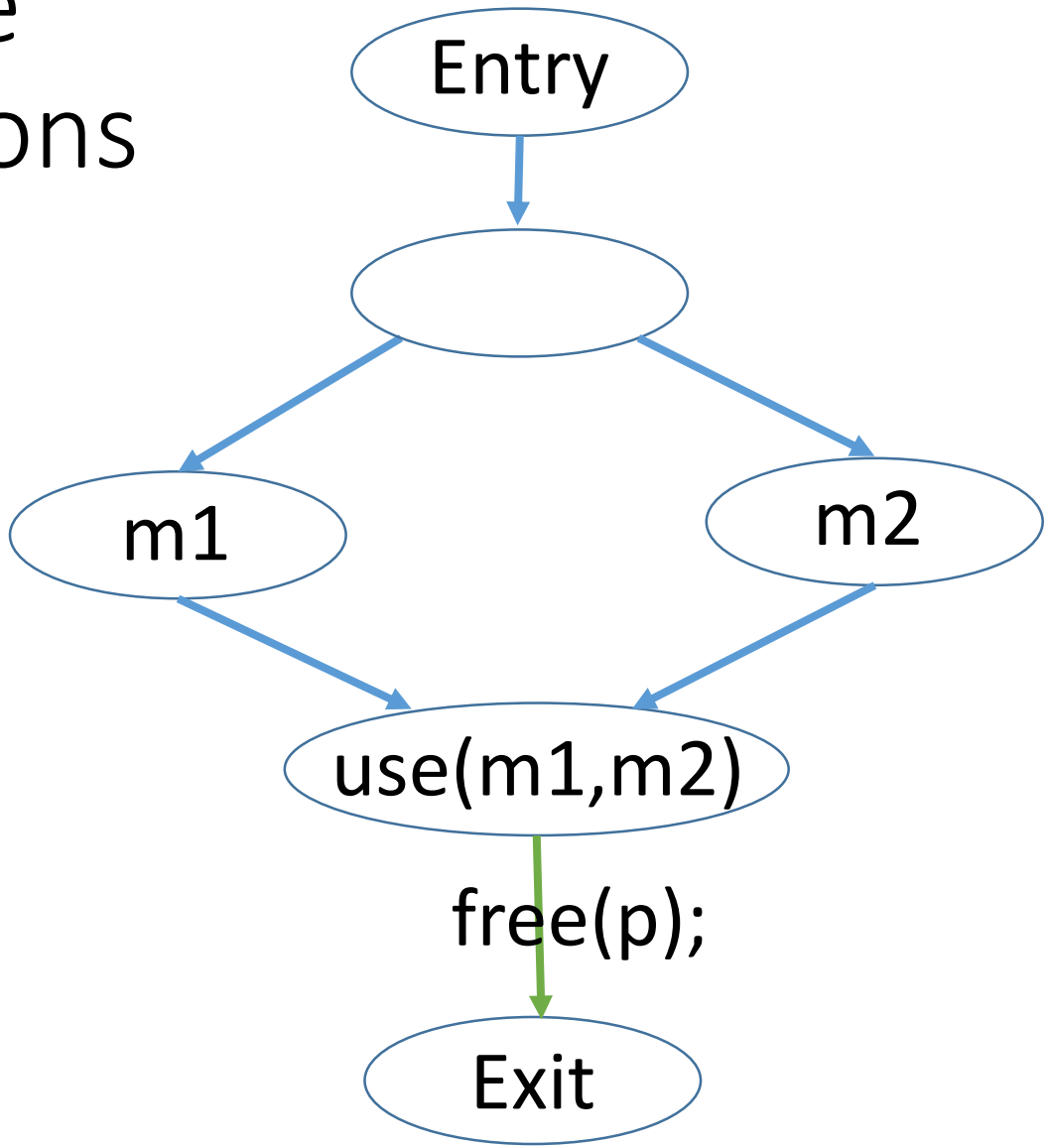




Multiple Allocations



Multiple Allocations



Loops

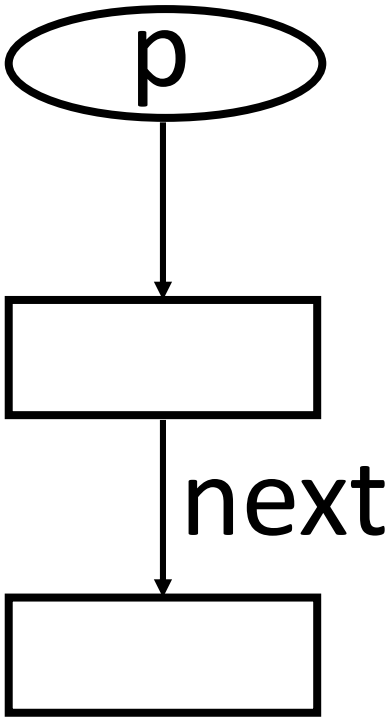
- In loops, unsafe locations are propagated all around
- Solution: Extract the loop

```
void f(bool b){
  for (int i = 0; i < 10; ++i){
    int p=(int*)malloc(sizeof(int));
    if (b){
      free(p);
    }
    else{
      p=(int*)malloc(sizeof(int)*2);
    }
  }
}
```

Heap Structure

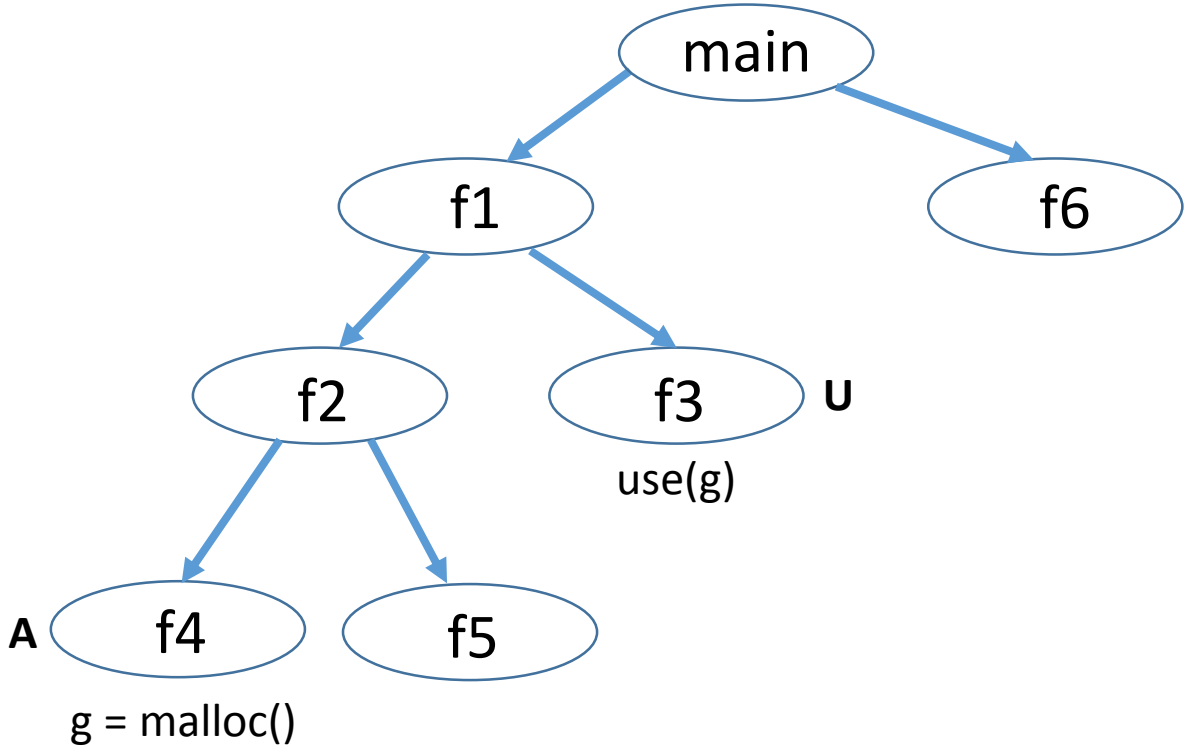
- References for heap structure
- Solution: based on points-to graph

```
p->next =(int*)malloc(sizeof(int));
if (b==0){
   free(p->next);
}
else{
   p->next=1;
   b=0;
}
```



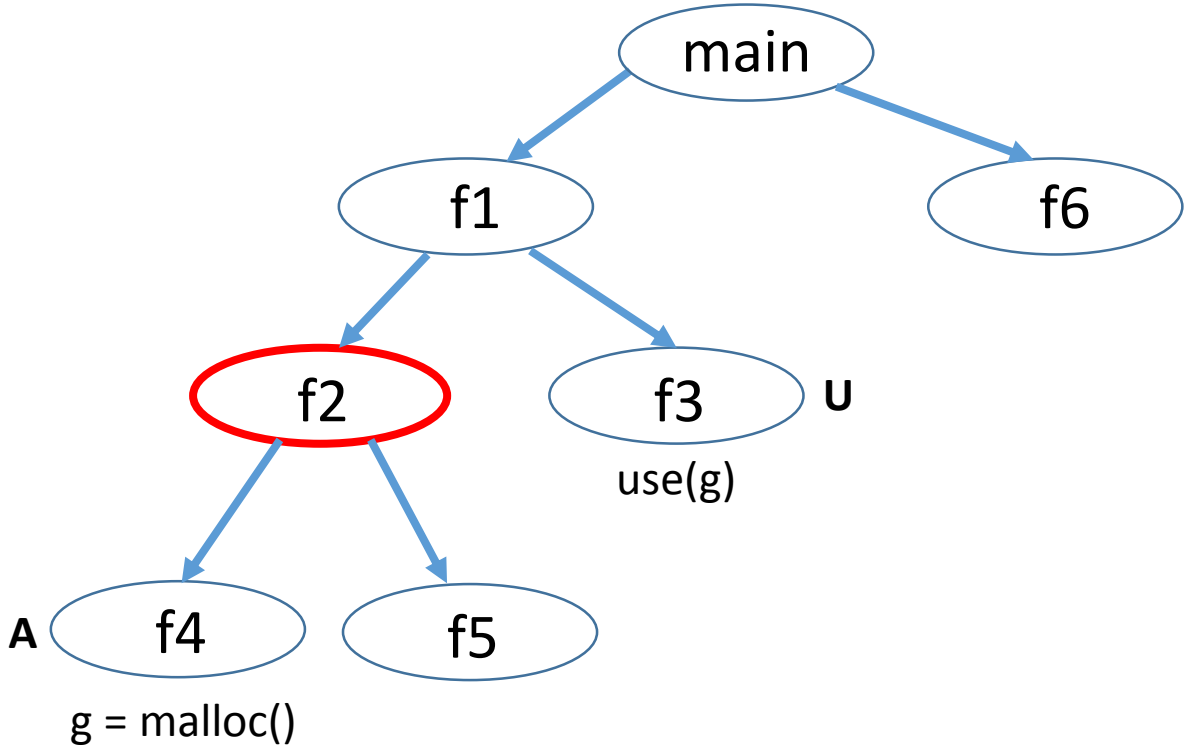
Global Variables

- Global variables escape from every function
- Solution: based on call graph



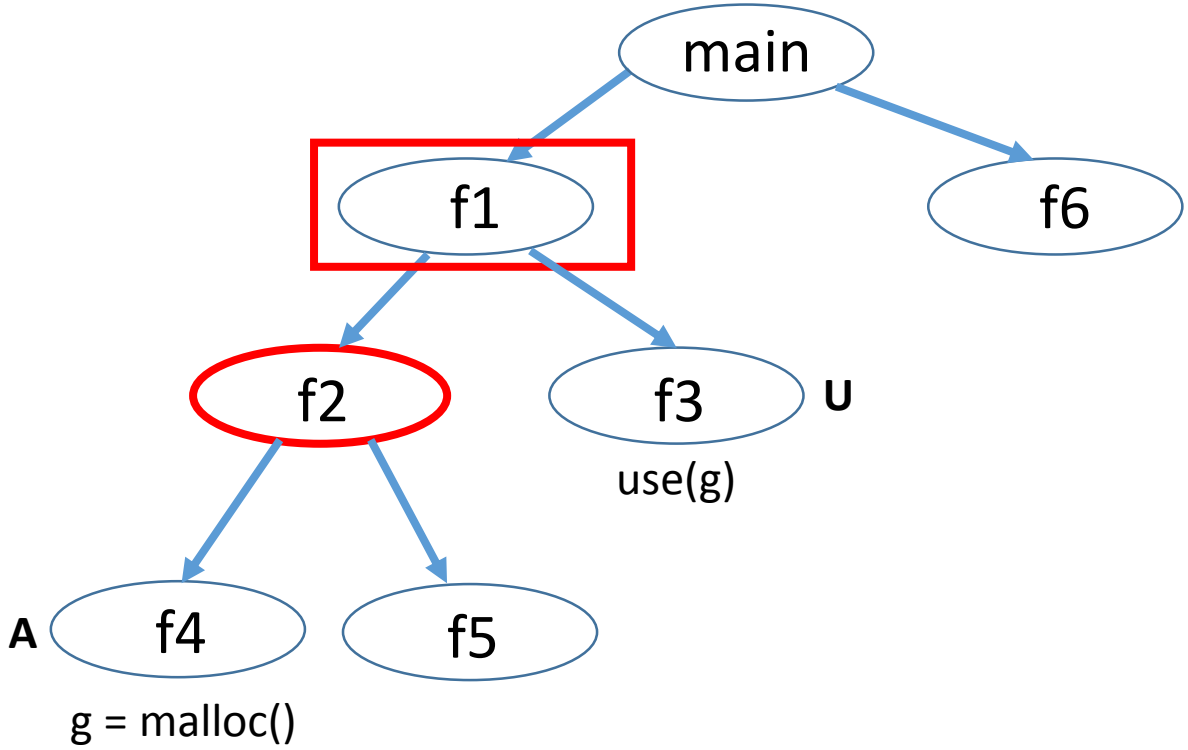
Global Variables

- Global variables escape from every function
- Solution: based on call graph



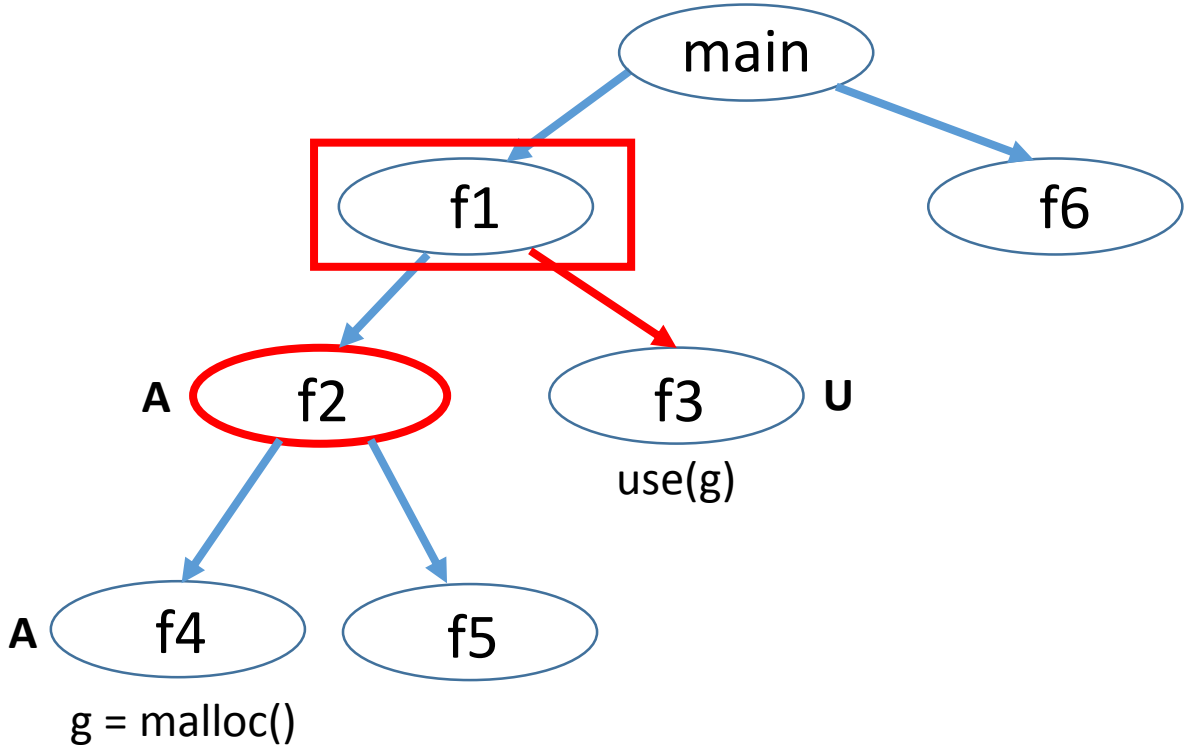
Global Variables

- Global variables escape from every function
- Solution: based on call graph



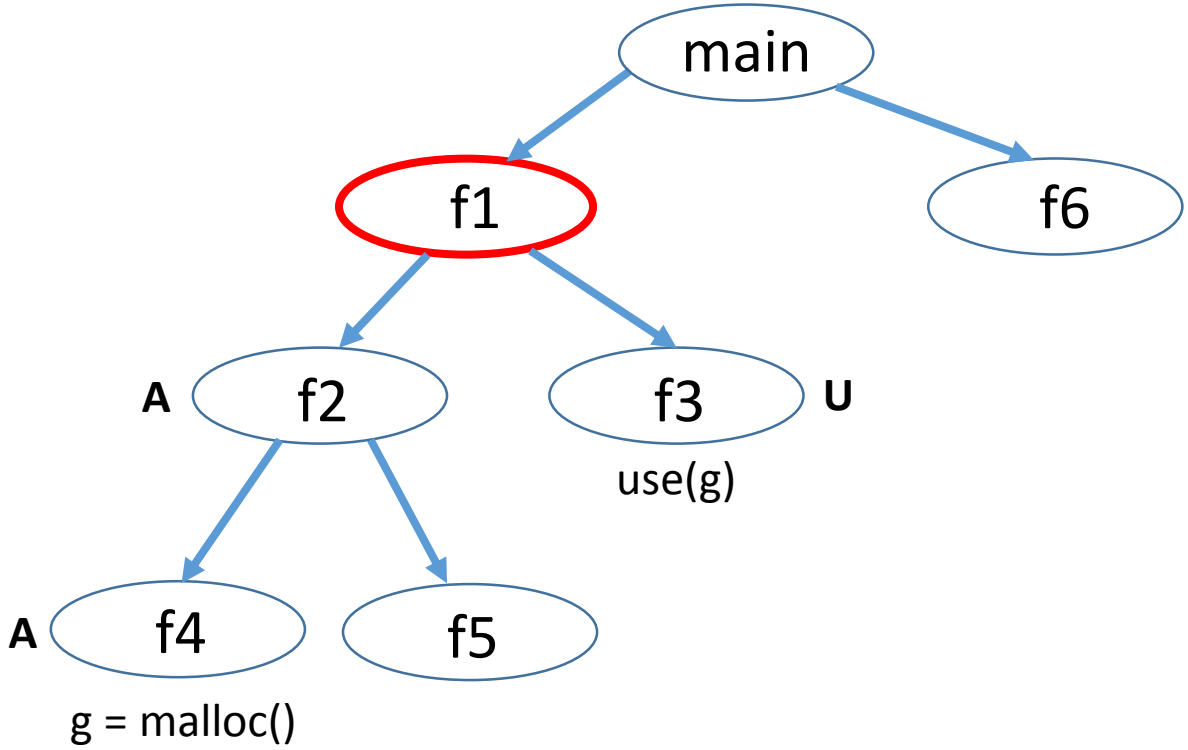
Global Variables

- Global variables escape from every function
- Solution: based on call graph



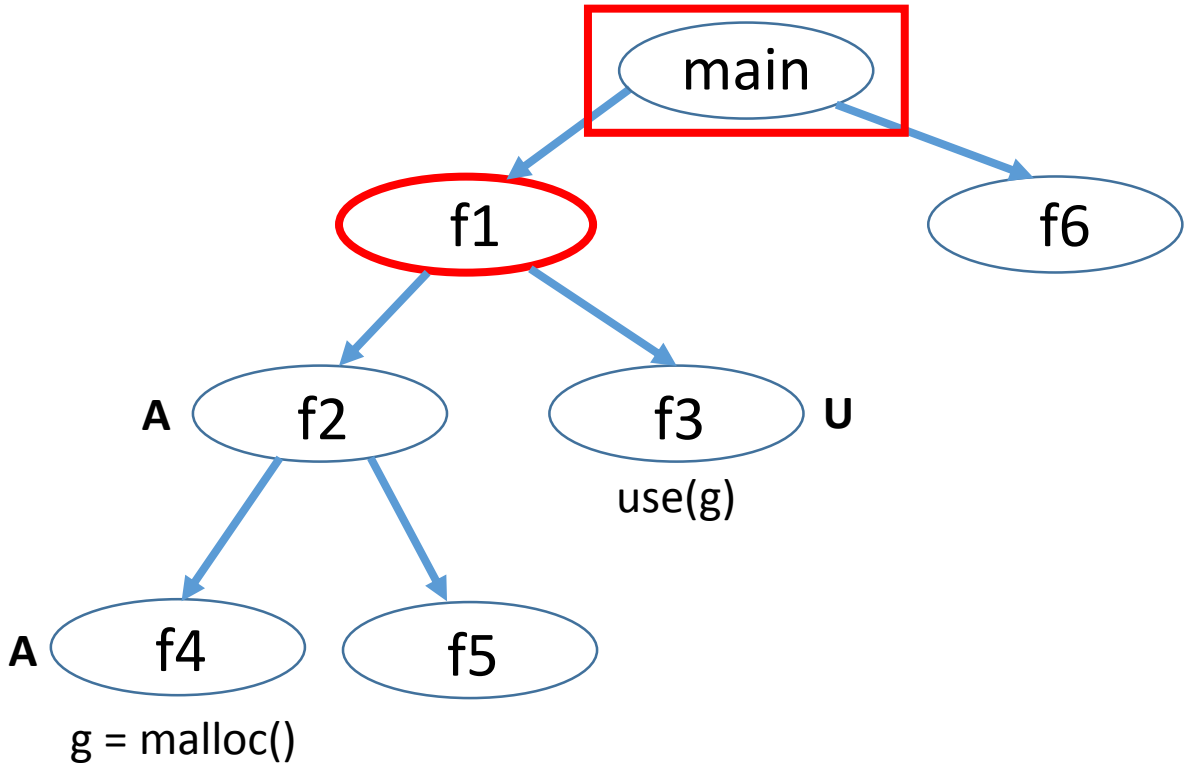
Global Variables

- Global variables escape from every function
- Solution: based on call graph



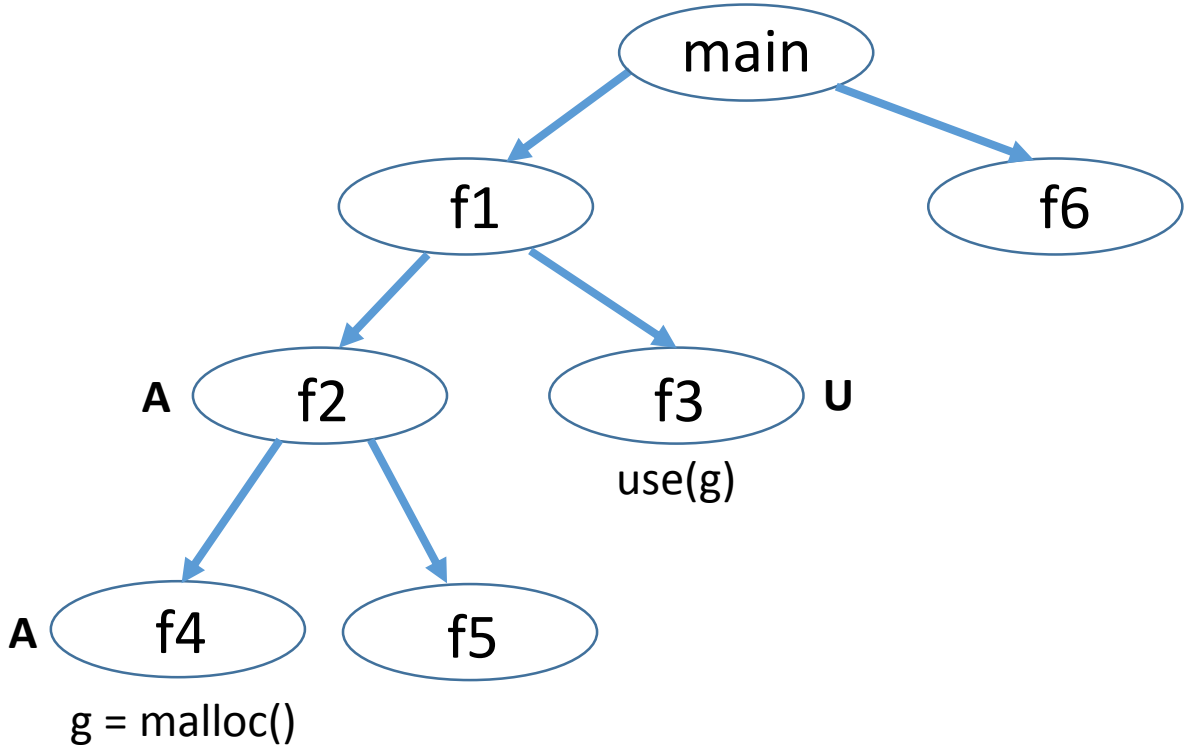
Global Variables

- Global variables escape from every function
- Solution: based on call graph



Global Variables

- Global variables escape from every function
- Solution: based on call graph



Implementation

- Implemented on LLVM
- Pointer analysis: DSA
- Open-source tool available
  - http://sei.pku.edu.cn/~gaoqing11/leakfix

Experimental results

- Benchmark: SPEC2000

Program	Size (Kloc)	#Func	#Allocation
art	1.3	44	11
equake	1.5	45	29
mcf	1.9	44	3
bzip2	4.6	92	10
gzip	7.8	128	5
parser	10.9	342	1
ammp	13.3	197	37
vpr	17.0	290	2
crafty	18.9	127	12
twolf	19.7	209	2
mesa	49.7	1124	67
vortex	52.7	941	8
perlbmk	58.2	1094	4
gap	59.5	872	2
gcc	205.8	2271	53

Existing memory-leak detectors

Program	LC	Fastcheck	SPARROW	SABER
art	1(0)	1(0)	1(0)	1(0)
equake	0(0)	0(0)	0(0)	0(0)
mcf	0(0)	0(0)	0(0)	0(0)
bzip2	1(1)	0(0)	1(0)	1(0)
gzip	1(2)	0(0)	1(4)	1(0)
parser	0(0)	0(0)	0(0)	0(0)
ammp	20(4)	20(0)	20(0)	20(0)
vpr	0(0)	0(1)	0(9)	0(3)
crafty	0(0)	0(0)	0(0)	0(0)
twolf	0(0)	2(0)	5(0)	5(0)
mesa	2(0)	0(2)	9(0)	7(4)
vortex	0(26)	0(0)	0(1)	0(4)
perlbmk	1(0)	1(3)	N/A[1]	8(4)
gap	0(1)	0(0)	0(0)	0(0)
gcc	N/A[1]	35(2)	44(1)	40(5)
total	26(34)	59(8)	81(15)	83(20)

Fixing results

Program	#Fixed	#Maximum Detected	Percentage(%)	#Fixes	#Useless Fixes
art	0	1	0	0	0
equake	0	0	N/A	0	0
mcf	0	0	N/A	0	0
bzip2	1	1	100	1	0
gzip	1	1	100	1	0
parser	0	0	N/A	0	0
ammp	20	20	100	36	0
vpr	0	0	N/A	0	0
crafty	0	0	N/A	0	0
twolf	0	5	0	0	0
mesa	0	9	0	0	0
vortex	0	0	N/A	0	0
perlbmk	1	8	13	1	0
gap	0	0	N/A	0	0
gcc	2	44	5	2	0
total	25	89	28	41	0

Time consumption

Program	Compiling and Linking (sec)	LeakFix (sec)		Total (sec)	Percentage(%)
		Pointer Analysis	Fix Analysis		
art	0.20	0.02	0.01	0.23	13.0
equake	0.21	0.01	0.02	0.24	12.5
mcf	1.19	0.02	0.01	1.22	2.5
bzip2	0.36	0.03	0.02	0.41	12.2
gzip	1.31	0.04	0.04	1.39	5.8
parser	1.68	0.18	0.07	1.93	13.0
ammp	2.98	0.12	0.37	3.47	14.1
vpr	2.51	0.20	0.31	3.02	16.9
crafty	3.53	0.16	0.23	3.92	9.9
twolf	6.22	0.27	0.20	6.69	7.0
mesa	9.36	5.36	5.97	20.69	54.8
vortex	9.00	0.94	0.83	10.77	16.4
perlbmk	9.50	18.20	39.20	66.90	85.8
gap	6.03	7.36	22.36	35.75	83.1
gcc	10.99	31.76	95.81	142.99	89.2

Leaks failed to fix

- Flow-insensitive pointer analysis

```
void f(bool b){
  int p=(int*)malloc(sizeof(int));
  if (b){
    free(p);
  }
  else{
    *p=1;
  }
  int a;
  p=&a;
}
```

Leaks failed to fix

- Leak pattern in gcc

```
void f(bool b){
  char *p= "";
  if (b){
    p=(char*)malloc(sizeof(char)*10);
  }
  use(p);
}
```

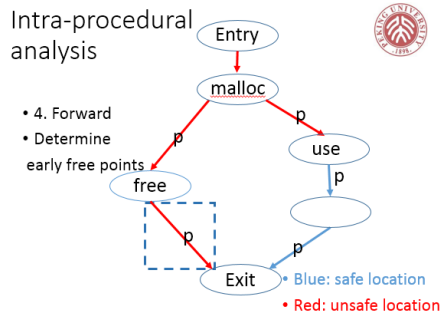
Summary

A safe fix

- The fix is after an allocation
- There is no double free
- There is no use after free
- There is an expression that always points to the allocation

Safe fix definition

Intra-procedural analysis



Intra-procedure analysis

Building summaries

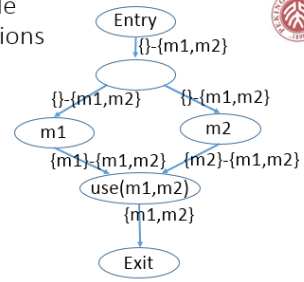
- Function type for each memory chunk
  - Allocation（A）
  - Free（F）
  - Use（U）
  - Escape (E)

```
void h(int *p){
  *p=1;
}
```

```
void g(int *p){
  free(p);
}
```

Inter-procedure analysis

Multiple Allocations



Multiple allocation

Loops

```
void f(bool b){
  for (int i = 0; i < 10; ++i){
    int p=(int*)malloc(sizeof(int));
    if (b){
      free(p);
    }
    else{
      p=(int*)malloc(sizeof(int)*2);
    }
  }
}
```

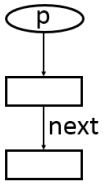
- Extract the loop

Loop

Heap Structure

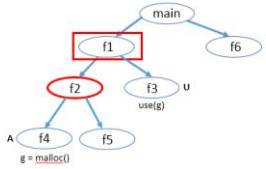
- Find references for heap structure
- Solution: based on points-to graph

```
p->next =(int*)malloc(sizeof(int));
if (b==0){
  free(p->next);
}
else{
  p->next=1;
  b=0;
}
```



Heap structure

Global Variables



Global variable

Fixing results

Program	#Fixed	#Maximum Detected	Percentage(%)	#Fixes	#Useless Fixes
art	0	1	0	0	0
equake	0	0	N/A	0	0
mcf	0	0	N/A	0	0
bzip2	1	1	100	1	0
gzip	1	1	100	1	0
parser	0	0	N/A	0	0
ammp	20	20	100	36	0
vpr	0	0	N/A	0	0
crafty	0	0	N/A	0	0
twolf	0	5	0	0	0
mesa	0	9	0	0	0
vortex	0	0	N/A	0	0
perlbmk	1	8	13	1	0
gap	0	0	N/A	0	0
gcc	2	44	5	2	0
total	25	89	28	41	0

Evaluation

Thank you !