Prelude

- Copying an array

```c
do { /* count > 0 assumed */
  *to++ = *from++;
} while (--count > 0);
```

Duff's device

- Why is this faster?

```c
int n = (count + 7) / 8;
switch (count % 8)
{
  case 0:        do {  *to++ = *from++;
  case 7:              *to++ = *from++;
  case 6:              *to++ = *from++;
  case 5:              *to++ = *from++;
  case 4:              *to++ = *from++;
  case 3:              *to++ = *from++;
  case 2:              *to++ = *from++;
  case 1:              *to++ = *from++;
} while (--n > 0);
```

Heterogeneous data structures

- Simple example: shapes
  - Abstract type “shape”
  - Concrete subtypes: square, circle, triangle

```c
C: typedef struct Shape {
  int kind;  /* 0=square, 1=Circle, 2=triangle */
  union {
    struct Square { . . . } square;
    struct Circle { . . . } circle;
    struct Triangle { . . . } triangle;
  } info;

  Shape, *ShapePtr;
```

Labeled unions

- What's good?
  - Code for “area” all in one place
  - Probably fast

- What's bad?
  - Have to make sure to get kinds right
    (Could use enum to help)
  - No control over access to fields
  - New shapes: fix all the switch statements

```c
Shape * all_shapes[50];
float area = 0.0;
for (int i = 0; i<50; i++) {
  Shape * cur_shape = all_shapes[i];
  switch (cur_shape->kind) {
    case 0: area += cur_shape->stuff.square.side *
             cur_shape->stuff.square.side;
             break;
    case 1: . . .
    case 2: . . .
  }
}
Objects

- Common superclass
  - Common functionality
  - Required functionality

```
class Shape {
    abstract float area();
}
class Square extends Shape {
    float side;
    float area() { return side * side; }
}
class Circle extends Shape {
    float radius;
    float area() { return pi * radius * radius; }
}
class Triangle extends Shape {
    float base;
    float height;
    float area() { return 0.5 * base * height; }
}
```

- What’s good
  - Encapsulation – all fields protected
  - No type errors
  - Easy to add new shapes

- What’s bad
  - How do we add new functions?
  - May need to modify all subclasses

Visitor

- Method for each kind
  - `class Visitor {
      void visitSquare(Square sq) {}
      void visitCircle(Circle ci) {}
      void visitTriangle(Triangle tr) {}
    }`

- Idea:
  - Describe what to do with each kind of object
  - Object calls its visit class

```
class areaVisitor extends Visitor {
    float area = 0.0;
    void visitSquare(Square sq) {
        area += sq.side() * sq.side();
    }
    void visitCircle(Circle ci) {
        area += pi * ci.radius() * ci.radius();
    }
    void visitTriangle(Triangle tr) {
    }
}
```

- Note:
  - May need to carry data in the visitor

Supporting visitors

- In the shapes classes
  - `class Shape {
      void accept(Visitor vis) {
          vis.visitShape(this);
      }
  }
class Square extends Shape {
    float side;
    void accept(Visitor vis) {
        vis.visitSquare(this);
    }
  }
```

Using a visitor

- Three parts
  - Create visitor instance
  - Call “accept” method on each shape
  - Read out the result at end

```
Shape[] all_shapes;
Visitor vis = new areaVisitor();
for (int i = 0; i<50; i++) {
    Shape cur_shape = all_shapes[i];
    cur_shape.accept(vis);
}
float result = vis.area;
```
Visitor

- What’s good
  - Easy to add new “passes”
  - All code is in one place

- What’s bad
  - New shapes: may have to modify all visitors

Last time: type checking

- Type system
  - Set of rules for assigning types to parts of the program
  - Inference rules for operators

- Type checker
  - Start with declared types
  - Apply typing rules
  - At each step, make sure resulting types obey the rules of the language

- Why type checking?
  - Alert the programmer to errors
  - Help compiler generate a correct translation

Polymorphism

- Ordinary procedures
  - Accept fixed type signature
  - Example: search(char c, string s)

- Generic procedures
  - Work on arguments of different types

- User-defined generics
  - Define interface with type variables
  - Example: search(E1 c, E1 [] list)

Type checking polymorphism

- Checking generic code:
  - C++ templates
    - Checked at instantiation
    - Plug in the type variables
    - Compile and check resulting code
  - Java generics
    - Constrain type variables
    - Declare “type E1 must implement interface X”
    - Can check generic code independent of use

- Problem:
  - How do we check generics statically?
  - What do we need to know?
    - Does E1 have a method “compareTo”?
    - Is A an array of type int?

- Checking uses of generics
  - Is search(5, A) correct? (Let’s say int A[20];)
  - Find a mapping from E1, E1[] to type(5), type(A)
  - Mapping: (E1, E1[]) --> (int, int[]) when E1 is int
  - This process is called unification

- Unification
  - Given two type expressions, one with type variables
  - Find a consistent substitution of type variables that turns it into the other type expression
  - Used in many other areas: theorem proving,
Template metaprogramming

- C++ templates are very powerful

```
template<int N>
class Factorial {
public:
    enum { value = N * Factorial<N-1>::value }; 
};

class Factorial<1> {
public:
    enum { value = 1 }; 
};
x = Factorial<8>::value;
```

More...

```
template<bool C> class IfThen { };

class IfThen<true> {
public:
    static inline void do() { statement1; } // true case 
};

class IfThen<false> {
public:
    static inline void do() { statement2; } // false case 
};

// Replacement for 'if/else' statement:
IfThen<condition>::do();
```

Type inference

- Languages without declarations
  - ML, Haskell
  - Still statically typed
  - Types determined by use

- Requires type inference algorithm
  - Determine constraints from program
  - Results in type expressions with type variables
  - Compute a consistent assignment to variables

Back to Mars Orbiter

- Language support for units
  - Idea: make units a part of the type
  - Use polymorphism for operators
  - Type-check the computations

- Example:

  ```
  double<kg> weight;
  double<s> time;
  double<m> distance;
  double<m s^-2> gravity = 9.8 * m/(s * s);
  double<kg m s^-2> force = weight * gravity;
  ```

- Issues:
  - Type checker must understand algebra
  - Generics are tricky
    (what is the type of sqrt?)

Where are we?

- The latter half of a compiler contains more open problems, more challenges, and more gray areas than the front half
  - Implementing promised behavior
    What defines the meaning of the program?
  - Managing target machine resources
    Registers, memory, issue slots, locality, power, ...
  - These issues determine the quality of the compiler

Procedure abstraction

- Compile-time versus run-time
  - The compiler takes a static program and generates code for dynamic execution
  - Most of the tricky issues arise in "procedures"

- Issues
  - Finding storage, and mapping names to addresses
  - Emit code to compute addresses that the compiler cannot know at compile-time!
  - Interfaces with other programs, other languages, and the OS
  - Efficiency of implementation
Procedure: three abstractions

- **Control** abstraction
  - Well defined entries & exits
  - Mechanism to return control to caller
  - Some notion of parameterization (usually)
- **Clean name space**
  - Clean slate for writing locally visible names
  - Local names may obscure identical, non-local names
  - Local names cannot be seen outside
- **External interface**
  - Access is by procedure name & parameters
  - Clear protection for both caller & callee
  - Invoked procedure can ignore calling context

The Procedure (Realist's View)

Procedures are the key to building large systems

- Requires **system-wide contract**
  - Conventions on memory layout, protection, resource allocation calling sequences, & error handling
  - Must involve architecture (ISA), OS, & compiler
- Provides shared **access to system-wide facilities**
  - Storage management, flow of control, interrupts
  - Interface to input/output devices, protection facilities, timers, synchronization flags, counters, ...
- Establishes a **private context**
  - Create private storage for each procedure invocation
  - Encapsulate information about control flow & data abstractions

Run Time versus Compile Time

*These concepts are often confusing to the newcomer*

- **Linkage** is the code that implements the procedure interface
- Code for the linkage is emitted at **compile time**
- Linkages execute at **run time**
- The linkage is designed long before either of these

- More confusion:
  - Dynamic linking and shared libraries
  - Just-in-time compilers

The Procedure (More Abstract View)

- A procedure is an abstract structure constructed via software
- Underlying hardware does explicitly not support:
  - Entries and exits
  - Interfaces and parameter passing
  - Call and return mechanism *(may be special instructions)*
  - Name spaces and nested scopes
- Abstraction created by cooperation between:
  - Compiler
  - Run-time system
  - Linkage editor and loader
  - Operating system

The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call

- Invoked at a call site, with some set of actual parameters
- Control returns to call site, immediately after invocation
The Procedure as a Control Abstraction

Procedures have well-defined control-flow

The Algol-60 procedure call
- Invoked at a call site, with some set of actual parameters
- Control returns to call site, immediately after invocation

```
int p(a,b,c)
int a, b, c;
{
    int   d;
d = q(c,b);
    ...
}

s = p(10,t,u);
...
```

```
int q(x,y)
int x,y;
{
    return x + y;
}
```

```
s = p(10,t,u);
...
```

Most languages allow recursion

Implementing procedures with this behavior
- Requires code to save and restore a “return address”
- Must map actual parameters to formal parameters (c→x, b→y)
- Must create storage for local variables (and, maybe, parameters)
- p needs space for d (and, maybe, a, b, & c)
- Where does this space go in recursive invocations?
The Procedure as a Name Space

Each procedure creates its own name space
- Any name (almost) can be declared locally
- Local names hide identical non-local names (shadowing)
- Local names cannot be seen outside the procedure
- We call this set of rules & conventions **lexical scoping**

Examples
- C has global, static, local, and block scopes
- Blocks can be nested, procedures cannot
- Scheme has global, procedure-wide, and nested scopes
- Procedure scope (typically) contains formal parameters

Examples
- In C++ and Java

```c
{ 
  for (int i=0; i < 100; i++) { 
    ... 
  }
  for (Iterator i=list.iterator(); i.hasNext();)
    ... 
}

This is actually useful!
```

Lexically-scoped Symbol Tables

- The problem
  - Compiler needs a distinct entry for each declaration
  - Nested lexical scopes admit duplicate declarations
- The interface
  - `enter()` – enter a new scope level
  - `insert(name)` – creates entry for name in current scope
  - `lookup(name)` – lookup a name, return an entry
  - `exit()` – leave scope, remove all names declared there

Dynamic vs static

- Static scoping
  - Most compiled languages – C, C++, Java, Fortran
  - Scopes only exist at compile-time
  - We’ll see the corresponding run-time structures that are used to establish addressability later.
- Dynamic scoping
  - Interpreted languages – Perl, Common Lisp

```c
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

Example

```c
procedure p() 
  int x, y, z
  procedure q() 
    int v, b, x, w
    procedure r() 
      int x, y, z
      ... 
    
    procedure s() 
      int x, a, v
      ...
      ... q ...
    ...
  ...
L0: 
L1: 
L2a: 
L2b: 
```

Solution: **Lexically scoped symbol tables**
Chained implementation

- Create a new table for each scope, chain them together for lookup

Sheaf of tables' implementation
- `enter()` creates a new table
- `insert()` adds at current level
- `lookup()` walks chain of tables & returns first occurrence of name
- `exit()` throws away table for level `p`, if it is top table in the chain

Individual tables can be hash tables.

Stack implementation

Implementation
- `enter()` puts a marker in stack
- `insert()` inserts at `nextFree`
- `lookup()` searches linearly from `nextFree`-1 forward
- `exit()` sets `nextFree` back to the previous marker.

Advantage
- Uses less space
Disadvantage
- Lookups can be expensive

Threaded stack implementation

Implementation
- `insert()` puts new entry at the head of the list for the name
- `lookup()` goes direct to location
- `exit()` processes each element in level being deleted to remove from head of list

Advantage
- Lookup is fast
Disadvantage
- exit takes time proportional to number of declared variables in level

Symbol tables in C

Identifiers
- Mapping from names to declarations
- Fully nested – each `}` opens new scope
Labels
- Mapping from names to labels (for goto)
- Flat table – one set of labels for each procedure
Tags
- Mapping from names to struct definitions
- Fully nested
Externals
- Record of extern declarations
- Flat table – redundant extern declarations must be identical

In general, rules can be very subtle

Examples

- Example of typedef use:
  ```c
  typedef int T;
  struct S { T T; };
  ```
  /* redefinition of T as member name */

- Example of proper declaration binding:
  ```c
  int; /* syntax error: vacuous declaration */
  struct S; /* no error: tag is defined or elaborated */
  ```

- Example of declaration name spaces
  - Declare "a" in the name space before parsing initializer
    ```c
    int a = sizeof(a);
    ```
  - Declare "b" with a type before parsing "c"
    ```c
    int b, c[sizeof(b)];
    ```

Next time…

- Midterm review
- More on language abstractions
  - Methods (vs procedures)
  - Object-oriented programs