CSE 504: Compilers
Evaluation and Runtime Environments

R. Sekar
Expression evaluation

- Order of evaluation
- For the abstract syntax tree

the equivalent expression is \((x + 3) + (2 + 4) + 5\)
Expression evaluation (Continued)

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

- This constrains optimization that uses mathematical properties of operators
  (e.g. commutativity and associativity)
  
  - e.g., it may be preferable to evaluate of $e_1 + (e_2 + e_3)$ instead of $(e_1 + e_2) + e_3$
  - $(x+0)+(y+3)+(z+4) = x+y+z+0+3+4 = x+y+z+7$

  - The compiler can evaluate $0+3+4$ at compile time, so that at runtime, we have two fewer addition operations.
Some languages leave order of evaluation unspecified.
- order of evaluation of procedure parameters are also 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 with expressions with side-effects forced to specify evaluation order

Still, a bad programming practice to use expressions where different orders of evaluation can lead to different results
- Impacts readability (and maintainability) of programs
Left-to-right evaluation

Left-to-right evaluation with short-circuit semantics is appropriate for boolean expressions.

- \( e_1 \&\& e_2 \): \( e_2 \) is evaluated only if \( e_1 \) evaluates to true.
- \( e_1 \| e_2 \): \( e_2 \) is evaluated only if \( e_1 \) evaluates to false.

This semantics is convenient in programming:

- Consider the statement: \( \text{if}((i<n) \&\& a[i]! = 0) \)
- With short-circuit evaluation, \( a[i] \) is never accessed if \( i \geq n \)
- Another example: \( \text{if} ((p!=\text{NULL}) \&\& p->value>0) \)
Disadvantage:
- In an expression like "if((a==b)||(c=d))"
- The second expression has a statement. The value of c may or may not be the value of d, depending on if a == b is true or not.

Bottom-up:
- No order specified among unrelated subexpressions.
- Short-circuit evaluation of boolean expressions.

Delayed evaluation
- Delay evaluation of an expressions until its value is absolutely needed.
- Generalization of short-circuit evaluation.
Control Statements

- Structured Control Statements:

- Case Statements:
  - Implementation using if-then-else
  - Understand semantics in terms of the semantics of simple constructs
  - actual implementation in a compiler

- Loops
  - while, repeat, for
If-Then-Else

- If-then-else. It is in two forms:
  - if cond then s1 else s2
  - if cond then s1

- evaluate condition: if and only if evaluates to true, then evaluate s1 otherwise evaluate s2.
Case (Switch) Statement

- Case statement

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

- Evaluate "<expr>" to get value v. Evaluate the case that corresponds to v.

- Restriction:
  - "<value>" has to be a constant of an original type e.g., int, enum
  - Why?
Implementation of case statement

- **Naive algorithm:**
  - Sequential comparison of value \( v \) with case labels.
  - This is simple, but inefficient. It involves \( O(N) \) comparisons.

```java
switch(e) {
    case 0: s0;
    case 1: s1;
    case 2: s2;
    case 3: s3;
}
```

can be translated as:

```java
v = e;
if (v == 0) s0;
else if (v == 1) s1;
else if (v == 2) s2;
else if (v == 3) s3;
```
Implementation of case statement (Continued)

- 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

```java
v = e;
if (v<=1)
    if (v==0) s0;
    else if (v==1) s1;
else if (v>=2)
    if (v==2) s2;
    else if (v==3) s3;
```
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 (expected).
Control Statements (contd.)

- **while:**
  
  \[
  \text{let } s_1 = \text{while } C \text{ do } S \\
  \text{then it can also be written as} \\
  s_1 = \text{if } C \text{ then } \{S; s_1\}
  \]

- **repeat:**
  
  \[
  \text{let } s_2 = \text{repeat } S \text{ until } C \\
  \text{then it can also be written as} \\
  s_2 = S; \text{ if } (!C) \text{ then } s_2
  \]

- **loop**
  
  \[
  \text{let } s = \text{loop } S \text{ end} \\
  \text{its semantics can be understood as } S; s
  \]
  
  S should contain a break statement, or else it won’t terminate.
Semantics of for \((S2; C; S3)\) can be specified in terms of while:

- \(S2; \text{while } C \text{ do } \{ S; S3 \}\)

In some languages, additional restrictions imposed to enable more efficient code:

- Value of index variable can’t change loop body, and is undefined outside the loop
- Bounds may be evaluated only once
Unstructured Control Flow

Unstructured control transfer statements (goto) can make programs hard to understand:

40: if (x > y) then goto 10
   if (x < y) then goto 20
   goto 30
10: x = x - y
   goto 40
20: y = y - x
   goto 40
30: gcd = x
Unstructured Control Flow (Continued)

- Unstructured control transfer statements (goto) can make programs hard to understand:

  40: if (x > y) then goto 10
     if (x < y) then goto 20
     goto 30
  10: x = x - y
     goto 40
  20: y = y - x
     goto 40
  30: gcd = x

- Equivalent program with structured control statements is easier to understand:

  while (x!=y) {
    if (x > y) then x=x-y
    else y=y-x
  }
goto should be used in rare circumstances
  e.g., error handling.

Java doesn’t have goto. It uses labeled break instead:

```java
l1: for ( ... ) {
    while ( ... ) {
        ....
        break l1.
    }
}
```

break l1 causes exit from loop labeled with l1
Restrictions in use of goto:

- jumps across procedures
- jumps from outer blocks to inner blocks or unrelated blocks

```c
goto l1;
if (...) then {
    int x = 5;
    x = 5;
    l1: y = x*x;
}
```

Jumps from inner to outer blocks are permitted.
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
  - In an actual implementation, we will simply look up the local variables in a different environment (callee’s environment)
- Renaming captures this semantics without having to model environments.
- replace procedure call with the body of called procedure
Parameter-passing semantics

- Call-by-value
- Call-by-reference
- Call-by-value-result
- Call-by-name
- Call-by-need
- Macros
Call-by-value

- Evaluate the actual parameters
- Assign them to corresponding formal parameters
- Execute the body of the procedure.

```c
int p(int x) {
    x = x + 1;
    return x;
}
```

An expression \( y = p(5+3) \) is executed as follows:
- evaluate \( 5+3 = 8 \), call \( p \) with \( 8 \), assign \( 8 \) to \( x \), increment \( x \), return \( x \) which is assigned to \( y \).
Preprocessing
- create a block whose body is that of the procedure being called
- introduce declarations for each formal parameter, and initialize them with the values of the actual parameters

Inline procedure body
- Substitute the block in the place of procedure invocation statement.
If-Then-Else Parameter Passing Mechanisms

Call-by-value (Continued)

Example:

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

Replacing the invocation `p(y)` as described yields:

```c
int z;
main(){
    int y;
    {
        int x1=y;
        z = 2*x1;
    }
}
```
"Name Capture"

- Same names may denote different entities in the called and calling procedures.
- To avoid name clashes, need to rename local variables of called procedure.
  - Otherwise, local variables in called procedure may be confused with local variables of calling procedure or global variables.
Call-by-value (Continued)

**Example:**

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

**After replacement:**

```c
int z;
main(){
    int y;
    {
        int x1=y;
        int y1=2;
        z = y1*x1;
    }
}
```
Call-by-reference

- Evaluate actual parameters (must have l-values)
- Assign these l-values to formal parameters
- Execute the body.
  
  ```
  int z = 8;
  y = p(z);
  ```
- After the call, y and z will both have value 9.
- Call-by-reference supported in C++, but not in C
  
  Effect realized by explicitly passing l-values of parameters using “&” operator
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);
```

```
g(List z) { 
    int x;
    int y;
    x = 5;
    y = 3;
    x = y;  
    x = 2;  
    y is still 3  
}
g(y) 
List x = new ...
List y = new ...
  x = y;
x, f(...)  
  y also changes
```
If-Then-Else Parameter Passing Mechanisms

Call-By-Reference (Continued)

Example:

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

After replacement:

```c
int z;
main(){
    int y;
    {
        int& x1=y;
        int y1=2;
        z = y1*x1;
    }
}
```
Call-by-value-result

- Works like call by value but in addition, formal parameters are assigned to actual parameters at the end of procedure.

```c
void p (int x, int y) {
    x = x + 1;
    y = y + 1;
}
...
int a = 3;
p(a, a);
```

- After the call, `a` will have the value 4, whereas with call-by-reference, `a` will have the value 5.
The following is the equivalent of call-by-value-result call above:

\[
\begin{align*}
x &= a; \\ y &= a; \\ x &= x + 1; \\ y &= y + 1; \\ a &= x; \\ a &= y;
\end{align*}
\]

thus, at the end, \(a = 4\).
Example:

```c
void p(int x, y){
    x = x + 1;
    y = y + 1;
}
main(){
    int u = 3;
    p(u,u);
}
```

After replacement:

```c
main(){
    int u = 3;
    {
        int x1 = u;
        int y1 = u;
        x1 = x1 + 1;
        y1 = y1 + 1;
        u = x1; u = y1;
    }
}
```
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

Preprocessing:
- Substitute formal parameters in procedure body by actual parameter expressions.
- Rename as needed to avoid “name capture”

Inline:
- Substitute the invocation expression with the modified procedure body.
Example:

```c
void p(int x, y){
    if (x==0)
        then x=y;
    else{
        x=y+1;
    }
}
main(){
    int u=5; int v=0;
    p(v,u/v);
}
```

After replacement:

```c
main(){
    int u=5; int v=0;
    {
        if (v==0)
            then v=u/v;
        else{
            v=u/v+1;
        }
    }
}
```
Call-By-Need

- Similar to call-by-name, but the actual parameter is evaluated at most once
  - Has same semantics as call-by-name in functional languages
    - This is because the value of expressions does not change with time
  - Has different semantics in imperative languages, as variables involved in the actual parameter expression may have different values each time the expression is evaluated in C-B-Name
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 (Continued)

- **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; z = 3*z;}
  }
  ```
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; // z renamed as z1
        z1 = 2*z; // y replaced by z without confusion with original z
        z = 3*z1; // confusion with original z
    }
}
```
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 in which formals are assigned back to the actual parameters.
- Otherwise, relatively easy to understand.
If-Then-Else Parameter Passing Mechanisms

Difficulties With CBVR (Continued)

Example:

```c
int f(int x, int y) {
    x=4;
    y=5;
}
void g() {
    int z;
    f(z, z);
}
```

If assignment of formal parameter values to actual parameters were performed left to right, then \( z \) would have a value of 5.

If they were performed right to left, then \( z \) will be 4.
Difficulties in Using CBR

- **Aliasing can create problems.**
  ```c
  int rev(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 `rev(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 rev will be 4,3,3,4.
CBN is complicated, and can be confusing in the presence of side-effects.

- actual parameter 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.
If the same variable is used in multiple parameters.

```c
void swap(int x, int y) {
    int tp = x;
    x = y;
    y = tp;
}
```

```c
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 2, 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 and load-time constants.

**Stack area:** for execution-time data that obeys a last-in first-out lifetime rule.
- Examples: nested declarations and temporaries.

**Heap:** a dynamically allocated area for “fully dynamic” data, i.e. data that does not obey a LIFO rule.
- Examples: objects in Java, lists in OCaml.
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
    - ARs go on the stack.
Procedures and the environment

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

Structure of AR:

- Actual parameters
- Return value
- Return address
- Saved BP (control link)
- Local variables
- Temporary variables

Direction of stack growth
Procedures and the environment

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

<table>
<thead>
<tr>
<th>Base Pointer</th>
<th>Actual parameters</th>
<th>Return value</th>
<th>Return address</th>
<th>Saved BP (control link)</th>
<th>Local variables</th>
<th>Temporary variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>+offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Direction of stack growth

push X → decr SP
move X to *SP
push BP
Access to Local Variables

- 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:

```c
{int x; int y;
 { int z; }
 { int w; }
}
```

<table>
<thead>
<tr>
<th>x:</th>
<th>-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>y:</td>
<td>-12</td>
</tr>
<tr>
<td>z:</td>
<td>-16</td>
</tr>
<tr>
<td>w:</td>
<td>-16</td>
</tr>
</tbody>
</table>
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 (unless return is through a register)
- Call: 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 [BP-k], parameters using [BP+l]
If-Then-Else Parameter Passing Mechanisms

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 and parameters
  - 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.
Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope
Access Link vs Control Link

- **Control Link** is a reference to the AR of the caller.
- **Access Link** is a reference to the AR of the surrounding scope.

**Dynamic Scoping**: When an identifier is not found in the current AR, use *control link* to access caller’s AR and look up the name there.
- If not found, keep walking up the control links until name is found.
Access Link vs Control Link

- Control Link is a reference to the AR of the caller.
- Access link is a reference to the AR of the surrounding scope.
- **Dynamic Scoping**: When an identifier is not found in the current AR, use *control link* to access caller’s AR and look up the name there.
  - If not found, keep walking up the control links until name is found.
- **Static Scoping**: When an identifier is not found in the AR of the current function, use *access link* to get to AR for the surrounding scope and look up the name there.
  - If not found, keep walking up the access links until the name is found.
Access Link vs Control Link

- Control Link is a reference to the AR of the caller
- Access link is a reference to the AR of the surrounding scope

**Dynamic Scoping:** When an identifier is not found in the current AR, use control link to access caller’s AR and look up the name there
  - If not found, keep walking up the control links until name is found

**Static Scoping:** When an identifier is not found in the AR of the current function, use access link to get to AR for the surrounding scope and look up the name there
  - If not found, keep walking up the access links until the name is found.

**Note:** Except for top-level functions, access links correspond to function scopes, so they cannot be determined statically
  - They need to be “passed in” like a parameter.
If p used its caller’s BP to access x, then it ends up accessing the variable x defined within p
- This would be dynamic scoping.
- To get static scoping, access should use q’s BP

**Access link:** q explicitly passes a link to its BP
- Calls to self: pass access link without change.
- Calls to immediately nested functions: pass your BP
- Calls to outer functions: Follow your access link to find the right access link to pass
- Other calls: these will be invalid (like goto to an inner block)
Supporting Closures

- **Closures** are needed for
  - Call-by-name and lazy evaluation
  - Returning dynamically constructed functions containing references to variables in surrounding scope

- Variables inside closures may be accessed long after the functions defining them have returned
  - Need to “copy” variable values into the closure, or
  - Not free the AR of functions when they return,
    - i.e., allocate ARs on heap and garbage collect them
```cpp
std::function<int(int a, float b) { return [a, b]() { return a + b; }}
```

```cpp
void f() {
    auto f = q(2, 3);
    return f();
}
```

```cpp
x = y + z;
add y, z, x;
```
AST classes

- constructor
- print
- type check
- codegen
- eval

mem_alloc()
Example:

```cpp
int fac(int n) {
    if (n <= 0) throw (-1) ; else if (n > 15) throw ("n too large");
    else return n*fac(n-1); }
void g (int n) {
    int k;
    try { k = fac (n) ;
    catch (int i) { cout << "negative value invalid" ; }
    catch (char *s) { cout << s; }
    catch (...) { cout << "unknown exception" ;}
g(-1) will print “negative value invalid”, g(16) will print “n too large”
```
Exception Vs Return Codes

- Exceptions are often used to communicate error values from a callee to its caller. Return values provide alternate means of communicating errors.

- Example use of exception handler:

```c
float g (int a, int b, int c) {
    float x = fac(a) + fac(b) + fac(c) ; return x ; }
main() {
    try { g(-1, 3, 25); }
    catch (char *s) { cout << "Exception ‘" << s << "’raised, exiting\n"; }
    catch (...) { cout << "Unknown exception, exiting\n"; }
}
```

- We do not need to concern ourselves with every point in the program where an error may arise.
float g(int a, int b, int c) {
    int x1 = fac(a);
    if (x1 > 0) {
        int x2 = fac(b);
        if (x2 > 0) {
            int x3 = fac(c);
            if (x3 > 0)
                return x1 + x2 + x3;
            else return x3;
        }
        else return x2;
    }
    else return x1;
}

main() {
    int x = g(-1, 2, 25);
    if (x < 0) { /* identify where error occurred, print */ }
}
Use of Exceptions in C++ Vs Java

- In C++, exception handling was an after-thought.
  - Earlier versions of C++ did not support exception handling.
  - Exception handling not used in standard libraries
  - Net result: continued use of return codes for error-checking

- In Java, exceptions were included from the beginning.
  - All standard libraries communicate errors via exceptions.
  - Net result: all Java programs use exception handling model for error-checking, as opposed to using return codes.
Implementation of Exception Handling

- Exception handling can be implemented by adding “markers” to ARs to indicate the points in program where exception handlers are available.

- In C++, entering a try-block at runtime would cause such a marker to be put on the stack.

- When exception arises, the RTE gets control and searches down from stack top for a marker.

- Exception then "handed" to the catch statement of this try-block that matches the exception.

- If no matching catch statement is present, search for a marker is continued further down the stack, and the whole process is repeated.
Memory Allocation

- A variable is stored in memory at a location corresponding to the variable.
- Constants do not need to be stored in memory.
- Environment stores the binding between variable names and the corresponding locations in memory.
- The process of setting up this binding is known as storage allocation.
Static Allocation

- Allocation performed at compile time.
- Compiler translates all names to corresponding location in the code generated by it.
- Examples:
  - all variables in original FORTRAN
  - all global and static variables in C/C++/Java
Stack Allocation

- Needed in any language that supports the notion of local variables for procedures.
- Also called “automatic allocation”, but this is somewhat of a misnomer now.
- Examples: all local variables in C/C++/Java procedures and blocks.
- Implementation:
  - Compiler translates all names to relative offsets from a location called the “base pointer” or “frame pointer”.
  - The value of this pointer varies will, in general, be different for different procedure invocations.
Stack Allocation (Continued)

- The pointer refers to the base of the “activation record” (AR) for an invocation of a procedure.
- The AR holds such information as parameter values, local variables, return address, etc.

```c
int fact(int n){
    if n=0 then 1
    else{
        int rv = n*fact(n-1);
        return rv;
    }
}
main(){
    fact(5);
}
```
An activation record is created on the stack for each a call to function.

The start of activation record is pointed to by a register called BP.

On the first call to fact, BP is decremented to point to new activation record, n is initialized to 5, rv is pushed but not initialized.

New activation record is created for the next recursive call and so on.

When n becomes 0, stack is unrolled where successive rv’s are assigned the value of n at that stage and the rv of previous stage.
Heap Management

- Issues
  - No LIFO property, so management is difficult
  - Fragmentation
  - Locality

- Models
  - explicit allocation, free
    - e.g., malloc/free in C, new/delete in C++
  - explicit allocation, automatic free
    - e.g., Java
  - automatic allocation, automatic free
    - e.g., Lisp, OCAML, Python, JavaScript
Internal fragmentation: When asked for $x$ bytes, allocator returns $y > x$ bytes
- $y - x$ represents internal fragmentation

External fragmentation: When (small) free blocks of memory occur in between (i.e., external to) allocated blocks
- the memory manager may have a total of $M \gg N$ bytes of free memory available, but no contiguous block larger enough to satisfy a request of size $N$. 
Fragmentation

Allocate 32 bytes

You cannot satisfy although 64B is available
Approaches for Reducing Fragmentation

- Use blocks of single size (early LISP)
  - Limits data-structures to use less efficient implementations.

- Use bins of fixed sizes, e.g., $2^n$ for $n = 0, 1, 2, \ldots$
  - When you run out of blocks of a certain size, break up a block of next available size
  - Eliminates external fragmentation, but increases internal fragmentation

- Maintain bins as LIFO lists to increase locality

- malloc implementations (Doug Lea)
  - For small blocks, use bins of size $8k$ bytes, $0 < k < 64$
  - For larger blocks, use bins of sizes $2^n$ for $n > 9$
Coalescing

What if a program allocates many 8 byte chunks, frees them all and then requests lots of 16 byte chunks?

- Need to coalesce 8-byte chunks into 16-byte chunks
- Requires additional information to be maintained
  - for allocated blocks: where does the current block end, and whether the next block is free
Coalescing
Explicit Vs Automatic Management

- Explicit memory management can be more efficient, but takes a lot of programmer effort.
- Programmers often ignore memory management early in coding, and try to add it later on.
  - But this is very hard, if not impossible.

Result:
- Majority of bugs in production code is due to memory management errors:
  - Memory leaks
  - Null pointer or uninitialized pointer access
  - Access through dangling pointers
Managing Manual Deallocation

How to avoid errors due to manual deallocation of memory

- Never free memory!!!
- Use a convention of object ownership (owner responsible for freeing objects)
  - Tends to reduce errors, but still requires a careful design from the beginning. (Cannot ignore memory deallocation concerns initially and add it later.)
- Smart data structures, e.g., reference counting objects
- Region-based allocation
  - When a collection of objects having equal life time are allocated
  - Example: Apache web server’s handling of memory allocations while serving a HTTP request
Garbage Collection

- Garbage collection aims to avoid problems associated with manual deallocation
  - Identify and collect garbage automatically

- What is garbage?
  - Unreachable memory

- Automatic garbage collection techniques have been developed over a long time
  - Since the days of LISP (1960s)
Garbage Collection Techniques

- **Reference Counting**
  - Works if there are no cyclic structures

- **Mark-and-sweep**

- **Generational collectors**

- **Issues**
  - Overhead (memory and space)
  - Pause-time
  - Locality
Each heap block maintains a count of the number of pointers referencing it.

Each pointer assignment increments/decrements this count

Deallocation of a pointer variable decrements this count

When reference count becomes zero, the block can be freed
Disadvantages:

- Does not work with cyclic structures
- May impact locality
- Increases cost of each pointer update operation

Advantages:

- Overhead is predictable, fixed
- Garbage is collected immediately, so more efficient use of space
Reference Counting

How many pointers point to this block?

p

Ref Count

usable part

Memory leak

x = y

Inc ref count

Dec ref count

block freed by

p = NULL;
Mark-and-Sweep

- Mark every allocated heap block as “unreachable”
- Start from registers, local and global variables
- Do a depth-first search, following the pointers
  - Mark each heap block visited as “reachable”
- At the end of the sweep phase, reclaim all heap blocks still marked as unreachable
Mark-and-Sweep

"root pointers"

DFS starting from root pointers

on the stack

in global memory
Garbage Collection Issues

- Memory fragmentation
  - Memory pages may become sparsely populated
  - Performance will be hit due to excessive virtual memory usage and page faults
  - Can be a problem with explicit memory management as well
    - But if a programmer is willing to put in the effort, the problem can be managed by freeing memory as soon as possible

- Solution:
  - Compacting GC
    - Copy live structures so that they are contiguous
  - Copying GC
Copying Garbage Collection

- Instead of doing a sweep, simply copy over all reachable heap blocks into a new area.
- After the copying phase, all original blocks can be freed.
- Now, memory is compacted, so paging performance will be much better.
- Needs up to twice the memory of compacting collector, but can be much faster.
  - Reachable memory is often a small fraction of total memory.
Copying Garbage Collection

- Current heap
- Copy all live objects
- Next gen heap
- All allocations
- Go here
- Make this current heap after GC
Generational Garbage Collection

- Take advantage of the fact that most objects are short-lived
- Exploit this fact to perform GC faster
- Idea:
  - Divide heap into generations
  - If all references go from younger to older generation (as most do), can collect youngest generation without scanning regions occupied by other generations
  - Need to track references from older to younger generation to make this work in all cases
Garbage collection in Java

- Generational GC for young objects
- “Tenured” objects stored in a second region
  - Use mark-and-sweep with compacting
- Makes use of multiple processors if available

References


GC for C/C++: Conservative Garbage Collection

- Cannot distinguish between pointers and nonpointers
  - Need “conservative garbage collection”

- The idea: if something “looks” like a pointer, assume that it may be one!
  - Problem: works for finding reachable objects, but cannot modify a value without being sure
    - Copying and compaction are ruled out!

- Reasonable GC implementations are available, but they do have some drawbacks
  - Unpredictable performance
  - Can break some programs that modify pointer values before storing them in memory
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;
}
Implementation Aspects of OO-Languages

```java
class C {
    int k, l; B b;
}
```

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>int k</td>
<td>0</td>
</tr>
<tr>
<td>int l</td>
<td>4</td>
</tr>
<tr>
<td>int i</td>
<td>8</td>
</tr>
<tr>
<td>double d</td>
<td>12</td>
</tr>
<tr>
<td>char c</td>
<td>16</td>
</tr>
<tr>
<td>float f</td>
<td>20</td>
</tr>
</tbody>
</table>
```

alignment requirement as the largest alignment of any of B's fields.
class D: public C {
    double x;
}

class E: public C, public B {
    C's part
    public B's part

    int k
    int l
    int i
    double d
    char c
    float f
    double x

    new fields of E
    new fields of D start here
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.
• 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.
class B {
  int i ;
  char c ;
  virtual void g();
  virtual void h();
}

B b1, b2;

B's VMT

VMT ptr

Ptr to B's g

Ptr to B's h

Offset: 0 in VMT

Offset: 1 in VMT

*$(&b1+2)+1$
• 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.
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:

  ```cpp
  class A: public B {
      float f;
      void h(); // reuses implementation of G from B;
      virtual void k();
  }
  ```

  A a;

  a’s layout
  
<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>VMT ptr</th>
<th>Float f</th>
</tr>
</thead>
</table>

  Virtual Method Table (VMT) for class A
  
<table>
<thead>
<tr>
<th>B’s part</th>
<th>A’s new fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>B’s g</td>
<td>A’s h</td>
</tr>
<tr>
<td>B’s ptr VMT</td>
<td>A’s k</td>
</tr>
</tbody>
</table>
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 in its superclass.