ECE326
PROGRAMMING LANGUAGES

Lecture 19 : Variance and Data Types

Kuei (Jack) Sun
ECE
University of Toronto
Fall 2019

Enumerated Type

- A data type consisting of named values
- Each named value behaves as a constant
- A variable of an enumerated type can be assigned any of the named value

```
enum Suit {
    CLUB,
    DIAMOND,
    HEART,
    SPADE,
};
```

```
Suit suit = SPADE;

// can be compared
if (suit > HEART) {
    win();
}
```

Enumerated Type

- In principle, their in-memory representation should not be exposed by the programming language
- In practice, C++ treats them like integers (mostly)

```
int a = CLUB;
Suit b = HEART;

cout << a << " " << b << endl;   // prints 0 2

b = (Suit)5;       // this is OK in C++

Suit * sp = &b;
int * ip = sp;    // this is not OK without a cast
```

C Enum

- Named values can be assigned an integer expression
- The type's size can be altered via base specifier
  - If you prefer your enum to take up less memory

```
enum Suit : char {
    CLUB = 1,

    // 2, always previous value + 1 if not specified
    DIAMOND,
    HEART = 5,

    // this is very bad but is allowed: DIAMOND == SPADE
    SPADE = CLUB + 1,
};
```

Enum Class

- A more type-safe version of C Enum
- Disallows coercion to integer and other enums
- Named values are also “scoped”

```
enum class Suit {
    CLUB,
    DIAMOND,
    HEART,
    SPADE = 5,
};

// requires name resolution
Suit suit = Suit::SPADE;
```

```
// no longer allowed
int a = Suit::CLUB;

enum Foo { A = 1 };
enum Bar { B = 1 };

// allowed
bool b = A == B;

// not allowed
b = A == Suit::DIAMOND;
```

Union

- Stores different data types in same memory location
- Type-unsafe construct, but can save memory
- Can be used in C to implement private members

```
// size of union is always the largest member
// in this case it's the double (8 bytes)
union Mangle {
    int i;
    unsigned char c;
    double d;
};

Mangle m = { .i = 0xffff };
cout << (unsigned)m.c << endl;  // prints 255
```

```
union Private {
    const int readonly;
    int __writeable;
};
```

Union Cast

- Used to bypass strict aliasing rules
- C/C++ does not type check other union members
- Used to see memory representation of other types

```
union Pun {
    double d;
    unsigned char c[sizeof(double)];
};

Pun p = { .d = 1.0 };
for (unsigned i = 0; i < sizeof(double); i++)
    cout << (unsigned)p.c[i] << “ ”;

0 0 0 0 0 0 240 63
```

Tagged Union

- A union that has a tag field to indicate which union member is being used
- Also known as variant type, or algebraic type

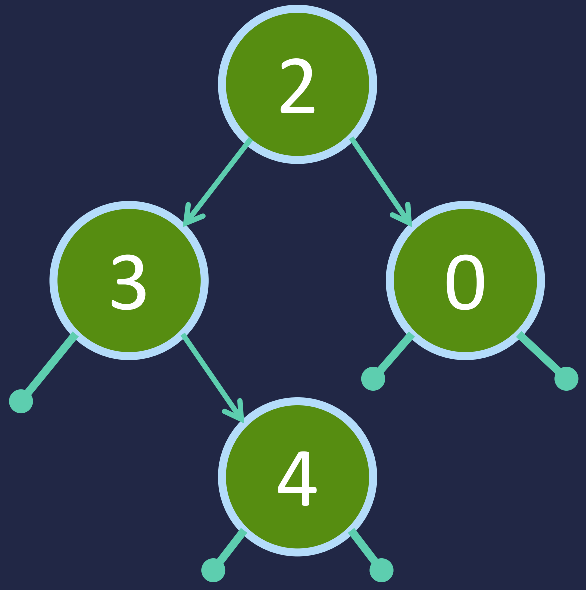
```
struct Tagged {
	enum { INT, STR } tag;
	union {                     // anonymous union (members
		int * i;                // can enter parent scope)
		string * s;
	};
	Tagged(int i) : tag(INT), i(new int(i)) {}
	Tagged(const char * s) : tag(STR), s(new string(s)) {}
	~Tagged() {
		if (tag == INT) delete i; else delete s;
	}
};
```

Rust Enums

- Natively supports tagged union by mixing it with enum

```
let tree = Tree::Node(2,
        Box::new(Tree::Node(0,
                Box::new(Tree::Leaf),
                Box::new(Tree::Leaf),
     )),
        Box::new(Tree::Node(3,
                Box::new(Tree::Leaf),
                Box::new(Tree::Node(4,
                        Box::new(Tree::Leaf),
                        Box::new(Tree::Leaf),
                )),
        )),
);
```

```
enum Tree {
  Leaf,
  Node(i64,
        Box<Tree>,
        Box<Tree>)
}
```

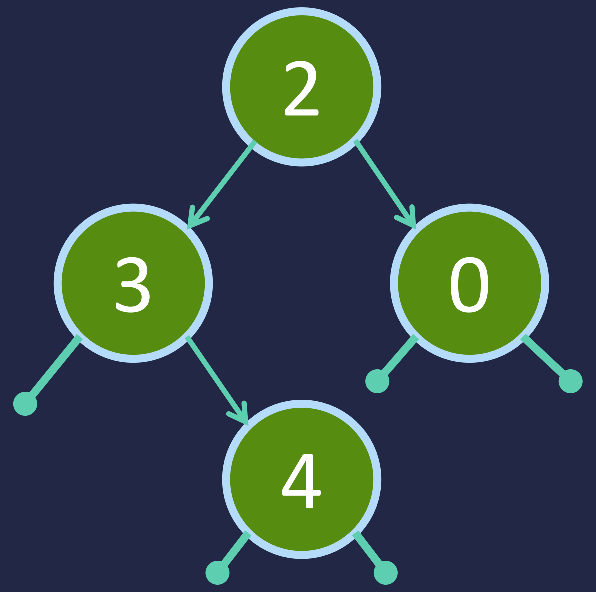


Rust Enums

```
enum Tree {
  Leaf,
  Node(i64, Box<Tree>, Box<Tree>)
}

fn add_values(tree: Tree) -> i64 {
    match tree {
        Tree::Node(v, a, b) => {
            v + add_values(*a) +
            add_values(*b)
        },
        Tree::Leaf => 0
    }
}

assert_eq!(add_values(tree), 9);
```



You do not need to say **return** in Rust. Just leave an expression by the end of a function. In the case of multiple branches, all branches must return the same type.

Subtype Compatibility

and Contract Programming

Variance

- Compatibility of types and their subtypes
- Substitutability
  - If S is a subtype of T, then objects of type T may be replaced with objects of type S without altering program correctness
- Complication arises with dealing with complex types
- Refers to how subtyping relation changes
  - E.g. when used as a parameter in a virtual function
  - E.g. when used as a return type in a virtual function
  - E.g. when used as a parameterized type

Variance

- Suppose Apple is a subtype of Fruit
  - Is a list of apple a subtype of list of fruit?

```
template<typename T>
struct List<T> {
    T * get(unsigned i);
    void add(T *);
    unsigned count() const;
};
```

```
struct Fruit {
    int calories() const=0;
    float weight() const=0;
    float sweetness() const=0;
};
```

```
int total_calories(const List<Fruit> & basket) {
      int total = 0;
      for (int i = 0; i < basket->count(); i++)
            total += basket->get(i)->calories();
      return total;
}
```

Covariance

- Allows use of more derived type than specified
  - Ordering of types is preserved
    - type can be substituted by subtype

```
int total_calories(const List<Fruit> & basket) {
      int total = 0;
      for (int i = 0; i < basket->count(); i++)
            total += basket->get(i)->calories();
      return total;
}

List<Apple> apples = { new Apple(“Gala”, 0.3),
                       new Apple(“Fuji”, 0.4), … };

// covariant, but NOT allowed in C++
cout << total_calories(apples) << endl;
```

Covariance

- For parameterized types, only safe if immutable

```
List<Apple> apples = { new Apple(“Gala”, 0.3),
                       new Apple(“Fuji”, 0.4), … };

List<Fruit> * fruits = &apples;

// THIS IS UNSAFE
fruits->add(new Orange(“Navel”, 0.5));
```

- However, Java allows the above to occur
  - Will cause a runtime java.lang.ArrayStoreException
    - Unless List<Fruit> is actually implemented as a list of assorted fruits

Contravariance

- Allows use of more generic type than specified
  - Ordering of types is *reversed*

```
struct Canine {
    virtual void eat(Meat *)=0;
};

struct Wolf : public Canine {
   …
};

struct Dog : public Canine {
    virtual void eat(Food *) override;
}
```

```
struct Food {};
struct Meat : public Food {};
struct Fruit : public Food {};
struct Apple : public Fruit {};
```

`eat(Food *)` is a subtype of `eat(Meat *)`. In other words, eat is contravariant because the parameter of its subtype is more generic.

```
Dog fido;   Apple * apple = new Apple(“Honeycrisp”, 0.33);
fido.eat(apple);   // contravariant, but NOT allowed in C++
```

Contravariant Parameter

```
struct Canine {
    virtual void eat(Meat *)=0;
};
```

```
struct Food {};
struct Meat : public Food {};
struct Fruit : public Food {};
struct Apple : public Fruit {};
```

```
struct Dog : public Canine {
    virtual void eat(Food *) override;
}

Dog fido;   Apple * apple = new Apple(“Honeycrisp”, 0.33);
fido.eat(apple);   // this is type safe, a dog can eat any food

Canine * canine = &fido;      // now fido is a canine

// still type safe, fido can only eat meat as a canine,
// but meat is also food so a dog can eat it
canine->eat(new Meat(“Beef”, “10oz”));
```

Covariant Parameter

```
struct Canine {
    virtual void eat(Food *)=0;
};
```

```
struct Food {};
struct Meat : public Food {};
struct Fruit : public Food {};
struct Apple : public Fruit {};
```

```
struct Wolf : public Canine {
    virtual void eat(Meat *) override;
}

Wolf kevin;
// this is type safe, a wolf eats meat only
kevin.eat(new Meat(“Venison”, “8oz”);

Canine * canine = &kevin;     // now kevin is a canine

// NOT TYPE SAFE, wolf cannot eat anything but meat!
canine->eat(new Apple(“Red Delicious”, 0.5));
```

Invariance

- Nonvariant
- Only the specified type is accepted
- Most conservative, least flexible
- C++ allows covariant return types

```
struct AnimalShelter {
    virtual Animal * adopt();
};

struct CatShelter : public AnimalShelter {
    // OK – return type is a subtype of Animal
    virtual Cat * adopt() override;
}
```

Liskov's Substitution Principle

- Correctness of function subtyping is guaranteed if:
  - Method parameters are contravariant
  - Method return type is covariant
- Correctness of behavioural subtyping is guaranteed if:
  - Precondition
    - Condition that must be true before execution of some code
    - Cannot be strengthened in a subtype
  - Postcondition
    - Condition that must be true after execution of some code
    - Cannot be weakened in a subtype

Square Rectangle Problem

- Base class can incorrectly mutate its derive class
- Violation of Liskov's substitution principle
  - Square has a stronger precondition than Rectangle

```
class Rectangle {
      int width, height;
public:
      Rectangle(int w, int h) : width(w), height(h) {}
      void set_width(int w) { width = w; }
      void set_height(int h) { height = h; }
};

struct Square : public Rectangle {
      Square(int s) : Rectangle(s, s) {}
};

Square sq(5);      sq.set_width(6);     // oops
```

subclass cannot properly adhere to the interface of the interface

Example

- Weakened post-condition

```
class Shipment {
      double dimensions[3];
public:
      Shipment(double w, double h, double d);
      virtual double cost() const;
};

class DiscountShipment : public Shipment {
      double discount;
public:
      virtual double cost() const override;
};

Shipment * package = new DiscountShipment(…);

package->cost();    // -$10.00 !?!?
```

Contract Programming

- Language support for specifying precondition, postcondition, errors, and invariants
- Allows business logic to be written more “aggressively”
  - Without having to type check or verify assumptions
- Opposite of “defensive programming”
- Fills the gap that type checking cannot accomplish
- Some can be done statically
- Most are done at runtime

Assertion

- Verifies that an expression is true at runtime

```
assert(i > 0);      // crashes program if expression not true
assert(j < 0, “j must be negative”); // supported by D language
```

- Static version available

```
// syntax in D programming language
static assert(Message.sizeof <= 1024,
    “size of message exceeds maximum packet size”);

// in C++
static_assert(sizeof(Message) <= 1024);
```

Contracts

- Defining precondition and postcondition

```
int fun(ref int a, int b)
in {
    assert(a > 0);
    assert(b >= 0, "b cannot be negative!");
}
out (r) {   // r binds to the return value of fun
    assert(r > 0, "return must be positive");
    assert(a != 0);
}
do {
    a = (b – 3)*(b + 2) + 1;
    return b*b;
}
```

Invariants

- Characteristics of a class that must always be true
  - Except in methods, when temporary violation can occur

```
class Date {
    int day, hour;

    this(int d, int h) {
        day = d; hour = h;
    }

    void add_hours(int h) {
        hour += h;
        if (hour >= 24) {          // does not trip the invariant!
            d += 1; hour -= 24;
        }
    }

    invariant {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24, "hour out of bounds");
    }
}
```