DigiPen
INSTITUTE OF TECHNOLOGY

CS170#12.3

Abstract Classes. More Examples

Vadim Surov

Example: RangedWeapon

- In this slightly more complicated example, we want to extend our Weapon class to handle shooting
  - We use the std::string class for the name of a Weapon (rather than our home-made String class in the previous lecture)
  - We also want to add a data member called ammo that gives the amount of ammunition the RangedWeapon contains

Example: RangedWeapon

- Our old Weapon class (slightly changed):

```
class Weapon {
public:
  Weapon(const std::string& name, int min_damage,
      int max_damage, float weight);
  void set_name(const std::string& name);
  void set_damage(int min_damage, int max_damage);
  void set_weight(float weight);
  void display(void) const;
private:
  std::string name;
  int min_damage, max_damage;
  float weight;
};
```

Example: RangedWeapon

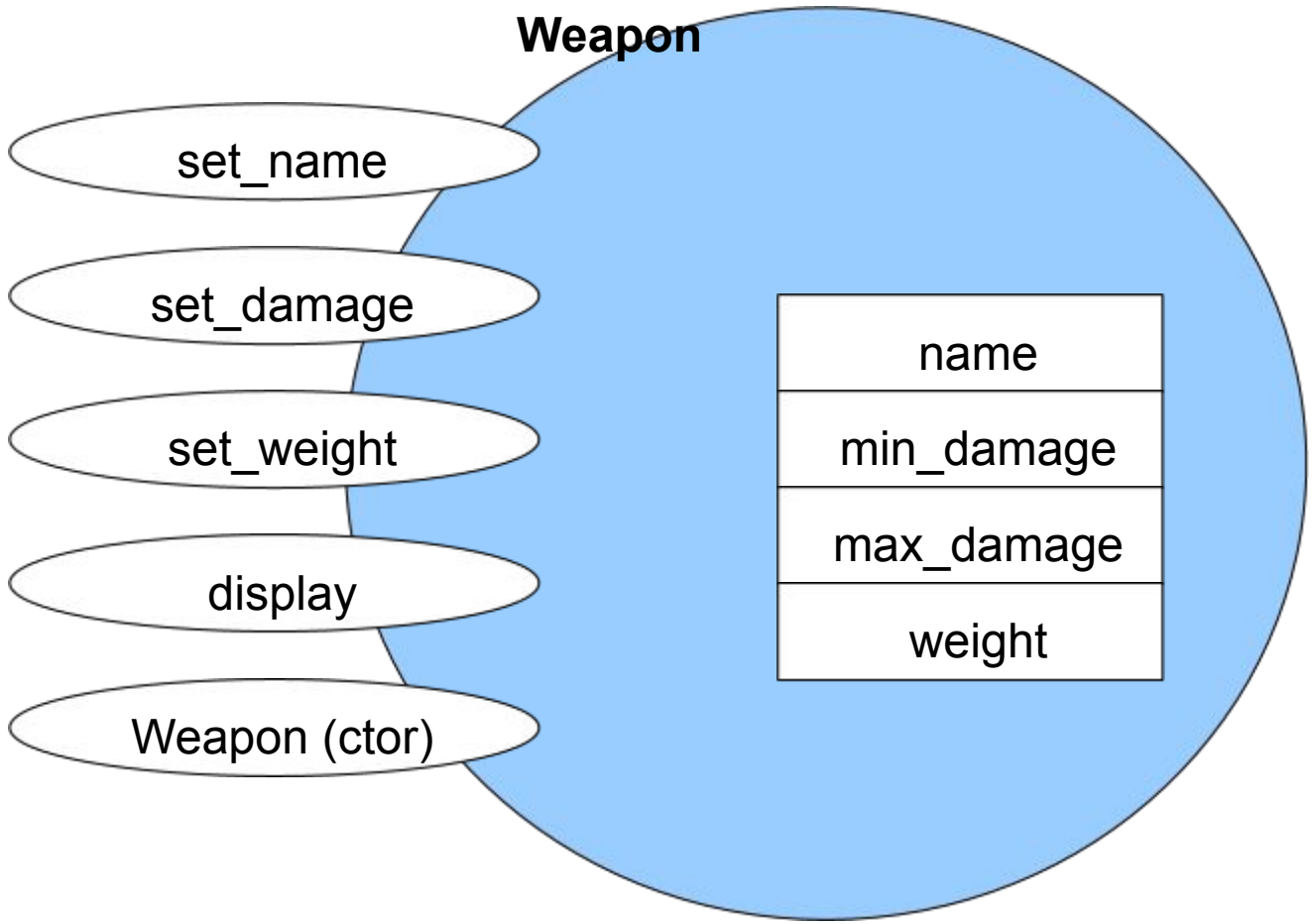
- The constructor and display() function for Weapon:

```
Weapon::Weapon(const std::string& name,
    int min_damage, int max_damage, float weight)
     : name(name), min_damage(min_damage),
        max_damage(max_damage), weight(weight) {}

void Weapon::display(void) const {
   cout << "Name: " << name << endl;
   cout << "Min damage: " << min_damage << endl;
   cout << "Max damage: " << max_damage << endl;
   cout << "Weight: " << weight << "g" << endl;
}
```

Example: RangedWeapon

- How is Weapon organized in memory?



Example: RangedWeapon

- We now want to extend the Weapon class to allow shooting and call it a RangedWeapon:

```
#include "Weapon.h"
class RangedWeapon : public Weapon {
public:
  RangedWeapon(const std::string& name,
        int min_damage, int max_damage,
        float weight, int ammo);
  void set_ammo(int ammo);
  void display(void) const;
private:
  int ammo;
};
```

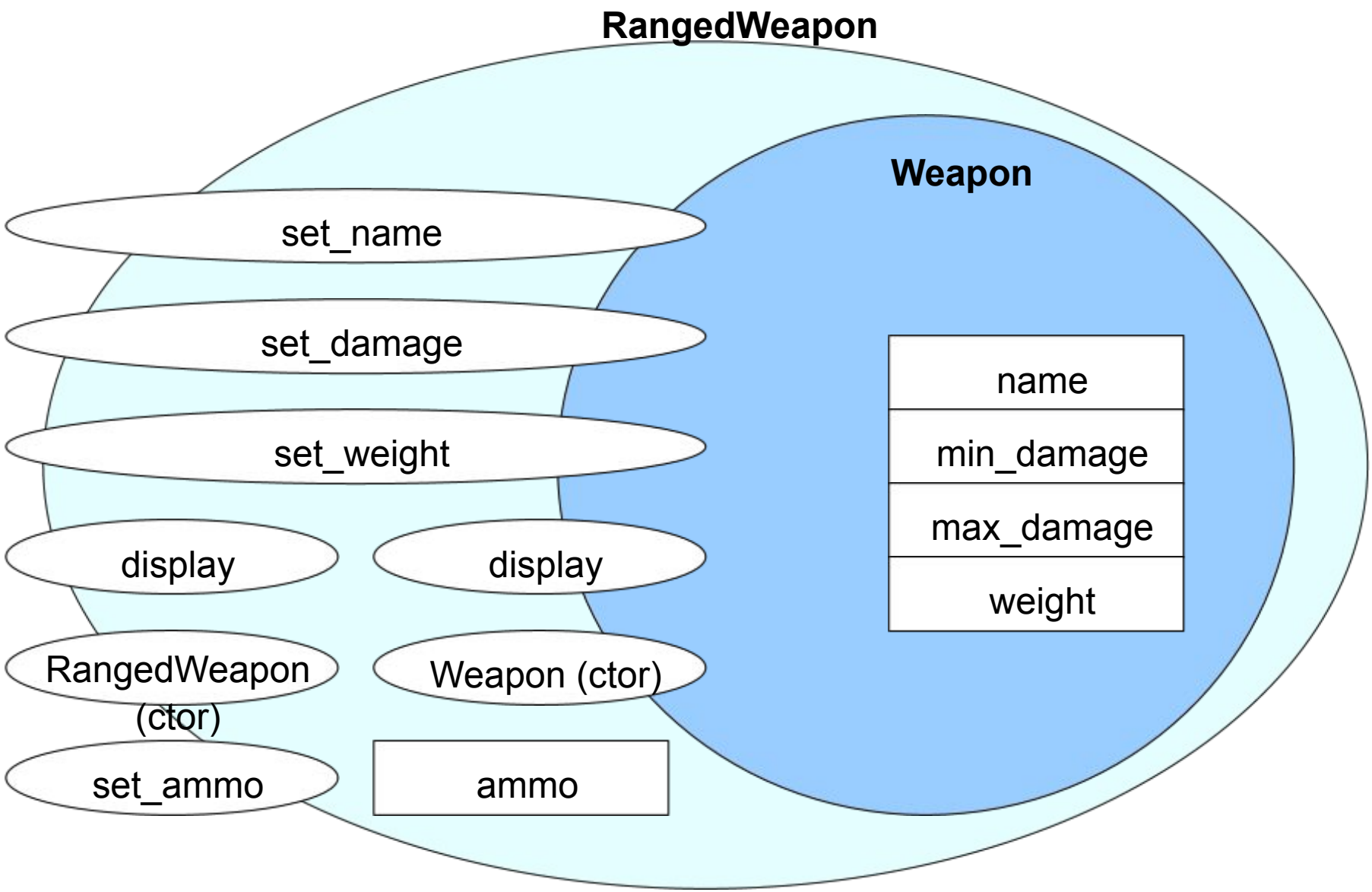
Example: RangedWeapon

- The constructor and display() function for RangedWeapon:

```
RangedWeapon::RangedWeapon(const std::string&
      name, int min_damage, int max_damage,
      float weight, int ammo)
: Weapon(name, min_damage, max_damage, weight),
  ammo(ammo) {};

void RangedWeapon::display(void) const
{
  Weapon::display();
  cout << "Ammo: " << ammo << endl;
}
```

Example: RangedWeapon



Example: RangedWeapon

- Using the classes:

```
Weapon w1("Dagger", 5, 10, 1.5f);
w1.display();
std::cout << std::endl;
w1.set_damage(7, 12); // Changing the damage
w1.display();
std::cout << std::endl;
// a RangedWeapon object
RangedWeapon r1("Pistol", 20, 30, 100.0f, 6);
r1.display();
std::cout << std::endl;
r1.set_damage(25, 35); // Changing the damage
r1.display();
std::cout << std::endl;
```

Example: RangedWeapon

- Output:

```
Name: Dagger
Min damage: 5
Max damage: 10
Weight: 1.5g

Name: Dagger
Min damage: 7
Max damage: 12
Weight: 1.5g

Name: Pistol
Min damage: 20
Max damage: 30
Weight: 100g
Ammo: 6

Name: Pistol
Min damage: 25
```

Virtual methods

- Does the following code compile?

```
void funcWeapon(const Weapon& w){
  w.display(); std::cout << std::endl;
}
void funcRanged(const RangedWeapon& r) {
  r.display(); std::cout << std::endl;
}
int main(void) {
  Weapon w1("Dagger", 5, 10, 1.5f);
  RangedWeapon r1("Pistol", 20, 30, 100.0f, 6);
  funcWeapon(w1);
  funcRanged(r1);
  funcWeapon(r1);
  funcRanged(w1);
  ...
```

Virtual methods

- Output after removing "funcRanged(w1);":

```
Name: Dagger
Min damage: 5
Max damage: 10
Weight: 1.5g

Name: Pistol
Min damage: 20
Max damage: 30
Weight: 100g
Ammo: 6

Name: Pistol
```

Virtual methods

- Does the following code compile?

```
Weapon w1("Dagger", 5, 10, 1.5f);
RangedWeapon r1("Pistol", 20, 30, 100.0f, 6);
Weapon* wptr = &w1;
RangedWeapon* rptr = &r1;

wptr->display(); std::cout << std::endl;
rptr->display(); std::cout << std::endl;

wptr = &r1; // point a Weapon pointer to a
                              RangedWeapon
wptr->set_name("M16");
wptr->set_ammo(30);
wptr->display(); std::cout << std::endl;
```

Virtual methods

- Output ("wptr->set_ammo(30);" removed):

```
Name: Dagger
Min damage: 5
Max damage: 10
Weight: 1.5g

Name: Pistol
Min damage: 20
Max damage: 30
Weight: 100g
Ammo: 6

Name: M16
Min damage: 25
Max damage: 35
```

Virtual methods

- Consider this program:

```
Weapon w1("Dagger", 5, 10, 1.5f);
Weapon w2("Knife", 3, 12, 1.0f);
RangedWeapon r1("Pistol", 20, 30, 100.0f, 6);
RangedWeapon r2("M16", 30, 50, 3600.0f, 30);
Weapon* inventory[4]; // An array of weapons
inventory[0] = &w1;
inventory[1] = &w2;
inventory[2] = &r1;
inventory[3] = &r2;
for (int i = 0; i < 4; i++) {
  inventory[i]->display();
  std::cout << std::endl;
}
```

Virtual methods

- Output:

```
Name: Dagger
Min damage: 5
Max damage: 10
Weight: 1.5g

Name: Knife
Min damage: 3
Max damage: 12
Weight: 1g

Name: Pistol
Min damage: 20
Max damage: 30
Weight: 100g

Name: M16
Min damage: 30
Max damage: 50
```

Virtual methods

- Notice that it is the Weapon version of display() that is called, not the RangedWeapon version
  - This is because inventory is an array of Weapon*
  - This is the default behaviour of the C++ compiler
- This type of code generation is called *static binding* or *early binding*; it is done at compile time

Virtual methods

- What we really want is to call the correct version of display() depending on the type of the object
  - Which function to call is only determined at run-time
  - Delaying the binding until run-time is called *dynamic binding*
- To achieve this in C++, make the base class function ***virtual***:

```
virtual void display(void) const;
```

Virtual methods

- Once you declare the base class function **virtual**, then the correct function will be called
  - The previous example will now work properly
- Dynamic binding is not the default because
  - It is slightly less efficient
  - You may not want to redefine the function in the derived class
- Basically, *a virtual function allows a derived class to override it*

Abstract base classes

- A class hierarchy goes from more general to more specific
  - E.g., Weapon → RangedWeapon → ProjectileWeapon → Rifle → …
- Derived classes are more specific than their base classes
  - The more "derived", the more "specific"

Abstract base classes

- Some base classes are so general that they do not (and cannot) represent anything specific
  - E.g., what is a figure?
- Such classes should not be instantiated
- We call these types of classes *abstract base classes*