THEORY OF COMPILATION

LECTURE 11

MEMORY, OOP AND ERROR HANDLING
You are here

Source text ➔ Lexical Analysis ➔ Syntax Analysis ➔ Semantic Analysis ➔ IR Optimization ➔ Code Generation ➔ Executable code + Runtime
Today

• Memory management
  ‣ Manual
  ‣ Garbage collection
• Object-oriented: overloading, polymorphism
• Error handling (poor man’s version)
Representing Data at Runtime

• Source language types
  – int, boolean, string, object types

• Target language types
  – Single bytes, integers, address representation

• Compiler should map source types to some combination of target types
  – Implement source types using target types
Where do we allocate data?

• Activation records on the stack
  – Lifetime of allocated data limited by procedure lifetime
  – Stack frame deallocated (popped) when procedure return
  – (that was the whole idea)

• Dynamic memory allocation on the heap
Memory Layout

- **Stack**: Grows down (towards lower addresses)
- **Heap**: Grows up (towards higher addresses)
- **Static Data**
- **Code**

Higher addresses:

Lower addresses: 0x0000
Allocating memory

- In C — `malloc`
  
  ```c
  void *malloc(size_t size)
  ```

- Why does `malloc` return `void*`?
Allocating memory

• In C — malloc
  
  ```c
  void *malloc(size_t size)
  ```

• Why does malloc return void* ?
  ‣ It just allocates a chunk of memory, without regard to its type

• How does malloc guarantee alignment?
  ‣ After all, you don’t know what type it is allocating for
Allocating memory

- In C — malloc
  
  ```c
  void *malloc(size_t size)
  ```

- Why does malloc return void* ?
  - It just allocates a chunk of memory, without regard to its type

- How does malloc guarantee alignment?
  - After all, you don’t know what type it is allocating for
  - It has to align for the largest primitive type
Allocating memory

• In C — malloc

\[ \text{void *malloc(size_t size)} \]

• Why does malloc return void* ?
  ‣ It just allocates a chunk of memory, without regard to its type

• How does malloc guarantee alignment?
  ‣ After all, you don’t know what type it is allocating for
  ‣ It has to align for the largest primitive type
  ‣ In practice optimized for 8 byte alignment (glibc-2.17)
Basic Types

- int, boolean, char, void
- Arithmetic operations
  - Addition, subtraction, multiplication, division, remainder
- Could be mapped directly to target language types and operations
Pointer Types

- Represent addresses of source language data structures
- Usually implemented as an unsigned integer
- Pointer dereferencing — retrieves pointed value
- Pointer arithmetic — access elements of array/buffer

- May produce an error
  - Null pointer dereference ($p = 0$)
  - Segmentation violation ($p$ outside heap area)
Memory Management

- Manual deallocation
- Automatic deallocation
Manual Memory Management

- `malloc` (new)
- `free` (delete)

```c
a = malloc(...);  
// do something with a  
free(a);  
```
free

- Free too late – waste memory (memory leak)
- Free too early – dangling pointers / crashes
- Free twice – error
When can we free an object?
When can we free an object?

```c
a = malloc(…); 
b = a;
free (a); ?
c = malloc (…);
if (b == c)
    printf(“unexpected equality”);
```
When can we free an object?

```c
a = malloc(…) ;
b = a;
free (a); ?
c = malloc (…) ;
if (b == c)
    printf("unexpected equality");
```

Cannot free an object if it has a reference with a future use!
Automatic Memory Management

• Automatically free memory when it is no longer needed

• prevalent in OO languages such as Java, C#
  ‣ but not limited to OO languages

• Challenge: how can we know an object will not be used, since we cannot tell the future??
Garbage collection

- **approximate** reasoning about object liveness
- use reachability to approximate liveness
- **assume reachable objects are live**
  - non-reachable objects are dead
Garbage Collection – Classical Techniques

• Reference counting
• Mark and sweep
• Copying
GC using Reference Counting

• Add a reference-count field to every object
  ‣ tracks how many pointers refer to it
• when (refcnt==0), the object is unreachable
  ‣ not reachable ⇒ dead
  ‣ can be collected (deallocated)
Managing Reference Counts

- Each object has a reference count `o.RC`
- A newly allocated object `o` gets `o.RC = 1`

- write-barrier for reference updates
  ```c
  update(x,old,new) {
    old.RC--;
    new.RC++;
    if (old.RC == 0) collect(old);
  }
  ```

- `collect(old)` will decrement RC for all children and recursively collect objects whose RC reached 0.
The Problem — Cycles!

- Cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
The Problem — Cycles!

• Cannot identify non-reachable cycles
  ‣ reference counts for nodes on the cycle will never decrement to 0

• several approaches for dealing with cycles
  ‣ ignore
  ‣ weak references
  ‣ periodically invoke a tracing algorithm to collect cycles
  ‣ specialized algorithms for collecting cycles
\[ x \rightarrow n \rightarrow n \rightarrow n \]

\[ RC = 2 \quad RC = 1 \quad RC = 1 \]

\[ x = null \]

\[ RC = 1 \quad RC = 1 \quad RC \leq 1 \]
The Mark-and-Sweep Algorithm
[McCarthy 1960]

• **Marking phase**
  – mark roots
  – trace all objects transitively reachable from roots
  – mark every traversed object

• **Sweep phase**
  – scan all objects in the heap
  – collect all unmarked objects
The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.
The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.

Note! This is not the heap data structure!
**Triggering**

Garbage collection is triggered by allocation

\[
\text{New}(A) =
\begin{align*}
\text{if free_list is empty} & \\
& \text{mark_sweep()} \\
& \text{if free_list is empty} \\
& \text{return ("out-of-memory")} \\
\text{pointer} = \text{allocate}(A) \\
\text{return (pointer)}
\end{align*}
\]
Basic Algorithm

```
mark_sweep() =
for Ptr in Roots
    mark(Ptr)
sweep()
```
Basic Algorithm

\textbf{mark\_sweep}()=
for \textbf{Ptr} in \textbf{Roots}
mark\(\textbf{Ptr}\)
sweep()

\textbf{mark}(\textbf{Obj})=
if mark\_bit(\textbf{Obj}) == \textbf{unmarked}
mark\_bit(\textbf{Obj})=\textbf{marked}
for \textbf{C} in \text{Children}(\textbf{Obj})
mark\(\textbf{C}\)
**Basic Algorithm**

**mark_sweep()**

```python
for Ptr in Roots
    mark(Ptr)
sweep()
```

**mark(Obj)**

```python
if mark_bit(Obj) == unmarked
    mark_bit(Obj) = marked
    for C in Children(Obj)
        mark(C)
```

**Sweep()**

```python
p = Heap_bottom
while (p < Heap_top)
    if (mark_bit(p) == unmarked) then free(p)
    else mark_bit(p) = unmarked;
    p = p + size(p)
```
Mark&Sweep Example

r1

r2
Mark&Sweep in Depth

• How much memory does it consume?
  – Recursion depth?
  – Can you traverse the heap without worst-case $O(n)$ stack?
    • Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)
Mark&Sweep in Depth

\texttt{mark(Obj)=}
\texttt{if mark\_bit(Obj) == unmarked}
\texttt{mark\_bit(Obj)=marked}
\texttt{for C in Children(Obj)}
\texttt{mark(C)}

• How much memory does it consume?
  – Recursion depth?
  – Can you traverse the heap without worst-case O(n) stack?
    • Deutch-Schorr-Waite algorithm for graph marking without recursion or stack (works by reversing pointers)
Properties of Mark & Sweep

• Most popular method today
• Simple
• Does not move objects, and so heap may fragment
• Complexity
  🧄Mark phase: live objects (dominant phase)
  😞Sweep phase: heap size
• Termination: each pointer traversed once
• Engineering tricks used to improve performance
Mark-Compact

- During the run objects are allocated and reclaimed.
- Gradually, the heap gets fragmented.
- When space is too fragmented to allocate, a compaction algorithm is used.
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations.
- Compaction is very costly and we attempt to run it infrequently, or only partially.
Mark-Compact

- During the run objects are allocated and reclaimed
- Gradually, the heap gets fragmented
- When space is too fragmented to allocate, a compaction algorithm is used
- Move all live objects to the beginning of the heap and update all pointers to reference the new locations
- Compaction is very costly and we attempt to run it infrequently, or only partially
Parallel Mark&Sweep GC

Parallel GC: mutator is stopped, GC threads run in parallel
Parallel Mark&Sweep GC

Thread 1
Thread 2

Parallel GC: mutator is stopped, GC threads run in parallel
Concurrent Mark&Sweep GC

Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator
Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator
Concurrent Mark & Sweep GC

Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator
Concurrent Mark&Sweep GC

Concurrent GC: mutator and GC threads run in parallel, no need to stop mutator
Problem: Interference

SYSTEM = MUTATOR || GC

1. GC traced B
Problem: Interference

SYSTEM = MUTATOR | | GC

1. GC traced B
2. Mutator links C to B
Problem: Interference

SYSTEM = MUTATOR || GC

1. GC traced B
2. Mutator links C to B
3. Mutator unlinks C from A
Problem: Interference

SYSTEM = MUTATOR || GC

1. GC traced B
2. Mutator links C to B
3. Mutator unlinks C from A
4. GC traced A

C LOST
The 3 Families of Concurrent GC Algorithms

1. Marks C when C is linked to B (DIJKSTRA)

2. Marks C when link to C is removed (YUASA)

3. Rescan B when C is linked to B (STEELE)
The 3 Families of Concurrent GC Algorithms

1. Marks C when C is linked to B (DIJKSTRA)

2. Marks C when link to C is removed (YUASA)

3. Rescan B when C is linked to B (STEELE)
The 3 Families of Concurrent GC Algorithms

1. Marks C when C is linked to B (DIJKKSTRA)

2. Marks C when link to C is removed (YUASA)

3. Rescan B when C is linked to B (STEELE)
The 3 Families of Concurrent GC Algorithms

1. Marks C when C is linked to B (DIJKSTRA)

2. Marks C when link to C is removed (YUASA)

3. Rescan B when C is linked to B (STEELE)
Copying GC

• Partition the heap into two parts
  – old space
  – new space

• Copying GC algorithm
  – copy all \texttt{reachable} objects from old space to new space
  – swap roles of old/new space
Example

Roots

old

new

A
B
C
D
E
Example
Properties of Copying Collection

• Compaction for free
• Major disadvantage: half of the heap is not used
• Advantage: “touches” only live objects
  ▸ Good when most objects are dead
  ▸ Observation: usually, most new objects are dead
    ⇒ Some methods use a small space for young objects and collect just this space using copying garbage collection
Modern Memory Management

• Considers standard program properties
• Handle parallelism
  – Stop the program and collect in parallel on all available processors
  – Run collection concurrently with the program run
• Cache consciousness
• Real-time
Conservative GC

• How do you track pointers in languages such as C?
  – Any value can be cast down to a pointer
Conservative GC

• How do you track pointers in languages such as C?
  – Any value can be cast down to a pointer

• Easy – be conservative, consider anything that can be a pointer to be a pointer

• Practical! (e.g., Boehm collector)
Conservative GC
Conservative GC

• Can you implement a conservative copying GC?

• What is the problem?

• Cannot update pointers to the new address... you don’t know whether the value is a pointer, cannot update it
Switching Topic

COMPILING OBJECT ORIENTED
Object Types

• Basic operations
  – Field selection
    • computing address of field, dereferencing address
  – Copying
    • copy block (shallow) or field-by-field copy (deep)
  – Method invocation
    • Identifying method to be called, calling it

• How does it look at runtime?
class Foo {
    int x;
    int y;

    void rise() {...}
    void shine() {...}
}

Object Types

Compile time information

Runtime memory layout for object of class Foo
Field Selection

```cpp
Foo f;
int q;
q = f.x;

Foo f;
int q;
q = f.x;

t1 = f

int q;
q = f.x;

t2 = t1 + 4

q = *t2
```

**Runtime memory layout** for object of class Foo

```
DispatchVecPtr
x
y
```

**Compile time** information

```
rise
shine
```
Field Selection

Foo f;
int q;
q = f.x;

t1 = f

\[ t2 = t1 + 4 \]

\[ q = *t2 \]

Runtime memory layout for object of class Foo

Compile time information
class Foo {
    int x;
    int y;

    void rise() {...}
    void shine() {...}
}

class Bar extends Foo{
    int z;
    void twinkle() {...}
}
class Foo {
    ... 
    void rise() {...} 
    void shine() {...} 
}

class Bar extends Foo {
    ... 
}

class Main {
    void main() {
        Foo f = new Bar();
        f.rise();
    }
}
Dynamic Binding

- Finding the right method implementation
- Done at runtime according to object type
- Using the Dispatch Vector (sometimes called Dispatch Table)

```java
class Foo {
    ...
    void rise() {...}
    void shine() {...}
}

class Bar extends Foo{
    void rise() {...}
}

class Main {
    void main() {
        Foo f = new Bar();
        f.rise();
    }
}
```
Dispatch Vectors

- Vector contains addresses of methods
- Indexed by method-id number
- A method signature has the same id number for all subclasses
Representing dispatch tables

class A {
    void rise() {…}
    void shine() {…}
    {…}
}
class B extends A {
    void rise() {…}
    void shine() {…}
    void twinkle() {…}
}

# data section
.data
.align 4
_DV_A:
    .long _A_rise
    .long _A_shine
_DV_B:
    .long _B_rise
    .long _B_shine
    .long _B_twinkle
class C {
    ...
}

class D extends C{
    ...
}

class E {
    void foo(C c) {...}
    void foo(D d) {...}
}
class C {
    ...
}
class D extends C{
    ...
}
class E {
    void foo(C c) {...}
    void foo(D d) {...}
}
Switching Topic Again

ERROR HANDLING
Runtime checks

- Generate code for checking attempted illegal operations
  - Null pointer check
    - MoveField, MoveArray, ArrayLength, VirtualCall
    - Reference arguments to library functions should not be null
  - Array bounds check
  - Array allocation size check
  - Division by zero
    - ...

- If check fails jump to **error handler** code that prints a message and gracefully exits program
- Alternatively, use an exception-handling mechanism
Null pointer check

```
# null pointer check
cmp $0,%eax
je labelNPE
```

Single generated handler for entire program

```
labelNPE:
push $strNPE       # error message
call __println
push $1            # error code
call __exit
```
Array allocation size check

# array size check
cmp $0, %eax  # eax = array size
jle labelASE  # eax <= 0 ?

labelASE:
  push $strASE   # error message
  call __println
  push $1        # error code
  call __exit

Single generated handler for entire program
Array bounds check

# array bounds check
mov -4(%eax),%ebx  # ebx = length
mov $0,%ecx        # ecx = index
cmp %ecx,%ebx
jle labelABE       # ebx <= ecx ?
cmp $0,%ecx
jl  labelABE       # ecx < 0 ?

Single generated handler for entire program

labelABE:
push $strABE        # error message
call __println
push $1              # error code
call __exit
Exceptions

```java
main() {
    foo(1)
}

foo(int n) {
    bar(n - 1, 1)
}

bar(int x, int y) {
    if (x < y)
        throw new Exception()
}
```

Exception thrown
try { ... } catch { ... }

- Allows programmer to customize error handler

```java
if (exc.type == IO_ERROR) {
    ... something something ...
    jump to <after catch block>
}
else if (exc.type == HW_ERROR) {
    ... something something ...
    jump to <after catch block>
}
else {
    unwind
    throw exc
}
```

- (Default handler throws unconditionally)
try { ... } catch { ... }

- Allows programmer to customize error handler

if (exc.type == IO_ERROR) {
    ... something something ...
    jump to <after catch block>
}
else if (exc.type == HW_ERROR) {
    ... something something ...
    jump to <after catch block>
}
else {
    unwind
    throw exc
}

- (Default handler throws unconditionally)
try { ... } catch { ... }

• Allows programmer to customize error handler

```java
if (exc.type == IO_ERROR) {
    ... something something ...
    jump to <after catch block>
}
else if (exc.type == HW_ERROR) {
    ... something something ...
    jump to <after catch block>
}
else {
    unwind
    throw exc
}
```

• (Default handler throws unconditionally)
Exception Handling Overhead

- Handler address has to be updated
  - whenever a try {} block is entered
  - and when it ends successfully
- Implies constant cost even when no exception is thrown
  - C++ programmers cannot have that
Exception Handling Overhead

- **Idea:** active handler is known, at compile time, per emitted location

```c
void foo(int n) {
    try {
        throw;
        try {
            throw;
        } catch () { }
        throw;
    } catch () { }
    catch() { }
    throw;
}
```
Exception Handling Overhead

- **Idea**: active handler is known, at compile time, per emitted location

```c
foo(int n) {
    try {
        throw
        try {
            throw
        }
        catch() { } throw
    }
    catch() { }
    throw
}
```
Exception Handling Overhead

• **Idea**: active handler is known, at compile time, per emitted location

```c
foo(int n) {
    try {
        throw
        try {
            throw
        }
        catch() { }
        throw
    } catch() { }
    catch() { }
    throw
}
```
Exception Handling Overhead

- **Idea**: active handler is known, at compile time, per emitted location

```c
1 foo(int n) {
2     try {
3         throw
4         try {
5             throw
6         }
7         catch() { }
8         throw
9     } catch() { }
10    catch() { }
11    throw (default)
12 }
```
Exception Handling Overhead

- **Idea**: active handler is known, at compile time, per emitted location
Exception Handling Overhead

• **Solution**: store a hash table that maps pc → handler address.
Unhandled Exception?

org.eclipse.swt.SWTException: Graphic is disposed
  at org.eclipse.swt.SWT.error(SWT.java:3744)
  at org.eclipse.swt.SWT.error(SWT.java:3662)
  at org.eclipse.swt.SWT.error(SWT.java:3633)
  at org.eclipse.swt.graphics.GC.getClipping(GC.java:2266)
  at com.aelitis.azureus.ui.swt.views.list.ListRow.doPaint(ListRow.java:260)
  at com.aelitis.azureus.ui.swt.views.list.ListRow.doPaint(ListRow.java:237)
  at com.aelitis.azureus.ui.swt.views.list.ListView.handleResize(ListView.java:867)
  at com.aelitis.azureus.ui.swt.views.list.ListView$5$2.runSupport(ListView.java:406)
  at org.gudy.azureus2.core3.util.AERunnable.run(AERunnable.java:38)
  at org.eclipse.swt.widgets.RunnableLock.run(RunnableLock.java:35)
  at org.eclipse.swt.widgets.Synchronizer.runAsyncMessages(Synchronizer.java:130)
  at org.eclipse.swt.widgets.Display.runAsyncMessages(Display.java:3323)
  at org.eclipse.swt.widgets.Display.readAndDispatch(Display.java:2985)
  at org.gudy.azureus2.ui.swt.mainwindow.SWTThread.<init>(SWTThread.java:183)
  at org.gudy.azureus2.ui.swt.mainwindow.SWTThread.createInstance(SWTThread.java:67)

(Java)
Unhandled Exception?

Abort

(C++)