Runtime Environment
Runtime Environment

- **Runtime Environment**: The ‘environment’ in which the program executes in at runtime
  - Includes HW: Main memory, CPU, ...
  - Includes OS: Environment variables, ...

- 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()’

- All program binaries include two parts:
  - Code implementing semantics of program
  - Runtime code
Runtime Code

- **Runtime code**: any code not implementing semantics
  - Code to manage runtime environment
    - Manage main memory storage (e.g. heap/stack)
    - Manage CPU register storage
    - Manage multiple CPUs (for languages with threading)
  - Code to implement language execution model
    - Code to pass function arguments according to model
    - Code to do dynamic type checking (if applicable)
    - Code to ensure security (if applicable)
  - May even include compiler itself! (just-in-time compiler)

- Some runtime codes are pre-fabricated libraries
  - E.g. Heap data management, threading library ...

- Some generated by compiler, interleaved in program code
  - E.g. Stack data management, register management, argument passing, type checking, ...
3 Types of Memory Management

- **Static** data management
  - Store data in statically (compile-time) allocated area
  - Good for data that must last for entire duration of program
    - Program code: code must last for duration of program
    - Global variables: globals can be accessed at any point

- **Stack** data management
  - Store data in area managed like a stack
  - Good for data with lifetime associated with (nested) scopes
    - Data pushed / popped when scope entered / exited
    - Local variables: locals are only valid for that scope

- **Heap** data management
  - Store data in area that allows on-demand allocation
  - Good for data that is created / deleted on-demand
    - Data managed by malloc() / free() in C
    - Data created by new Object() in C++ and Java
Example Memory Layout

Example layout of program memory at runtime

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000000</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>code of g()</td>
</tr>
<tr>
<td></td>
<td>code of f()</td>
</tr>
<tr>
<td></td>
<td>code of main()</td>
</tr>
<tr>
<td>0xBFFFFFF</td>
<td>global data segment</td>
</tr>
<tr>
<td></td>
<td>heap</td>
</tr>
<tr>
<td></td>
<td>stack</td>
</tr>
<tr>
<td></td>
<td>env</td>
</tr>
</tbody>
</table>
Static Storage Management

- Laid out at compile-time since contents never change at runtime

- Case study: Fortran’s data allocation
  - Allocation strategy
    - Lay out code+data for each function sequentially in memory
    - Address of variable = address of function + offset in function
  - Limitations
    - No recursive functions (need multiple instances of function)
    - More space: need to allocate sum of storage for all functions even though only a subset will be active at one time
  - But faster thanks to better data locality due to static layout
    - Accesses to same variable will always land in same location
    - Results in more cache hits and more page hits
Name Address Translation

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

FUNCTION F1(...)
  ...
END
FUNCTION F2(...)
  ...
END
FUNCTION FN(...)
  A = ...
  ...
END

F1’s AR
F2’s AR
FN’s AR
Name Address Translation

- 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

- offset
Stack Based Storage Management

- **Allocation strategy**
  - Organize all local variables within a function in one AR (Activation Record) allocation unit
  - Manage ARs like a stack in memory:
    - On function entry: AR instance allocated at top of stack
    - On function return: AR instance 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 base address of current AR
      (can calculate variable addresses as an offset from FP)
More About Stack-based Storage Management

- **Lifetime and scope**
  - Scoping: static concept (is A lexically nested in B?)
  - Lifetime: dynamic concept (is A active while B is still active?)

- Nested lifetime — P2 is allocated on top of P1
- Disjoint lifetime — P2 reuses storage of P1

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 recursive functions
- Allocate only as much storage as needed at one time

Disadvantages:
- Bad locality: AR for same function may land in different locations depending on call stack below function
- Security concerns
  - Buffer overflow attack (BOA)
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}
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}
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

y

x=3
(result)

main's AR

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()
global data segment
heap

location (①)
x=3 (result)
main's AR

fpmain

① ...
② ...

y
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

x=3 (result) location (①)
f_{main} fp_{main}
y ② ...

x=3
location (①)
f_{f(3)} fp_{f(3)}
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);
        y
        ... 
        return y;
    }

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

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

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

Code of g()

Code of f()

Code of main()

Global data segment

Heap

tmp=x-1

y

Location (1)

f_p_{main}

x=3
(result)

main's AR

f_p_{f(3)}

f_p_{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()

global data segment

heap

y
location (②)
fp_{f(3)}
x=2
(result)
tmp=x-1
y
location (①)
fp_{main}
x=3
(result)
main's AR

fp_{f(2)}

fp_{f(3)}

fp_{main}
### Contents of Activation Record (AR)

#### Layout of an AR for a function

<table>
<thead>
<tr>
<th>Section</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>Parameters</td>
</tr>
<tr>
<td>Return Value</td>
</tr>
</tbody>
</table>
Calling Convention

- **Caller’s responsibility**
  - Caller evaluates arguments and 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**
  - 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**

- Includes rules on sharing CPU registers
  - Some registers are saved/restored to/from stack by caller
    (A subset called **caller-saved registers**)
  - Some registers are saved/restored to/from stack by callee
    (A subset called **callee-saved registers**)

AR Layout

- Designed for easy code generation
  - Code to create/access AR must be generated by compiler
  - E.g. placing "return value" before "local variables" in callee’s frame enables caller to find value easily (even when callee is library function, and caller does not know how many local variables there are)

- Designed for execution speed
  - First four arguments of MIPS passed in registers (Register accesses are faster than stack accesses)

- Who decides on the calling convention?
  - Entirely up to the compiler writer
  - When linking modules generated by different compilers, care must be taken that same conventions are followed (e.g. Java Native Interface allows calls from Java to C)
Translating IR to Machine Code

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

- When generating machine code
  - Symbols have to be translated to memory addresses

- For static data (global variables):
  - Allocated in global data segment
  - Statically known: `global_base + offset`

- For stack data (local variables):
  - Using relative addressing from $FP$ i.e. $FP + offset$
    - $FP$ — Generate code to update at function call / return
    - offset — statically known (from the symbol table)
Stack Data

Note how all data can be addressed relative to FP

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

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

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

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

y
location (2)
fp f(3)
x=2
(result)
tmp=x-1
y
location (1)
fp main
x=3
(result)
main's AR

fp f(2)
fp f(3)
fp main

1 2
```

```plaintext
x=3
location
x=2
location
```
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

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
sw R1, offset(R2)  # store R1 to offset+R2
add R1, R2, R3 # R1 ← R2 + R3
addiu R1, R2, imm # R1 ← R2 + imm, overflow unchecked
move R1, R2 # R1 ← R2
li R1, imm # R1 ← imm
```
Code Generation for Expressions

cgen(e1+e2):
    # stores result in $t0
    cgen(e1)
    # push $t0 on stack
    addiu $sp, $sp, -4
    sw $t0, 0($sp)
    # stores result in $t0
    cgen(e2)
    # pop value of e1 to $t1
    lw $t1, 0($sp)
    addiu $sp, $sp, 4
    # performs addition
    add $t0, $t1, $t0

cgen(if (e1==e2) then e3 else e4):
    cgen(e1)
    # push $t0 on stack
    addiu $sp, $sp, -4
    sw $t0, 0($sp)
    cgen(e2)
    # pop value of e1 to $t1
    lw $t1, 0($sp)
    addiu $sp, $sp, 4
    # 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)
    addiu $sp, $sp, -4
    sw $t0, 0($sp)
    # push old FP
    addiu $sp, $sp, -4
    sw $fp, 0($sp)
    # push return address
    addiu $sp, $sp, -4
    sw $ra, 0($sp)
    # 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
        lw $ra, 0($sp)
        addiu $sp, $sp, 4
        # pop old FP
        lw $fp, 0($sp)
        addiu $sp, $sp, 4
        # jump to return address
        jr $ra
Code Generation for Variables

- Local variables are referenced from an offset from $fp
  - In example, $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
  Temporaries
  Local Variables
  Return Address
  Old $fp
  X1
  X2
  ...
  Xn

new $fp
```

first local variable: -4($fp)
argument X1: +8($fp)
Buffer Overflow Attacks (BOAs)

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

```c
int foo()
{
    int a[4];
    int i=0;
    while (1) {
        a[i] = getc();
        if (a[i] == '.')
            break;
        i++;
    }
}

void main()
{
    foo();
}
```
When Return Address is Overwritten

What happens when *foo()* finishes its execution

```assembly
foo: ...
    # pop return address
    lw $ra, 0($sp)
    addiu $sp, $sp, 4
    ...
    # jump to return address
    jr $ra
```

When provided a malicious input string?

```
"... ... 00 10 00 00 "
(20Bytes) (addr of malicious 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

- Control flow integrity
  - Compiler does call graph analysis to record legal return targets at the end of each function body
  - Compare return address to legal targets before returning

- Randomization
  - Randomize size of AR to make attacks harder

- Many other defending techniques
Translating Parameters

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

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

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

- How about parameters and arguments?
  ```
  int func1(int a, int b) { ... }
  ...
  ... z = z + func1(x, y);
  - Parameters `a, b` — the names used when a function is declared
  - Arguments `x, y` — the expressions passed when a function is called
Calling Convention

- **Call by value**
  - Semantic: Copy value of argument to parameter
  - Caller copies argument value to parameter location
  - (Stack location or register depending on calling convention)

- **Call by reference**
  - Semantic: Parameter references same location as arg
    - Updates to parameters is reflected on argument
    - Argument must be a location, not merely a value
  - Caller copies pointer to argument to parameter
    - Callee accesses argument through pointer

- **Call by name**
  - Semantic: Arg evaluated whenever parameter accessed
    - Whenever a parameter is accessed in callee, argument expression evaluated in the environment of caller
    - Arguments are not evaluated immediately on function call
    - Good for efficiency: unused arguments not evaluated
  - Caller copies address of evaluation code to parameter
Call by Value

```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);
}
```
Call by Value

```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);
}
```
Call by Value

```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);
}
```

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>x</td>
<td>15</td>
<td>y</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
Call by Value

```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);
}
```

<table>
<thead>
<tr>
<th>a</th>
<th>=20</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>=6</td>
</tr>
<tr>
<td>x</td>
<td>=15</td>
</tr>
<tr>
<td>y</td>
<td>=6</td>
</tr>
</tbody>
</table>
Call by Value

```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);
}
```

```
a  =20
b  =6

x  =15
y  =22
```
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);
}
```
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);
}
```
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);
}
```
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);
}
```

![Diagram showing the changes in variables]

```
a = 25
b = 32
x
y
```
Call by Name

- Used in many functional languages such as Haskell
- Rule
  - Goal: **lazy evaluation** of arguments
    - Premise 1: arguments are often complex expressions
    - Premise 2: arguments often go unused inside callee
    - Only evaluate arguments when they are used
      (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
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));
}
```

The output of the program is:
```
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));
}
```

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

- $b=2$
- $x=?$
- $y=?$
- $a=1$
- $b=1$
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));
}
```
Runtime Management of Classes and Objects

- 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) reference type
      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 jump to a (particular) function

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

**Dynamic Dispatch**: selects 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
- dispatch ptr
- attribute 1
- attribute 2
- ...
- attribute n

- Class tag is an integer
  - to identify this class for purposes of dynamic type checking
- Dispatch ptr is a pointer to the dispatch table
- Member variables are allocated in subsequent slots
Inheritance and Subclasses

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 reference type (class)
  - 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 reference type
  - 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
    ```
    TreeNode* createTREE() {
    ....
    p = (TreeNode*)malloc(sizeof(TreeNode));
    return p;
    }
    ```
  - Cannot reclaim memory 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
    Runtime code automatically reclaims memory when it determines 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, nondeterministic data corruption or crashes ...
    - Memory bugs are extremely hard to find and fix
  - Plenty of research on memory bug detection tools ...
    - AddressSanitizer: compiler pass to insert checks on memory accesses and malloc/frees (implemented on clang, gcc)
    - Valgrind: tool that inserts checks at runtime using dynamic binary recompilation techniques
  - But tools have limitations
    - Too much overhead to be used during production runs
    - All runtime detection tools have limited coverage
Why Automatic Memory Management?

- Leads to a more secure system
  - Secure languages run on virtual machines (VMs)
    (E.g. Java, JavaScript, PHP ...)
  - Virtual machine: a secure runtime environment sandboxed from underlying environment, implemented by runtime code
    - E.g. JavaScript app running on web browser is sandboxed from rest of device (can’t manipulate your camera)
    - Must guarantee security against even malicious code
  - Control of memory management essential for security
    - Otherwise, app can manipulate memory used by VM
      → Nullifying all security guarantees
    - Runtime must keep track of where each piece of data is (and what it is used for)
    - Impossible to do with manual memory management
Why Manual Memory Management?

- Manual memory management is typically more efficient
  - Programmers know when data is no longer needed

- Runtime code must somehow *detect* when data is no longer needed and recycle that memory
  - Time overhead: process of detection is often costly
  - Memory overhead:
    - Detection can be done every so often
      (Typically only when program runs out of memory)
    - Runtime may keep around data longer than necessary
    - Results in larger memory footprint

- Bad response time:
  - Program must be paused during detection phase
  - Program will be unresponsive during that time
Implementation: Automatic and Manual

- **Similarities between automatic and manual**
  - Runtime code maintains used/unused spaces in heap (e.g. linked together in the form of a list)
  - On allocation: memory of given size found in unused space and moved to used space
  - On free: given memory moved from used to unused space

- **Only in automatic memory management**
  - Compiler inserted calls to memory management routines
  - Memory management library
    - Routines to perform detection of unused memory
    - Routines to perform freeing of unused memory

- **We will focus on automatic memory management**
The Difficulty: Detection

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

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
  - A 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 is 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 garbage and are reclaimed
    3. Restarts the program
Scheme 1: Reference Counting

- Intuition: if no references point to an object, that object is unreachable
  - If an object is referenced by only unreachable objects, that object is also unreachable

- Implementation: each object has a counter for the number of references pointing to that object
  - Counter referred to as reference counter
  - Compiler inserts code to update reference counters, whenever program modifies references
Implementation Details

- Initialization \( x = \text{new Object}() \):
  \[
  x \leftarrow \text{new Object}(); \\
  \text{rc(new object)} \leftarrow 1;
  \]

- Reference assignment \( x = y \). Assume:
  \( x \) references object \( X \), and \( y \) references object \( Y \)
  \[
  x \leftarrow y; \\
  \text{rc}(Y) \leftarrow \text{rc}(Y) + 1; \quad \text{// rc}(Y): \text{Y's reference counter} \\
  \text{rc}(X) \leftarrow \text{rc}(X) - 1; \\
  \text{if (rc}(X)==0) \text{ then} \\
   \quad \text{mark } X \text{ as garbage to reclaim}; \\
   \quad \text{for each object } O \text{ referenced by } X \\
   \quad \quad \text{rc}(O) \leftarrow \text{rc}(O) - 1; \quad \text{// X's reference no longer counts} \\
   \quad \quad \text{if (rc}(O)==0) \text{ then} \\
   \quad \quad \quad \text{mark } O \text{ as garbage to reclaim}; \\
   \quad \quad \text{recursively repeat for } O;
  \]
Reference Counting Example

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

Heap:

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

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

Heap

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

int a
int *p
int *q
F1’s AR

main’s AR

free list

Heap

1 ...
2 ...
3 ...
0 ...

nil
Reference Counting Example

int a
int *p
int *q
F1’s AR
main’s AR

free list

Heap

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

RC cannot handle **circular data structures**
Problem of Reference Counting

RC cannot handle **circular data structures**

![Diagram showing reference counting and circular data structures](image-url)
Problem of Reference Counting

RC cannot handle **circular data structures**

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

Free list

Heap

...
Problem of Reference Counting

RC cannot handle **circular data structures**

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

```
free list
```

```
Heap
```

```
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
 ➢ Good response time
   - Garbage is collected whenever there is opportunity
   - No need to pause program for long time → responsive
     (Unless freeing a long chain of objects!)
Discussion of Reference Counting Scheme

- **Advantages:**
  - Relatively easy to implement
    - Compiler only needs to insert RC manipulation code at reference assignments
  - Good response time
    - Garbage is collected whenever there is opportunity
    - No need to pause program for long time → responsive
      (Unless freeing a long chain of objects!)

- **Disadvantages:**
  - Cannot collect circular data structures
    (Must rely on tracing GC for these corner cases)
  - Bad performance
    - Manipulating RCs at each assignment is high overhead
    - RC must be synchronized with multithreading → even slower
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 struct in heap
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 struct in heap
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 struct in heap
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 struct in heap
Discussion of Tracing Scheme

Advantages:

- Is guaranteed to collect even cyclic references
- Good performance
  - Overhead proportional to traced (live) objects
  - Garbage (dead) objects do not incur any overhead!
  - Most objects have short lifetimes
    → Dead by the time tracing GC runs

Disadvantages:

- Bad response time: Requires pausing program
- Prone to heap thrashing
  - Thrashing: frequent GCs to collect small amounts of garbage
  - If heap does not have extra 'headroom' beyond working set
    → GC becomes very frequent
    → Most objects are now live (bad performance)
Discussion of Tracing Scheme

Advantages:
- Is guaranteed to collect even cyclic references
- Good performance
  - Overhead proportional to traced (live) objects
  - Garbage (dead) objects do not incur any overhead!
  - Most objects have short lifetimes
    → Dead by the time tracing GC runs

Disadvantages:
- Bad response time: Requires pausing program
- Prone to heap thrashing
  - Thrashing: frequent GCs to collect small amounts of garbage
  - If heap does not have extra ‘headroom’ beyond working set
    → GC becomes very frequent
    → Most objects are now live (bad performance)
Flavors of Tracing Collection

- To move or not to move objects?
  - Garbage collection can leave ‘holes’ inside heap
  - Objects can be moved during GC to "compress" holes
  - **Mark-and-Sweep**: example of non-moving GC
  - **Semispace**: example of moving GC

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

- The two choices are orthogonal to each other
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 all reachable objects starting from root objects
  - Sweep phase: reclaims unreachable objects

Implementation

- Each object has an extra mark bit
  - Set mark bit to 0 on object allocation
- Mark phase:
  - Do a depth-first traversal starting from root objects
  - Set mark bit to 1 for all reachable objects while tracing
- Sweep phase:
  - Scan entire heap from beginning to end
  - Reclaim all objects with mark bit 0 (unreachable objects)
  - Reset mark bits to 0 for all remaining objects
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 ...
nil
0 ...
0 ...
0 ...
inl
nil
Mark-and-Sweep Example

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

free list

Heap

1 ... 1 ... 0 ...
1 ...nil
0 ...
nil
Mark-and-Sweep Example

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

Heap

1 ... 1 ...
1 ... 1 ...
1 ... nil
1 ... nil
nil
Mark-and-Sweep Example
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 space fragmented across heap
  - On allocation: must search through free list to find appropriate size
  - Search process can become costly with long lists
- Loss of Locality: objects allocated together spatially apart
  - Due to fragmentation, objects allocated temporally close together may be spatially far apart
  - Objects allocated together tend to be accessed together
  - Loss of locality can cause CPU cache misses, page faults
Discussion of Mark-and-Sweep Collection

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

قة Disadvantages:
usaha Fragmentation: free space fragmented across heap
usaha On allocation: must search through free list to find appropriate size
usaha Search process can become costly with long lists
usaha Loss of Locality: objects allocated together spatially apart
usaha Due to fragmentation, objects allocated temporally close together may be spatially far apart
usaha Objects allocated together tend to be accessed together
usaha Loss of locality can cause CPU cache misses, page faults
Flavor 2: Semispace

**Rules**

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

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

From space

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

To space
Flavor 2: Semispace

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

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

free list
```

```plaintext
From space

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

To space
```

From space

0 ...
```
Flavor 2: Semispace

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

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

```
free list
```

```
From space
1 ... 1 ... 0 ...
1 ... nil 0 ...
0 ...
```

```
To space
1 ... nil
1 ... nil
0 ...
0 ...
```
Flavor 2: Semispace

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

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

```
free list
```

```
From space

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

```
To space

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

From space

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

To space
Flavor 2: Semispace

Rules

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

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

```
From space

1 ...
1 ...
1 ...
hil
0 ...
0 ...
1 ...
```

```
To space

1 ...
1 ...
1 ...
hil
0 ...
0 ...
1 ...
```
Flavor 2: Semispace

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

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

```
free list
```

```
From space
```

```
To space
```

```
1 ... 1 ... nil
0 ... 0 ...
1 ... nil
1 ...
```
Flavor 2: Semispace

Rules

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

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

```
From space
1 ... 1 ... nil
0 ... 0 ...
```

```
To space
1 ... nil
1 ...
1 ...
```

```
Flavor 2: Semispace

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

- **Advantages:**
  - No fragmentation → fast allocation
  - Better locality → faster program execution
    - Objects are compacted on the heap (no holes)
    - Objects are laid out contiguously in the order of allocation
      (Objects allocated together land on same cache line/page)

- **Disadvantages:**
  - Only half of heap space usable → more frequent GC
  - Objects need to be copied → longer GC
Discussion of Semispace Collection

Advantages:
- No fragmentation → fast allocation
- Better locality → faster program execution
  - Objects are compacted on the heap (no holes)
  - Objects are laid out contiguously in the order of allocation
    (Objects allocated together land on same cache line/page)

Disadvantages:
- Only half of heap space usable → more frequent GC
- Objects need to be copied → longer GC
Flavors of Tracing Collection

- To move or not to move objects?
  - Garbage collection can leave ‘holes’ inside heap
  - Objects can be moved during GC to "compress" holes
  - **Mark-and-Sweep**: example of non-moving GC
  - **Semispace**: example of moving GC

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

The two choices are orthogonal to each other
Flavor 3: Incremental

Rules

- Divide heap and collect one partition at a time
- Needs **write barrier** to handle cross-partition pointers
  - Idea: conservatively add cross-partition pointers to root set
  - Write barrier: code to enforce above at every pointer update

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

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

```
From space#2        To space#2
0 ... 0 ... 0 ...
```
Flavor 3: Incremental

Rules

- Divide heap and collect one partition at a time
- Needs **write barrier** to handle cross-partition pointers
  - Idea: conservatively add cross-partition pointers to root set
  - Write barrier: code to enforce above at every pointer update

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

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

```
From space#1
From space#2
```

```
To space#1
To space#2
```

**writer barrier**
Flavor 3: Incremental

- Divide heap and collect one partition at a time
- Needs **write barrier** to handle cross-partition pointers
  - Idea: conservatively add cross-partition pointers to root set
  - Write barrier: code to enforce above at every pointer update
Flavor 3: Incremental

- Divide heap and collect one partition at a time
- Needs **write barrier** to handle cross-partition pointers
  - Idea: conservatively add cross-partition pointers to root set
  - Write barrier: code to enforce above at every pointer update

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

main's AR

free list
```

```
From space#1

1 ... nil
1 ...
1 ...
1 ...
1 ...

To space#1

From space#2

0 ...

To space#2

writer barrier
```
Discussion of Incremental Collection

Advantages:

- Good response time: due to smaller partitions

Disadvantages:

- Overhead of write barriers: must check for cross-partition point

Conservatism of write barriers

Cross-partition pointers could be from garbage objects

If cycles cross partitions → will never be collected

(Need periodic full heap GC to remove these)
Discussion of Incremental Collection

Advantages:
- Good response time: due to smaller partitions

Disadvantages:
- Overhead of write barriers: must check for cross-partition pointers on every pointer modification
- Conservatism of write barriers
  - Cross-partition pointers could be from garbage objects
  - If cycles cross partitions → will never be collected (Need periodic full heap GC to remove these)
Flavor 4: Generational Garbage Collection

- **Motivation:** Generational Hypothesis (empirical observation)
  - 90% of objects die very young (locals, temporaries)
  - 10% of objects live for a long time (non-local data)

- **Rules**
  - Divide heap into two partitions: young / old generations
  - Stages in an object’s life:
    1. Objects 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 generational collection is efficient
  - Young generation GC is frequent (many short-lived objects)
  - Old generation GC is infrequent (few long-lived objects)
  - Most GC happens in young where GC is efficient
    (In young GC most objects are dead, per hypothesis)

- Advantages:
  - Better response time: most GCs are young generation GCs

- Disadvantages:
  - Same problem with write barriers as incremental
  - Infrequently, old generation needs collection
    - With young generation to collect cross-partition cycles
    - Will lead to long pause times
    - Techniques exist to shorten: parallel GC, concurrent GC
Semispace and Generational Collectors in Graphic

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 the following 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)

- 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 (II)

- Mutator time: user program execution time
  - Semispace: best performance due to spatial locality
  - Mark-Sweep: worst performance due to less locality
  - Generational: good performance since young generation is Semispace (and most accesses happen there)
  - Generational also incurs overhead of write barriers
Comparison of Different GC algorithms (III)

- Normalized total time
  - Generational collectors perform best
    - Generational hypothesis: less collection
    - Even if hypothesis false, locality due to semispace nursery
  - Choice of old generation collector sometimes matters
    - GenMS: less collector overhead (no need to move objects)
    - GenCopy: less mutator overhead (due to better locality)