Runtime Environment
Compiler’s Role in Program Execution

- When a program is invoked
  - The OS allocates memory for the program
  - The code is loaded into the main memory
  - Program initializes runtime environment
  - Program jumps to entry point 'main()'

- **Runtime Environment**: The 'environment' in which the program executes in at runtime
  - Includes Main Memory, CPU, ...

- A compiler generates two types of code:
  - Code to implement semantics of program
  - Code to manage runtime environment

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>code of g()</td>
<td>global data segment</td>
</tr>
<tr>
<td>code of f()</td>
<td>heap</td>
</tr>
<tr>
<td>code of main()</td>
<td>stack</td>
</tr>
<tr>
<td>env</td>
<td></td>
</tr>
</tbody>
</table>

Memory References:
- 0x8000000
- 0xBFFFFFF
The Runtime Environment

- Runtime environment management code:
  - Allocate/deallocates storage in main memory (e.g. heap/stack)
  - Allocate storage in CPU (register allocation)
  - Map variables to physical allocated storage
  - Enforce language semantics e.g. static/dynamic scoping, ...

- We are going to focus on memory management in this section
  (Wait for code optimization lecture for register allocation!)

- Memory management core issue
  - Identifying the runtime address of a given name (e.g. variable, proc name) at compile time
  - Solution: generate addresses as offsets from addresses of allocated storage (which are decided at runtime)
Types of Memory 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 call
  - Deallocates storage on procedure return
    - e.g. local variables in C, C++, Java

- **Heap data management**
  - Allocates storage for objects that live across procedure boundaries
  - Deallocate when storage is no longer needed
    - e.g. malloc() in C, new in C++/Java
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 functions, FORTRAN compiler lays them out sequentially in memory
    - Address of local variable = offset in function + address of function
      (Can be determined statically)
  - 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
A list of AR (activation record) with their sizes known at compile time

FUNCTION F1(...)  
...  
END
FUNCTION F2(...)  
...  
END
FUNCTION FN(...)  
A = ...  
...  
END
A list of AR (activation record) with their sizes known at compile time

FUNCTION F1(...)  
...  
END  
FUNCTION F2(...)  
...  
END  
...  
FUNCTION FN(...)  
A = ... // variable access can be hard-coded  
... // using Global Base + offset  
END  

// store R1, 0x10002008;

Global Base:  

F1’s AR  
F2’s AR  
FN’s AR
Stack Based Storage Management

- Allocation strategy
  - Organize all local variables of a procedure in one AR (Activation Record) allocation unit
  - Manage ARs like a stack in memory
  - On function call: AR instance allocated at top of stack
  - On function return: AR removed from top of stack

- Hardware support
  - Stack pointer (SP) register
    - SP stores address of top of the stack
    - Allocation/de-allocation can be done by incrementing/decrementing SP
  - Frame pointer (FP) register
    - FP stores address of current AR
      - assists address mapping within AR
More About Stack-based Storage Management

- **Lifetime and scope**
  - Static scoping rule — static concept
  - Lifetime is dynamic concept
    - start: when the storage is allocated
    - end: when the storage is deallocated

- Stack storage management leverages lifetimes
  - Nested lifetime — P2 is allocated on top of P1
  - Disjoint lifetime — P2 reuses storage of P1
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)
class C {
    int g() {
        return 1;
    }

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

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

How does it work?

- code of g()
- code of f()
- code of main()
- global data segment
- heap
- main's AR
- fp_{main}
class C {
    int g() {
        return 1;
    }

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

    int main() {
        f(3);
        return 0;
    }
}
Example

How does it work?

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

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

    int main() {
        f(3);
        return 0;
    }
}

Code of g()

Code of f()

Code of main()

Global data segment

Heap

Main's AR

fp_{main}

x=3
(result)

y

1 ...
Example

How does it work?

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

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

    int main() {
        f(3);
        return y;
    }
}

code of g()
code of f()
code of main()
global data segment
heap

main's AR

y
location (①)
x=3 (result)

fp_{main}
Example

How does it work?

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

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

int main() {
    f(3);
}

code of g()
code of f()
code of main()

heap

main's AR

x=3
(result)

location (①)

fp

fp

fp

1 ... 2 ...
How does it work?

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

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

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

- code of g()
- code of f()
- code of main()
- global data segment
- heap

### Memory Layout

- `fp_main`
- `fp_f(3)`
- `x=3` (result)
- main's AR
- location (1)
- tmp=x-1
How does it work?

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

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

    int main() {
        f(3);
    }
}

code of g()
code of f()
code of main()
global data segment
heap

main's AR

x=3
(result)

fp

fp_{\text{main}}

fp_{f(3)}

tmp=x-1

location

y

\text{location (①)}
Example

How does it work?

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

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

    int main() {
        f(3);
    }
}

code of g()
code of f()
code of main()
global data segment
heap

main's AR

fp

main

x=3
(result)

fp

f

x=2
(result)
y

location (2)
n

location (1)

fp

f

f

x

fp

f(3)

fp

f(2)

x

fp

fp

main

fp
## Contents of Activation Record (AR)

### Layout of an AR for a function

<table>
<thead>
<tr>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporaries</td>
</tr>
<tr>
<td>Local variables</td>
</tr>
<tr>
<td>Saved Caller’s Register Values</td>
</tr>
<tr>
<td>Saved Return Address at Caller</td>
</tr>
<tr>
<td>Saved Caller’s AR Frame Pointer</td>
</tr>
<tr>
<td>Access Link to static parent’s AR</td>
</tr>
<tr>
<td>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>
Calling Convention

- **Caller’s responsibility**
  - Caller evaluates actual parameters in stores them in stack
  - Caller stores return address and its own FP in callee’s AR
  - Caller sets FP register to new position for callee’s AR

- **Callee’s responsibility**
  - Before execution:
    - Callee saves registers and other machine status information that was being used by caller
  - After execution:
    - Callee restores saved registers
    - Callee restores FP register to the caller’s FP
    - Callee jumps to return address at caller

- Rules on how the caller and callee interact is called the **calling convention**
AR Layout

- AR layout is determined at compile-time according to calling convention
  - Compiler already knows the number of local variables

- AR layout is designed for easy code generation
  - Placing the result as the first entry in callee’s frame simplifies caller finding the value

- AR layout is designed for execution speed
  - Some values (e.g. the first four parameters) can be kept in registers to speed up execution
    (for machine ISAs with many registers)

- Calling convention of system usually follows OS
  - OS has a certain convention decided by compiler used
  - Programs running on OS usually follows OS convention
    (since programs often have to do system calls into OS)
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
We use symbol names in 3-address code (IR) e.g. \texttt{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
  - Using relative addressing from $FP$ i.e. $FP + \text{offset}$
    - $FP$ — fixed for the lifetime of the corresponding function invocation
    - $\text{offset}$ — statically known (from the symbol table)
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);
        y
        return y;
    }

    int main() {
        f(3);
        return ...
    }
}

code of g()
code of f()
code of main()
global data segment
heap

main’s AR

fp$_{main}$
f$_{f(3)}$
x=3
(result)
y
location (①)

fp$_{f(2)}$
f$_{f(3)}$
x=2
(result)
y
location (②)

fp$_{main}$
fp$_{f(2)}$
How about Non-Local Variables?

- For languages with nested procedures
  - e.g. PASCAL, ALGOL 68
- How do we calculate address of non-local variables?
How about Non-Local Variables?

- For languages with nested procedures
  - e.g. PASCAL, ALGOL 68
- How do we calculate address of non-local variables?
- A possible guess .... X + offset?
How about Non-Local Variables?

- For languages with nested procedures
  - e.g. PASCAL, ALGOL 68
- How do we calculate address of non-local variables?
- A possible guess .... X + offset?
  - a good guess
  - but what is X? The FP of parent’s AR, of course.
How about Non-Local Variables?

- For languages with nested procedures
  - e.g. PASCAL, ALGOL 68
- How do we calculate address of non-local variables?
- A possible guess .... X + offset?
  - a good guess
  - but what is X? The FP of parent’s AR, of course.
- What is the difficulty?
  - AR of parent procedure may be at different relative locations on the stack at runtime
  - Impossible to calculate parent’s FP based on current FP
  - Need to traverse access links to find parent’s FP at runtime
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?

```
... 
P3 find x
P2 int x
P5
P5
P2 int x
P5
P4
P1
```
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 variable
  - variable \( x \) is translated to \((\text{diff}, \text{offset})\)
    - \( \text{diff} \) — 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

```plaintext
// 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 $parentfp
load $R2, off_{fp}($R2)
// variable y is saved in $parentfp + off_y
load $R3, off_y($R2)
```
Discussion of This Approach

- offset<sub>fp</sub> — a constant that indicates the distance of access link from $fp$.
- offset<sub>y</sub> — the offset of variable 'y' from $fp$. It takes different values for different variables.
Another Example

```c
void P0() {
    int I;
    int J;
}

void P1() {
    int K;
    int L;
    use I
}

void P2() {
    use K;
    use J;
    use I
}

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

<table>
<thead>
<tr>
<th>NestingLevel</th>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>J</td>
<td>4</td>
</tr>
<tr>
<td>P1</td>
<td>K</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>P3</td>
<td>H</td>
<td>0</td>
</tr>
</tbody>
</table>
Another Example

```c
void P0() {
    int I;
    int J;
    void P1() {
        int K;
        int L;
        void P2() {
            use K;
            use J;
        }
        use I
    }
    use I
}
void P3() {
    int H;
    use J
    use I
}
```

<table>
<thead>
<tr>
<th></th>
<th>NestingLevel</th>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>J</td>
<td>4</td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td>K</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>H</td>
<td>0</td>
</tr>
</tbody>
</table>

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

=> (P2'nestingLevel - P1'nestingLevel, K's offset)  
   => (1,0)
Another Example

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

<table>
<thead>
<tr>
<th>NestingLevel</th>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>J</td>
<td>4</td>
</tr>
<tr>
<td>P1</td>
<td>K</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>L</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>P3</td>
<td>H</td>
<td>0</td>
</tr>
</tbody>
</table>

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

=> (P2'nestingLevel-P1'nestinglevel, K's offset)
=> (1,0)
Example: P0 calls P1 calls P3 calls P1 calls P2

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 in most CPUs

- To speed up the access, we use **display**
  - Observation: given a nesting level $L$ at a point in execution, we have exactly $L$ active ARs to traverse
  - Display: a runtime data structure in memory that records the list of accessible ARs
  - Displays are implemented as arrays of accessible FPs
  - Now need only one memory operation to load FP for correct nesting level from 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

```c
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)
        }
    }
    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.

```c
void P0() {
    void P1() {
        void P2() {
        }
    }
    void P3() {
    }
}
```
The display needs to be updated when:

- a procedure is called, and
- a procedure is terminated

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

```
+----+----+----+
|    |    |    |
|    |    |  3 |
|  2 |  1 |  0 |
+----+----+----+
     display
```

```
void P0() {

}
```
The display needs to be updated when

- a procedure is called, and
- a procedure is terminated

```c
void P0() {
}

void P1() {
  void P2() {
  }
}

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

```
0 1 2 3
```

display

```
P0:
P1:
```

0 1 2 3
The display needs to be updated when

- a procedure is called, and
- a procedure is terminated

```c
void P0() {}
void P1() {
    void P2() {
    }
}
void P3() {
    void P4() {
    }
}
```
The display needs to be updated when

- a procedure is called, and
- a procedure is terminated

```c
void P0() {
}

void P1() {
    void P2() {
    }
}

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

```
   display
     3
     2
     1
     0
   P3
   P2
   P1:
   P0:
```
The display needs to be updated when

- a procedure is called, and
- a procedure is terminated

```c
void P0() {
}

void P1() {
    void P2() {
    }
}

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

```
<table>
<thead>
<tr>
<th></th>
<th>P0:</th>
<th>P1:</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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() {
    }
}
```

```
<table>
<thead>
<tr>
<th>display</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
</tr>
<tr>
<td>P1:</td>
</tr>
<tr>
<td>P0:</td>
</tr>
</tbody>
</table>
```
Saving/Restoring Displays

When a procedure terminates, the old display has to be restored.

There are three restore approaches:

1. Save entire Caller display on stack and restore on Callee termination
   - When P1 calls P2, save D[0], D[1] of P1 on stack
   - When P2 returns, restore both D[0] and D[1] from stack

2. Use access links to reconstruct the display
   - P1’s display is not saved on stack but reconstructed
   - Needed only when P2’s nesting Level (n2) ≤ P1’s nesting Level (n1),
   - Fix d[n2], d[n2+1], ..., d[n1] by traversing access links of P1

3. Save and restore one for each call
   - If P2’s nesting level is n1, save D[n1] and restore D[n1]
   - Idea: only save/restore the display entry overwritten
Approach 2: when P2 call P3

... P3’s nesting level is 1

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

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

```plaintext
display

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

P0: P1:
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

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

```c
void P0() {
    void P1() {
        void P2() {
        }
    }
}
void P3() {
    void P4() {
    }
}
void P4() {}
```
Approach 2 Illustrated

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

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
display

P0: P1: P2: P3: P4:
```
Approach 2 Illustrated

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

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

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

```
<table>
<thead>
<tr>
<th>display</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P2</td>
<td></td>
<td>P1</td>
<td>P0</td>
</tr>
</tbody>
</table>
```
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];

```c
void P0() {
}
void P1() {
    void P2() {
    }
}
void P3() {
    void P4() {
    }
}
void P4() {
}
```
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];
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];
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: 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 because all entries are saved/restored.
- Approach 2 only incurs overhead when $n_2 \leq n_1$ but has to traverse $n_1 - n_2$ access links to restore as many entries.
- Approach 3 only incurs overhead when $n_2 \leq n_1$ and has constant overhead (i.e. one save/restore per call).
Translating Parameters

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

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

- **Call by value**
  - Formal parameter is treated like a local variable
  - Caller places value in storage element for formal parameter
  - Good for guaranteeing actual parameters stay constant

- **Call by reference**
  - Address for parameter is passed as the value of parameter
  - If actual is an expression then compute the expression into a temporary and pass the address of temporary
  - Good when actual parameters need to change
  - Good for passing large data structures efficiently

- **Call by name**
  - When the function is called, parameters are not evaluated
  - When a parameter is used, evaluate the parameter in the environment of caller
  - Good for efficiency since parameter evaluations can be skipped if not used
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);
}
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);
}
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);
}
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);
}
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

```c
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);
}
```
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);
}
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 Name

- Originated in ALGOL, still used in many functional languages such as Haskell

- Rule
  - Goal: avoid evaluation of parameters when not needed
    - When function is called, parameters are not evaluated
    - When parameters is used, evaluate the parameters in the environment of caller
  - Implementation:
    - Code called **thunk** generated inside caller for evaluation
    - Pass address of thunk to callee
    - Callee jumps to thunk whenever parameter value is needed
  - How to recreate environment of caller:
    - On thunk entry, update FP to that of caller
    - On thunk exit, update FP to that of callee
int f(int \diamond x, int \diamond 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

```c
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?

```c
b=2
x=?
y=?
a=1
b=1
```
The Problem of Call-by-Name

```c
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));
}
```

Variables initialized:
- a = 1
- b = 1
- f(): x = ?, y = ?
After binding variables to declarations using the symbol table, we are ready to translate IR (AST in our case) to binary code.
After binding variables to declarations using the symbol table, we are ready to translate IR (AST in our case) to binary code.

However, we will only generate very inefficient code:

- Inefficient use of registers
- Inefficient use of instruction types
After binding variables to declarations using the symbol table, we are ready to translate IR (AST in our case) 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

\begin{align*}
\text{lw } & R1, \text{ offset}(R2) & \text{# load one word from offset + R2 to R1} \\
\text{add } & R1, R2, R3 & \text{# } R1 \leftarrow R2 + R3 \\
\text{addiu } & R1, R2, \text{ imm} & \text{# } R1 \leftarrow R2 + \text{ imm, overflow unchecked} \\
\text{sw } & R1, \text{ offset}(R2) & \text{# store R1 to offset+R2} \\
\text{li } & R1, \text{ imm} & \text{# } R1 \leftarrow \text{ imm}
\end{align*}
cgen(e1+e2):

# stores result in $t0
cgen(e1)
# saves $t0 on stack
sw $t0, 0($sp)
addiu $sp, $sp, -4
# stores result in $t0
cgen(e2)
# restores value of e1
addiu $sp, $sp, 4
lw $t1, 0($sp)
# performs addition
add $t0, $t1, $t0

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

cgen(e1)
# saves $t0 on stack
sw $t0, 0($sp)
addiu $sp, $sp, -4
cgen(e2)
# restores value of e1
addiu $sp, $sp, 4
lw $t1, 0($sp)
# performs comparison
beq $t0, $t1, Tpath

Fpath: cgen(e4)
b End

Tpath: cgen(e3)
End: ...
Code Generation for Function Call

cgen(f(e1)) =
# push arguments
  cgen(e1)
  sw $t0, 0($sp)
  addiu $sp, $sp, -4
# push old FP
  sw $fp, 0($sp)
  addiu $sp, $sp, -4
# push return address
  sw $ra, 0($sp)
  addiu $sp, $sp, -4
# begin new AR in stack
  move $fp, $sp
  j fEntry

cgen(def(f(...):e)) =
  fEntry: cgen(e)
  # remove AR from stack
  move $sp, $fp
  # pop return address
  addiu $sp, $sp, 4
  lw $ra, 0($sp)
  # pop old FP
  addiu $sp, $sp, 4
  lw $fp, 0($sp)
  # jump to return address
  jr $ra
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

First local variable: -4($fp)
Argument X1: +4($fp)

New $sp

New $fp

...  Temp
Local Variable
Return Address
X1
X2
...  Xn
old(fp)
Buffer Overflow Attacks (BOAs)

- BOA is a major type of security threat

Code example

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

void main()
{
    foo();
}
```

Diagram:
- Stack grows downwards.
- Array grows to the left.
- Program execution flow:
  - `main()` calls `foo()`
  - `foo()` fills `a[0]` to `a[3]` from left to right.
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?

- Array bounds checks
  - Java Virtual Machine inserts these for every array access

- Canary word
  - A random word in stack before return address
  - Check canary word before returning
  - BOA will overwrite canary before return address

- Legal jump targets
  - Compiler does call graph analysis to hardcode legal return targets into binary
  - Compare return address to legal targets before returning

- Randomization
  - Randomize size of AR to make attacks harder

- Many other defending techniques
Objects are like structures in C

- Objects are laid out in contiguous memory
- Each member variable is stored at a fixed offset in object

Unlike structures, objects have member methods

- Two types of member methods:
  - Nonvirtual member methods: cannot be overridden
    Parent obj = new Child();
    obj.nonvirtual(); // Parent::nonvirtual() called
    Method called depends on (static) type of reference
    Compiler can decide call targets statically
  - Virtual member methods: can be overridden by child class
    Parent obj = new Child();
    obj.virtual(); // Child::virtual() called
    Method called depends on (runtime) type of object
    Need to call different targets depending on runtime type
Static Dispatch and Dynamic Dispatch

- **Dispatch**: to send to a particular place for a purpose
  - In CS: to select which implementation of a polymorphic method to jump to

- **Static Dispatch**: to select call target at compile time
  - Nonvirtual methods implemented using static dispatch
  - Implication for code generation:
    Can hard code function address into binary

- **Dynamic Dispatch**: to select call target at runtime
  - Virtual methods implemented using dynamic dispatch
  - Implication for code generation:
    Must generate code to select correct call target
  - How?
    - At compile time, generate a dispatch table for each class
      Contains call targets for all virtual methods for that class
    - At runtime, each object maintains a pointer to the dispatch table for its runtime type (class)
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 dispatch table
- Member variables are allocated in subsequent slots
Invariant — the offset of a member variable or member method is the same in a class and all of its subclasses. For example, if $A_3 \leq A_2 \leq A_1$.
Inheritance and Subclasses

- **Member variable access**
  - Generate code using offset for type (class) of reference
  - Object may be of child type, but will still have same offset

- **Member method call**
  - Generate code to load call target from dispatch table using offset for type of reference
  - Again, object may be of child type, but still same offset

- **No inheritance in our project**
  - No dynamic dispatching
  - Statically bind a function call to its address
Heap Memory Management
Garbage Collection
Heap Management

- Heap data
  - Lives beyond the lifetime of the procedure that creates it
  ```c
  TreeNode* createTREE() {
  ....
  TreeNode* p = (TreeNode*)malloc(sizeof(TreeNode));
  return p;
  }
  ```
  - Cannot reclaim automatically using a stack

- Problem: when and how do we reclaim that memory?

- Two approaches
  - Manual memory management
    Programmer inserts deallocation calls. E.g. “free(p)”
  - Automatic memory management
    Compiler generates code to automatically reclaim memory when that data is ‘no longer needed’
Why Automatic Memory Management?

- Leads to fewer bugs
  - With manual management, programmers may
    - forget to free unused memory → memory leak
    - free memory too early → dangling pointer access
    - free memory twice (double free)
  - Symptoms: slowly increasing memory footprint, unexplained data corruption, program crashes ...
  - Memory bugs are notoriously hard to find and fix

- Leads to a more secure system
  - Secure languages run on top of a virtual machine
    (E.g. Java, JavaScript, PHP ...)
  - Control of memory management enables security guarantees
    - Compiler knows where every piece of data is
    - Compiler can prevent applications from accessing data it should not have access to
Why Not Automatic Memory Management?

- Manual memory management is typically more efficient
- Runtime system must ‘figure out’ when data is no longer needed
  - Memory wastage: runtime may keep around data longer than necessary
  - Time wastage: process of figuring out is often costly
Automatic Memory Management

Old but new
- Old idea: Studied since 1950s for LISP programming language
- New significance: Taking on new significance due to the importance of security and secure languages such as Java

Basic idea:
- Runtime system maintains list of used/unused spaces in heap memory
- On object creation: unused space of its size is allocated
- When an object is no longer used, it is automatically moved to unused space
The Difficulty

- How to determine an object will no longer be used
  - In general, impossible to tell exactly
    - After all, how can you predict the future?
    - Requires knowledge of program beyond what compiler has
  - But can tell when it *can* no longer be used

- **Axiom**: an object can only be used if it can be referenced
  - **Corollary 1**: if an object cannot be referenced, the object can no longer be used
    
    ```java
    Object obj = new Object();
    ...
    obj = NULL; // above object can no longer be used
    ```

- Heap allocated objects are called **nameless objects**
- Rest (including references) are called **named objects**

- **Corollary 2**: if there is no path from a named object, the object can no longer be used
Reachable Objects and Garbage

- Named objects can be
  - global variables in global data segment
  - local variables in
    - stack memory
    - registers

- An object $x$ is **reachable** iff
  - A named object contains a reference to $x$, or
  - Another reachable object $y$ contains a reference to $x$

- An unreachable object is referred to as **garbage**
  - Garbage can no longer be used and can be reclaimed
  - This reclamation process is called **garbage collection**
Two Garbage Collection Schemes

- **Reference Counting**
  - Maintains a counter inside each object that counts the number of references to object
  - When counter becomes 0, the object is no longer reachable
  - Immediately reclaim the unreachable object as garbage

- **Tracing**
  - When the heap runs out of memory to allocate:
    1. Stops the program and traces through all reachable objects starting from the named objects
    2. Remaining objects are accumulated garbage and are reclaimed
    3. Restarts the program
Scheme 1: Reference Counting

- Intuition: if no references point to an object, that object is unreachable
  - It is a sufficient condition, but is it a necessary condition?
  - In other words: Is it necessary for there to be no references, for that object to be unreachable?
  - No. Problems with cyclic references that we will visit soon.

- Implementation: each object has a counter that counts the number of references pointing to the object
  - The counter of object x is referred as its reference counter
  - Notation $rc(x)$ refers to reference counter inside object x
  - Each reference assignment requires manipulation of reference counters
Initialization $x = \text{new Object}()$: 

- $x \leftarrow \text{new Object}()$;
- $rc(\text{new object}) \leftarrow 1$;

Reference assignment $x = y$:

Assume references $x, y$ point to objects $p, q$ respectively

- $x \leftarrow y$;
- $rc(q) \leftarrow rc(q) + 1$;
- $rc(p) \leftarrow rc(p) - 1$;
- if $(rc(p)=0)$ then
  - mark $p$ as garbage to reclaim;
  - for each object $o$ referenced by $p$
    - $rc(o) \leftarrow rc(o) - 1$;
    - if $(rc(o)=0)$ then
      - mark $o$ as garbage to reclaim;
      - recursively repeat for $o$;
Reference Counting Example

- int a
- int *p
- int *q
- F1’s AR
- main’s AR
- free list

Heap:

- 1...
- 2...
- 1...
- 1...
- 2...
- 1...
- nil
Reference Counting Example

int a
int *p
int *q
F1's AR
main's AR
free list

Heap

1 ...
2 ...
1 ...
1 ...
2 ...
1 ...
1 ...
nil
Reference Counting Example

int a
int *p
int *q
F1’s AR
main’s AR

free list

Heap

1...
1...
1...
2...
3...
0...
nil
Reference Counting Example

Free list

Heap

int a
int *p
int *q
F1's AR
main's AR

1 ...
2 ...
1 ...
2 ...
1 ...
nil
Problem of Reference Counting

RC cannot handle **circular data structures**

![Diagram showing reference counting issue with circular data structures]

- int a
- int *p
- int *q
- F1’s AR
- main’s AR
- free list

Heap

1...
2...
1...
2...
1...
nil
RC cannot handle **circular data structures**

```
int a
int *p
int *q
F1's AR
main's AR
free list
```

Heap

```
1 ...
1 ...
1 ...
1 ...
nil
2 ...
1 ...
2 ...
```

...
RC cannot handle **circular data structures**

```
int a
int *p
int *q
F1's AR
main's AR
```

```
free list
```
RC cannot handle circular data structures

- int a
- int *p
- int *q
- F1’s AR
- main’s AR
- free list

Heap

1 ...
2 ...
1 ...
2 ...
1 ...

Garbage but cannot reclaimed
Discussion of Reference Counting Scheme

Advantages:

- Relatively easy to implement
  - Compiler only needs to insert RC manipulation code at reference assignments

- Collects incrementally
  - Garbage is collected immediately ‘on the fly’
  - No need to pause program for a long time to collect accumulated garbage
    (Can take minutes depending on the size of the heap!)
  - Program is more responsive compared to tracing collection

Disadvantages:

- Cannot collect circular data structures
- Manipulating RCs at each assignment is slow
- Counter manipulation must be synchronized in multithreaded programs
  → even slower
- RC typically has better worst-case response times compared to tracing but worse overall performance
Discussion of Reference Counting Scheme

Advantages:

- Relatively easy to implement
  - Compiler only needs to insert RC manipulation code at reference assignments
- Collects incrementally
  - Garbage is collected immediately ‘on the fly’
  - No need to pause program for a long time to collect accumulated garbage
    - (Can take minutes depending on the size of the heap!)
  - Program is more responsive compared to tracing collection

Disadvantages:

- Cannot collect circular data structures
- Manipulating RCs at each assignment is slow
  - Counter manipulation must be synchronized in multithreaded programs → even slower
  - RC typically has better worst-case response times compared to tracing but worse overall performance
Scheme 2: Tracing

- Start from named objects (also called root objects)
  - If object is value: no further action
  - If object is reference: follow reference
  - If object is struct: go through each field

- Example: F1’s local variable references heap object

```
int a
int *p
struct x *q

F1's AR
main's AR
```
Scheme 2: Tracing

- Start from named objects (also called **root objects**)
  - If object is value: no further action
  - If object is reference: follow reference
  - If object is struct: go through each field

- Example: F1’s local variable references heap object

```
int a
int *p
struct x *q
F1's AR
main's AR
```
Scheme 2: Tracing

- Start from named objects (also called root objects)
  - If object is value: no further action
  - If object is reference: follow reference
  - If object is struct: go through each field

- Example: F1’s local variable references heap object

```
int a
int *p
struct x *q
F1’s AR
main’s AR
```
Scheme 2: Tracing

- Start from named objects (also called **root objects**)
  - If object is value: no further action
  - If object is reference: follow reference
  - If object is struct: go through each field

- Example: F1’s local variable references heap object
Discussion of Tracing Scheme

- Advantages:
  - Is guaranteed to collect all garbage eventually (even when there are circular references)
  - No need to track reference counts on each reference update
Discussion of Tracing Scheme

Advantages:
- Is guaranteed to collect all garbage eventually (even when there are circular references)
- No need to track reference counts on each reference update

Disadvantages:
- Requires pausing program while tracing through references
- Requires tracing through all live objects for every collection
  - Even when the amount of garbage collected is small
  - Requires heap to have extra ‘headroom’
  - If heap is too tight, results in repeated GCs to collect small amounts of garbage (heap thrashing)
Flavors of Tracing

- To move or not to move?
  - Garbage collection can leave ‘holes’ inside heap
  - Holes can cause: fragmentation, loss of access locality
  - Objects can be moved in the course of GC to prevent holes
  - **Mark-and-Sweep**: example of non-moving GC
  - **Semispace**: example of moving GC

- To collect at once or incrementally?
  - Tracing GC requires programs to be paused
  - Tracing entire heap can lead to long pause times
  - It is possible to collect only a part of the heap
  - **All-at-once**: naive GC with no partitions
  - **Incremental**: divides heap into multiple partitions
  - **Generational**: divides heap into generations

- Possible to mix-and-match the two different flavors
Flavor 1: Mark-and-Sweep

- When it is about to run out of memory, GC stalls program execution and executes two phases
  - Mark phase: traces through reachable objects starting from root 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

- int a
- int *p
- int *q

F1’s AR

main’s AR

free list

Heap

- 0 ...
- 0 ...
- 0 ...
- 0 ...
- 0 ...
- 0 ...
- nil

Heap pointer
Mark-and-Sweep Example

- int a
- int *p
- int *q
- F1’s AR
- main’s AR

free list

Heap

1 ... nil
0 ...

0 ...

1 ...

1 ...

0 ...

nil
Mark-and-Sweep Example

F1’s AR
- int a
- int *p
- int *q

main’s AR

free list

Heap
- 1...
- 1...
- 1...
- nil
- nil
Mark-and-Sweep Example

```
int a
int *p
int *q
F1's AR
main's AR
free list

Heap
```

```
0 ...
0 ...
0 ...
0 ...
0 ...
nil
```
Discussion of Mark-and-Sweep Collection

Advantages:

- No need to move objects → simpler implementation
- GC time is shorter since objects stay in place
Discussion of Mark-and-Sweep Collection

Advantages:
- No need to move objects → simpler implementation
- GC time is shorter since objects stay in place

Disadvantages:
- Fragmentation: free list is fragmented across heap
  - On allocation: must search through free list to find appropriate size
  - Might be unable to find appropriate size even though there is enough total free space
- Loss of Locality: increasing number of ‘holes’ in heap
  - Accesses to objects become increasingly distributed across a wide range of address → loss of locality
  - Loss of locality can cause CPU cache misses, page faults
Flavor 2: Semispace

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

```
<table>
<thead>
<tr>
<th></th>
<th>From space</th>
<th>To space</th>
</tr>
</thead>
<tbody>
<tr>
<td>int a</td>
<td>1 ...</td>
<td>0 ...</td>
</tr>
<tr>
<td>int *p</td>
<td>1 ...</td>
<td>0 ...</td>
</tr>
<tr>
<td>int *q</td>
<td>1 ...nil</td>
<td>0 ...</td>
</tr>
<tr>
<td>F1's AR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>main's AR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>free list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Flavor 2: Semispace

- Use half of the heap space
- When collecting garbage, copy live objects to the other half
- Install **forward pointers** to assist moving objects
Flavor 2: Semispace

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

```
<table>
<thead>
<tr>
<th>int a</th>
<th>int *p</th>
<th>int *q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1’s AR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>main’s AR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
1 ... ~   1 ...
           1 ... nil   0 ...
0 ...     0 ...
1 ... nil
```

From space

To space

Free list
Flavor 2: Semispace

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

```
int a
int *p
int *q
F1's AR
main's AR
```

```
From space

1 ... nil
0 ...
```

```
To space

1 ... nil
1 ...
```

Free list
Flavor 2: Semispace

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

```c
int a
int *p
int *q
F1's AR
main's AR
free list
```

Diagram:
- From space:
  - `1 ... nil` → `1 ...
  - `0 ...
- To space:
  - `1 ... nil` → `1 ...
  - `0 ...

From space:
- `1 ...` → `0 ...
- `0 ...
  - `1 ...
- `1 ... nil` → `0 ...
  - `1 ...
- `0 ...
  - `1 ...
- `1 ... nil` → `0 ...
  - `1 ...
- `0 ...
```
### Rules
- Use half of the heap space
- When collecting garbage, copy live objects to the other half
- Install **forward pointers** to assist moving objects
Flavor 2: Semispace

- Use half of the heap space
- When collecting garbage, copy live objects to the other half
- Install **forward pointers** to assist moving objects
Flavor 2: Semispace

**Rules**

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

```
int a
int *p
int *q
F1's AR
main's AR
```

```
1 ... nil 1 ... 1 ...
```

```
From space
```

```
To space
```

```
free list
```
Advantages:

- No fragmentation → fast allocation
- Better locality → faster program execution
  - Objects are compacted on the heap (no holes)
  - Objects are laid out in the order of allocation
    (Objects allocated together tend to in same working set)
Advantages:

- No fragmentation → fast allocation
- Better locality → faster program execution
  - Objects are compacted on the heap (no holes)
  - Objects are laid out in the order of allocation
    (Objects allocated together tend to in same working set)

Disadvantages:

- Only half of heap space usable → more frequent GC
- Objects need to be copied → longer GC
Flavor 3: Incremental

- Rules
  - Divide heap into smaller partitions and collect one partition at a time
  - Needs **write barrier** to ensure correctness
    - i.e. Adds pointers from other partitions to root set
Flavor 3: Incremental

Rules

- Divide heap into smaller partitions and collect one partition at a time
- Needs **write barrier** to ensure correctness
  i.e. Adds pointers from other partitions to root set
Flavor 3: Incremental

- Rules
  - Divide heap into smaller partitions and collect one partition at a time
  - Needs **write barrier** to ensure correctness
    i.e. Adds pointers from other partitions to root set

```
int a
int *p
int *q
F1's AR
main's AR
```

```
free list
```

```
From space#1
1 ... 1 ... 1 ...
1 ... nil
0 ... 1 ...
To space#1
1 ... nil
1 ...
1 ...

From space#2
1 ...

To space#2
0 ...
```

**writer barrier**
Flavor 3: Incremental

- **Rules**
  - Divide heap into smaller partitions and collect one partition at a time
  - Needs **write barrier** to ensure correctness
    - i.e. Adds pointers from other partitions to root set

Diagram:

- **Free list**
- **F1's AR**
- **Main's AR**

Diagram shows the movement of objects between two spaces, with a writer barrier to indicate correctness.

```
int a
int *p
int *q
F1's AR
main's AR
free list
```
Discussion of Incremental Collection

Advantages:

- Pause time is shorter due to smaller partitions

Disadvantages:

- Each invocation reclaims smaller amount of free space
- Overhead of write barriers: must check for cross-partition references on every reference modification
Discussion of Incremental Collection

- Advantages:
  - Pause time is shorter due to smaller partitions

- Disadvantages:
  - Each invocation reclaims smaller amount of free space
  - Overhead of write barriers: must check for cross-partition references on every reference modification
Flavor 4: Generational Garbage Collection

- **Motivation: Generational Hypothesis**
  - Empirical observation that most objects die young

- **Rules**
  - Divide heap into two partitions: young / old generations
  - Stages in an object’s life:
    1. Objects are initially allocated to young generation
    2. Most objects die young and are collected in the next young generation GC
    3. The few objects that survive several young generation GCs are moved to the old generation
    4. Old generation objects eventually die and are collected in the next old generation GC
Discussion of Generational Collection

- How does generational collection help?
  - Most GC happens in the young generation where GC is very efficient since there are few surviving objects to trace.
  - Old generation GC happens only when old generation overflows → very infrequently.
  - Typical GC collects only young generation → very short.

- Advantages:
  - Most GC pause times are short since only young generation collection is required.

- Disadvantages:
  - Sometimes old generation has to be collected, albeit infrequently.
  - Overhead of write barriers: must check for cross-generation references on every reference modification.
Diagram 7.4 Garbage collection pauses: a two-space copying collector (top) vs. a generational copying collector (bottom).
Comparison of Different GC algorithms

Based on this publication


Studied 6 configurations on Java benchmarks:

- RefCount: reference counting
- Semispace: moves objects from ‘from’ to ‘to’ space
- MarkSweep: adds objects to free list without moving
- GenCopy: young = Semispace, old = Semispace
- GenMS: young = Semispace, old = MarkSweep
- GenRC: young = Semispace, old = Reference Counting
Comparison of Different GC algorithms (I)

- **Mutator time**: user program execution time
  - Semispace has best performance due to improved locality
  - Generational has good performance since young generation is Semispace (and most accesses happen in young generation)
  - Mark-Sweep has worst performance due to little locality
  - Generational versions incur overhead of write barriers
Garbage collection time: GC overhead

- Heap thrashing observed at close to minimum heap sizes
- Generational versions tend to incur lower overhead
- Reference counting performs badly due to constant RC updates (but has better response times not shown here)
Comparison of Different GC algorithms (III)

- Normalized total time
  - There is no one GC algorithm that performs the best for all applications and all heap sizes
  - Software companies have dedicated teams to tune garbage collector to application characteristics

(a) .202.jess Total Time
(b) .209.db Total Time
(c) .213.javac Total Time