Run-time Environments

Dragon Book, Ch. 7
Run-time Environment

Key question: how does the interpreter or compiler implement the abstractions of the programming language, as the program is running?

– For example: in Project 3, you probably used a map from variable names to numeric values

Goal 1: overview of language abstractions
Goal 2: run-time organization of memory
Goal 3: mapping of program elements (variables, etc.) to this memory (“storage/memory allocation”)
Goal 1 (abstractions): Procedures

Terminology
– **Subroutine**: the general historical term
  • **Procedure**: subroutine that does not return a value
  • **Function**: subroutine that returns a value
  • **Method**: subroutine in some object-oriented languages
– More often, people use “procedure” as the general term
– **Procedural languages**: imperative languages in which procedures are a major abstraction mechanism (C, Fortran)
  • **Imperative**: based on statements that change the state (as opposed to **functional**, in which there is no state)

Reusable procedural abstraction: a collection of statements is abstracted by **name**, list of **formal parameters**, and (optionally) **return value**
Example: Old Context-Free Grammar

\<\text{program}\> ::= \<\text{funcDefList}\> \\
\<\text{funcDefList}\> ::= \<\text{funcDef}\> \ <\text{funcDefList}\> \ | \ \varepsilon \\
\<\text{funcDef}\> ::= \<\text{varDecl}\> ( \ <\text{formalDeclList}\> \ ) \ \{ \ <\text{stmtList}\> \ \} \\
\<\text{varDecl}\> ::= \text{int} \ \text{ident} \ | \ \text{float} \ \text{ident} \\
\<\text{stmt}\> ::= \ldots \ | \ \{ \ <\text{stmtList}\> \ \} \\
\<\text{expr}\> ::= \ldots \ | \ \text{ident} ( \ <\text{exprList}\> \ ) \ \text{function call} \\

Example:

\text{int } f \ (\text{int } x, \text{int } y) \ { \int z = x+y; \ \text{return } z;} \ \}

\text{int } g(\text{int } x) \ { \int z = 5; \ { \int t = x+z; \ \text{return } t;} \ \} \\
\text{int } \text{main} \ (\text{int } w) \ { \text{return } f(w+1,w+2) + g(8); \ \}
Basic Mechanism

1. A caller (another procedure) makes a call
   – The caller provides arguments (a.k.a. actual parameters) – expressions that are evaluated immediately before the call

2. Parameter passing: the actual parameters are “mapped” to the formal parameters
   – We will focus on “call by value”

3. Memory is allocated for the formal parameters and the local variables of the called procedure

4. The flow of control enters the procedure
   – And eventually returns back to the caller
Goal 1 (abstractions): Scopes

**Static question:** which declarations (of variables, procedures, ...) are visible in which code parts?

**Dynamic question:** what is the corresponding lifetime?

– Example: Fortran/C without blocks

<table>
<thead>
<tr>
<th>Main procedure</th>
<th>Procedure $S_1$</th>
<th>...</th>
<th>Procedure $S_n$</th>
</tr>
</thead>
</table>

Procedure names are visible everywhere
Local variables and formal parameters are visible only in the declaring procedure
Global variables are visible everywhere
Nested Blocks

What if procedure bodies have blocks?

Example:

```c
int f (int x, float y) {
    int z ...
    {
        float x ...
        f(z,x)
    }
    {
        int v, y, z ...
        f(z,y)
    }
}
```

remember Project 2 ...
Static Scope Rule

Algol, Pascal, Modula-2, C, C++, Java, ...

– Entities accessible in a scope are (1) entities declared in that scope, and (2) entities declared in the surrounding scopes, minus those with name conflicts

– A name declared in an inner scope “hides” a name declared in a surrounding scope

This is **static scoping**: just by looking at the code, we can determine the mapping from any name occurrence to the corresponding declaration

Some languages use **dynamic scoping**: the mapping dynamically changes [we will ignore this]
Compile time vs. Run time Abstractions

At **compile time**, we consider the scopes and their nesting

– Determines which entities (variables, etc.) are accessible in which parts of the code
  
  • Additional restrictions with so called “access modifiers” [private, protected, etc. – we will ignore]

At **run time**, each scope has a **lifetime**

– Anything declared in this scope has this lifetime – it becomes alive at the start of the scope, and “dies” at the end of the scope
Java Example: Scopes [ignore public/private/etc.]

```java
class Int {
    static Scanner scanner;
    static void main(String[] args) {
        InputStream is = System.in;
        Int.scanner = new Scanner(is);
        int result = Parser.parse();
        PrintStream ps = System.out;
        ps.println(result); } }

class Parser { // adds up a list of ints
    static int parse() {
        Scanner s = Int.scanner;
        int result = 0;
        if (s.hasNextInt()) {
            int num = s.nextInt();
            int rest = Parser.parse();
            result = num+rest;
        }
        return result; } }
```

Global scope:
- Int
- Int.scanner
- Int.main
- Parser
- Parser.parse

+ from Java libraries:
- Scanner
- System
- System.in
- InputStream
- Scanner.Scanner
- constructor
- System.out
- PrintStream
- PrintStream.println
- Scanner.hasNextInt
- Scanner.nextInt

Local scope for Int.main:
- args
- is
- result
- ps

Local scope for Parser.parse:
- s
- result

Scope for block:
- num
- rest
Lifetimes for fields, formals, and locals

class Int {
    static Scanner scanner;
    static void main(String[] args) {
        InputStream is = System.in;
        Int.scanner = new Scanner(is);
        int result = Parser.parse();
        PrintStream ps = System.out;
        ps.println(result); }
}

class Parser { // adds up a list of ints
    static int parse() {
        Scanner s = Int.scanner;
        int result = 0;
        if (s.hasNextInt()) {
            int num = s.nextInt();
            int rest = Parser.parse();
            result = num+rest;
        }
        return result; } }
Goal 2: Run-time Organization of Memory

Memory regions

- **Code segment**: code for all procedures
- **Global (static) segment**: global variables, static fields
- **Heap**: dynamically-allocated memory (e.g. “new” in Java/C++/C#, malloc() in C)
- **Run-time call stack**: local variables of procedures, including from nested blocks:
  - Needed because of recursion
  - Otherwise, we can just put the locals in the static segment (earlier version of Fortran do this)
  - All locals from nested blocks are allocated at the beginning of the lifetime of the procedure

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Free memory</td>
</tr>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>
class Int {
  static Scanner scanner;
  static void main(String[] args) {
    InputStream is = System.in;
    Int.scanner = new Scanner(is);
    int result = Parser.parse();
    PrintStream ps = System.out;
    ps.println(result); } }

class Parser { // adds up a list of ints
  static int parse() {
    Scanner s = Int.scanner;
    int result = 0;
    if (s.hasNextInt()) {
      int num = s.nextInt();
      int rest = Parser.parse();
      result = num+rest;
    }
    return result; } }
Goal 3: Storage Allocation

class Int {
    static Scanner scanner;
    static void main(String[] args) {
        InputStream is = System.in;
        Int.scanner = new Scanner(is);
        int result = Parser.parse();
        PrintStream ps = System.out;
        ps.println(result); } }

class Parser { // adds up a list of ints
    static int parse() {
        Scanner s = Int.scanner;
        int result = 0;
        if (s.hasNextInt()) {
            int num = s.nextInt();
            int rest = Parser.parse();
            result = num+rest;
        }
        return result; } }

---

### Code

#### Static
- Int.scanner
- System.in
- System.out

#### Heap
- Strings for command-line arguments + String array to hold them

#### Free memory

#### Stack
- args, is, result, ps

---

activation record for Int.main
class Int {
    static Scanner scanner;
    static void main(String[] args) {
        InputStream is = System.in;
        Int.scanner = new Scanner(is);
        int result = Parser.parse();
        PrintStream ps = System.out;
        ps.println(result); }
}

class Parser { // adds up a list of ints
    static int parse() {
        Scanner s = Int.scanner;
        int result = 0;
        if (s.hasNextInt()) {
            int num = s.nextInt();
            int rest = Parser.parse();
            result = num+rest;
        }
        return result; }
    }
class Int {
  static Scanner scanner;
  static void main(String[] args) {
    InputStream is = System.in;
    Int.scanner = new Scanner(is);
    int result = Parser.parse();
    PrintStream ps = System.out;
    ps.println(result); } }

class Parser { // adds up a list of ints
  static int parse() {
    Scanner s = Int.scanner;
    int result = 0;
    if (s.hasNextInt()) {
      int num = s.nextInt();
      int rest = Parser.parse();
      result = num+rest;
    }
    return result; } }
class Int {
    static Scanner scanner;
    static void main(String[] args) {
        InputStream is = System.in;
        Int.scanner = new Scanner(is);
        int result = Parser.parse();
        PrintStream ps = System.out;
        ps.println(result); } }

class Parser { // adds up a list of ints
    static int parse() {
        Scanner s = Int.scanner;
        int result = 0;
        if (s.hasNextInt()) {
            int num = s.nextInt();
            int rest = Parser.parse();
            result = num+rest;
        }
        return result; } }
Call Stack: Sample Implementation

When a procedure P begins execution:

- An activation record for that incarnation of P is created on the stack (has space for local variables)
- During this incarnation of P, the activation record pointer (AP) register will contain the (starting) address of this activation record
- The stack pointer (SP) register will contain the address of the location immediately beyond this a.r.

When this incarnation of P finishes, control returns to the caller, SP is set to the current AP, and AP set to the address of the activation record of the caller
Call Stack: Sample Implementation

Activation record for P

- Space for locals & formals of P
- Caller’s AP value
- Return Address

Activation record for P’s caller

- Space for locals & formals of P’s caller
- SP
- AP

Code segment

- Code for P’s caller
- Instruction: call P
- ...
Call Stack: Formals and Return Value

<table>
<thead>
<tr>
<th>activation record for P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local\textsubscript{n}</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Local\textsubscript{1}</td>
</tr>
<tr>
<td>Return value of P</td>
</tr>
<tr>
<td>Formal parameter\textsubscript{n}</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Formal parameter\textsubscript{1}</td>
</tr>
<tr>
<td>Caller’s AP value</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
</tbody>
</table>

- The formal parameters and the return values are at offsets (w.r.t. AP) that are known at compile time.
- The caller of P can access them using its value of SP (the top of the stack), before and after the call.
Parameter Passing Modes

**Call-by-value**: C, Pascal, C++, Java, ...

– The formal parameter is essentially a local variable initialized with the corresponding argument

```c
void swap(int x, int y) // does not work
{
    int z; z = x; x = y; y = z;
}
```

**Call-by-reference**: C++, Pascal, ...

– The parameter is not a new variable, but a new reference to the corresponding argument

– For large objects, could be more efficient than call-by-value (no need to copy large amounts of memory)
Lifetimes and Storage Allocation

**Static allocation**: address determined once and retained throughout the execution of the program

– Global variables in C, Pascal, etc.
– `static` fields in C++, Java, etc.
– Local variables in languages without recursion
  • E.g., earlier versions of Fortran
– `static` local variables in C
– Large constants – e.g., string/array constants
Lifetimes and Storage Allocation

**Stack-based allocation**: address determined when the call happens; lifetime ends when the call ends

- Push the activation record on the run-time call stack
  - Sometimes the activation record is called a **stack frame**
- Local variables in languages with recursion
- Relative address within the stack frame is determined at compile time

**Heap-based allocation**: space allocated and deallocated manually by the programmer

- C: `A* a = (A*)malloc(sizeof(A)); ... free (a);`
- C++: `A* a = new A(); ... delete a;`
- Java: `A a = new A();` dealloc with garbage collection
Garbage Collection

– Slides based on course materials by Prof. Kathryn McKinley and Prof. Mike Bond

– Explicit (manual) memory management e.g. C/C++
  – More code to maintain
  – Correctness
    • Free an object too soon - crash
    • Free an object too late - waste space
    • Never free - at best waste, at worst fail
  – Efficiency can be very high
  – Gives programmers more control over the run-time behavior of the program
Garbage Collection

Automatic management through garbage collection

– Reduces programmer burden: less user code compared to manual memory management

– Eliminates sources of errors
  • Less user code to get correct
  • Protects against some classes of memory errors: no free(), thus no premature free(), no double free(), or forgetting to free()

– Not perfect, memory can still leak
  • Programmers still need to eliminate all pointers to objects the program no longer needs

– Integral to modern languages
  • Java, C#, PHP, JavaScript
Key Issues

For both manual/automatic\[we will not discuss these]\:
- Fast allocation
- Fast reclamation
- Low fragmentation (wasted space)
- How to organize the memory space

Garbage collection
- Distinguish live objects from garbage
  - \textbf{Live} object will be used in the future
  - Prove that object is \textit{not live (i.e., dead)}, and deallocate it
  - Deallocation as soon as possible after last use
What is Garbage?

In theory, any object the program will never reference again
  – But compiler & runtime system cannot figure that out
In practice, any object the program cannot reach is garbage
  – Approximate liveness with reachability

Modern languages couple GC with “safe” pointers
  – Programs may not access arbitrary addresses in memory (e.g., Java/C# vs. C/C++)
  – The compiler can identify and provide to the garbage collector all the pointers, thus enforcing “Once garbage, always garbage”
  – Runtime system can move objects by updating pointers
Reachability

- The runtime memory management system examines all global variables, stack variables, and live registers that could refer to objects on the heap (i.e., the **roots** of reachability)
- We can **trace** these pointers through the heap (following object fields that themselves point to heap objects) to find all reachable objects
Reachability

Tracing collector
– Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
Reachability

Tracing collector

– Marks the objects reachable from the roots as **live objects**, and then performs a reachability computation from them

![Diagram](image.png)
Reachability

Tracing collector
- Marks the objects reachable from the roots as **live objects**, and then performs a reachability computation from them.
Reachability

Tracing collector

- Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
- Unmarked objects are **dead**
Reachability

Tracing collector
- Marks the objects reachable from the roots as **live objects**, and then performs a **reachability** computation from them
- Unmarked objects are **dead**
Mark-and-Sweep Implementation

Free-lists organized by size
– blocks of same size, or
– individual objects of same size
Most objects are small < 128 bytes
Mark-and-Sweep Implementation

Allocation
  – Grab a free object off the free list
Mark-and-Sweep Implementation

Allocation

– Grab a free object off the free list
Mark-and-Sweep Implementation

Allocation
– Grab a free object off the free list

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

free lists

heap
Mark-and-Sweep Implementation

Allocation

– If there is no more memory of the right size, a garbage collection is triggered
– Mark phase - find the live objects
– Sweep phase - put free ones on the free list

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>free lists</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

heap
Mark-and-Sweep Implementation

Mark phase
– Reachability computation on the heap, marking all live objects

Sweep phase
– Sweep the memory for free objects, and populate the free lists

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
</table>

free lists
Mark-and-Sweep Implementation

Mark phase
- Reachability computation on the heap, marking all live objects

Sweep phase
- Sweep the memory for free objects, and populate the free lists

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
</table>

free lists

heap
Mark-and-Sweep Implementation

Mark phase
  – Reachability computation on the heap, marking all live objects

Sweep phase
  – Sweep the memory for free objects, and populate the free lists
Mark-and-Sweep Implementation

Mark phase
- Reachability computation on the heap, marking all live objects

Sweep phase
- Sweep the memory for free objects, and populate the free lists

<table>
<thead>
<tr>
<th>4</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

free lists