Other data types

Nested records without implicit pointers, as in C

```
struct S1 {
   int x;
   struct S2 {
      double y;
      S3* z;
   } s2;
   int w;
} s1;
```

Unions, as in C

```
union U {
   int x;
   double y;
   S3* z;
   int w;
} u;
```


Other data types

Multidimensional arrays: *T*`[][]`...

- rectangular matrix?
- array of arrays?

Strings

- null-terminated arrays of characters, as in C
- length-prefixed array of characters, as in Java

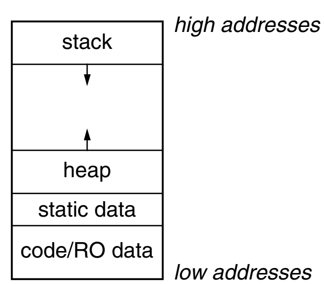
Storage layout

Where to allocate space for each variable/data structure?

Key issue: what is the **lifetime** (**dynamic extent**) of a variable/data structure?

- whole execution of program (global variables)
  ⇒ **static** allocation
- execution of a procedure activation (formals, local vars)
  ⇒ **stack** allocation
- variable (dynamically-allocated data)
  ⇒ **heap** allocation

Parts of run-time memory



Code/RO data area

- read-only data & machine instruction area
- shared across processes running same program

Static data area

- place for read/write variables at fixed location in memory
- can start out initialized, or zeroed

Heap

- place for dynamically allocated/freed data
- can expand upwards through `sbrk` system call

Stack

- place for stack-allocated/freed data
- expands/contracts downwards automatically

Static allocation

Statically allocate variables/data structures with global lifetime

- global variables in C, `static` class variables in Java
- `static` local variables in C, all locals in Fortran
- compile-time constant strings, records, arrays, etc.
- machine code

Compiler uses symbolic address

Linker assigns exact address, patches compiled code

`ILGlobalVarDecl` to declare statically allocated variable

`ILFunDecl` to declare function

`ILGlobalAddressExpr` to compute address of statically allocated variable or function

Stack allocation

Stack-allocate variables/data structures with **LIFO** lifetime

- last-in first-out (stack discipline):
  data structure doesn't outlive previously allocated data structures on same stack

Activation records usually allocated on a stack

- a stack-allocated a.r. called a **stack frame**
- frame includes formals, locals, static link of procedure
- dynamic link = stack frame above

Fast to allocate & deallocate storage

Good memory locality

`ILVarDecl` to declare stack allocated variable

`ILVarExpr` to reference stack allocated variable

- both with respect to some `ILFunDecl`

Problems with stack allocation

Stack allocation works only when can't have references to stack allocated data after containing function returns

Violated if first-class functions allowed

```
(int(*)(int)) curried(int x) {
   int nested(int y) { return x+y; }
   return &nested;
}

(int(*)(int)) f = curried(3);
(int(*)(int)) g = curried(4);

int a = f(5);
int b = g(6);

// what are a and b?
```


Problems with stack allocation

Violated if inner classes allowed

```
Inner curried(int x) {
   class Inner {
      int nested(int y) { return x+y; }
   };
   return new Inner();
}

Inner f = curried(3);
Inner g = curried(4);

int a = f.nested(5);
int b = g.nested(6);

// what are a and b?
```

Problems with stack allocation

Violated if pointers to locals allowed

```
int* addr(int x) { return &x; }

int* p = addr(3);
int* q = addr(4);

int a = (*p) + 5;
int b = (*p) + 6;

// what are a and b?
```


Heap allocation

Heap-allocate variables/data structures with unknown lifetime

- `new`/`malloc` to allocate space
- `delete`/`free`/garbage collection to deallocate space

Heap-allocate activation records (environments at least) of first-class functions

Put locals with address taken into heap-allocated environment, or make illegal, or make undefined

Relatively expensive to manage

Can have dangling references, storage leaks if don't `free` right

- use automatic garbage collection in place of manual `free` to avoid these problems

`ILAllocateExpr`, `ILArrayedAllocateExpr` to allocate heap memory

Garbage collection implicitly frees heap memory

Parameter passing

When passing arguments, need to support right semantics

An issue: when is argument expression evaluated?

- before call, or if & when needed by callee?

Another issue: what happens if formal assigned in callee?

- effect visible to caller? if so, when?
- what effect in face of aliasing among arguments, lexically visible variables?

Different choices lead to different representations for passed arguments and different code to access formals

Some parameter passing modes

Parameter passing options:

- call-by-value, call-by-sharing
- call-by-reference, call-by-value-result, call-by-result
- call-by-name, call-by-need
- ...

Call-by-value

If formal is assigned, caller's value remains unaffected

```
class C {
   int a;
   void m(int x, int y) {
      x = x + 1;
      y = y + a;
   }
   void n() {
      a = 2;
      m(a, a);
      System.out.println(a);
   }
}
```

Implement by passing copy of argument value

- trivial for scalars: ints, booleans, etc.
- inefficient for aggregates: arrays, records, strings, ...

Call-by-sharing

If implicitly reference aggregate data via pointer
(e.g. Java, Lisp, Smalltalk, ML, ...)
then call-by-sharing is call-by-value applied to implicit pointer

- "call-by-pointer-value"

```
class C {
   int[] a = new int[10];
   void m(int[] x, int[] y) {
      x[0] = x[0] + 1;
      y[0] = y[0] + a[0];
      x = new int[20];
   }
   void n() {
      a[0] = 2;
      m(a, a);
      System.out.println(a);
   }
}
```

- efficient, even for big aggregates
- assignments of formal to a different aggregate (e.g. `x = ...`) don't affect caller
- updates to contents of aggregate (e.g. `x[...] = ...`) visible to caller immediately

Call-by-reference

If formal is assigned, actual value is changed in caller

- change occurs immediately

```
class C {
   int a;
   void m(int& x, int& y) {
      x = x + 1;
      y = y + a;
   }
   void n() {
      a = 2;
      m(a, a);
      System.out.println(a);
   }
}
```

Implement by passing pointer to actual

- efficient for big data structures
- references to formal do extra dereference, implicitly

Call-by-value-result: do assign-in, assign-out

- subtle differences if same actual passed to multiple formals

Call-by-result

Write-only formals, to return extra results;
no incoming actual value expected

- "out parameters"
- formals cannot be read in callee, actuals don't need to be initialized in caller

```
class C {
   int a;
   void m(int&out x, int&out y) {
      x = 1;
      y = a + 1;
   }
   void n() {
      a = 2;
      int b;
      m(b, b);
      System.out.println(b);
   }
}
```

Can implement as in call-by-reference or call-by-value-result

Call-by-name, call-by-need

Variations on **lazy evaluation**

- only evaluate argument expression if & when needed by callee function

Supports very cool programming tricks

Hard to implement efficiently in traditional compiler

Incompatible with side-effects

$\Rightarrow$ only in purely functional languages, e.g. Haskell, Miranda