Expression Evaluation, Runtime Environments

Expression evaluation

- Order of evaluation
- For the abstract syntax tree

\[ (x + 3) + (2 + 4) + 5 \]

Expression evaluation (Contd.)

- One possible semantics:
  - evaluate AST bottom-up, left-to-right.

- Problem:
  - constrains optimizations based on mathematical properties, e.g., commutativity and associativity
  - Consider \((x+0)+(y+3)+(z+4)\)
  - Using associativity and commutativity, the compiler can simplify this to \(x+y+z+7\) (3 additions at runtime)
  - A strict left-to-right evaluation would require 5 addition operations at runtime.

Expression evaluation (Contd.)

- Some languages leave order of eval unspecified.

- Problem:
  - Semantics of expressions with side-effects, e.g., \((x++) + x\)
  - If initial value of \(x\) is 5, left-to-right evaluation yields 11 as answer, but right-to-left evaluation yields 10

- So, languages that allow expressions with side-effects are forced to specify order of evaluation

- Still, it is bad programming practice to use expressions where different orders of evaluation can lead to different results
  - Impacts readability (and maintainability) of programs
Evaluation of Boolean Expressions

- Left-to-right evaluation with short-circuit semantics is appropriate for boolean expressions.
  * In `e1 && e2`, `e2` is evaluated only if `e1` evaluates to true.
  * In `e1 || e2`, `e2` is evaluated only if `e1` evaluates to false.
- This semantics is convenient in programming:
  * Consider statement: `if ((i<n) && a[i]!=0)`
  * With short-circuiting, `a[i]` never accessed if `i>= n`
  * Another example: `if((p!=NULL) && p->value>0)`

Switch Statement

```
switch (<expr>) {
    case <value> :
    case <value> :
    ... 
    default : 
}
```

- Evaluate `<expr>` to get value `v`.
- Evaluate the case that corresponds to `v`.
- Restrictions:
  * `<value>` has to be constant, of an ordinal type e.g., int
  * Why?

Implementation of Switch statement

- Naive algorithm:
  Sequential comparison of value `v` with case labels.
  This is simple, but inefficient. It involves O(N) comparisons.
  ```
  switch (e) {
    case 0 : s0 ;
    case 1 : s1 ;
    case 2 : s2 ;
    default: s3 ; }
  ```
  can be translated as
  ```
  v = e ;
  if (v == 0) s0 ;
  else if (v == 1) s1 ;
  else if (v == 2) s2 ;
  else s3 ;
  ```
- Binary search:
  O(log N) comparisons, a drastic improvement over sequential search for large N.
- Using this, the above case statement can be translated as
  ```
  v = e ;
  if (v <= 1) {
    if(v == 0) s0 ;
    else if (v == 1) s1 ;
  }
  else if (v == 2) s2 ;
  else s3 ;
  ```
Implementation of switch statement (Contd.)

- Another technique is to use hash tables.
- This maps the value \( v \) to the case label that corresponds to the value \( v \).
- This takes constant time (average case).

Loops

- **while:**
  - Consider the statement: \( \text{while } C \text{ do } S \)
  - Its semantics is equivalent to: \( \text{if } C \text{ then } \{S; s1\} \)
- **repeat:**
  - Consider: \( \text{repeat } S \text{ until } C \)
  - Its semantics is equivalent to \( S; \text{ if } (!C) \text{ then } S2 \)
- **for:**
  - Semantics of "\( \text{for } (S2; C; S3) \text{ S} \)" is the same as that of "\( S2; \text{ while } C \text{ do } \{S; S3\} \)"

Control Statements (contd.)

- Procedure calls:
  - Communication between the calling and the called procedures takes place via parameters.
- Semantics:
  - substitute formal parameters with actual parameters
  - rename local variables so that they are unique in the program
  - replace procedure call with the body of called procedure

Parameter-passing semantics

- Call-by-value
- Call-by-reference
- Call-by-value-result
- Call-by-need
  - Differences with macros
**Call-by-value**

- Evaluate the actual parameters
- Assign them to corresponding formal parameters
- Execute the body of the procedure.
- We need to ensure that the names of local variables and formal parameters of callee do not clash with the variable names visible in the caller
  - If they do, the variables in the callee should be renamed to avoid any clash.

**Call-By-Value (Contd.)**

- Example:
  ```
  int z;
  void p(int x) {
    z = 2*x;
    main() {
      int y;
      ==> int x = y;
      z = 2*x;
      main() {
        int y;
        ==> z = 2*x;
        p(y);
      }
    }
  }
  ```

**Call-By-Value-Result**

- In addition to the steps in CBV, add:
  - assignment statements to copy values of formal parameters to actuals at the end of callee code

  ```
  int z;
  void p(int x) {
    z = 2*x;
    ==> z = 2*x;
  }
  main() {
    int y;
    ==> int y;
    p(y);
  }
  ```

**Call-By-Reference**

- Works like CBV, with one important difference:
  - l-values (rather than r-values) are passed in.

  ```
  int z;
  void p(int x) {
    z = 2*x;
    ==> int x = y;
  }
  main() {
    int y;
    ==> z = 2*x;
    int y;
    p(y);
  }
  ```

- Call-by-reference supported in C++ but not C
  - Effect realized in C by explicitly passing l-values of parameters using the "&" operator
Call-by-reference (contd.)

- Explicit simulation in C provides a clearer understanding of the semantics of call-by-reference:

```c
int p(int *x) {
    *x = *x + 1;
    return *x;
}
...  
int z;
y = p(&z);
```

CBVR Vs CBR

- Consider
  ```c
  void p(int x, int y) {
      x = x+1; y = y+1;
  }
  ...
  int a = 3; p(a, a);
  ```
- With CBVR, `a` will have the value 4
- With CBR, `a` will have the value 5

Call-by-Name

- Instead of assigning l-values or r-values, CBN works by substituting actual parameter expressions in place of formal parameters in the body of callee

```c
int z;
void p(int x,y) {          main() {
    z = x+x+y;                 int z1;
}                 ==>         }
int y=0;                          z = (y++)+(y++)
main() {                         +(y--);     p(y++, y--);
    z = (y++)+(y++)             + (y--); }
}                          
```

Macros

- Macros work like CBN, with one important difference:
  - no renaming of “local” variables
- This means that possible name clashes between actual parameters and variables in the body of the macro will lead to unexpected results.
Macros (Contd.)

- given

```c
#define sixtimes(y) {int z=0; z = 2*y; y = 3*z;}
main() {
    int x = 5, z = 3;
    sixtimes(z);
}
```

After macro substitution, we get the program:

```c
main() {
    int x = 5, z = 3;
    {
        int z=0;
        z = 2*z;  
        y = 3*z;
    }
}
```

Macros (Contd.)

- It is different from what we would have got with CBN parameter passing.
- In particular, the name confusion between the local variable z and the actual parameter z would have been avoided, leading to the following result:

```c
main() {
    int x = 5, z = 3;
    {
        int z1=0;
        z1 = 2*z;
        z = 3*z1;
    }
}
```

Difficulties in Using the Parameter Passing Mechanisms

- CBV: Easiest to understand, no difficulties or unexpected results.
- CBVR:
  - When the same parameter is passed in twice, the end result can differ depending on the order.

```c
void p(int x, int y) {
    x = x+1; y = y+2;
}
...  
int a = 3; p (a, a); // a=4 or a=5?
```
  - Otherwise, relatively easy to understand.

- Otherwise, relatively easy to understand.

Difficulties in Using CBR

- Aliasing can create problems.

```c
int arev(int a[], int b[], int size) {
    for (int i = 0; i < size; i++)
        a[i] = b[size-i-1];
}
```
- The above procedure will normally copy b into a, while reversing the order of elements in b.
- However, if a and b are the same, as in an invocation arev(c,c,4), the result is quite different.
- If c is {1,2,3,4} at the point of call, then its value on exit from arev will be {4,3,3,4}.  

Difficulties in Using CBN

- CBN is probably the most complicated of the parameter passing mechanisms, and can be quite confusing in several situations.
- If the actual parameter is an expression with side-effects:
  ```c
  void f(int x) {
    int y = x;
    int z = x;
  }
  main() {
    int y = 0;
    f(y++);
  }
  ```
  Note that after a call to f, y's value will be 2 rather than 1.

Difficulties in Using CBN (Contd.)

- If the same variable is used in multiple parameters:
  ```c
  void swap(int x, int y) {
    int tp = x;
    x = y;
    y = tp;
  }
  main() {
    int a[] = {1, 1, 0};
    int i = 2;
    swap(i, a[i]);
  }
  ```
  When using CBN, by replacing the call to swap by the body of swap: i will be 0, and a will be {0, 1, 0}.

Difficulties in Using Macro

- Macros share all of the problems associated with CBN.
- In addition, macro substitution does not perform renaming of local variables, leading to additional problems.

Components of Runtime Environment (RTE)

- **Static area** allocated at load/startup time.
  - Examples: global/static variables
  - Variables mapped to absolute addresses at compile time
- **Stack area** for execution-time data that obeys a last-in first-out lifetime rule.
  - Examples: local variables, parameters, temporary vars
- **Heap area** for "fully dynamic" data, i.e. data that doesn't obey LIFO rule.
  - Examples: objects in Java, lists in Scheme.
Languages and Environments

- Languages differ on where activation records must go in the environment:
  - (Old) Fortran is static: all data, including activation records, are statically allocated.
    - Each function has only one activation record—no recursion!
  - Functional languages (Scheme, ML) and some OO languages (Smalltalk) are heap-oriented:
    - Almost all data, including AR, allocated dynamically.
  - Most languages are in between: data can go anywhere (depending on its properties)
    - ARs go on the stack.

Stack Allocation

- An Activation Record (AR) is created for each invocation of a procedure
- Structure of AR:

<table>
<thead>
<tr>
<th>Base Pointer</th>
<th>Direction of stack growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual parameters</td>
<td>Return value</td>
</tr>
<tr>
<td>Return address</td>
<td></td>
</tr>
<tr>
<td>Saved BP (control link)</td>
<td></td>
</tr>
<tr>
<td>Local variables</td>
<td></td>
</tr>
<tr>
<td>Temporary variables</td>
<td></td>
</tr>
</tbody>
</table>

Simple stack-based allocation

- Local variables are allocated at a fixed offset on the stack
  - Accessed using this constant offset from BP
    - Example: to load a local variable at offset 8 into the EBX register (x86 architecture)
      \[
      \text{mov 0x8(%ebp), %ebx}
      \]
  - Example:
    \[
    \begin{align*}
    &\{ \text{int } x; \text{ int } y; \\
    &\quad \{ \text{ int } z; \} \\
    &\quad \{ \text{ int } w; \}
    &\}
    \end{align*}
    \]

Steps involved in a procedure call

- Caller
  - Save registers
  - Evaluate actual parameters, push on the stack
    - Push l-values for CBR, r-values in the case of CBV
  - Allocate space for return value on stack
  - Save return address
  - Jump to the beginning of called function
- Callee
  - Save BP (control link field in AR)
  - Move SP to BP
  - Allocate storage for locals and temporaries (Decrement SP)
  - Local variables accessed as [FP+k], parameters using [FP-I]
Steps in return

- **Callee**
  - Copy return value into its location on AR
  - Increment SP to deallocate locals/temporaries
  - Restore BP from Control link
  - Jump to return address on stack

- **Caller**
  - Copy return values
  - Pop parameters from stack
  - Restore saved registers

Example (C):

```c
int x;
void p( int y)
{ int i = x;
  char c; ... }
void q ( int a)
{ int x;
  p(1);
}
main()
{ q(2);
  return 0;
}
```

Non-local variable access

- Requires that the environment be able to identify frames representing enclosing scopes.
- Using the control link results in dynamic scope (and also kills the fixed-offset property).
- If procedures can't be nested (C), the enclosing scope is always locatable:
  - it is global/static (accessed directly)
- If procedures can be nested (Ada, Pascal), to maintain lexical scope a new link must be added to each frame:
  - access link, pointing to the activation of the defining environment of each procedure.

Implementation Aspects of OO-Languages

- Allocation of space for data members: The space for data members is laid out the same way it is done for structures in C or other languages. Specifically:
  - The data members are allocated next to each other.
  - Some padding may be required in between fields, if the underlying machine architecture requires primitive types to be aligned at certain addresses.
  - At runtime, there is no need to look up the name of a field and identify the corresponding offset into a structure; instead, we can statically translate field names into relative addresses, with respect to the beginning of the object.
  - Data members for a derived class immediately follow the data members of the base class.
  - Multiple inheritance requires more complicated handling, we will not discuss it here.
Implementation Aspects of OO-Languages

class B {
    int i; double d;
    char c; float f;
}

// Integer requires 4 bytes
// pad,
// Double requires 8 bytes
// char needs 1 byte, 3 are padded
// float to be aligned on 4-byte
// require 4-bytes of space

int i
XXXXXXXXXXXX
double d
char c|XXXXX
float f

Implementation Aspects of OO-Languages

class C {
    0
    int k, l; B b;
}

int k
int l
XXXXXXXXXXXX
double d
char c|XXXXX
float f

Implementation Aspects of OO-Languages

class D: public C {
    0
    double x;
}

int k
int l
int i
XXXXXXXXXXXX
double d
char c|XXXXX
float f
double x

Implementation of Virtual Functions

- Approach 1:
  - Lookup type info at runtime, and then call the function defined by that type.
  - Problem: very expensive, require type info to be maintained at runtime.
Implementation of Virtual Functions (Contd.)

- Approach 2:
  - Treat function members like data members:
    - Allocate storage for them within the object.
    - Put a pointer to the function in this location, and translate calls to the function to make an indirection through this field.
  - Benefit:
    - No need to maintain type info at runtime.
    - Implementation of virtual methods is fast.
  - Problem:
    - Potentially lot of space is wasted for each object.
    - Even though all objects of the same class have identical values for the table.

Benefit:
- No need to maintain type info at runtime.
- Implementation of virtual methods is fast.

Problem:
- Potentially lot of space is wasted for each object.
- Even though all objects of the same class have identical values for the table.

Implementation of Virtual Functions (Contd.)

Approach 3:
- Introduce additional indirection into approach 2.
- Store a pointer to a table in the object, and this table holds the actual pointers to virtual functions.
- Now we use only one word of storage in each object.

Impact of subtype principle on Implementation

- The subtype principle requires that any piece of code that operates on an object of type B can work "as is" when given an object belonging to a subclass of B.
- This implies that runtime representation used for objects of a subtype A must be compatible with those for objects of the base type B.
- Note that the way the fields of an object are accessed at runtime is using an offset from the start address for the object.
- For instance, b1.i will be accessed using an expression of the form *(&b1+0), where 0 is the offset corresponding to the field i.
- Similarly, the field b1.c will be accessed using the expression *(&b1+1)
Impact of subtype principle on Implementation (Contd.)

• an invocation of the virtual member function b1.h() will be implemented at runtime using an instruction of the form:
  call *(&b1+2)+1
  • &b1+2 gives the location where the VMT ptr is located
  • *(&b1+2) gives the value of the VMT ptr, which corresponds to the location of the VMT table
  • *(&b1+2)+1 yields the location within the VMT table where the pointer to virtual function h is stored.

Impact of subtype principle on Implementation (Contd.)

• The subtype principle imposes the following constraint:
  • Any field of an object of type B must be stored at the same offset from the base of any object that belongs to a subtype of B.
  • The VMT ptr must be present at the same offset from the base of any object of type B or one of its subclasses.
  • The location of virtual function pointers within the VMT should remain the same for all virtual functions of B across all subclasses of B.

Impact of subtype principle on Implementation (Contd.)

• We must use the following layout for an object of type A defined as follows:
  class A: public B {
    float f;
    void h(); // reuses implementation of G from B;
    virtual void k();
  }
  A a;

Impact of subtype principle on Implementation (Contd.)

• In order to satisfy the constraint that VMT ptr appear at the same position in objects of type A and B, it is necessary for the data field f in A to appear after the VMT field.
• A couple of other points:
  • a) non-virtual functions are statically dispatched, so they do not appear in the VMT table
  • b) when a virtual function f is NOT redefined in a subclass, the VMT table for that class is initialized with an entry to the function f defined its superclass.