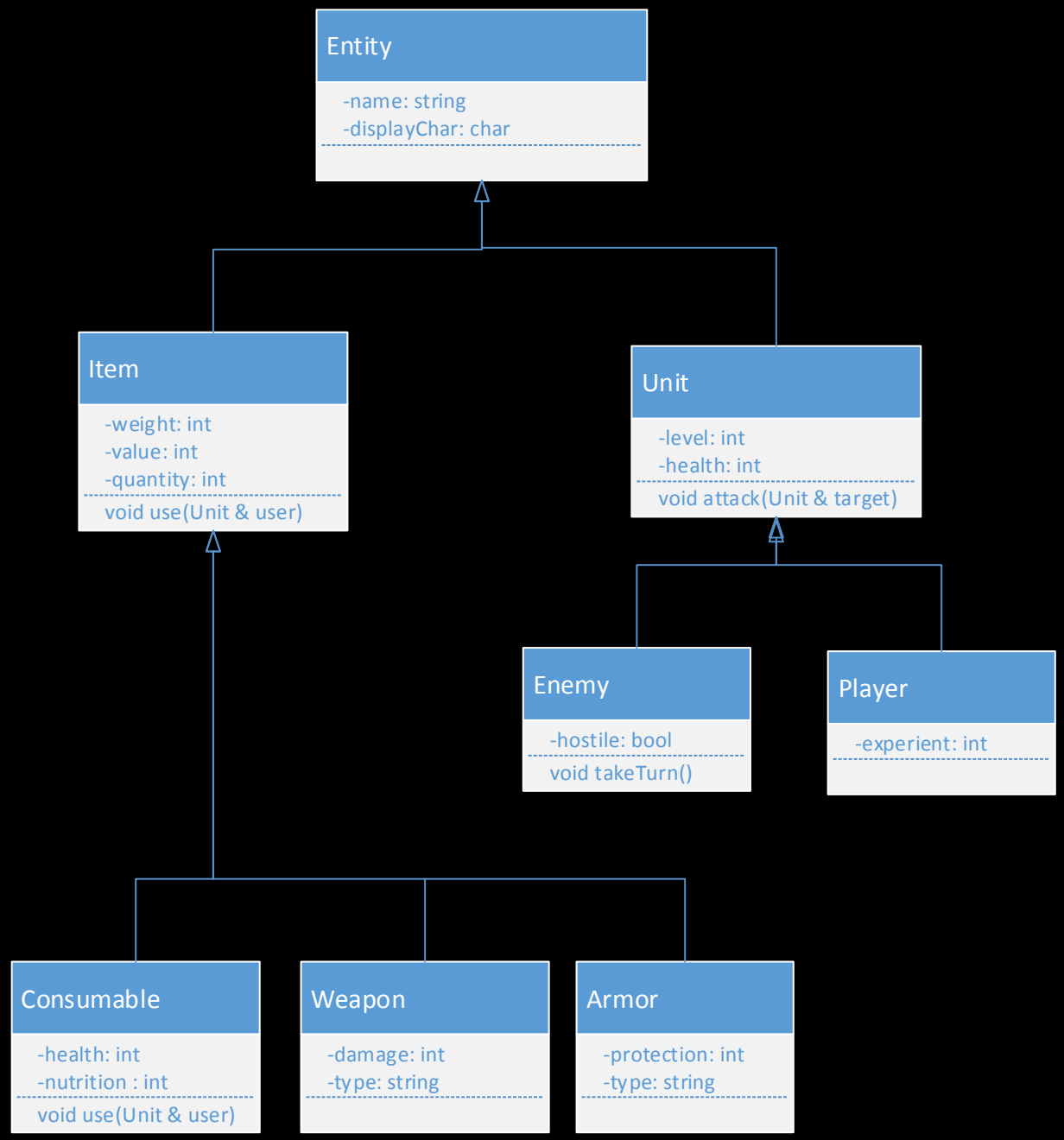
CS 216

Lecture 3

January 31st, 2013

Administrivia

Programming Assignment 1!


Entity
-name: string
-displayChar: char
Item
-weight: int
-value: int
-quantity: int
void use(Unit & user)
Unit
-level: int
-health: int
void attack(Unit & target)
Enemy
-hostile: bool
void takeTurn()
Player
-experient: int
Consumable
-health: int
-nutrition : int
void use(Unit & user)
Weapon
-damage: int
-type: string
Armor
-protection: int
-type: string

Split up into two parts:

Setting up the classes

(Due 2/9 at midnight)

XML serialization

(Due TBD)

Back to Encapsulation

```
class Critter
{
public:
    // stuff here
    int m_iConstitution;

    // more stuff down here
};
```

```
// Elsewhere in the code...
pCritter->m_iConstitution = 14;
```

```
class Critter
{
public:
    // stuff here
    void setConstitution(int iConstitution);

private:
    int m_iConstitution;
    // more stuff down here
};
```

```
// Somehwere in a .cpp file...
void Critter::setConstitution(int iConstitution)
{
    m_iConstitution = iConstitution;
}
```

```
// Elsewhere in the code...
pCritter->setConstitution(14);
```

Mechanically, encapsulation is accomplished with the private access declaration.

```cpp
// Somehwere in a .cpp file...
void Critter::setConstitution(int iConstitution)
{
    m_iConstitution = iConstitution;
}
```

```cpp
// Somehwere in a .cpp file...
void Critter::setConstitution(int iConstitution)
{
    m_iConstitution = iConstitution;
    recalculateHitPoints();
}
```

So, why *wouldn't* we do this?

Inline expansion

Time is our constraint.

Betting time on beating compiler optimization is usually bad.

Betting time against beating *future* compiler optimization is *terrible.*

Back to Inheritance

```
class Critter
{
public:
    int getHitPoints();

// All sorts of other stuff
};
```

```
class Adventurer : public Critter
{
public:
    bool isGameOver();

// All sorts of other stuff
};
```

```
bool Adventurer::isGameOver()
{
    return getHitPoints() <= 0;
}
```

A class derived from a base class has access to the base class's members

Inheritance of Behavior (or Implementation)

Inheritance of Interface

virtual is the mechanism that allows this in C++

```
class Critter
{
public:
    int getHitPoints();
    virtual void setHitPoints(int iHP);
    virtual void attack(Critter & victim);

// All sorts of other stuff
};
```

```
class Adventurer : public Critter
{
public:
    bool isGameOver();
    virtual void attack(Critter & victim);
    void addExperience(int iExperience);

// All sorts of other stuff
};
```

```
void Critter::attack(Critter & victim)
{
    victim.setHitPoints(victim.getHitPoints() - 5);
}
```

```
void Adventurer::attack(Critter & victim)
{
    Critter::attack(victim);

    if( victim.getHitPoints() <= 0 )
        addExperience(10);
}
```

So.
virtual methods
can be overridden.

Non-virtual methods cannot.

```
class Critter
{
public:
    int getHitPoints();
    virtual void setHitPoints(int iHP);
    virtual void attack(Critter & victim);

// All sorts of other stuff
};
```

```
class Adventurer : public Critter
{
public:
    int getHitPoints();
    bool isGameOver();
    virtual void attack(Critter & victim);
    void addExperience(int iExperience);

// All sorts of other stuff
};
```

```
int Adventurer::getHitPoints()
{
    return Critter::getHitPoints() + 5;
}
```

```
Critter critter;
Adventurer adventurer;

Critter * pCritter1 = &critter;
Critter * pCritter2 = &adventurer;

Critter & rCritter1 = critter;
Critter & rCritter2 = adventurer;
```

```
pCritter1->getHitPoints();
pCritter2->getHitPoints();

pCritter1->attack(*pCritter2);
pCritter2->attack(*pCritter1);
```

Virtual methods:
Call the method in the object pointed to/referenced.

Non-virtual methods:
Call the method in the class of the calling expression
(which is typically just a pointer or reference).

Remember!

C++ is a mathematical notation.

In C++, *almost* everything is an expression.

Expressions, like variables, have types. They also have values.

```
5
```

```
"Hello, World"
```

```
5 + 7.0
```

```
cout
```

```
cout << "Hi"
```

```
pCritter1
```

```
Critter critter;
Adventurer adventurer;

Critter * pCritter1 = &critter;
Critter * pCritter2 = &adventurer;

Critter & rCritter1 = critter;
Critter & rCritter2 = adventurer;
```

pCritter1

pCritter2

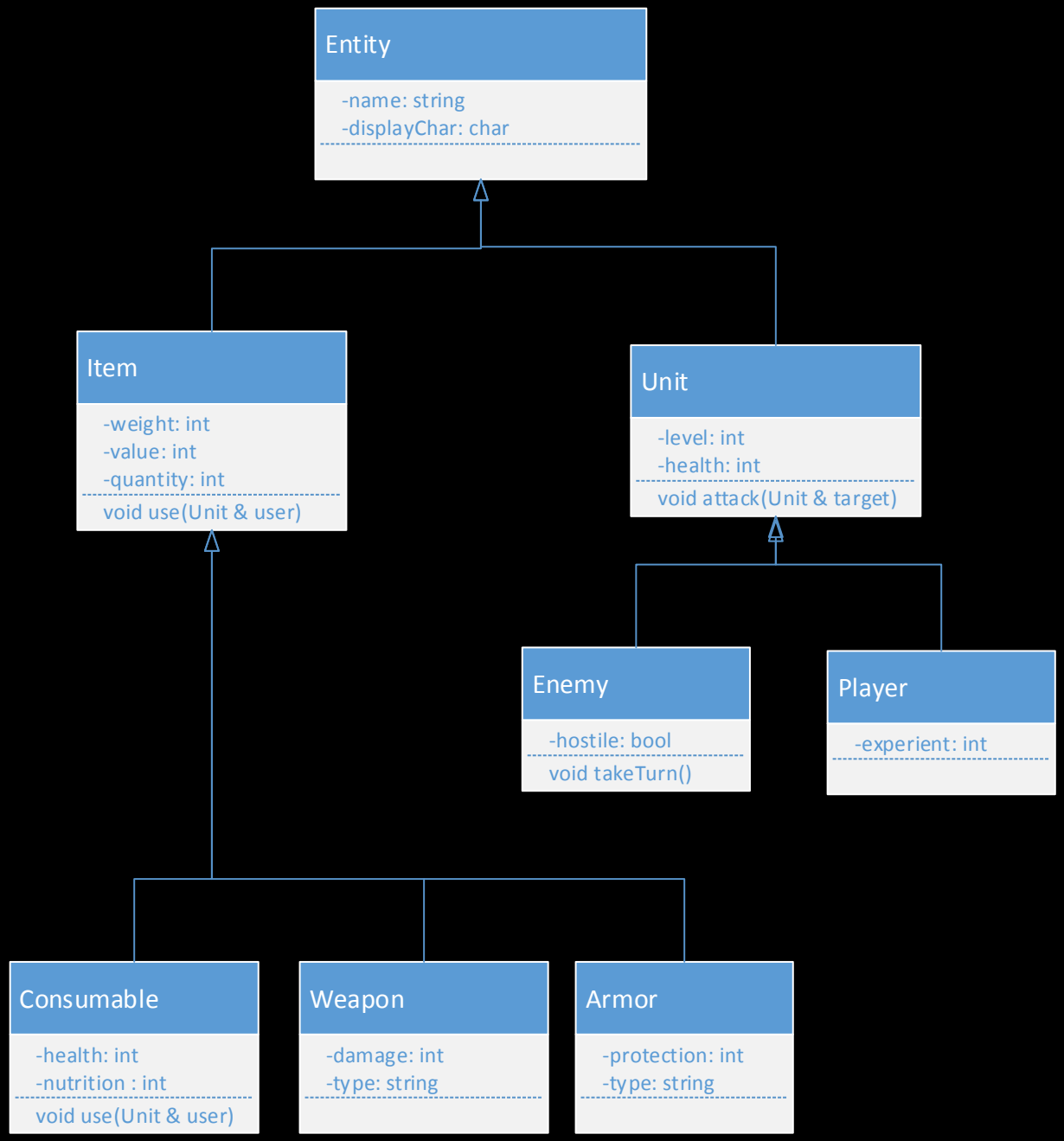
```
class Critter
{
public:
    int getHitPoints();
    virtual void setHitPoints(int iHP);
    virtual void attack(Critter & victim);

// All sorts of other stuff
};
```

```
Critter critter;
Adventurer adventurer;

Critter * pCritter1 = &critter;
Critter * pCritter2 = &adventurer;

Critter & rCritter1 = critter;
Critter & rCritter2 = adventurer;
```

```
pCritter1->getHitPoints();
pCritter2->getHitPoints();

pCritter1->attack(*pCritter2);
pCritter2->attack(*pCritter1);
```

Virtual: what matters is the object pointed to

Non-virtual: what matters is the type of the calling expression


Entity
-name: string
-displayChar: char
Item
-weight: int
-value: int
-quantity: int
void use(Unit & user)
Unit
-level: int
-health: int
void attack(Unit & target)
Enemy
-hostile: bool
void takeTurn()
Player
-experient: int
Consumable
-health: int
-nutrition : int
void use(Unit & user)
Weapon
-damage: int
-type: string
Armor
-protection: int
-type: string