CSCI 334:
Principles of Programming Languages

Lecture 19: C++

Instructor: Dan Barowy

Williams

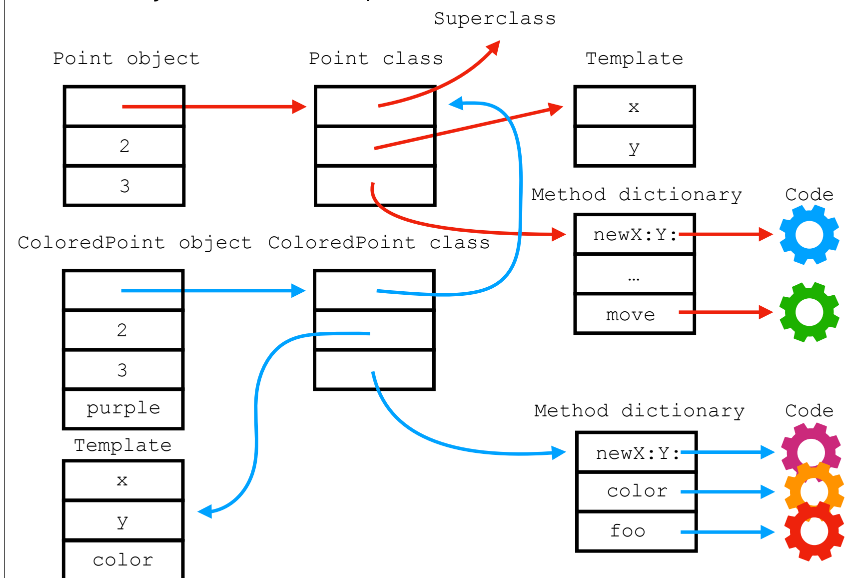
Announcements

HW8 pro tip: the HW7 solutions have a complete, correct implementation of the CPS version of bubble sort in SML. All you need to do is encode the same logic in C++.

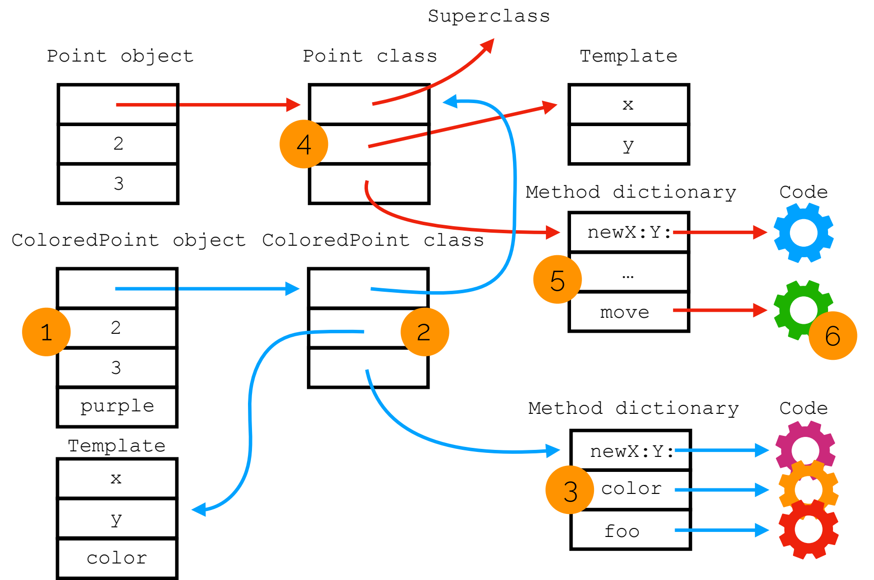
Announcements

I've decided to skip Java.

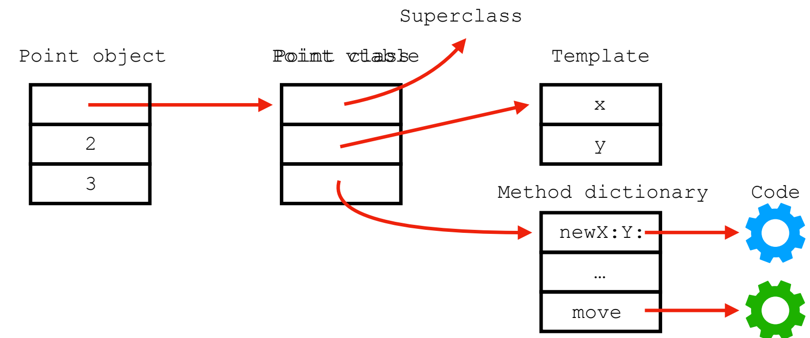
Dynamic Dispatch (ala Smalltalk)



Call `move` on ColoredPoint

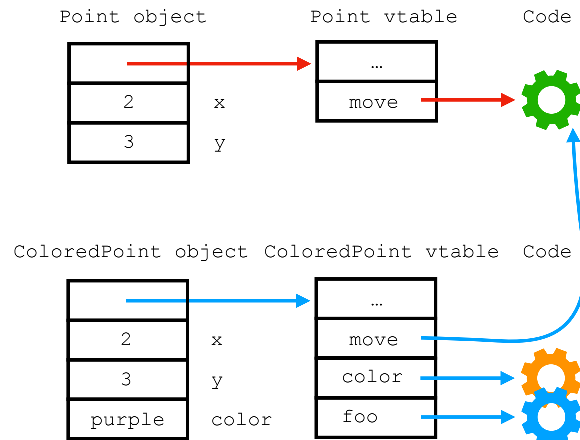


Optimize Dynamic Dispatch



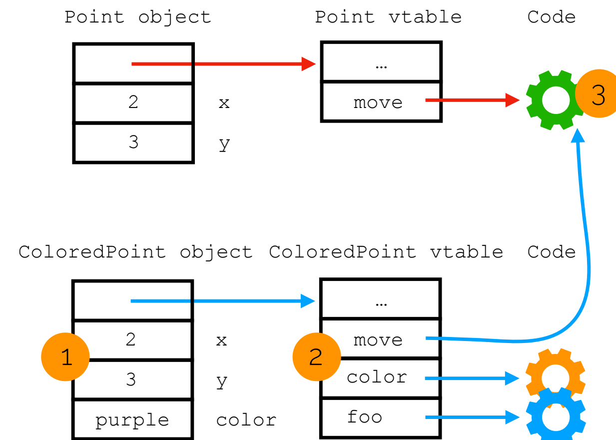
- Static types let us do some optimization.
- Sacrificing runtime polymorphism-by-default also allows optimization.
- Statically determine locations of `x` and `y`.
- Non-virtual method lookup determined statically.
- Copy virtual methods from superclasses into method dictionary.
- Eliminate class object; just a "virtual function table" now.

Optimize Dynamic Dispatch



Call `move` on ColoredPoint

C++ virtual dispatch *never searches* as in SmallTalk!



Runtime polymorphism

- You may have forgotten how OO polymorphism works.
- It's easy to forget with Smalltalk, which is dynamically typed.
- In Java—and especially C++—you need to think about this or it will bite you.

```
class Animal {
public:
  string myName() {
    return "Animal";
  }
};
```

```
class Human : public Animal {
public:
  string myName() {
    return "Human";
  }
};
```

```
Animal *a = new Animal();
cout << a->myName() << endl;

Human *h = new Human();
cout << h->myName() << endl;

a = h;
cout << a->myName() << endl;
```

- What will the last line print?

Runtime polymorphism

- You may have forgotten how OO polymorphism works.
- It's easy to forget with Smalltalk, which is dynamically typed.
- In Java—and especially C++—you need to think about this or it will bite you.

```
class Animal {
public:
  virtual string myName() {
    return "Animal";
  }
};
```

```
class Human : public Animal {
public:
  virtual string myName() {
    return "Human";
  }
};
```

```
Animal *a = new Animal();
cout << a->myName() << endl;

Human *h = new Human();
cout << h->myName() << endl;

a = h;
cout << a->myName() << endl;
```

- What will the last line print?

Inheritance vs Subtyping

```
class Pirate : public Person {
public:
  Pirate(string name);
  virtual void sayHello();
}
```

- Recall that inheritance and subtyping are not the same.
- Example: implement a stack using a dequeue.
- Inheritance of the form:
  `class <subclass> : <superclass>`
  is mere inheritance; the C++ compiler will not treat `<subclass>` as a subtype of `<superclass>`
- Inheritance of the form:
  `class <subclass> : public <superclass>`
  is inheritance with subtyping; the compiler will treat `<subclass>` as an instance of `<superclass>` when needed.

Initializer Lists

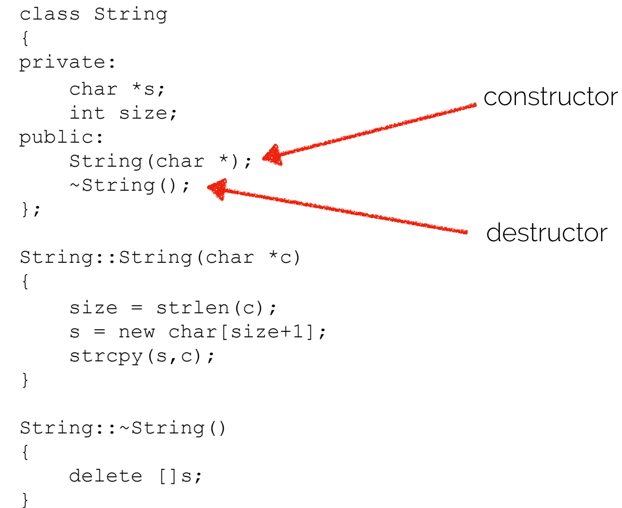
```
Pirate::Pirate(name) : Person(name) {}
```

- In C++, the base class constructor is called automatically for you for no-argument constructors.
- When calling a superclass constructor with an argument from a subclass, you must use initializer list syntax. This is different from Java.
- You can (and should) also use the initializer list to call constructors for instance variables if they need initialization.
- Initializer lists only work for instance variables that have constructors; primitives do not have constructors.
- For primitives, initialize the old-fashioned way (in constructor body).

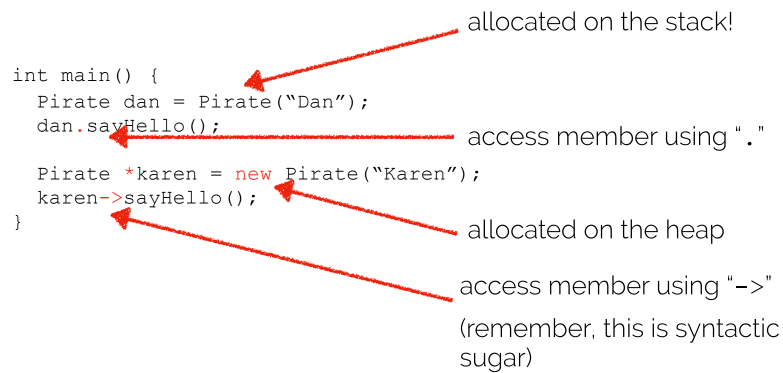
Manual Memory Management

- In C++, you need to think explicitly about allocation and deallocation, just as you do in C.
- While you can use `malloc` and `free` in C++, you should generally favor `new` and `delete` instead.
- `new` does more than `malloc`: it also calls the class constructor.
- `delete` does more than `free`: it also calls the class destructor.

Destructors



Automatic vs. Heap Allocation



Type Inference!

- C++ has a restricted form of type inference.

```
int main() {
  auto dan = Pirate("Dan");
  dan.sayHello();

  auto karen = new Pirate("Karen");
  karen->sayHello();
}
```

- `auto` makes life wonderful. Use it unless you are confused about inferred types (in which case, write the type manually)

Lambda expressions

- C++ has lambda expressions.
- They are a tad more verbose than in SML.
- Three main components.

[3] (1) { 2 }

1. Parameter list
2. Function body
3. Capture list

Lambda expressions

[3] (1) { 2 }

Let's rewrite this SML lambda expression in C++:

```
fn (x: int) => x + 1

[](int x){return x + 1;};
```

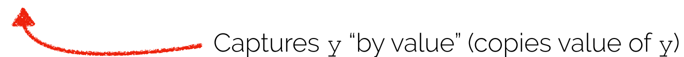
Lambda expressions

[3] (1) { 2 }

Let's rewrite this SML lambda expression in C++:

```
val y = 2
fn (x: int) => x + y

int y = 2;
[ y](int x){return x + y;};
```



Captures y "by value" (copies value of y)

Lambda expressions

[3] (1) { 2 }

Let's rewrite this SML lambda expression in C++:

```
val y = 2
fn (x: int) => x + y

int y = 2;
[&y](int x){return x + y;};
```

Captures y "by reference" (refers to y)

Lambda expressions

What is the type of a lambda expression?

```
[&y](int x){return x + y;};
```

This one takes an `int` and returns and `int`

```
std::function<int(int)>
```

More generally..

```
std::function<T(U1,…,Un)>
```

Lambda subtleties

Capture of closed-over lambda parameters is only necessary for variables with "automatic storage duration".

(demo)

Templates

- C++ lets you program "generically" just like Java or SML.
- Syntax is a little different.
- Mechanism is very different.

```
class Box {
public:
  int x;
}
…
Box b = new Box();
b->x = 2;
```

Templates

- No restriction on template parameter like Java (int vs Integer)
- Works by generating specialized code (literally, a new class) at compile-time for each parameter type used.

```
template <typename T>
class Box {
public:
  T x;
}
…
Box<double> b = new Box<double>();
b->x = 2.2;
```

Templated Lambdas

- You can even template lambda expressions.
- Here's a generic identity function.

```
template <typename T>
auto Identity = [](T x){return x;};
```

- Call it like:

```
Identity<string>("hello");
```

Typedef'd Templates

- You can even `typedef` templated types.
- Unfortunately, the `typedef` mechanism does not understand template parameters, which means that you have to fix the template parameter if you use it.
- C++0x introduced a generalization of `typedef` to address this called `using`.

```
std::function<bool(T,T)>
```

- Won't work; doesn't understand `T`:

```
template <typename T>
typedef Comparison std::function<bool(T,T)>;
```

- Will work:

```
template <typename T>
using Comparison = std::function<bool(T,T)>;
```

Next class

- Scala wrap-up
- Logic programming