Announcements

- HW10 due November 27
- Rainbow grades:
  - HW 1-8
  - Exam 1-2
  - Quiz 1-6
  - If you have any questions/concerns, let us know ASAP
- Now grading: HW9 and Quiz 7

Last Class

- Type equivalence
- Type in C
- Primitive types
- Composite types
  - Records (Structs)
  - Variants (Unions)
  - Arrays
  - Pointers

Today’s Lecture Outline

- Control Abstraction
- Parameter Passing Mechanisms
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name
  - Call by sharing

Control Abstraction and Parameter Passing

Read: Scott, Chapter 9.1-9.3
(lecture notes cover mostly 9.3)

Abstraction

- Abstraction: hiding unnecessary low-level detail
- Data abstraction: types
  - Type integer is an abstraction
  - Type struct Person is an abstraction
- Control abstraction: subroutines
  - A subroutine abstracts away an algorithm
  - A subroutine provides an interface: name, argument types, return type: e.g., int binarySearch(int a[], int v)
- Classes/objects in OO, Abstract Data Types (ADTs) are a higher level of abstraction

Subroutines

- Other terms: procedures and functions
- Modularize program structure
  - Argument: information passed from the caller to the callee (also called actual parameter or actual argument)
  - Parameter: local variable in the callee, whose value is received from the caller (also called formal parameter)
Parameter Passing Mechanisms

- How does the caller pass information to the callee?
  - Call by value
    - C, Pascal, Ada, Algol68
  - Call by reference
    - Fortran, C++, Pascal var params, sometimes Cobol
  - Call by value-result (copy-in/copy-out)
  - Ada
  - Call by name (outmoded)
    - Algol60

Discussion applies to value model for variables

Parameter Passing Modes

- Most languages use a single parameter passing rule
  - E.g., Fortran, C

- Other languages allow different modes, in other words, programmer can choose different parameter passing rules in different contexts
  - E.g., C++ has two parameter passing mechanisms, Pascal too

Call by Value

- Value of argument is copied into parameter location

```
    m, n : integer;
    procedure R(k, j : integer)
    begin
        k := k+1;
        j := j+2;
    end R;
    ...
    m := 5;
    n := 3;
    R(m, n);
    write m, n;
```

By Value:

```
    k   j
    6   5
```

Output:

```
    5   3
```

Call by Reference

- Argument is an l-value; l-value is passed to the parameter

```
    m, n : integer;
    procedure R(k, j : integer)
    begin
        k := k+1;
        j := j+2;
    end R;
    ...
    m := 5;
    n := 3;
    R(m, n);
    write m, n;
```

```
    k,m   j,n
    6   5
```

Output:

```
    6   5
```

Call by Value vs. Call by Reference

- Call by value
  - Advantage: safe
  - Disadvantage: inefficient

- Call by reference
  - Advantage: more efficient
  - Disadvantage: potentially unsafe due to aliasing

  Aliasing (memory aliasing) occurs when two or more different names refer to the same memory location
  - E.g., m in main, and k in R are aliases for the same memory location during the call to R

Aliasing: Call by Reference

```
y : integer;
procedure P(x : integer)
begin
    x := x + 1;
    x := x + y;
end P;
...
y := 2;
P(y);
write y;
```

During the call, x and y are two different names for the same location!
No Aliasing: Call by Value

```
y: integer;
procedure P(x: integer)
begin
  x := x + 1;
  x := x + y;
end P;
...
y := 2;
P(y);
write y;
```

Output: 2

More Aliasing with Call by Reference

```
j,k,m: integer;
procedure Q(a,b: integer)
begin
  b := 3;
  a := m * a;
end Q;
...
s1: Q(m, k);
...
s2: Q(j, j);
...```

Questions

- **Aliasing** is an important concept in programming
- Can you think of other examples of aliasing?
- Why memory aliasing is considered dangerous?
- Can you think of other ways for creating memory aliasing?

Memory Aliasing is Dangerous

- One part of the program can modify a location through one alias, breaking invariants/expectations of other parts that use different aliases to the same location
- In general, we cannot know whether \( x \rightarrow f \) and \( y \rightarrow f \) are aliases to the same location
  - We “err” on the safe side
  - Aliasing makes reasoning about code hard
  - Aliasing prevents compiler optimization

Readonly Parameters

- What are some defenses against unwanted modification through aliases?
  - **const** parameters are an important paradigm in C/C++

```
log(const huge_struct * r) { ... } 
... log(my_huge_struct);
```

Readonly Parameters

- **const** can be tricky...

```
log(const huge_struct * r) {
  r->f = 0; // NOT OK
}
```

**VS.**

```
log(huge_struct * const r) {
  r->f = 0; // OK
}
```
Readonly Parameters

```cpp
class C {
    int f;
    public:
        int get() const
            { return f; }
        int set(int g)
            { f = g; }
};
```

More on Call by Reference

- What happens when someone uses an expression argument for a call-by-reference parameter?
  - $(2\times x)$?

Exercise

- Write a program that produces different result when the parameter passing mechanism is call by reference and when it is call by value-result.

Call by Value-Result

- Argument is copied in into the parameter at entry, parameter is copied out into the argument at exit.

```cpp
m,n : integer;
procedure R(k,j : integer)
begin
    k := k+1;
    j := j+2;
end R;
```

```cpp
m := 5;
n := 3;
R(m,n);
write m,n;
```

Call by Value-Result

```cpp
c : array [1..10] of integer;
m,n : integer;
procedure R(k,j : integer)
begin
    k := k+1;
    j := j+2;
end R;
```

```cpp
/* set c[i] = i */
m := 2;
R(m, c[m]);
write c[1], c[2], ..., c[10];
```

Call by Name

- An expression argument is not evaluated at call. It is evaluated within the callee, if needed.

```cpp
c : array [1..10] of integer;
m : integer;
procedure R(k,j : integer)
begin
    k := k+1;
    j := j+2;
end R;
```

```cpp
/* set c[i] to i */
m := 2;
R(m, c[m]);
write m,c[m];
```
Call by Name

- Call by name (Algol 60)
  - Case 1: Argument is a variable
    - Same as call by reference
  - Case 2: Argument is an expression
    - E.g., expressions c[m], f(x,y), x+z, etc.
    - Evaluation of the argument is deferred until needed
    - Argument is evaluated in the caller’s environment – the expression goes with a THUNK (a closure!) which carries the necessary environment
    - Generally inefficient
    - Difficult to implement

Call by Name vs. Call by Value

- Recall reduction strategies in the λ-calculus
  - What reduction strategy corresponds to call by name?
    - Normal order reduction
  - What reduction strategy corresponds to call by value?
    - Applicative order reduction

Reference Model for Variables

- So far, discussion applied to the value model for variables
- What is the parameter passing mechanism in languages that use the reference model for variables? Neither call by value, nor call by reference make sense for languages with the reference model
  - Call by sharing: argument reference (address) is copied into parameter. Argument and parameter references refer to the same object

Reference Model for Variables

- How does call by sharing relate to call by value?
  - Similarities?
  - Differences?
- How does call by sharing relate to call by reference?
  - Similarities?
  - Differences?

Immutability

- Immutability is a “defense” against unwanted mutation due to sharing
- In Scheme, methods are pure
- In Python, there are immutable datatypes
- In Java, not much... There is no const-like construct to protect the referenced object
  - final disallows re-assignment of a variable

```
final Point p = new Point();
p = q; // NOT OK
p.x = 0; r.y = 0; // ALL OK
```

Immutability

- Software engineering principles that help protect against unwanted mutation due to “sharing”
  - Avoid representation exposure (rep exposure)
  - Design immutable ADTs
  - Write specifications that emphasize immutable parameters
    - E.g., modifies: none
Exercise

- Construct a program which prints different result when parameter passing mechanism is
  - Call by value
  - Call by reference
  - Call by value-result
  - Call by name

Object-Oriented Programming Languages

Read: Scott, Chapter
10.1-10.4

Lecture Outline

- Object-oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding
- Polymorphism

Benefits of Object Orientation

- Abstraction
  - Classes bridge the gap between concepts in the application domain and software
  - E.g., domain concept of Customer maps to class Customer
- Encapsulation
  - Classes provide interface but hide data
  - Easier to understand and use
  - Can be changed internally with minimal impact
- Reuse
  - Inheritance and composition provide mechanisms for reuse
- Extensibility

Encapsulation and Inheritance

- Access control modifiers – public, private, and others
  - What portion of the class is visible to users?
  - Public, protected or private visibility
  - Java: Has package as default; protected is slightly different from C++
  - C++: Has friend classes and functions
  - Smalltalk and Python: all members are public

- With inheritance
  - What control does the superclass have over its fields and methods?
    - There are different choices
  - C++: a subclass can restrict visibility of superclass members
  - C#: Java: a subclass can neither increase nor restrict visibility of superclass members

Initialization and Finalization

- Reference model for variables used in Java, Smalltalk, Python
  - Every variable is a reference to an object
    - Explicit object creation: foo b = new foo();
- Value model for variables used in C++, Modula-3, Ada-95
  - A variable can have a value that is an object
  - Object creation can be implicit: e.g. foo b;
- How are objects destroyed?
**Question**

- Consider the following code:

```cpp
A a;  // a is a local variable of type A
a.m();  // We call method m on a
```

What happens in C++?
What happens in Java?

**More on implicit creation in C++**

- C++ requires that an appropriate constructor is called for every object implicitly created on the stack, e.g., `A a;`
- What happens here: `foo a;`
  - Compiler calls zero-argument constructor `foo::foo()`
- What happens here: `foo a(10, 'x');`
  - Calls `foo::foo(int, char)`

**More on implicit creation in C++**

- What happens here:
  ```cpp
  foo a;
  foo c = a;
  ```
  - Calls `foo::foo()` at `foo a;` calls copy constructor `foo::foo(foo&)` at `foo c = a;`
  - `=` operator here stands for initialization, not assignment!

**More on implicit creation in C++**

- What happens here:
  ```cpp
  foo a, c;  // declaration
  c = a;    // assignment
  ```
  - Calls `foo::foo()` twice at `foo a, c;` calls assignment operator `foo::operator=(foo&)` at `c = a;`
  - `=` operator here stands for assignment!

**Lecture Outline**

- Object Oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding
- Polymorphism

**Subtyping and Dynamic Method Binding**

- Subtyping and subtype polymorphism – the ability to use a subclass where a superclass is expected
  - Thus, dynamic method binding
  - Advantages?
  - Disadvantages?
  - C++: static binding is default, dynamic binding is specified with keyword `virtual`
  - Java: dynamic binding is default, static binding is specified with `final`
Benefits of Subtype Polymorphism

- Covered extensively in Principles of Software
- Enables extensibility and reuse
  - E.g., we can extend a type hierarchy with no modification to the client of hierarchy
  - Reuse through inheritance or composition
- Subtype polymorphism enables the Open/closed principle (credited to Bertrand Meyer)
  - Software entities (classes, modules) should be open for extension but closed for modification

“Science” of software design teaches Design Patterns

- Design patterns promote design for extensibility and reuse
- Nearly all design patterns make use of subtype polymorphism!

Benefits of Subtype Polymorphism

Lecture Outline

- Object-oriented programming
- Encapsulation and inheritance
- Initialization and finalization
- Subtyping and dynamic method binding
- Polymorphism

Polymorphism, more generally

- Subtype polymorphism
  - What we just discussed... Code can use a subclass $B$ where a superclass $A$ is expected
  - Standard in object-oriented languages
- Parametric polymorphism
  - Code takes a type as parameter
  - Explicit parametric polymorphism
  - Implicit parametric polymorphism
  - Standard in functional programming languages
- Ad-hoc polymorphism (overloading)

Explicit Parametric Polymorphism

- Occurs in Ada, Clu, C++, Java, Haskell (type classes)
- There is an explicit type parameter
- Explicit parametric polymorphism is also known as genericity
- E.g. in C++:
  ```
  template<class V>
  class list_node {
    list_node<V>* prev;
    list_node<V> header;
  }
  ```

Explicit Parametric Polymorphism

- Usually (but not always!) implemented by creating multiple copies of the generic code, one for each concrete type
  ```
  typedef list_node<int> int_list_node;
  typedef list<int> int_list;
  ```
- Object-oriented languages usually provide both subtype polymorphism and explicit parametric polymorphism, which is referred to as generics
Explicit Parametric Polymorphism

- Generics are tricky...
- Consider this C++ code (uses the STL):
  ```cpp
  list<int> l;
  sort(l.begin(), l.end());
  ```
- Compiler produces around 2K of text of error messages, referring to code in the STL
- The problem here is that the STL's `sort` requires a `RandomAccessIterator`, while the `list` container provides only a `Bidirectional Iterator`.

In Java, Bounded Types Restrict Instantiations by Client

- Generic code can perform operations permitted by the bound
  ```java
  class MyList1<E extends Object> {
    void m(E arg) {
      arg.intValue(); //compile-time error: Object does not have intValue()
    }
  }
  ```
  ```java
  class MyList2<E extends Number> {
    void m(E arg) {
      arg.intValue(); //OK. Number has intValue()
    }
  }
  ```

In Haskell, Type Predicates Restrict Instantiation of Generic Functions

- `sum :: (Num a) => a -> List a -> a` 
- `sum n Nil = n`
- `sum n (Cons x xs) = sum (n+x) xs`
- `a` is an explicit type parameter
- `(Num a)` is a predicate in type definition
- `(Num a)` constrains the types we can instantiate a generic function with

Implicit Parametric Polymorphism

- Occurs in Scheme, Python and others
- There is no explicit type parameter, yet the code works on many different types!
- Usually, there is a single copy of the code, and all type checking is delayed until runtime
  - If the arguments are of type as expected by the code, code works
  - If not, code issues a type error at runtime

Implicit Parametric Polymorphism

- `twice in Scheme: (define (twice f x) (f (f x)))`
- `(twice (lambda (x) (+ 1 x)) 1) yields ?`
- `(twice (lambda (x) (cons 'a x)) '(b c)) yields ?`
- `(twice 2 3) yields ?`
- `map, foldl, length are all implicitly parametric`

Implicit Parametric Polymorphism

- `def intersect(seq1, seq2):
  res = []
  for x in seq1:
    if x in seq2:
      res.append(x)
  return res`
- As long as arguments for seq1 and seq2 are of iterable type, intersect works
Happy Thanksgiving!