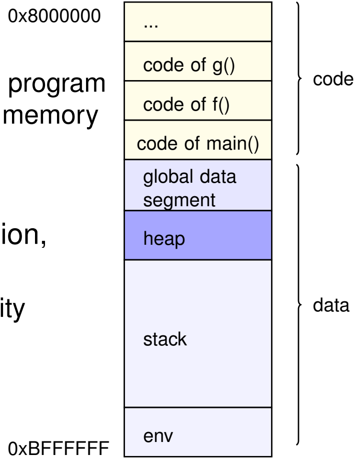
Runtime Environment

Compiler's Role for Program Execution

- □ When a program is invoked
  - ➢ The OS allocates memory for the program
  - ➢ The code is loaded into the main memory
  - ➢ Jump to the entry of 'main()'

- □ To support smooth program execution, a compiler generates the code for
  - ✓ completing the desired functionality
  - ☞ managing runtime memory



0x8000000

...

code of g()

code of f()

code of main()

global data segment

heap

stack

env

0xBFFFFFF

code

data

What Runtime Support?

- At runtime, the code needs to
  - allocate/deallocate storage in stack/heap area
  - access variables from the allocated storage
  - enforce the language semantics e.g. static/dynamic scoping, ...
- The core problem is
  - identify the runtime address of a given name e.g. variable, proc name, ...
- How to manage?
  - Generate appropriate code to finish the task

Types of Management

- Static data management
  - ➢ Variables are stored in statically allocated area
  - ➢ Addresses are known at compile time
    e.g. global variables in C, all variables in Fortran
- Stack data management
  - ➢ Allocates storage dynamically for each procedure invocation
    e.g. allocate storage for a recursive function 3 times if it is invoked 3 times
- Heap data management
  - ➢ Allocates storage for objects that live across procedure invocations
    e.g. pointer-objects, co-routines, tasks

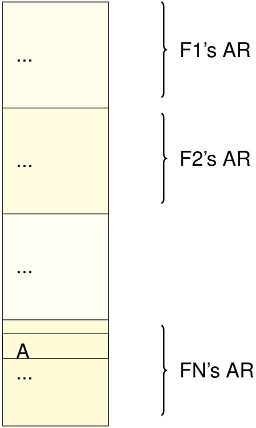
Static Storage Management

- Layout storage at compile time and the name/address binding will not change at runtime
- Case study 1: Fortran's data allocation
  - Allocation strategy
    - Given a program with many functions/procedures, FORTRAN first determines their order
    - Allocate variables within each function/procedure (we know how to do it)
  - Limitations
    - Cannot implement recursion, reentrant functions
    - Require maximum storage even though some functions are not activated at runtime
  - Advantages
    - Fast, less runtime overhead
    - Easy to manage

Name Address Translation

- A list of AR (activation record) with their sizes known at compile time

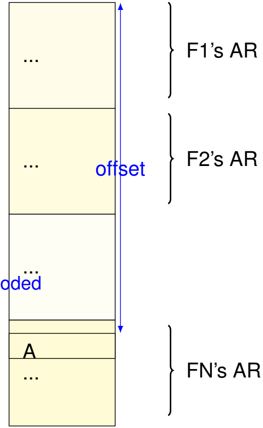
```
FUNCTION F1(...)
   ...
END
FUNCTION F1(...)
   ...
END
  ...
FUNCTION FN(...)
   A = ...
   ...
END
```



Name Address Translation

- A list of AR (activation record) with their sizes known at compile time

```
FUNCTION F1(...)
   ...
END
FUNCTION F1(...)
   ...
END
  ...
FUNCTION FN(...)
   A = ...  ☞a tuple (Global_Base, offset)
   ...      ...
END         // variable access can be hard-coded
            store R1, 0x10002008;
```

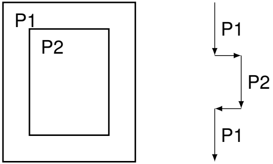


Stack Based Storage Management

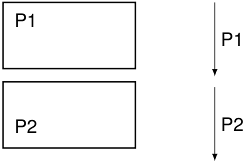
- Allocation strategy
  - Organize all locals of a procedure in one AR (activation record) unit
  - Manage ARs in a stack
  - A new AR instance is allocated when a function is called (or called again)
  - The corresponding AR is removed when a function finishes
- Hardware support
  - Stack pointer (SP) register
    - SP records the top of the stack
    - Allocation/de-allocation can be done by incrementing/decrementing SP
  - Frame pointer (FP) register
    - FP assists address mapping within AR

More About Stack-based Storage Management

- Two types of block structured languages
  - Flat nesting level — two levels: locals and globals
  - Fully block-structured language — three types: locals, non-locals, and globals
- Lifetime and scope
  - Static scoping rule — static concept
  - Lifetime is dynamic concept
    - start: when the storage is allocated
    - end: when the storage is deallocated



Nested lifetime



Disjoint lifetime

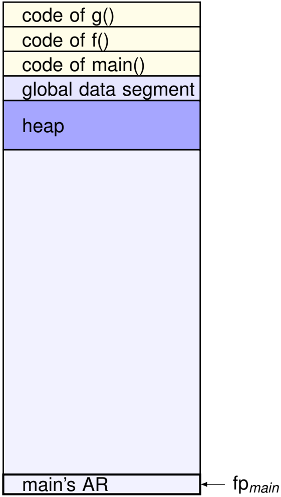
Discussion of Stack-based Management

- Advantages:
  - Support reentrant functions
  - Support recursive functions
  - Allocate storage as needed
- Disadvantages:
  - Management overhead
  - Security concerns
    - Buffer overflow attack (BOA)

Example

- How does it work?

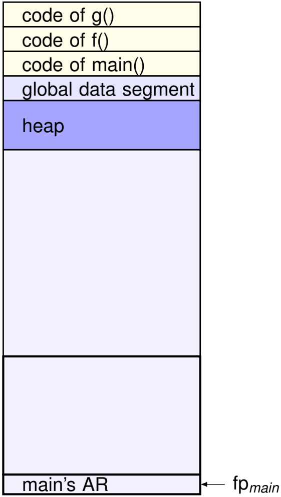
```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



Example

- How does it work?

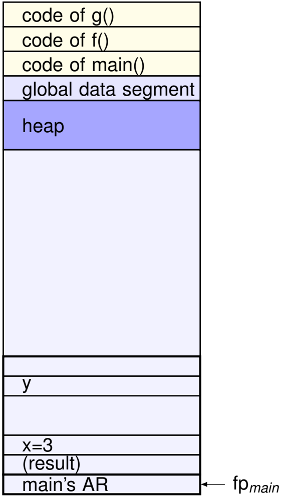
```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



Example

- How does it work?

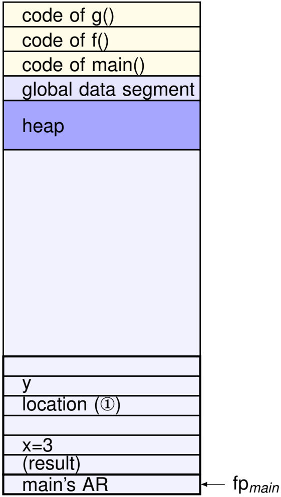
```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



Example

- How does it work?

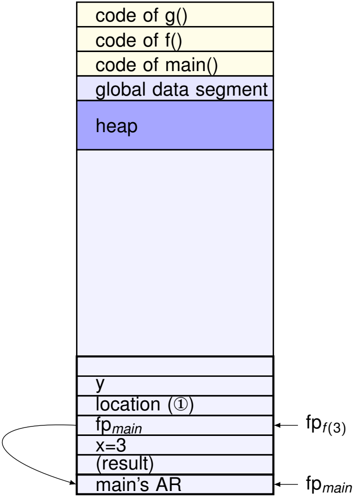
```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



Example

- How does it work?

```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



code of g()

code of f()

code of main()

global data segment

heap

y

location (①)

fp_{main}

x=3

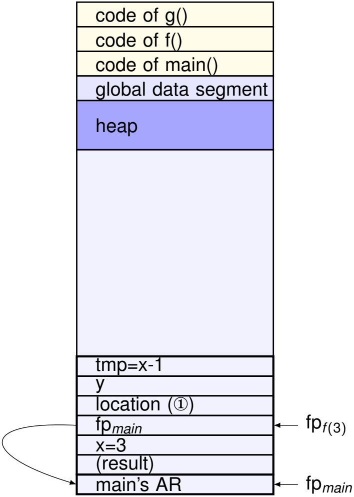
(result)

main's AR

Example

- How does it work?

```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



code of g()

code of f()

code of main()

global data segment

heap

tmp=x-1

y

location (①)

fp_{main}

x=3

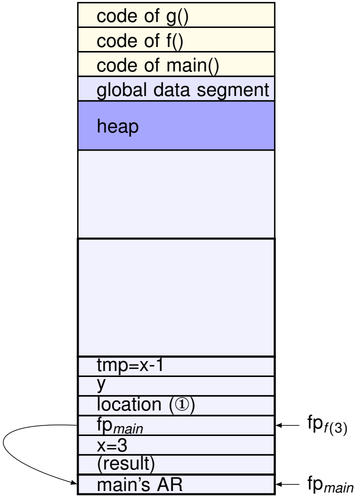
(result)

main's AR

Example

- How does it work?

```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



code of g()

code of f()

code of main()

global data segment

heap

tmp=x-1

y

location (①)

fp$_{main}$

x=3

(result)

main's AR

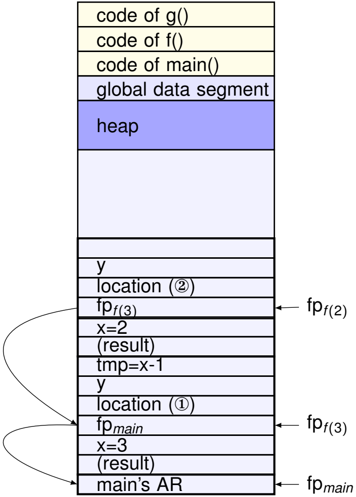
fp$_{f(3)}$

fp$_{main}$

Example

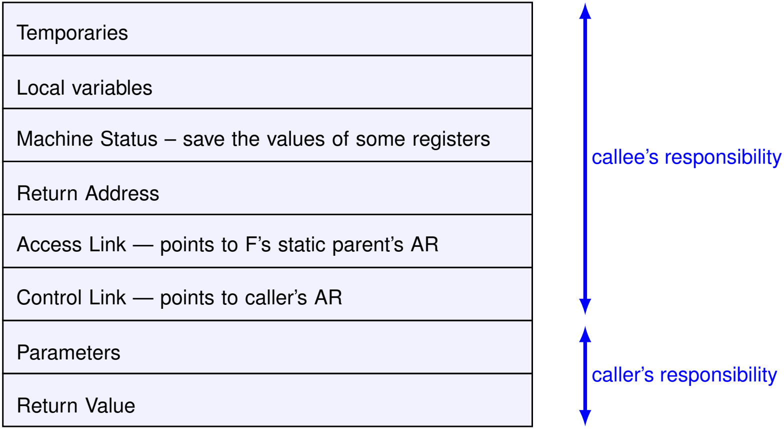
How does it work?

```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



Contents of Activation Record (AR)

- In a typical AR of function F, we have



Temporaries

Local variables

Machine Status – save the values of some registers

Return Address

Access Link — points to F's static parent's AR

Control Link — points to caller's AR

Parameters

Return Value

callee's responsibility

caller's responsibility

Calling Convention

- Caller's responsibility
  - Caller evaluates actual parameters
  - Caller stores return address and old FP in callee's AR
  - Callers sets FP register to its new position
- Callee's responsibility
  - Callee saves registers and other machine status information
  - Callee initializes its own data and begins execution

Discussion of AR

- The layout of AR is determined at compile-time
- The order can be rearranged but fixed (respect convention for better portability)
- Caller/callee responsibilities can be divided slightly differently
- Some values (e.g. the first four parameters) can be kept in registers to speed up execution
- Placing the result as the first entry in callee's frame simplifies caller finding the value

Translation IR to Binary Code

- We use symbol names in 3-address code (IR)
  e.g. **add a, b, c**

- When generating binary executable
  - Symbolic names have to be translated to memory addresses

Translation IR to Binary Code

- We use symbol names in 3-address code (IR)
  e.g. **add a, b, c**

- When generating binary executable
  - Symbolic names have to be translated to memory addresses

    .... but memory address is not fixed during execution

Translation IR to Binary Code

- We use symbol names in 3-address code (IR)
  e.g. **add a, b, c**

- When generating binary executable
  - ➢ Symbolic names have to be translated to memory addresses

    .... but memory address is not fixed during execution

- Recall how we translated global variables?
  - ➢ A tuple (global_base, offset)
  - ➢ Only one copy is kept for entire program execution
  - ➢ Allocated in global data segment
  - ➢ statically known

Translation Local Variables

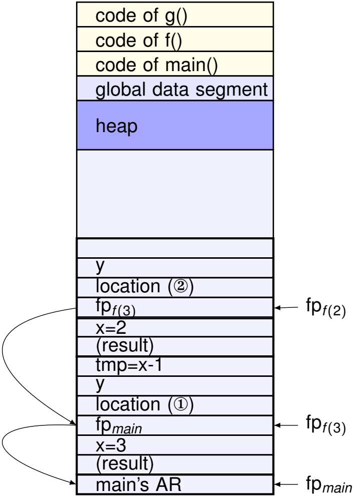
- Local variables can be translated similarly
  - Relative address to $FP i.e. **(FP, offset)**

    FP — fixed for the lifetime of the corresponding function invocation
    offset — statically known (from previous discussion)

Example

How does it work?

```
class C {
    int g() {
        return 1;
    }

    int f() {
        int y;
        if (x==2)
            y = 1;
        else
            y = x + f(x-1);
        ② ...
        return y;
    }

    int main() {
        f(3);
        ① ...
    }
}
```



How about Non-Local Variables ?

- For fully block-structured languages
  - e.g. PASCAL, ALGOL 68

How about Non-Local Variables ?

- For fully block-structured languages
  - e.g. PASCAL, ALGOL 68
- A possible guess .... a tuple (X, offset) ?

How about Non-Local Variables ?

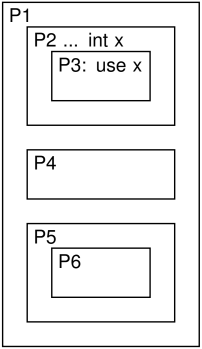
- For fully block-structured languages
  - e.g. PASCAL, ALGOL 68
- A possible guess .... a tuple (X, offset) ?
  - a good guess
  - but what is X?

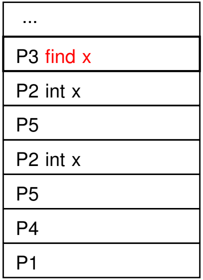
How about Non-Local Variables ?

- ❑ For fully block-structured languages
  - ➢ e.g. PASCAL, ALGOL 68
- ❑ A possible guess .... a tuple (X, offset) ?
  - ➢ a good guess
  - ➢ but what is X?
- ❑ What is the complication?
  - ➢ Non-locals can appear at different nesting level
  - ➢ Need to access them in different ARs

A Nested Procedure Declaration

- **P1** calls **P4** calls **P5** calls **P2** calls **P5** calls **P2** calls **P3**
- Problem:
  x is defined in **P2** but there are multiple P2's ARs, which one to use?



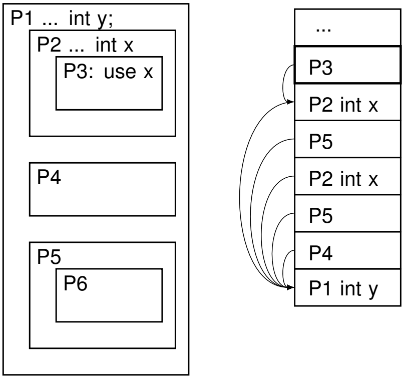


Access Link

- According to static semantic rule, variable **x** matches the one defined in its **textual parent** i.e. the closest enclosing definition
  - ➢ We need to add such information in our AR
- **Access link**
  - ➢ Access link is the FP of its textual parent
- When translating a non-local varaible
  - ➢ variable x is translated to **(diff, offset)**
    diff — nesting level difference
  - ➢ **Diff** indicates the number of jumps that we need to follow along the access link chain

Meaning of (diff, offset)

- How to use (diff, offset) to find variable at runtime?
  - Access link points to its textual parent
  - **diff** indicates the number of jumps to find the desired allocation base
  - **offset** indicates the offset to be added to the found base



P1 ... int y;

P2 ... int x

P3: use x

P4

P5

P6

...

P3

P2 int x

P5

P2 int x

P5

P4

P1 int y

// y is translated to (2, off$_y$)

// P3's access link can be found at \$fp+off$_{fp}$

load \$R2, off$_{fp}$(\$fp)

// jump twice along access link to get \$oldfp

load \$R2, off$_{fp}$(\$R2)

// variable y is saved in \$oldfp+off$_y$

load \$R3, off$_y$(\$R2)

Discussion of This Approach

- offset$_{fp}$ — a constant that indicates the distance to $fp where the access link is stored. It does not vary for different variables
- offset$_{y}$ — within P1, the offset to its allocation base (i.e. $fp). It takes a different value for a different variable

Another Example

```
void P0() {
     int I;
     int J;
     void P1() {
         int K;
         int L;
         void P2() {
             use K;
             use J;
         }

         use I
     }

     void P3() {
         int H;
         use J
     }
     use I
}
```

	NestingLevel	Variable	Offset
P0	0	I	0
		J	4
P1	1	K	0
		L	4
P2	2	-	-
P3	1	H	0

Another Example

```
void P0() {
    int I;
    int J;
    void P1() {
        int K;
        int L;
        void P2() {
            use K;
            use J;
        }

        use I
    }

    void P3() {
        int H;
        use J
    }
    use I
}
```

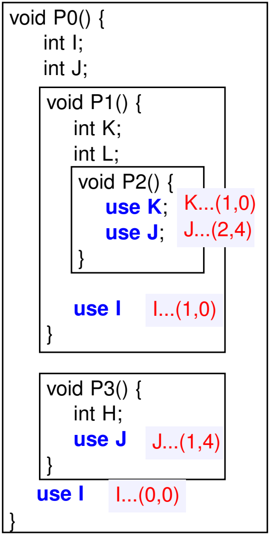
	NestingLevel	Variable	Offset
P0	0	I	0
		J	4
P1	1	K	0
		L	4
P2	2	-	-
P3	1	H	0

In P2: **use K** .... K is defined in P1 ...

=>(P2'nestingLevel-P1'nestinglevel, K's offset)

=>(1,0)

Another Example



```
void P0() {
    int I;
    int J;
    void P1() {
        int K;
        int L;
        void P2() {
            use K;   K...(1,0)
            use J;   J...(2,4)
        }

        use I    I...(1,0)
    }

    void P3() {
        int H;
        use J    J...(1,4)
    }
    use I    I...(0,0)
}
```

	NestingLevel	Variable	Offset
P0	0	I	0
		J	4
P1	1	K	0
		L	4
P2	2	-	-
P3	1	H	0

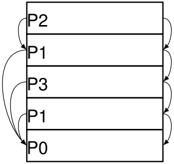
In P2: **use K** .... K is defined in P1 ...
=>(P2'nestingLevel-P1'nestinglevel, K's offset)
=>(1,0)

At Runtime

- Example: P0 calls P1 calls P3 calls P1 calls P2

```
void P0() {
     int I;
     int J;
     void P1() {
          int K;
          int L;
          void P2() {
               use K;   K...(1,0)
               use J;   J...(2,4)
          }

          use I      I...(1,0)
     }

     void P3() {
          int H;
          use J      J...(1,4)
     }
     use I      I...(0,0)
}
```

To access J in P2, need to jump twice along the access link chain



A Better Solution

- Using an access link chain has problems
  - Traverse the link chain requires multiple memory operations
  - Memory operations are slow

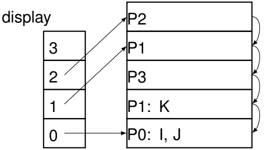
- To speed up the access, we use **display**
  - Observation: given a nesting level **L**, we have at most one active AR when enforcing static scoping rule
  - We therefore can use an array to record these FPs — display
  - Display tracks accessible ARs

An Example Showing the Use of Display

- Example: P0 calls P1 calls P3 calls P1 calls P2
  - ➢ Translates variable to **(Absolute Nesting Level, offset)**
  - ➢ Keep active pointers at each level in an array

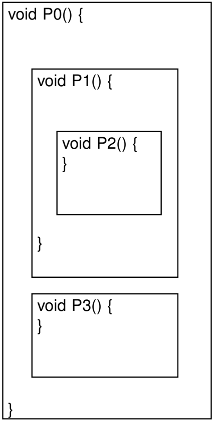
```
void P0() {
    int I;
    int J;
    void P1() {
        int K;
        int L;
        void P2() {
            use K;   K...(1,0)
            use J;   J...(0,4)
        }

        use I     I...(0,0)
    }

    void P3() {
        int H;
        use J     J...(0,4)
    }
    use I     I...(0,0)
}
```

To access J (defined in P0) in P2, we have (0,4) i.e. display[0]+4

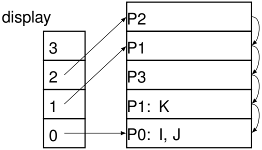


How to Update a Display?

- when procedures are called, or terminated, we need to update the display



```
void P0() {

    void P1() {

        void P2() {
        }

    }

    void P3() {
    }

}
```



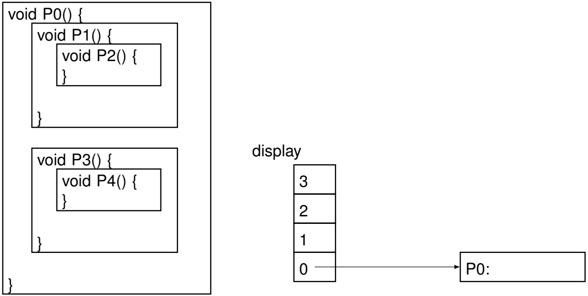
display

3
2
1
0

P2
P1
P3
P1: K
P0: I, J

Update the Display

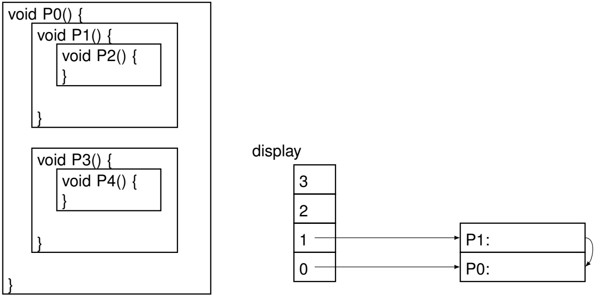
- The display needs to be updated when
  - a procedure is called, and
  - a procedure is terminated



```
void P0() {
  void P1() {
    void P2() {
    }
  }
  void P3() {
    void P4() {
    }
  }
}
```

Update the Display

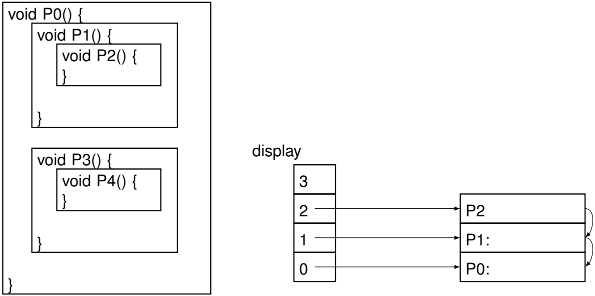
- The display needs to be updated when
  - a procedure is called, and
  - a procedure is terminated



```
void P0() {
    void P1() {
        void P2() {
        }

    }

    void P3() {
        void P4() {
        }

    }

}
```

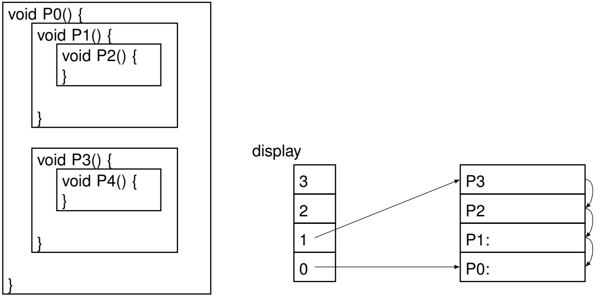
Update the Display

- The display needs to be updated when
  - a procedure is called, and
  - a procedure is terminated



Update the Display

- The display needs to be updated when
  - a procedure is called, and
  - a procedure is terminated

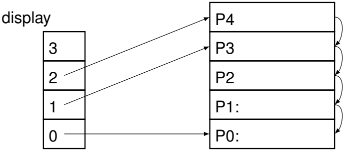


```
void P0() {
    void P1() {
        void P2() {
        }
    }

    void P3() {
        void P4() {
        }
    }
}
```

Update the Display

- The display needs to be updated when
  - a procedure is called, and
  - a procedure is terminated

```
void P0() {
    void P1() {
        void P2() {
        }

    }

    void P3() {
        void P4() {
        }

    }

}
```

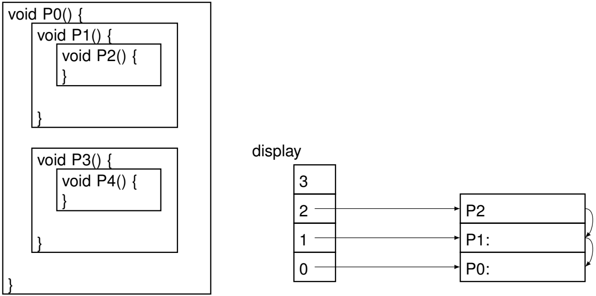

display

display
3
2
1
0

P4
P3
P2
P1:
P0:

Update the Display

- The display needs to be updated when
  - a procedure is called, and
  - a procedure is terminated



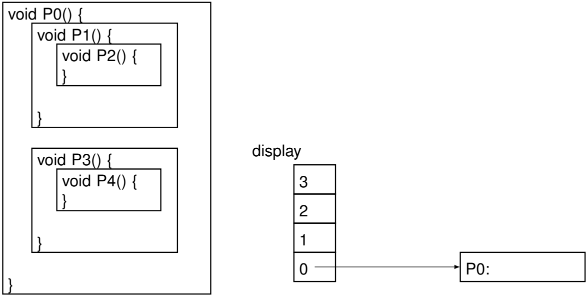
How to Update a Display?

When P4 terminates, there are three update approaches

1. Restore the entire display if it has been stored in the caller
   - ➢ Since P3 calles P4, and P3 uses D[0], D[1], we should have saved them before entering P4
   - ➢ Now we just need to restore both D[0] and D[1]
2. Use access links to reconstruct the display
   - ➢ Only when Callee's nesting Level (n2) $\leq$ Caller's nesting Level (n1)
   - ➢ And we only fix d[n2], d[n2+1], ..., d[n1]
3. Save and restore one for each call
   - ➢ Callee's nesting level is n1, save D[n1] and restore D[n1]

Approach 2 Illustrated

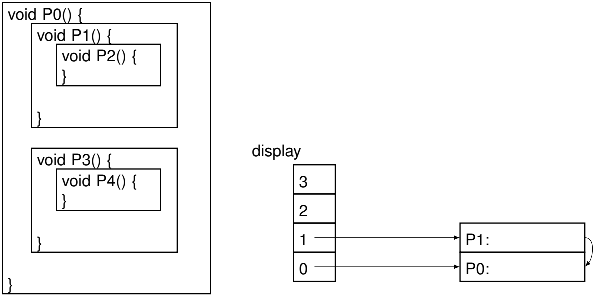
- Approach 2: when P2 call P3
  ... P3's nesting level is 1
  ... P2's nesting level is 2



```
void P0() {
    void P1() {
        void P2() {
        }
    }

    void P3() {
        void P4() {
        }
    }
}
```

Approach 2 Illustrated

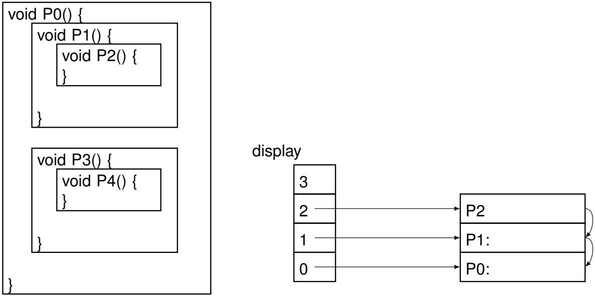
- Approach 2: when P2 call P3
  ... P3's nesting level is 1
  ... P2's nesting level is 2



```
void P0() {
    void P1() {
        void P2() {
        }
    }
    void P3() {
        void P4() {
        }
    }
}
```

Approach 2 Illustrated

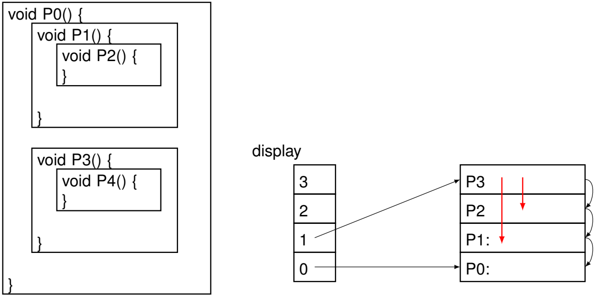
- Approach 2: when P2 call P3
  ... P3's nesting level is 1
  ... P2's nesting level is 2



```
void P0() {
    void P1() {
        void P2() {
        }
    }
    void P3() {
        void P4() {
        }
    }
}
```

Approach 2 Illustrated

- Approach 2: when P2 call P3
  ... P3's nesting level is 1
  ... P2's nesting level is 2

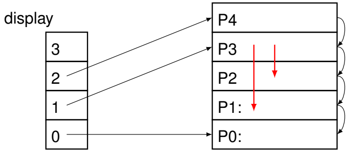


```
void P0() {
    void P1() {
        void P2() {
        }
    }
    void P3() {
        void P4() {
        }
    }
}
```

Approach 2 Illustrated

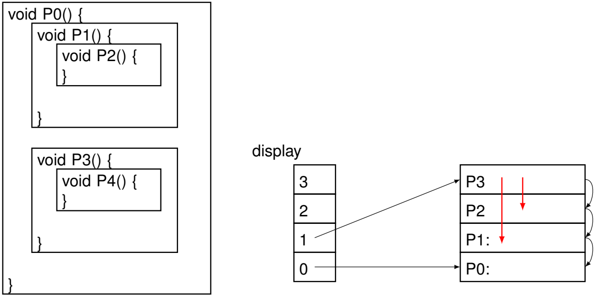
- Approach 2: when P2 call P3
  ... P3's nesting level is 1
  ... P2's nesting level is 2

```
void P0() {
    void P1() {
        void P2() {
        }
    }
    void P3() {
        void P4() {
        }
    }
}
```



Approach 2 Illustrated

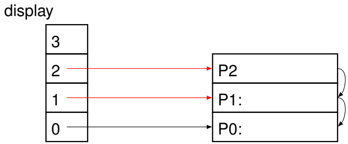
- Approach 2: when P2 call P3
  ... P3's nesting level is 1
  ... P2's nesting level is 2



Approach 2 Illustrated

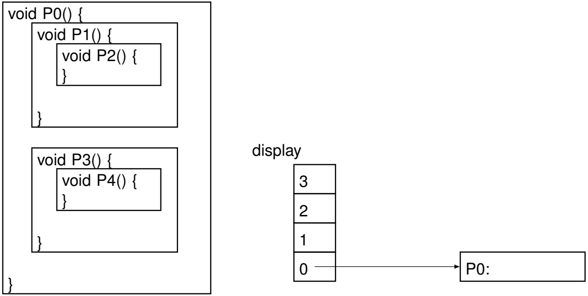
- Approach 2: when P2 call P3
  ... P3’s nesting level is 1
  ... P2’s nesting level is 2

```
void P0() {
    void P1() {
        void P2() {
        }
    }

    void P3() {
        void P4() {
        }
    }
}
```



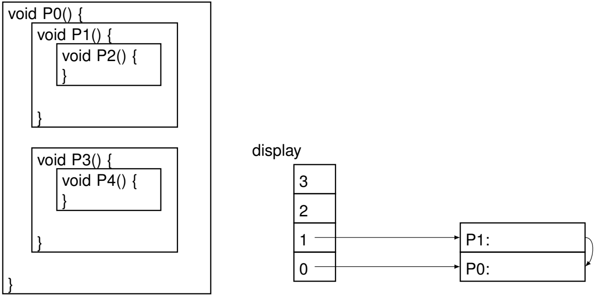
Approach 3 Illustrated

- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];



Approach 3 Illustrated

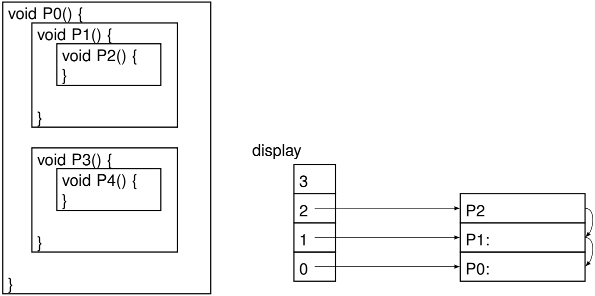
- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];



```
void P0() {
    void P1() {
        void P2() {
        }
    }

    void P3() {
        void P4() {
        }
    }
}
```

Approach 3 Illustrated

- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];



```
void P0() {
  void P1() {
    void P2() {
    }
  }

  void P3() {
    void P4() {
    }
  }
}
```

display

3

2

1

0

P2

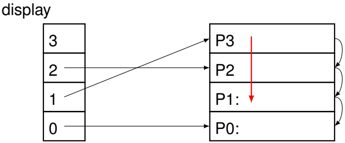
P1:

P0:

Approach 3 Illustrated

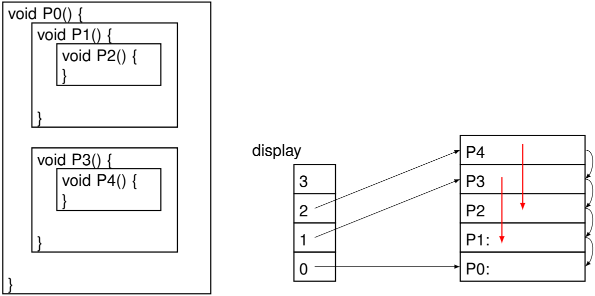
- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];

```
void P0() {
    void P1() {
        void P2() {
        }

    }

    void P3() {
        void P4() {
        }

    }
}
```



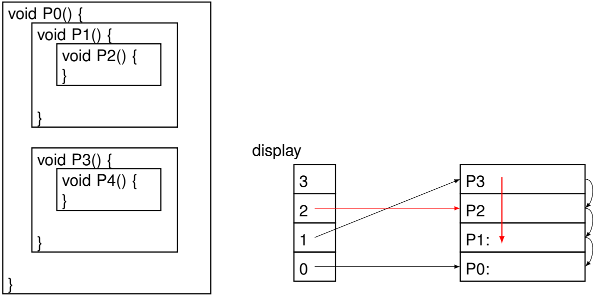
Approach 3 Illustrated

- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];



Approach 3 Illustrated

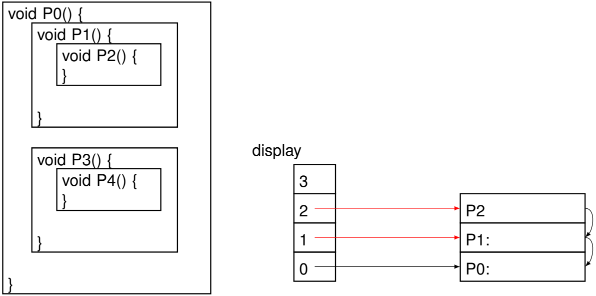
- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];



```
void P0() {
    void P1() {
        void P2() {
        }
    }
    void P3() {
        void P4() {
        }
    }
}
```

Approach 3 Illustrated

- Approach 3: saves/restores the entry to be overwritten
  ... when P3 is called, P3 saves/restores D[1];
  ... when P4 is called, P4 saves/restores D[2];



Comparing Three Approaches

- Approach 1 is always expensive
- Approach 2 only incurs overhead when n2 $\leq$ n1
- Approach 3 has constant overhead (i.e. one save/restore per call)

Translating Parameters

- Till now, we know how to translate
  - Globals
  - Locals
  - Non-locals

Translating Parameters

- Till now, we know how to translate
  - Globals
  - Locals
  - Non-locals
- How about parameters?

```
int func1(int a, int b) { ... }
...
... z = z + func1(x, y);
```

  - Formal parameters **a, b** — the names used when a function is declared
  - Actual parameters **x, y**— the names used when a function is called

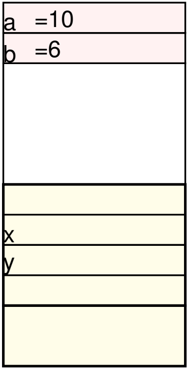
Calling Convention

Calling convention is also referred as parameter passing

- **Call by value**
  - Formal parameter is treated like a local variable
  - Caller evaluates and places the value in storage element for the formal parameter
- **Call by reference**
  - Address for parameter is passed as the value of the formal parameter
  - If actual is an expression then compute the expression into a temporary and pass the address of the temporary as formal parameter's value
- **Call by value**
  - Value passed to called procedures
  - Addresses of actual parameters are saved
  - Upon return, copy value of formals into address of actuals

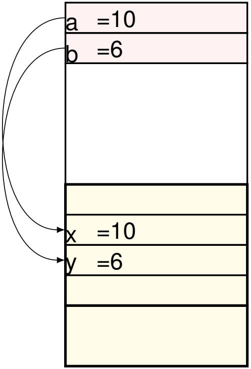
Call by Value

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



Call by Value

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



a =10

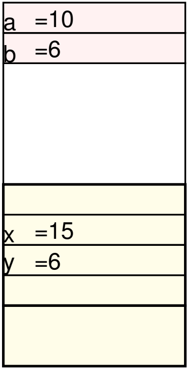
b =6

x =10

y =6

Call by Value

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



a =10

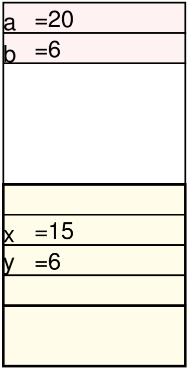
b =6

x =15

y =6

Call by Value

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



a =20

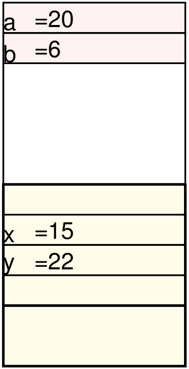
b =6

x =15

y =6

Call by Value

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf("a=%d,b=%d",a,b);
}
```



a =20

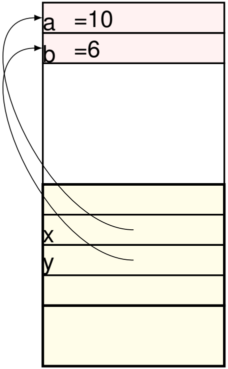
b =6

x =15

y =22

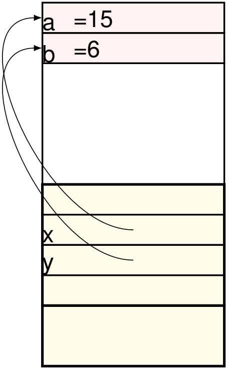
Call by reference

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



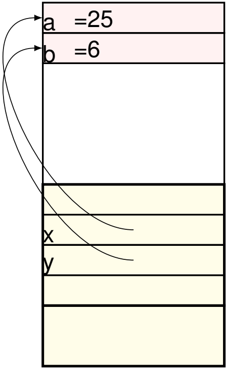
Call by reference

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



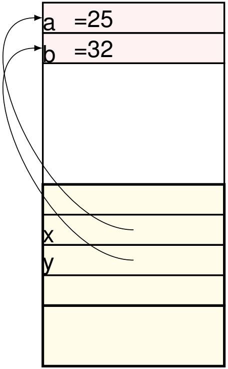
Call by reference

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



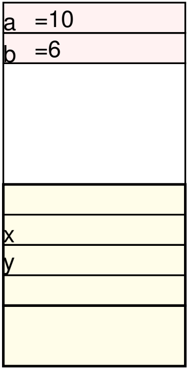
Call by reference

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



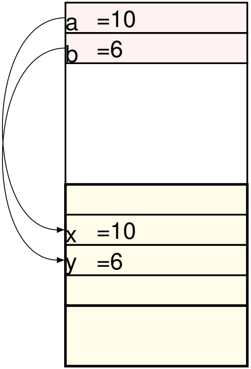
Call by Value-Result

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



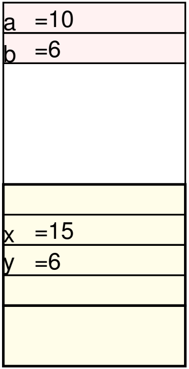
Call by Value-Result

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



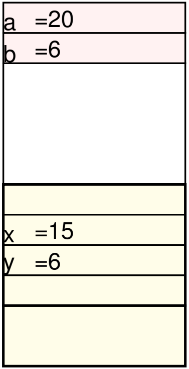
Call by Value-Result

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



Call by Value-Result

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



a =20

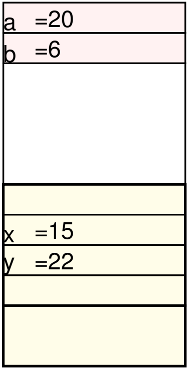
b =6

x =15

y =6

Call by Value-Result

```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



a =20

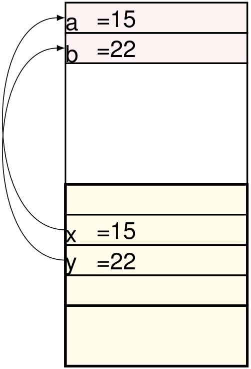
b =6

x =15

y =22

Call by Value-Result

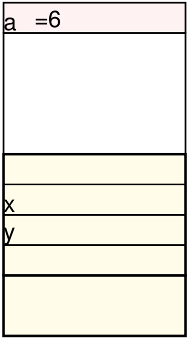
```
int a = 10;
int b = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,b);
    printf(“a=%d,b=%d”,a,b);
}
```



Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf(“a=%d,b=%d”,a,b);
}
```



a =6

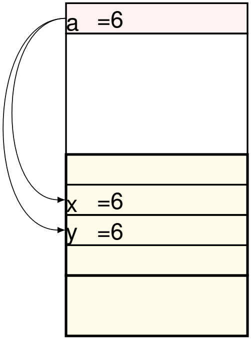
x

y

Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

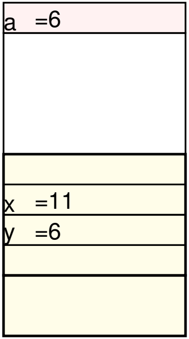
```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf(“a=%d,b=%d”,a,b);
}
```



Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

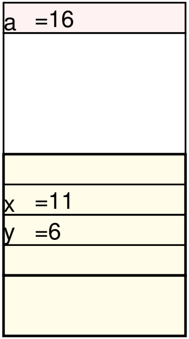
```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf(“a=%d,b=%d”,a,b);
}
```



Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

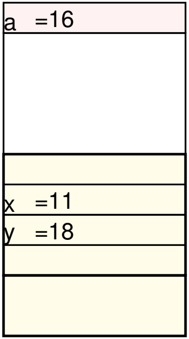
```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf(“a=%d,b=%d”,a,b);
}
```



Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

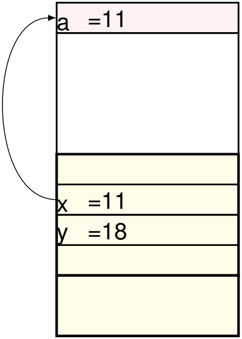
```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf("a=%d,b=%d",a,b);
}
```



Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

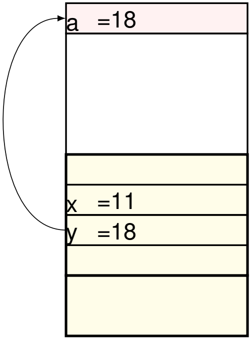
```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf(“a=%d,b=%d”,a,b);
}
```



Parameter Collision

- The order of evaluating parameters may affect results
  - left to right — x =?
  - right to left — x =?

```
int a = 6;
int f(int ◇x, int ◇y)
{
    x = a + 5;
    a = a + 10;
    y = x + 7;
}
void main()
{
    f(a,a);
    printf(“a=%d,b=%d”,a,b);
}
```



More About Parameter Collision

- Parameter collision creates alias
  - A memory location may be accessed using more than one variable names
- Assuming **call by value-result**, where to copy the results in the following cases?

  -
    ```
    int list[100];
    func(int a, int b) {...a...b...}
    main() { i=j; call func(list[i], list[j]); }
    ```

  -
    ```
    int list[100];
    func(int a) { i=100; ...}
    main() { i=10; call func(list[i]); }
    ```

  -
    ```
    int x=10;
    func(int a) { a=5; ...}
    main() { call func(x+20); }
    ```

Call by Name

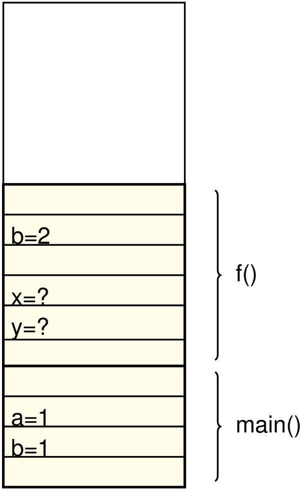
- Originated in ALGOL, now it is less popular
- It is a good case study to understand name translating in a compiler

- Rule
  - Evaluating parameters on-demand
  - When the function is called, parameters are not evaluated
  - When the parameters is used, evaluate the parameters in the environment of caller

  - The difficulty: the FP is now overwritten, caller may not be callee's textual parent

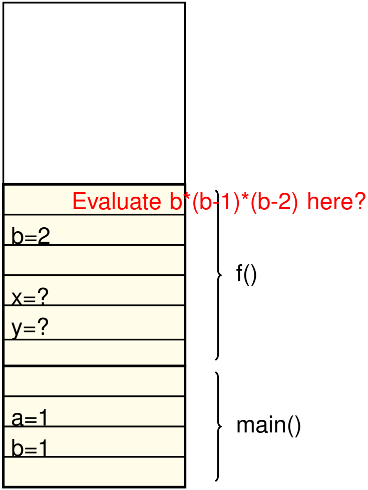
The Problem of Call-by-Name

```
int f(int ◇x, int ◇y)
{
    int b=2;
    if (x>0)
        x = y;
}
void main()
{
    int a=1;
    int b=1;
    f(a, b*(b-1)*(b-2));
}
```



The Problem of Call-by-Name

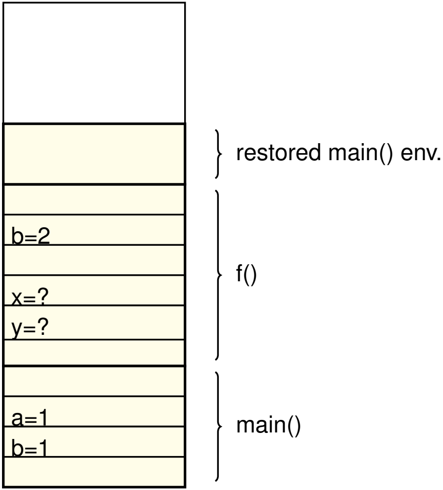
```
int f(int ◇x, int ◇y)
{
    int b=2;
    if (x>0)
        x = y;
}
void main()
{
    int a=1;
    int b=1;
    f(a, b*(b-1)*(b-2));
}
```



Evaluate b*(b-1)*(b-2) here?

The Problem of Call-by-Name

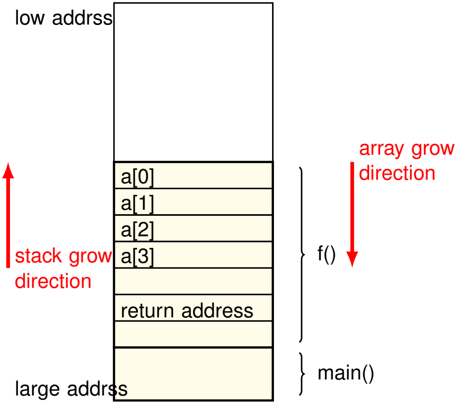
```
int f(int ◇x, int ◇y)
{
    int b=2;
    if (x>0)
        x = y;
}
void main()
{
    int a=1;
    int b=1;
    f(a, b*(b-1)*(b-2));
}
```



Buffer Overflow Attacks (BOAs)

- BOA is a major type of security threat
- Code example

```
int foo()
{
    int i=0, a[4];
    while (x>0) {
        a[i] = getc();
        if (a[i] == ’.’)
            break;
        i++;
    }
}
void main()
{
    foo();
}
```



When Return Address is Overwritten

- What may happen when *foo()* finishes its execution

```
foo: ...
    ld $ra, -4($fp) // get return address from stack
    ret; // jump to whatever found from stack
```

- When providing a nasty input

  "... ... 00 10 00 00 "
  (20Bytes) (entrance of bad code)

How to Defend BOA Attacks?

- Shadow word
  - A special/random word next to the return address/function pointer
  - Check the shadow word before returning
- Randomization
  - AR size is not fixed
- Save $ra in a different place
  - Function pointer could still be a problem
- Taint analysis
  - enforce information flow theory
  - high overhead
- Array bound check
- Many other defending techniques

After name translation, we are ready to translating IR to binary code

After name translation, we are ready to translating IR to binary code

- However, we will only generate very inefficient code
  - Inefficient use of registers
  - Inefficient use of instruction types

After name translation, we are ready to translating IR to binary code

- ❑ However, we will only generate very inefficient code
  - ➢ Inefficient use of registers
  - ➢ Inefficient use of instruction types

  - ➢ Will be addressed in **compiler optimization** phase

Generating MIPS Assembly

- Code generation is machine dependent
- In this course, we focus on MIPS architecture
  - RISC (Reduced Instruction Set Computer) machine
    - ALU instruction use registers for operands and results
    - load/store are the only instruction to access memory
  - 32 general purpose registers
    - $0 is always zero, $a0,...,$a4 are for arguments
    - $sp saves stack pointer, $fp saves frame pointer
  - 32 bits per word

Some Examples

```
lw R1, offset(R2)      ; load one word from offset + R2 to R1

add R1, R2, R3         ; R1 ← R2 + R3

addiu R1, R2, imm      ; R1 ← R2 + imm, overflow unchecked

sw R1, offset(R2)      ; store R1 to offset+R2

li R1, imm             ; R1 ← imm
```

Code Generation for Expressions

cgen(e1+e2):

```
    cgen(e1)
    sw $a0, 0($sp)
    addiu $sp, $sp, -4
    cgen(e2)
    lw $t1, 4($sp)
    add $a0, $t1, $a0
    addiu $sp, $sp, 4
```

cgen(if (e1==e2) then e3 else e4):

```
           cgen(e1)
           sw $a0, 0($sp)
           addiu $sp, $sp, -4
           cgen(e2)
           lw $t1, 4($sp)
           beq $a0, $t1, Tpath
    Fpath: cgen(e4)
           b End
    Tpath: cgen(e3)
    End:   ...
```

Code Generation for Function Call

cgen(f(e1))=

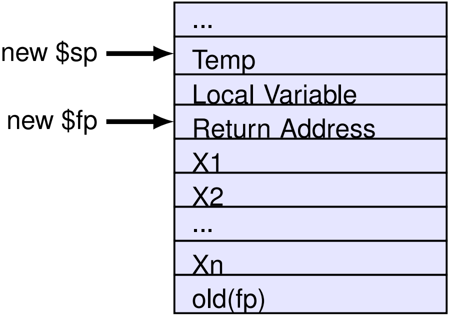
```
    sw $fp, 0($sp)
    addiu $sp, $sp, -4
    cgen(e1)
    sw $a0, 0($sp)
    j fEntry
```

cgen(def(f(...):e)=

```
            move $fp, $sp
            sw $ra, 0($sp)
            addiu $sp, $sp, -4
    fEntry:   cgen(e)
            lw $ra, 4($sp)
            addiu $sp, $sp, -4
            lw $sp, 0($sp)
            jr $ra
```

Code Generation for Variables

- Local variables are referenced from an offset from $fp
  - ➢ Traditionally $fp is pointing to the return address
  - ➢ Since the stack pointer changes when intermediate results are saved, local variables do not have fixed offset to $sp



new $sp → Temp

new $fp → Return Address

...
Temp
Local Variable
Return Address
X1
X2
...
Xn
old(fp)

first local variable: -4($fp)
argument X1: +4($fp)

Support Classes and Objects

- ❑ An object is like a structure in C
  - ➢ Object are laid out in contiguous memory
  - ➢ Each attribute (local variable declarations in its class) is stored at a fixed offset in object
- ❑ However, all objects of a class
  - ➢ All object of the same class share a table which stores the entries to all methods declared for this class
  - ➢ Each object contains a dispatch pointer which stores the entry of the table

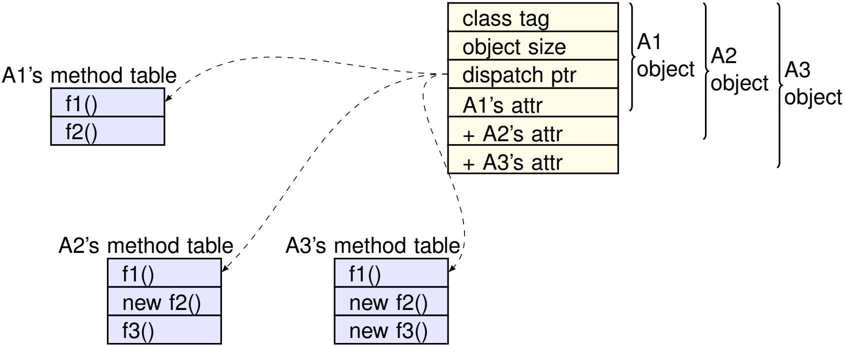
Object Layout

class tag
object size
dispatch ptr
attribute 1
...
attribute n

- ❑ Class tag is an integer
  - ➢ to identify this class from others
- ❑ Object size is an integer
  - ➢ to determine the size at runtime
- ❑ Dispatch ptr is a pointer to the method table
  - ➢ method table stores all declared methods
- ❑ Attributes are allocated in subsequent slots

Inheritance and Subclasses

- Single inheritance — the offset of an attribute is the same in a class and all of its subclasses
  e.g. A3 $\leq$ A2 $\leq$ A1



Dynamic Dispatching

- Inheritance
  - Override methods are assigned with the same offset in the dispatch table

- No inheritance in our project
  - No dynamic dispatching
  - Statically bind a function call to its address

Automatic Memory Management

Garbage Collection

Storage Management

- Programming language defines storage management scheme
- Runtime system provides automatic storage management

- Heap elements
  - ➢ Live beyond the lifetime of the procedure that create them
  - ➢ Cannot put in the stack area

```
....
TreeNode* createTREE() {
{
.... p = (TreeNode*)malloc(sizeof(TreeNode));
return p;
}
```

Why Automatic Memory Management?

- Heap elements can be reclaimed by programmers by calling “free(p)”
- However, programmers may
  - forget to free unused memory
  - dereference a dangling pointer
  - overwrite parts of a data structure by accident

  - Storage bugs are hard to find and fix
- Many languages e.g. Java, LISP, rely on automatic memory management
  - Automatic management drawbacks
    - Programmers have a better knowledge about the time to reclaim an object

Automatic Memory Management

- This is an old problem
    - Studied since 1950s for LISP programming language
    - Recently get popular because of Java/C++
        - due to memory management complexity and overhead
- The basic idea is
    - When an object is created, unused space of its size is automatically allocated
    - When an object becomes “never-be-used-again”, its space can be reclaimed
        - may not be reclaimed immediately
    - If the system is running out of space, then it proactively detects “never-be-used-again” objects and reclaim their space

The Difficulty

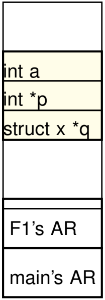
- How to determine an object reaches its last i.e. **never-be-used-again**
  - In general, impossible to tell
  - In C or PASCAL, the programmer decides when to reclaim
  - Automatic memory management uses heuristics
- **Foundation:** a program can only use an object if it can reference it
  - Named objects vs Nameless objects
  - Nameless objects i.e. heap objects are accessed through pointers
  - Pointers are named objects

Reachable Objects and Garbage

- An object **x** is **reachable** iff
  - A named object contains a pointer to **x**, or
  - Another reachable object **y** contains a pointer to **x**
- Here we define named objects (at runtime) can be
  - registers
  - global variables
  - stack objects
- An unreachable object is referred as **garbage**
  - cannot be used

Basic Approach to Track Reachable Objects

- To track objects, we need to know the layout of global and stack objects
- When analyzing an object with many fields, need to follow its pointer fields
  - value fields are skipped



int a

int *p

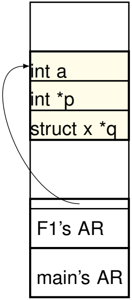
struct x *q

F1’s AR

main’s AR

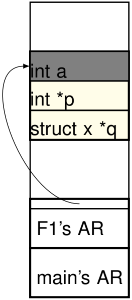
Basic Approach to Track Reachable Objects

- To track objects, we need to know the layout of global and stack objects
- When analyzing an object with many fields, need to follow its pointer fields
  - value fields are skipped



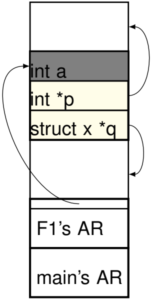
Basic Approach to Track Reachable Objects

- To track objects, we need to know the layout of global and stack objects
- When analyzing an object with many fields, need to follow its pointer fields
  - value fields are skipped



Basic Approach to Track Reachable Objects

- To track objects, we need to know the layout of global and stack objects
- When analyzing an object with many fields, need to follow its pointer fields
  - value fields are skipped



Elements of Garbage Collection

- Every garbage collection scheme has the following steps
  - ➢ Allocate space as needed for new objects
  - ➢ When space runs out
    - compute what objects might be used again, generally by tracking objects reachable from a set of **root** pointers
    - free space not used by objects from above
  - ➢ Some strategies perform garbage collection before the space actually runs out

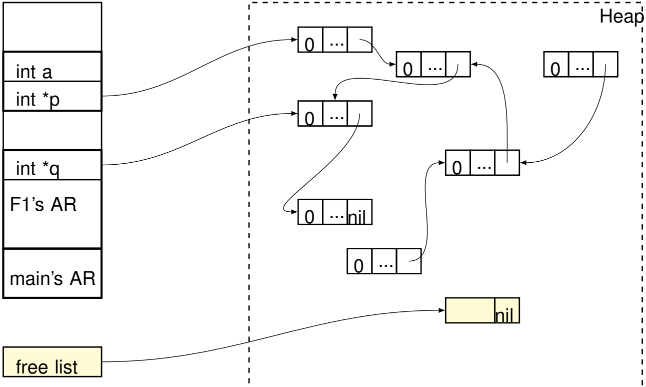
Algorithm 1: Mark and Sweep

- When it is about to run out of memory, GC stalls program execution and executes two phases
  - Mark phase: traces reachable objects
  - Sweep phase: reclaims garbage objects
- Implementation detail
  - Each object has an extra **mark** bit
  - The bit is initialized to 0
  - The bit is set to 1 for all reachable object in the mark phase
  - All objects with mark bit =0 are reclaimed in the sweep phase

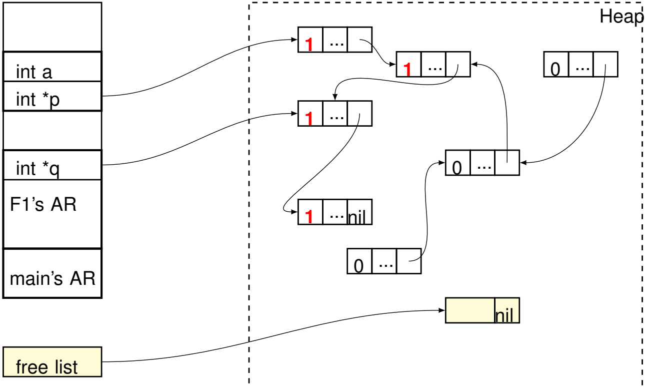
Implementation Details

```
mark() {
  todo = { all root objects };
  while (todo != NULL) {
    v ← one object in todo
    todo = todo - v;
    if (mark(v) == 0) {
      mark(v) = 1;
      extract all pointers pv1, pv2, ..., pvn from v;
      todo = todo ∪ {pv1, pv2, ..., pvn}
    }
  }
}

sweep() {
   p ← bottom(heap);
   while (p!=top(heap)) {
     if (mark(p)==1)
       mark(p) ← 0;
     else
       add p with sizeof(p) to freelist;
     p ← p + sizeof(p);
   }
```

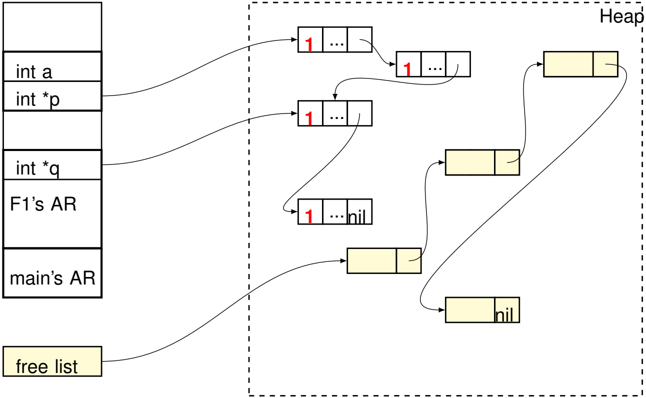
Mark and Sweep Example



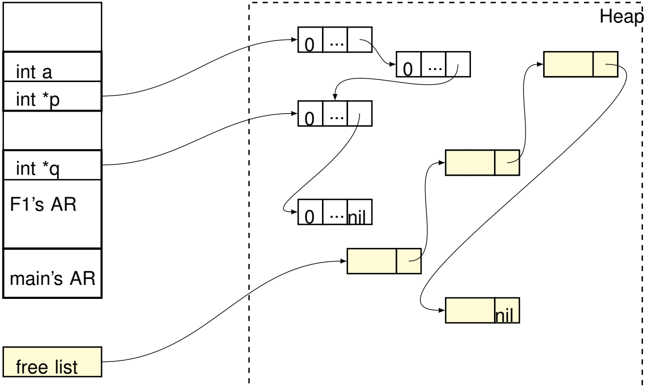
Mark and Sweep Example



Mark and Sweep Example



Mark and Sweep Example



Evaluation of Mark-and-Sweep Algorithm

- In summary, mark-sweep algorithm
  - is a pause-start algorithm
  - requires a large todo list to perform reachability analysis
  - can handle circular data structures

Evaluation of Mark-and-Sweep Algorithm

- In summary, mark-sweep algorithm
  - is a pause-start algorithm
  - requires a large todo list to perform reachability analysis
  - can handle circular data structures

- A serious problem of mark-sweep algorithm
  - The algorithm is invoked when it is about to run out of memory. However it requires large space to construct todo list.
  - The size of todo list is unbounded (not possible to reserve some space beforehand)

Algorithm 2: Reference Counting

- Idea: rather than waiting for the memory to be exhausted, let us reclaim an object when it becomes unreachable

- Solution: each object has a counter that counts the number of pointers pointing to the object
  - The counter of object x is referred as its **reference counter** i.e. rc(x)
  - Each pointer assignment requires additional manipulation of its reference counter

Implementation Details

Rules:

(rx(a) indicates the reference counter of object a)

- Initialization **x** ← **new()**:

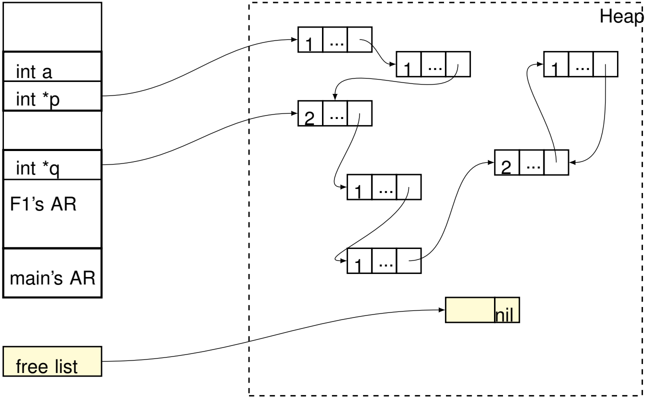
```
x ← new();
rx(x) ← 1;
```

- Pointer assignment **x** ← **y** :

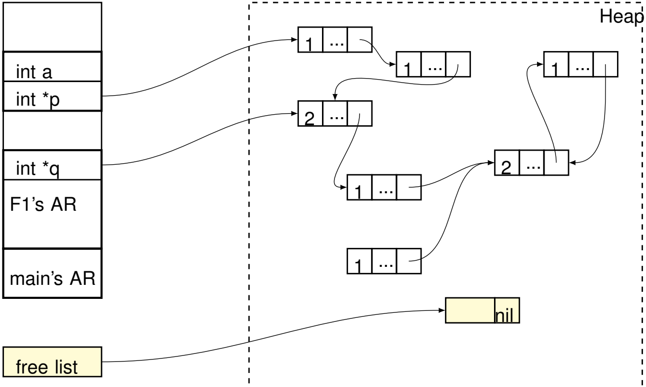
  assume pointers **x**,**y** point to objects **p**,**q** respectively

```
rx(q) ← rx(q) + 1;
rx(p) ← rx(p) + 1;
if (rc(p)==0) then
    mark p as garbage to reclaim;
x ← y;
```

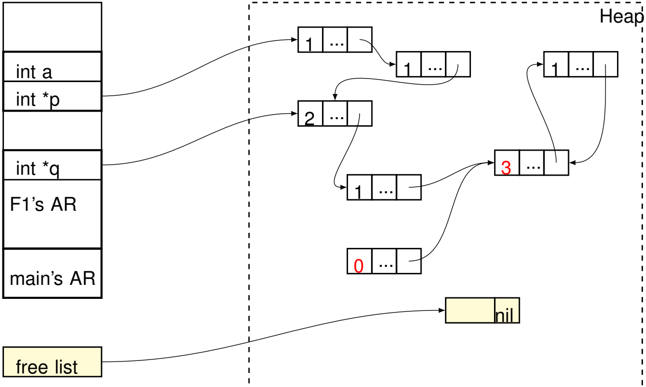
Reference Counting Example



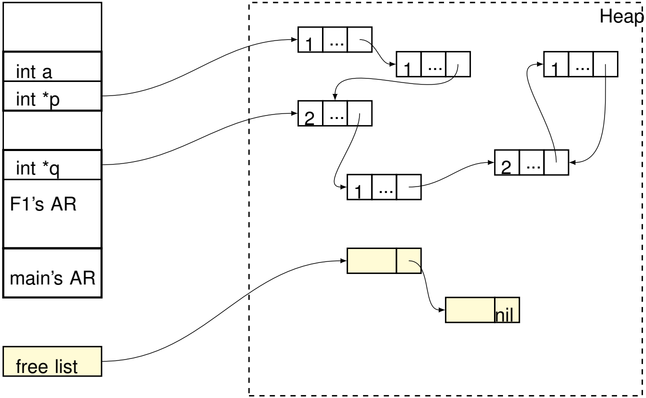
Reference Counting Example



Reference Counting Example

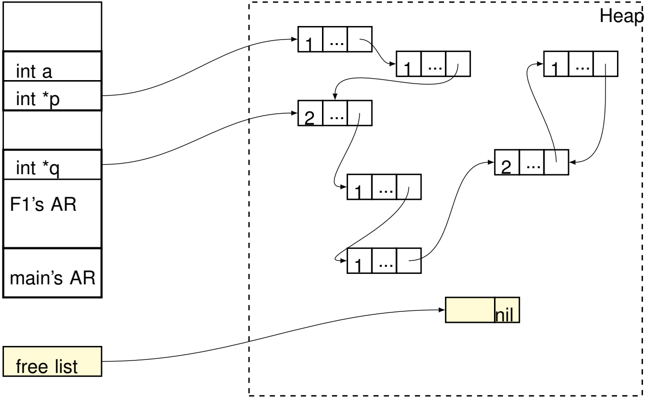


Reference Counting Example



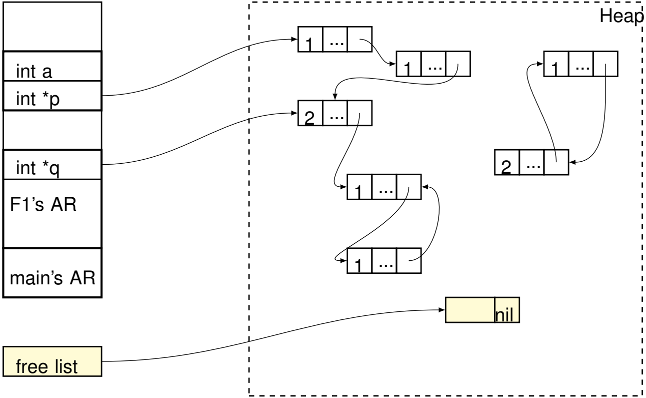
Problem of Reference Counting

- RC cannot handle **circular data structures**



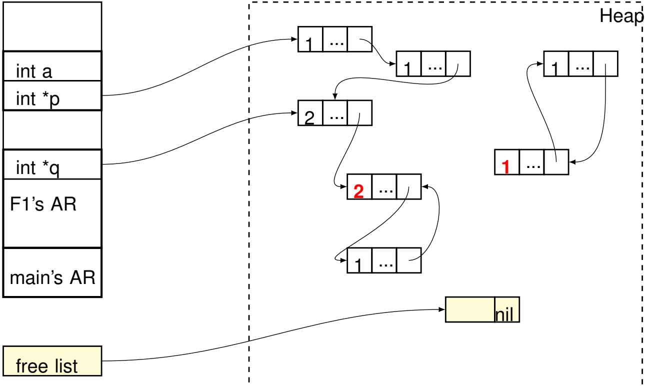
Problem of Reference Counting

- RC cannot handle **circular data structures**



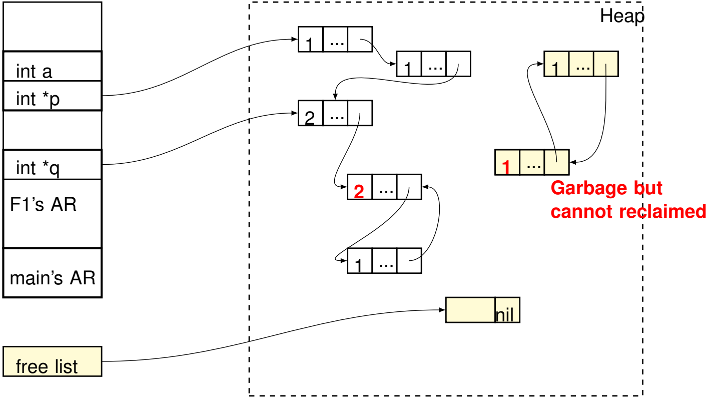
Problem of Reference Counting

- RC cannot handle **circular data structures**



Problem of Reference Counting

- RC cannot handle **circular data structures**



Garbage but cannot reclaimed

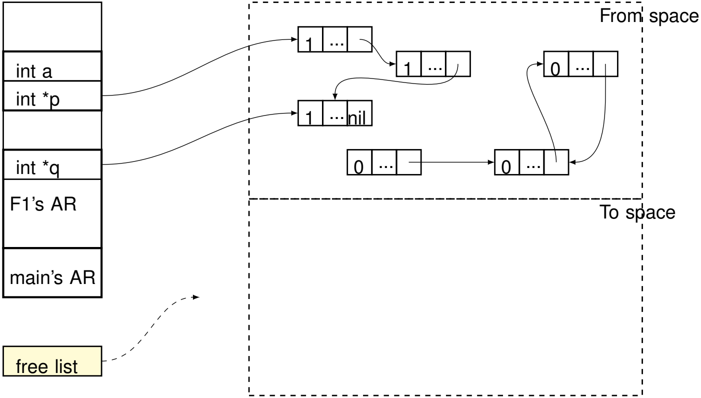
Evaluation of Reference Counting Algorithm

- Advantages:
  - Easy to implement
  - collects garbage incrementally without large pause during program execution
- Disadvantages:
  - cannot collect circular data structure
  - manipulating reference counters at each assignment is very slow

Algorithm 3: SemiSpace Collector

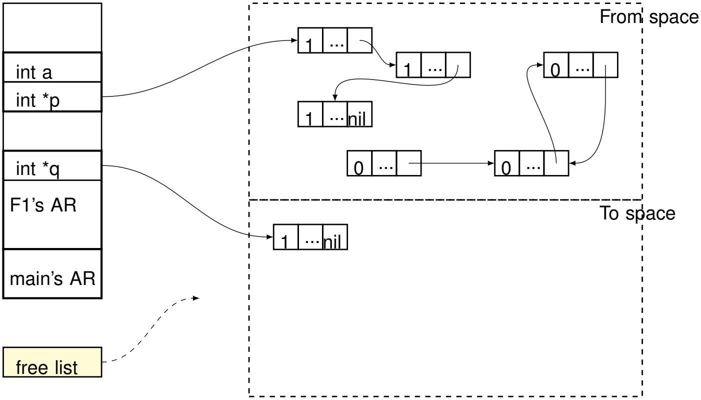
Rules

- Use half of the heap space
- When collecting garbage, copy live objects to the other half
- Install **forward pointers** to assist moving objects



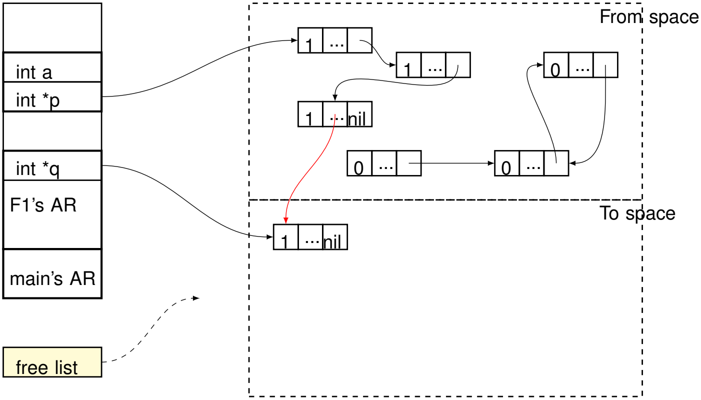
Algorithm 3: SemiSpace Collector

- Rules
  - Use half of the heap space
  - When collecting garbage, copy live objects to the other half
  - Install **forward pointers** to assist moving objects



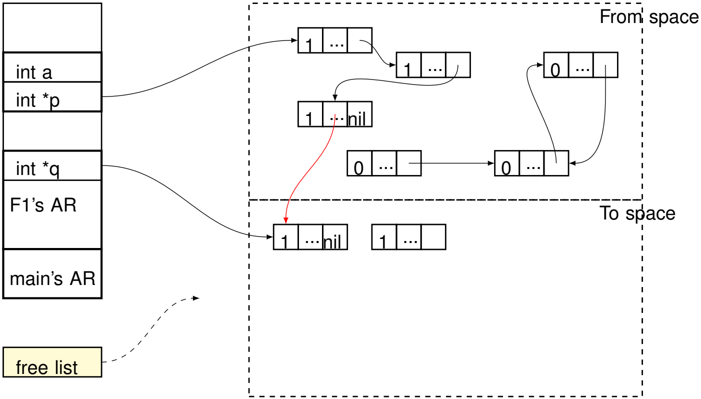
Algorithm 3: SemiSpace Collector

- Rules
  - ➢ Use half of the heap space
  - ➢ When collecting garbage, copy live objects to the other half
  - ➢ Install **forward pointers** to assist moving objects



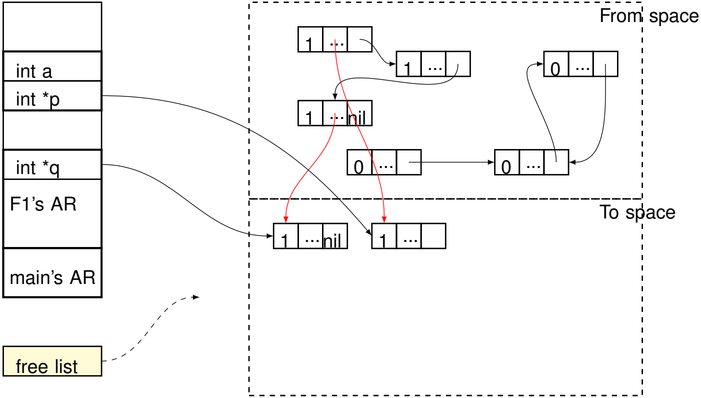
Algorithm 3: SemiSpace Collector

- Rules
  - Use half of the heap space
  - When collecting garbage, copy live objects to the other half
  - Install **forward pointers** to assist moving objects



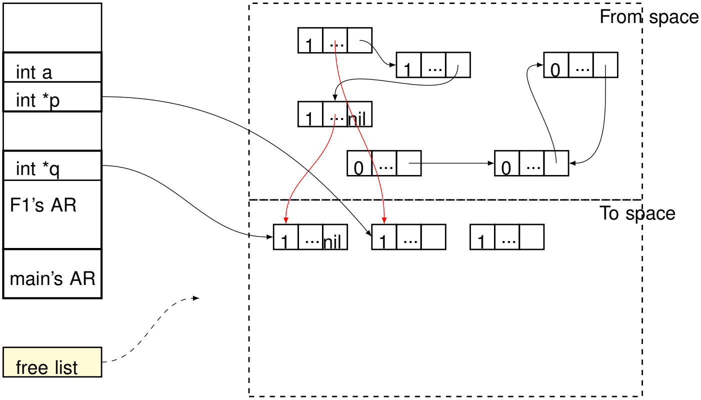
Algorithm 3: SemiSpace Collector

- Rules
  - Use half of the heap space
  - When collecting garbage, copy live objects to the other half
  - Install **forward pointers** to assist moving objects



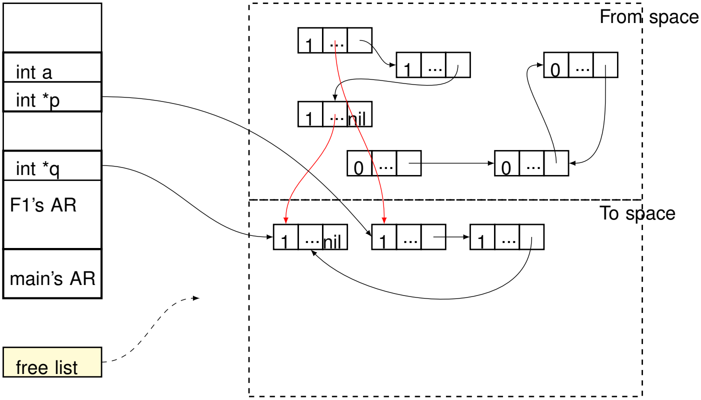
Algorithm 3: SemiSpace Collector

- Rules
  - Use half of the heap space
  - When collecting garbage, copy live objects to the other half
  - Install **forward pointers** to assist moving objects



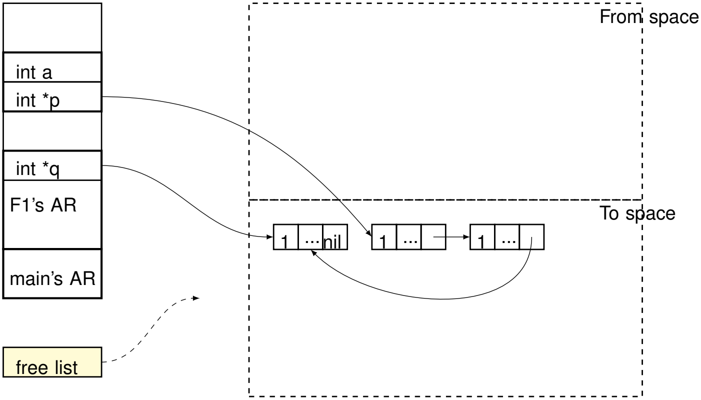
Algorithm 3: SemiSpace Collector

- Rules
  - ➢ Use half of the heap space
  - ➢ When collecting garbage, copy live objects to the other half
  - ➢ Install **forward pointers** to assist moving objects



Algorithm 3: SemiSpace Collector

- Rules
  - Use half of the heap space
  - When collecting garbage, copy live objects to the other half
  - Install **forward pointers** to assist moving objects



Evaluation of SemiSpace Collection

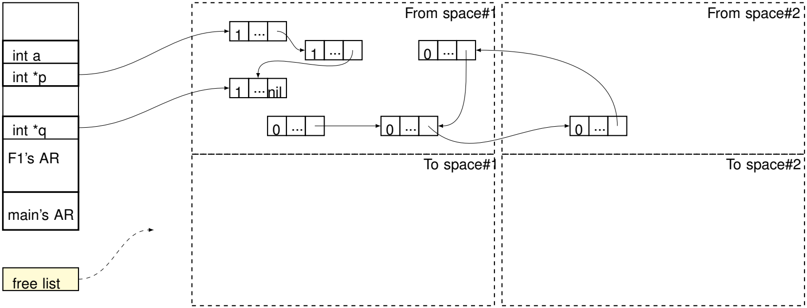
- Only use half of heap space
- Moving objects is slow
- Increase cache performance of following object accesses

Algorithm 4: Incremental Garbage Collection

Rules

- Divide heap into smaller chunks and collecting one chunk at a time
- Need **write barrier** to ensure correctness

  i.e. pointers from other chunks

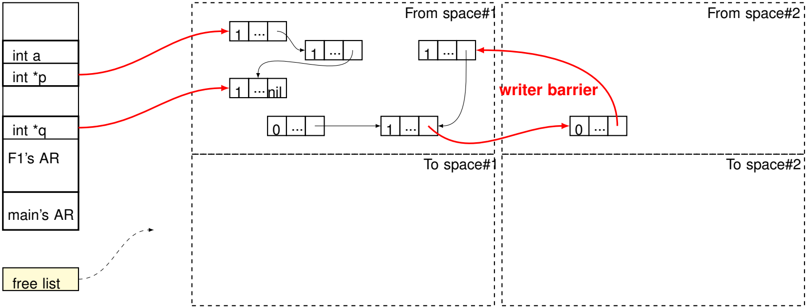


Algorithm 4: Incremental Garbage Collection

Rules

- Divide heap into smaller chunks and collecting one chunk at a time
- Need **write barrier** to ensure correctness

  i.e. pointers from other chunks

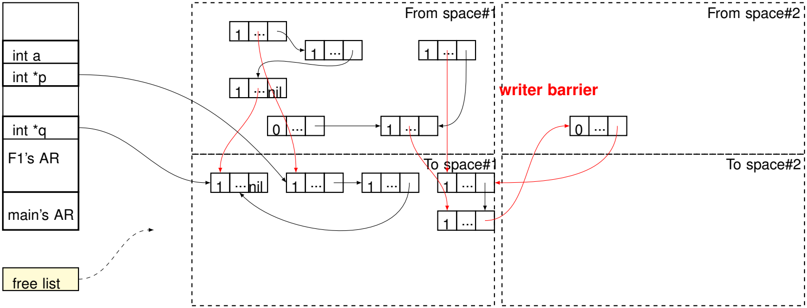


Algorithm 4: Incremental Garbage Collection

Rules

- Divide heap into smaller chunks and collecting one chunk at a time
- Need **write barrier** to ensure correctness

  i.e. pointers from other chunks

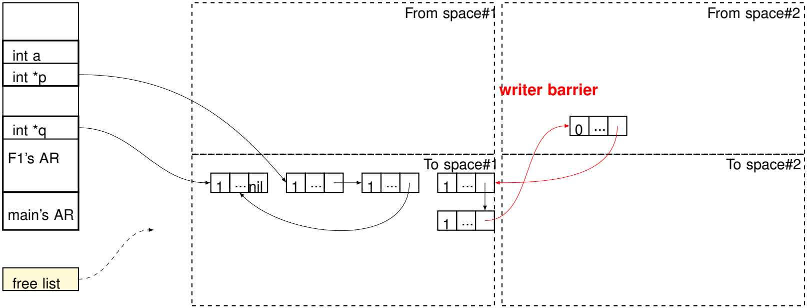


Algorithm 4: Incremental Garbage Collection

Rules

- Divide heap into smaller chunks and collecting one chunk at a time
- Need **write barrier** to ensure correctness

  i.e. pointers from other chunks



Evaluation of Incremental Garbage Collection

- Pause time is short due to smaller chunk to scan
- Each invocation reclaims smaller amount of free space
- Compatible with Semi-space, and Mark-and-sweep

Algorithm 5: Generational Garbage Collection

- Motivation
  - Most objects die young
- Rules
  - Divide the heap into several partitions (i.e. generations)
  - Objects are allocated from the current generation
  - When the current generation is full, move live objects to old generation

Comparing SemiSpace and Generational Collectors

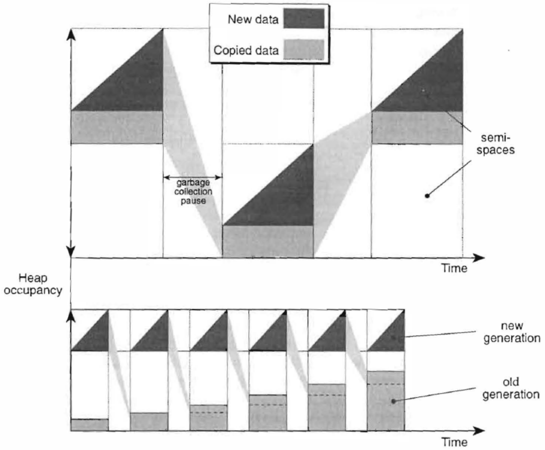
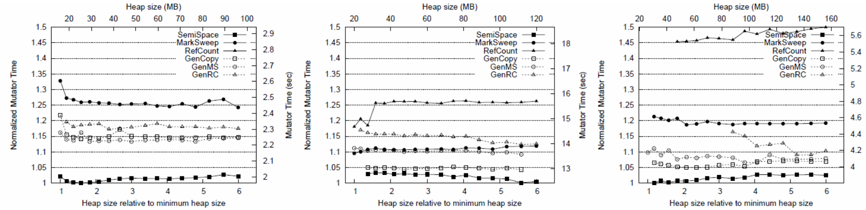


Diagram 7.4 Garbage collection pauses: a two-space copying collector (top) *vs.* a generational copying collector (bottom).

Comparison of Different GC algorithms (I)

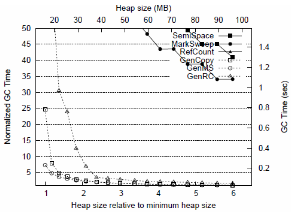
- Mutator time: user program execution time
  - ➢ Semi-space has the best performance due to improved locality
  - ➢ Mark-Sweep is the worst
  - ➢ Stable with heap size



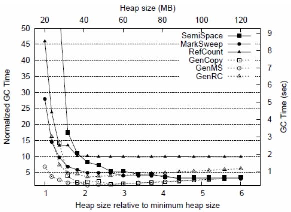
(g) _202_jess Mutator Time (h) _209_db Mutator Time (i) _213_javac Mutator Time

Comparison of Different GC algorithms (II)

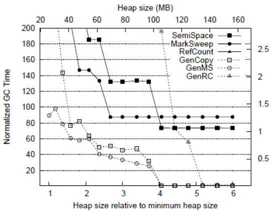
- Garbage collection time: GC overhead
  - Continuously tracking reference counters is expensive
  - Generational versions tend to incur low overhead



(d) _202_jess GC Time



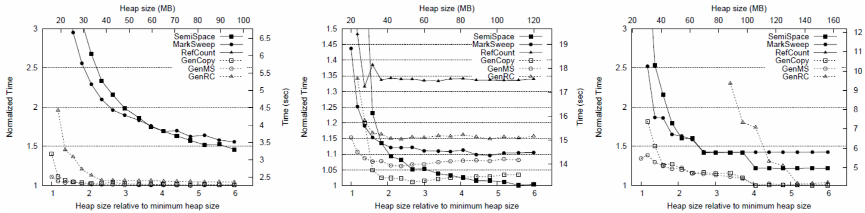
(e) _209_db GC Time



(f) _213_javac GC Time

Comparison of Different GC algorithms (III)

- Normalized total time



(a) _202_jess Total Time (b) _209_db Total Time (c) _213_javac Total Time