Run-Time Polymorphism

CS 211/Spring 2020
Definition

polymorphism, n. (from poly- + -morphism)

1. The ability to assume different forms or shapes.
2. (biology) The coexistence, in the same locality, of two or more distinct forms independent of sex, not connected by intermediate gradations, ...
3. (object-oriented programming) The feature pertaining to the dynamic treatment of data elements based on their type, allowing for an instance of a method to have several definitions.
4. (mathematics, type theory) The property of certain typed formal systems of allowing for the use of type variables and binders/quantifiers over those type variables; ...
5. (crystallography) ...
6. (genetics) ...
Parametric polymorphism (in OCaml)

```ocaml
let reverse xs0 = 
  let rec loop acc xs = 
    match xs with 
    | []    -> acc 
    | x :: xs' -> loop (x :: acc) xs' 
  in loop [] xs0
```
ML stands for meta-language

```ocaml
let reverse xs0 = 
  let rec loop acc xs = 
    match xs with 
    | []       -> acc 
    | x :: xs'  -> loop (x :: acc) xs'
  in loop [] xs0

OCaml infers a polymorphic type:
  reverse : 'a list -> 'a list
```

In C++ syntax:
```cpp
template <class T> List<T> reverse(List<T>);
```
ML stands for meta-language

```ocaml
let reverse xs0 =  
  let rec loop acc xs =  
    match xs with  
    | [] -> acc  
    | x :: xs' -> loop (x :: acc) xs'  
  in loop [] xs0
```

OCaml infers a polymorphic type:

```
reverse : 'a list -> 'a list
```

In C++ syntax:

```cpp
template<class T>
List<T> reverse(List<T>);
```
Ad-hoc polymorphism

Also known as overloading:

```cpp
bool test(int v, int lo, int hi)
{
    return lo <= v && v < hi;
}

bool test(double v, double lo, double hi)
{
    return low <= v && v <= hi;
}
```
Ad-hoc polymorphism

Also known as overloading:

```cpp
bool test(int v, int lo, int hi)
{
    return lo <= v && v < hi;
}

bool test(double v, double lo, double hi)
{
    return low <= v && v <= hi;
}
```

Overloading is dispatched statically.
Generic = parametric + ad-hoc

template<class T>
void filter(std::vector<T>& v, T lo, T hi)
{
    size_t dst = 0;

    for (T x : v)
        if (test(x, lo, hi))
            v[dst++] = x;

    v.resize(v.size() - dst);
}
Message/method polymorphism

Number subclass: Complex [
  | realpart imagpart |

"constructor and setter omitted..."

real [ ^realpart ]
imag [ ^imagpart ]

+ other [ ^Complex real: (realpart + other real)
               imag: (imagpart + other imag) ]

"etc..." ]
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$. 

Possible examples:

• int is-a double?
• Rectangle is-a Shape?
• Square is-a Rectangle?
• vector<Rectangle> is-a vector<Shape>?
• bool(*)(Shape) is-a bool(*)(Rectangle)?
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

• int is-a double?
• Rectangle is-a Shape?
• Square is-a Rectangle?
• vector<Rectangle> is-a vector<Shape>?
• bool(*)(Shape) is-a bool(*)(Rectangle)?
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- `int` is-a `double`?
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
Subtype polymorphism in theory

A type $\tau$ is a **subtype** of a type $\sigma$ (notation: $\tau \text{ is-a } \sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the *Liskov Substitution Principle*. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer **is-a** Real
- Rectangle **is-a** Shape?
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the *Liskov Substitution Principle*. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau \text{ is-a } \sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
- bool (*)(Shape) is-a bool (*)(Rectangle)?
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the *Liskov Substitution Principle*. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples:

- Integer is-a Real
- Rectangle is-a Shape
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
- bool (*)(Rectangle) is-a bool (*)(Shape)
Subtype polymorphism in theory

A type $\tau$ is a subtype of a type $\sigma$ (notation: $\tau$ is-a $\sigma$) iff every value of type $\tau$ is also a value of type $\sigma$.

(This is known as the Liskov Substitution Principle. Restated: A function that accepts an object of type $\sigma$ must work on objects of type $\tau$.)

Possible examples (but in C++, indirection is required):

- Integer& is-a Real&
- Rectangle& is-a Shape&
- Square is-a Rectangle
- vector<Rectangle> is-a vector<Shape>
- bool (*)(Rectangle&) is-a bool (*)(Shape&)

7
Subtype polymorphism in C++

```cpp
struct Base {
    int x;
};

struct Derived : Base {
    int y;
};
```
Subtype polymorphism in C++

```cpp
struct Base {
    int x;
};

struct Derived : Base {
    int y;
};

Then:

- Derived* is-a Basic*,
- Derived& is-a Base&, and
- and likewise for const versions, but
- Derived is-a Base – why not?
Adding “methods”

```cpp
struct Base
{
    int f() { return 0; }
};

struct Derived : Base
{
    int f() { return 1; }
};
```
Adding “methods”

```cpp
struct Base {
  int f() { return 0; }
};

struct Derived : Base {
  int f() { return 1; }
};

TEST_CASE("direct") {
  Base b;
  Derived d;
  CHECK( b.f() == 0 );
  CHECK( d.f() == 1 );
}
```
Adding “methods”

```cpp
struct Base {
    int f() { return 0; }
};

struct Derived : Base {
    int f() { return 1; }
};

int g(Base& b) { return b.f(); }

TEST_CASE("via reference")
{
    Base b;
    Derived d;
    CHECK( g(b) == 0 );
    CHECK( g(d) == 0 ); // ???
}
```
Static versus dynamic dispatch

To determine which function to call:

- Static dispatch uses the static type of the variable
- Dynamic dispatch uses the run-time class of the object
Static versus dynamic dispatch

To determine which function to call:

- Static dispatch uses the static type of the variable
- Dynamic dispatch uses the run-time class of the object

To get dynamic dispatch in C++, a function must be `virtual`
Introducing virtual functions

struct Base
{
  virtual int f() { return 0; }
};

struct Derived : Base
{
  int f() override { return 1; }
};

test_case("via reference")
{
  Base b;
  Derived d;
  CHECK( g(b) == 0 );
  CHECK( g(d) == 1 );
}
Introducing virtual functions

struct Base
{ virtual int f() { return 0; } }

struct Derived : Base
{ int f() override { return 1; } }

int g(Base& b) { return b.f(); }

TEST_CASE("via reference")
{
    Base b;
    Derived d;
    CHECK( g(b) == 0 );
    CHECK( g(d) == 1 );
}