Introduction to C, C++, and Unix/Linux

CS 60

Lecture 17: Inheritance

Today

→ C++ classes, inheritance

→ Reading [KR] Chapters 1-7

→ Read [So] chapters 1, 3, 4 (Boolean), 9, 13, 14 & 18.

Notes

- Questons?

Issue with default params

```
class Base {
  int x, y;
public:
  Base();
  Base(int x=0);
  ~Base();
};
```

This gives a compiler error

```
Base b1;
```

How does C++ know which constructor to call?

Need to either
- Leave out the default value of x
- Only use the second constructor

Defining operator +

```
class Thing {
private:
  int val;
public:
  Thing(int n) { val=n; }
  Thing operator+ (const int& i);
  Thing operator+ (const Thing& th);
  int GetVal() { return val; }
};
```

```
Thing c(a+b);
```

1. Evaluate a+b
2. Copy constructor

```
Thing Thing::operator+ (const int& i)
{
  Thing local(0);
  local.val = this->val + i;
  return local;
}
```

a+b ↔ a.+(b)

(analogous)

```
Thing Thing::operator+ (const Thing& th)
{
  Thing local(0);
  local.val = this->val + th.val;
  return local;
}
```

Note that we can access a private variable of an arg of type Thing!

Defining operator <<

- The left-shift operator << could be similarly defined
- What would it mean to define << for the Player class?
- General principle: If you define standard operators for a class (`<< >> = == ++ --` etc.) they should do "the expected thing"
- Exception, kind of: << and >> for standard I/O stream
  - `cout << "Hi\n";`
  - `cin >> x;`

`cout.<<("Hi\n");`
`cin.>>(x);`

Conceptually the same
(doesn't compile)

```
Thing Thing::operator << (const int& n)
{
  Thing local(0);
  local.val = this->val << n;
  return local;
}
```

```
Thing a(2);
Thing b = a<<3;
std::cout <<
  b.GetVal();
```

“16”

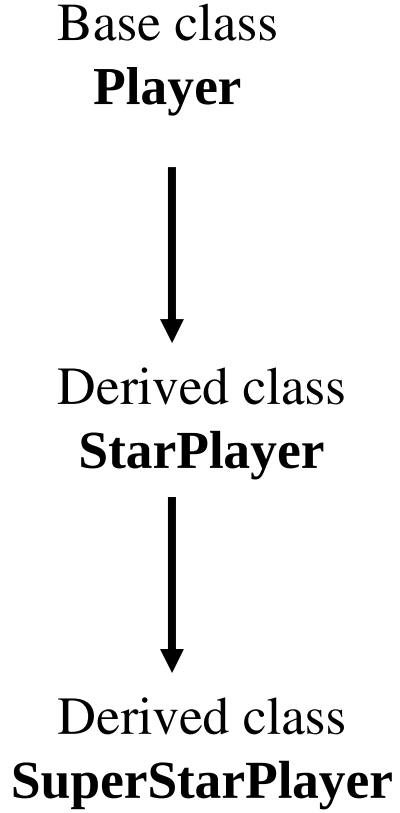
```
Thing Thing::operator >> (const int& n)
{
  Thing local(0);
  local.val = this->val >> n;
  return local;
}
```

Calling base constructors

- How can we call the Player constructor when creating a StarPlayer object?
- Define the StarPlayer constructor as:

```
StarPlayer::StarPlayer(char *name)
   : Player(name) { ... }

SuperStarPlayer::SuperStarPlayer
 (char *name) : StarPlayer(name)
 { ... }
```



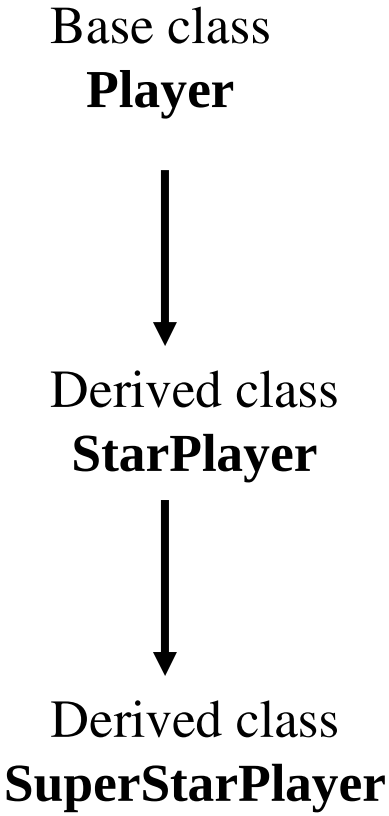
Functions on derived classes

```
Function1(Player p)

Function2(StarPlayer p)

Function3(SuperStarPlayer p)
```

Which one will work on *any* object of the base class or derived from the base class?



Functions on derived classes (cont.)

- We might like a function that takes a Player to be able to access the special overloaded functions of StarPlayer and SuperStarPlayer
  - E.g., **`Print()`**

- This brings up virtual functions….