Lecture 16
C++ Inheritance

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Fall 2021

Slides adapted from:
Jesse Tov (Northwestern), Hal Perkins (Washington), Godmar Back (Virginia Tech)

Northwestern

Administrivia

- Remember that project proposals are due on Friday
  - We’ve gotten about a third of proposals so far

Today’s Goals

- Finish walking through motion planning example
  - Start by debugging a little

- Introduce concept of inheritance for classes

- Describe inheritance process in C++

Getting the code for today

- Download code in a zip files from here: https://nu-cs211.github.io/cs211-files/lec/15_finalProject.zip https://nu-cs211.github.io/cs211-files/lec/16_inheritance.zip

- Extract code wherever

- Open with CLion
  - Make sure you open the folder with the CMakeLists.txt

Outline

- **Game Motion Planning**
- Concept of Inheritance
- Inheritance in C++
- GE211 Inheritance

Plan for game

- Image sprite that represents a character in the game
  - Moves towards a given position at a set velocity

- Text sprite to explain what position is being moved to

- Each character keeps a list of positions to move to
  - Moves towards the first position until it reaches it
  - Then starts moving towards the next position

- Add to list of positions with mouse clicks

Initial Character class

- Data members
  - Image_sprite sprite_
  - Posn<float> position_

- Interface
  - Constructor (from string for filename)
  - Getters/Setters for data members

Drawing the sprite

- Add sprite image to Resources/
- Add character to Model as a private member
    - Probably a std::vector of characters
- Add getter to allow View to access characters vector
- Update View to iterate through the characters and draw each one

Add motion to Character class

- Data members
  - Image_sprite sprite_
  - Posn<float> position_
  - float velocity_
  - Posn<float> destination_

- Interface
  - Constructor (from string for filename)
  - Getters/Setters for data members
  - update(double dt) called from on_frame()
  - distance_to_position_() helper function

Making the sprites move

- Add initial destinations upon creation in the Model

- Add on_frame() function to Controller and Model
  - Call Model’s on_frame()
  - Then call each character’s on_frame()

Add a text sprite to explain each character’s movement

- View gets new private members
  - ge211::Text_sprite explanation_
  - ge211::Font sans28_

- Build output string in draw()
  - Create an Image_sprite::Builder
  - Set a font and a Color
  - Set the string to be displayed based on the character
  - Reconfigure the Image_sprite
  - Add the sprite so it appears

Upgrade characters to hold a list of destinations

- Probably want to use an std::queue
    - push() positions to the end of the queue
    - pop() positions from the front of the queue

- Change to the next destination after we reach it
    - Occurs in on_frame()

- Make sure the initial destination is the initial position
    - Or we’ll start moving somewhere right away

Use mouse clicks to specify waypoints for a character

- Respond to mouse clicks in the Controller
  - Forward click to the model to act upon

- Model uses mouse click to add destination for first character

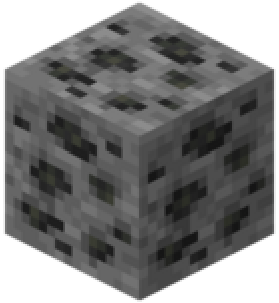
Outline

- Game Motion Planning
- **Concept of Inheritance**
- Inheritance in C++
- GE211 Inheritance

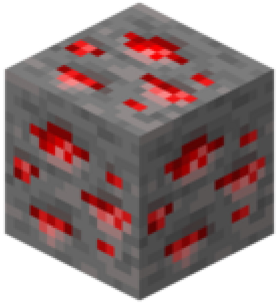
Duplicated behavior in separate classes

- Example: Minecraft
  - World is made of destructible blocks of various types
  - Blocks have different qualities
    - Sounds when hit, number of hits to break, what it drops when broken

Sand Block



Coal Ore Block



Redstone Ore Block

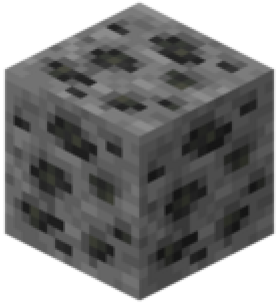
Example Class for a Sand Block

```
class Sand_block {
public:
  Sand_block(Posn<int>);


  void hit_block();
  void fall();

private:
  Posn<int> position_;
  int hits_remaining_;
}
```

These functions would probably take arguments and maybe return things. We'll ignore that for this example.

Example Class for a Coal Ore Block

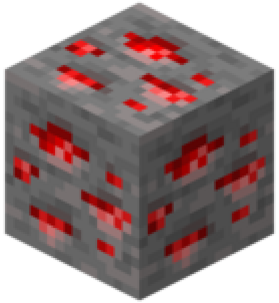
```
class Coal_ore_block {
public:
  Coal_ore_block(Posn<int>);


  void hit_block();
  void drop_item();

private:
  Posn<int> position_;
  int hits_remaining_;
}
```



These functions would probably take arguments and maybe return things. We’ll ignore that for this example.

Example Class for a Coal Ore Block

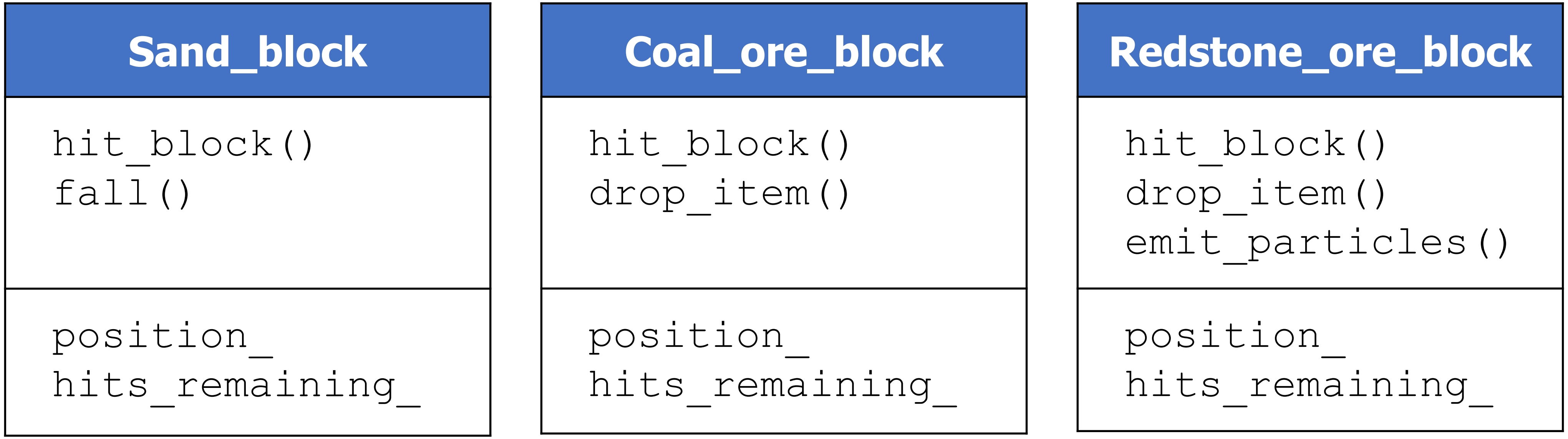
```
class Redstone_ore_block {
public:
  Redstone_ore_block(Posn<int>);


  void hit_block();
  void drop_item();
  void emit_particles();

private:
  Posn<int> position_;
  int hits_remaining_;
}
```



These functions would probably take arguments and maybe return things. We’ll ignore that for this example.

Design without inheritance

- One class per block type:



Sand_block

```
hit_block()
fall()
```

```
position_
hits_remaining_
```

Coal_ore_block

```
hit_block()
drop_item()
```

```
position_
hits_remaining_
```

Redstone_ore_block

```
hit_block()
drop_item()
emit_particles()
```

```
position_
hits_remaining_
```

- Feels pretty redundant. Lots of repeated information
- Cannot use multiple blocks as the same thing
    - Can’t have a `vector` of blocks, for instance

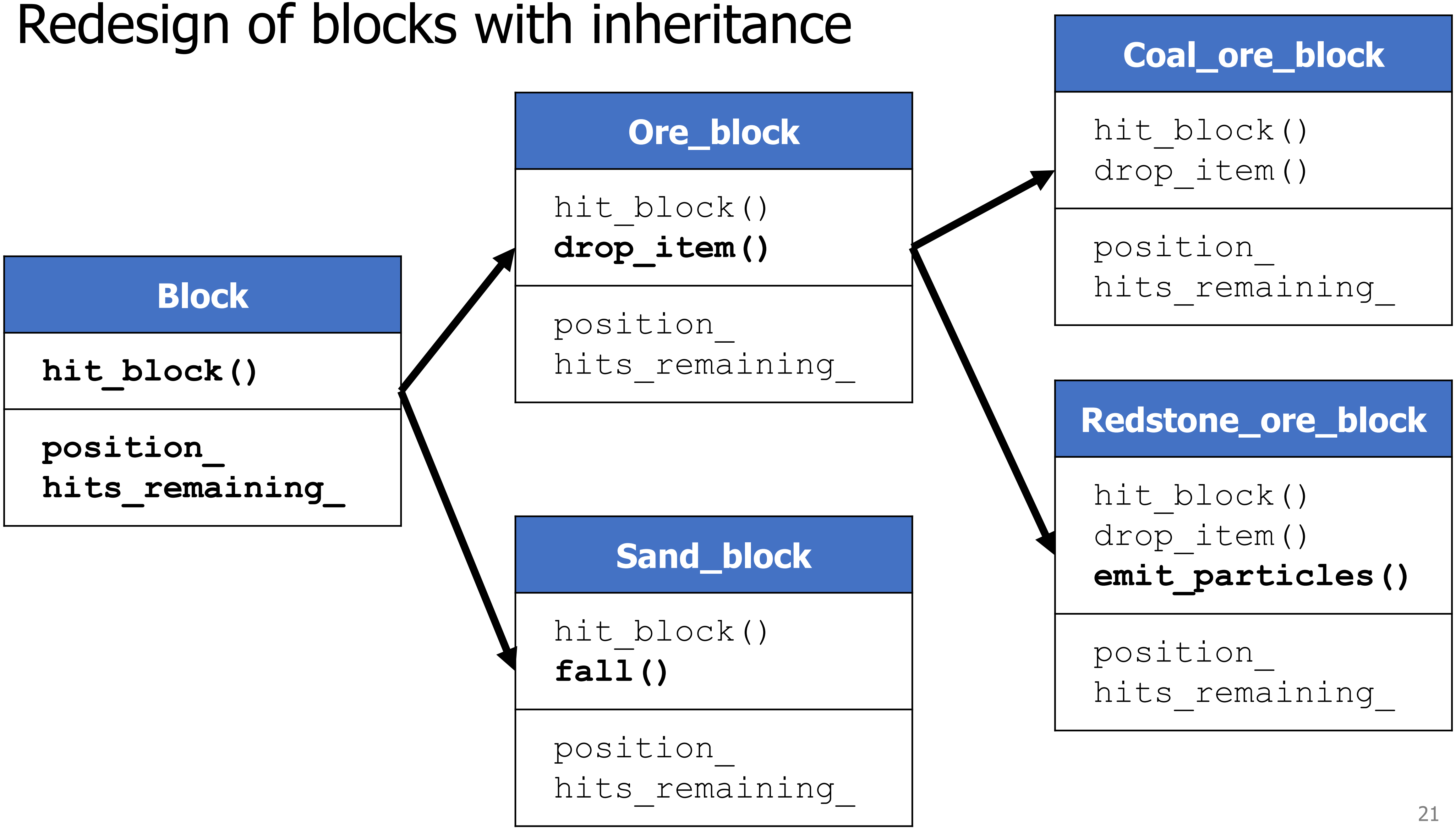
Concept: share common traits

- Inheritance allows one class to copy all the qualities of another
  - i.e. it inherits member functions and data members

- Allows us to form parent-child "is-a" relationship between classes
  - A child (derived class) extends a parent (base class)

- Objects can be treated anything they inherit from
  - Object can be treated as the base class to access general functionality
  - Or treated as the specific derived class to access specific functionality

Redesign of blocks with inheritance



Block

hit_block()

position_
hits_remaining_

Ore_block

hit_block()
drop_item()

position_
hits_remaining_

Sand_block

hit_block()
fall()

position_
hits_remaining_

Coal_ore_block

hit_block()
drop_item()

position_
hits_remaining_

Redstone_ore_block

hit_block()
drop_item()
emit_particles()

position_
hits_remaining_

Relationships between our blocks

- Redstone_ore_block is-a Ore_block TRUE
- Coal_ore_block is-a Ore_block TRUE

- Coal_ore_block is-a Block TRUE

- ~~Coal_ore_block is-a Redstone_ore_block~~ FALSE
- ~~Ore_block is-a Redstone_ore_block~~ FALSE

Derived classes can override inherited functionality

```
void Ore_block::hit_block() {
  hits_remaining--;
  if (hits_remaining == 0) { drop_item(); }
}

void Redstone_ore_block::hit_block() {
  hits_remaining--;
  emit_particles();
  if (hits_remaining == 0) { drop_item(); }
}
```

Benefits of inheritance

- Code reuse
  - Children can automatically inherit code from parents

- Extensibility
  - Children can add custom behavior by extending or overriding

- Polymorphism
  - Ability to redefine existing behavior but preserve the interface
  - Children can override the behavior of the parent
  - Other parts of the code can make calls on objects without knowing which part of the inheritance tree they are from

Outline

- Game Motion Planning
- Concept of Inheritance
- **Inheritance in C++**
- GE211 Inheritance

Simpler class for demonstrating inheritance

positions.hxx
positions.cxx

```
class Position {
public:
  Position(int x, int y);
  int distance_to(Position const& other) const;
  void print() const;

private:
  int x_;
  int y_;
};
```

Create a new class that inherits from Position

positions.hxx
positions.cxx

```
class Position3D: public Position {
public:
  Position3D(int x, int y, int z);
  int distance_to(Position3D const& other) const;
  void print() const;

private:
  int z_;
};
```

Needs its own unique constructor

positions.hxx
positions.cxx

```
class Position3D: public Position {
public:
  Position3D(int x, int y, int z);
  int distance_to(Position3D const& other) const;
  void print() const;

private:
  int z_;
};
```

Class derivation list

Position3D inherits from Position

Class derivation list

```
class Name : public BaseClass1, public BaseClass2
{ };
```

- Can inherit from any number of classes
  - Can add some difficulties outside the scope of this class

- `public` is an access specifier
  - Always want to use public
  - Private would make everything inherited private
    - Which would mean other things wouldn’t know you had them…

Needs its own unique constructor

positions.hxx
positions.cxx

```
class Position3D: public Position {
public:
  Position3D(int x, int y, int z);
  int distance_to(Position3D const& other) const;
  void print() const;

private:
  int z_;
};
```

Constructor

Must be unique for each class

Extending base class functionality

positions.hxx
positions.cxx

```
class Position3D: public Position {
public:
  Position3D(int x, int y, int z);
  int distance_to(Position3D const& other) const;
  void print() const;

private:
  int z_;
};
```

Extended functionality

Provides features that the original class does not

Overriding base class functionality

positions.hxx
positions.cxx

```
class Position3D: public Position {
public:
  Position3D(int x, int y, int z);
  int distance_to(Position3D const& other) const;
  void print() const;

private:
  int z_;
};
```

Overridden functionality

Redefines existing functionality
to do something different

Constructor for our derived class

positions.hxx
positions.cxx

```
Position3D::Position3D(int x, int y, int z)
    : Position(x, y),
      z_(z)
{ }
```

- Base class constructors are called first in the initializer list
  - C++ will automatically call the default constructor if one exists and you don’t

Access is not allowed to the base class's private members

```
int
Position3D::distance_to(Position3D const& other) const
{
    int diffx = other.x_ - x_;
    int diffy = other.y_ - y_;
    int diffz = other.z_ - z_;
    return std::sqrt(diffx*diffx + diffy*diffy
                     + diffz*diffz);
}
```

- **ERROR!** This won't work because `x_` and `y_` are private
  - Need some way to make them accessible to things that inherit from the class
  - Additional access specifier: `protected`

Classes meant to be inherited from use protected members

```
class Position {
public:
  Position(int x, int y);
  int distance_to(Position const& other) const;
  void print() const;

protected:
  int x_;
  int y_;
};
```

Compiler decides which version of an overridden function to call

```
Position p1 {0, 0};
Position3D p2 {0, 0, 0};
p1.print();
p2.print();
```

- How does the compiler know which version of `print()` to call?
  - Decides at compile time based on which type it is
  - This is known as “static dispatch”

Problem with static dispatch

- But often we would prefer to call the extended version of the function
  - Even if the object is treated as the base class

```
void print_position(Position const& p) {
  p.print();
}

Position p1 {0, 0};
Position3D p2 {0, 0, -5};
print_position(p1);
print_position(p2);// prints the 2D position version
```

Dynamic dispatch

- For some functions, have code use the overridden version if it exists
  - Need some way of specifying which functions should work this way

- This needs to be decided at runtime
  - Function can’t know in advance which type it is going to be called with
  - Language has to support this feature (C++ does!)

Declare functions virtual if dynamic dispatch should occur

```
class Position {
public:
  Position(int x, int y);
  int distance_to(Position const& other) const;
  virtual void print() const;

protected:
  int x_;
  int y_;
};
```

In derived class, mark function as override

```
class Position3D: public Position {
public:
  Position3D(int x, int y, int z);
  int distance_to(Position3D const& other) const;
  void print() const override;

private:
  int z_;
};
```

Important for compile-time errors.

Compiler will tell you if there isn’t a virtual function you’re overriding.

Repeat example but with dynamic dispatch

- Now our example works because the program decides which version of `print()` to call at run-time

```
void print_position(Position const& p) {
  p.print();
}

Position p1 {0, 0};
Position3D p2 {0, 0, -5};
print_position(p1);
print_position(p2);// prints the 3D position version!
```

Creating a class that MUST be overridden

- Sometimes we want to include a function in a base class but only implement it in derived classes
  - Back to Minecraft example:
    `hit_block()` might not have a default implementation

- We can make a function “pure virtual” in C++
  - No implementation is added for the base class
  - Any class that inherits is required to implement it

- The base class becomes an “abstract class”
  - It cannot be instantiated as an object because all of its functions aren’t implemented
  - It’s only useful as a class to inherit from

Making a pure virtual function

```
class Printable {
public:
  virtual void print()const = 0;
}


class Position : public Printable {
  void print() const override;
}
```

Outline

- Game Motion Planning
- Concept of Inheritance
- Inheritance in C++
- **GE211 Inheritance**

Inheritance in GE211

- https://github.com/tov/ge211/blob/main/include/ge211/base.hxx

- Abstract_game is an abstract base class
  - draw(Sprite_set&) is a pure virtual function
  - Any game MUST implement draw()

- Many other functions are marked virtual
  - Our Controller overrides them with its own implementation
    - on_key, on_mouse_move, etc.

- Some functions are implemented and we inherit directly
  - run() is a good example of this

Outline

- Game Motion Planning
- Concept of Inheritance
- Inheritance in C++
- GE211 Inheritance