Lecture 15
C++ Inheritance

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

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

Administrivia

- Homework 5 is due today
  - We'll do the best we can, but office hours will be *busy*

- Remember that project proposals are due on Friday!
  - We've gotten only a few proposals so far
  - I'm going to start emailing approvals later today

Today's Goals

- Introduce concept of inheritance for classes
- Describe inheritance process in C++
  - Understand some benefits and possible challenges
- Explore how GE211 uses inheritance

Getting the code for today

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

- Extract code wherever

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

Outline

- **Concept of Inheritance**
- Inheritance in C++
  - Overriding Functions
  - Storing Inherited Classes
- 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 Redstone 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	Coal_ore_block	Redstone_ore_block

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

`position_` `hits_remaining_`	`position_` `hits_remaining_`	`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

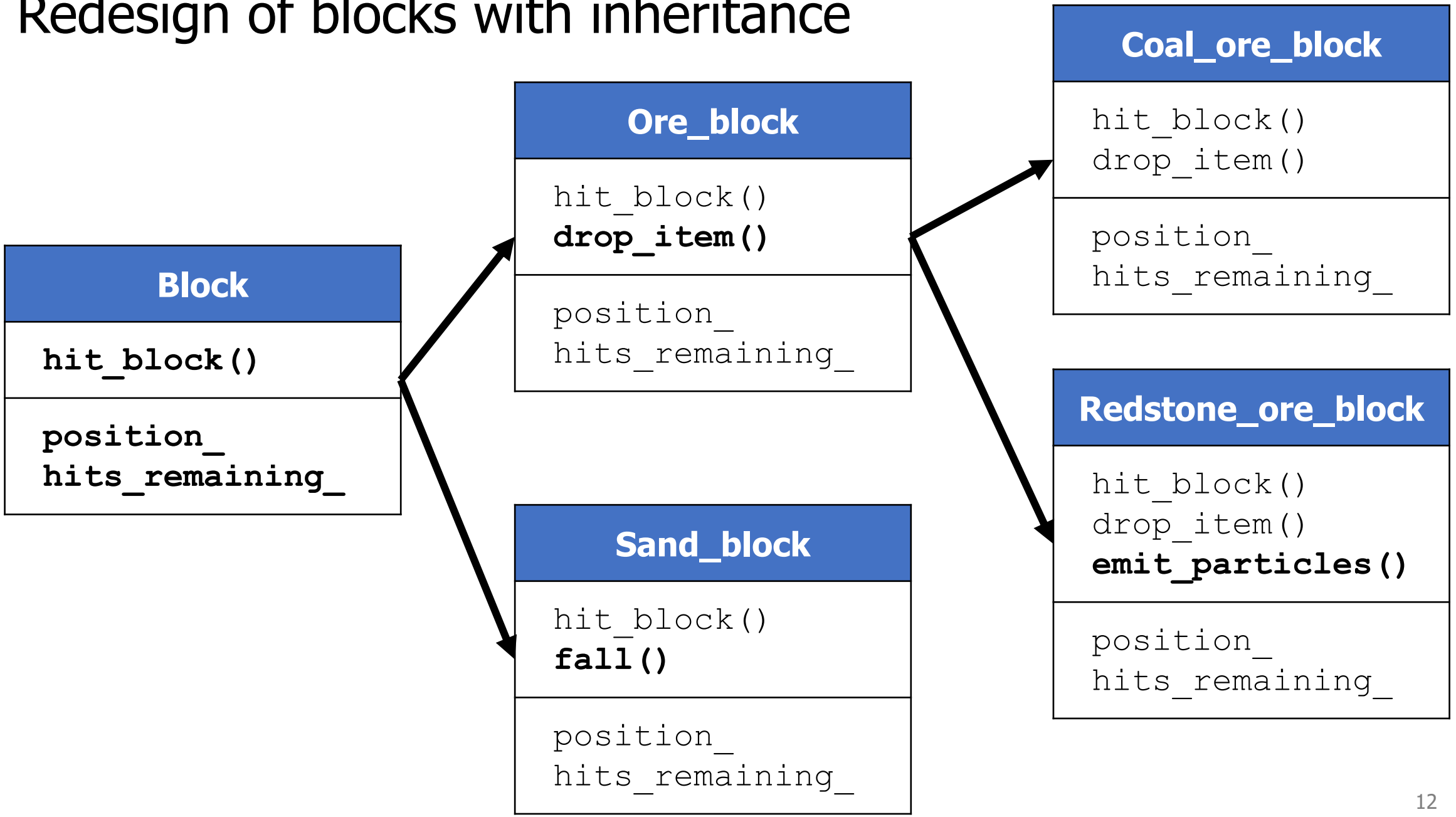
Concept: share common traits

- 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 as 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



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(); }
}
```

Derived classes can be treated as the parent class

- We can make a vector of generic "`Block`" and fill it with specific types of blocks
  - Although we have to do some extra work: using pointers in this example
  - More on this later

```
std::vector<Block*> blocks;
blocks.push_back(&Coal_ore_block());
blocks.push_back(&Redstone_ore_block());
blocks.push_back(&Coal_ore_block());
blocks.push_back(&Sand_block());

blocks[1]->hit_block(); // calls Redstone hit_block()
```

Benefits of inheritance

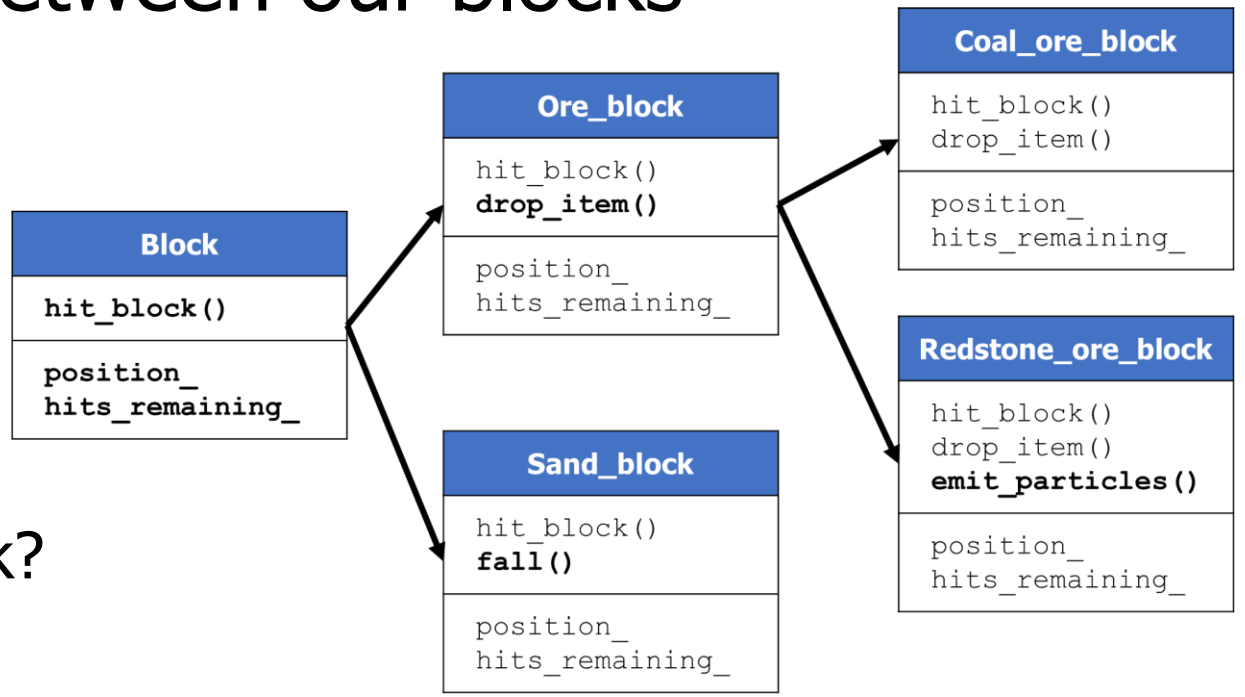
- Code reuse
  - Children can automatically inherit code from parents

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

- **Polymorphism** (biggest reason)
  - 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

Break + Quiz: Relationships between our blocks

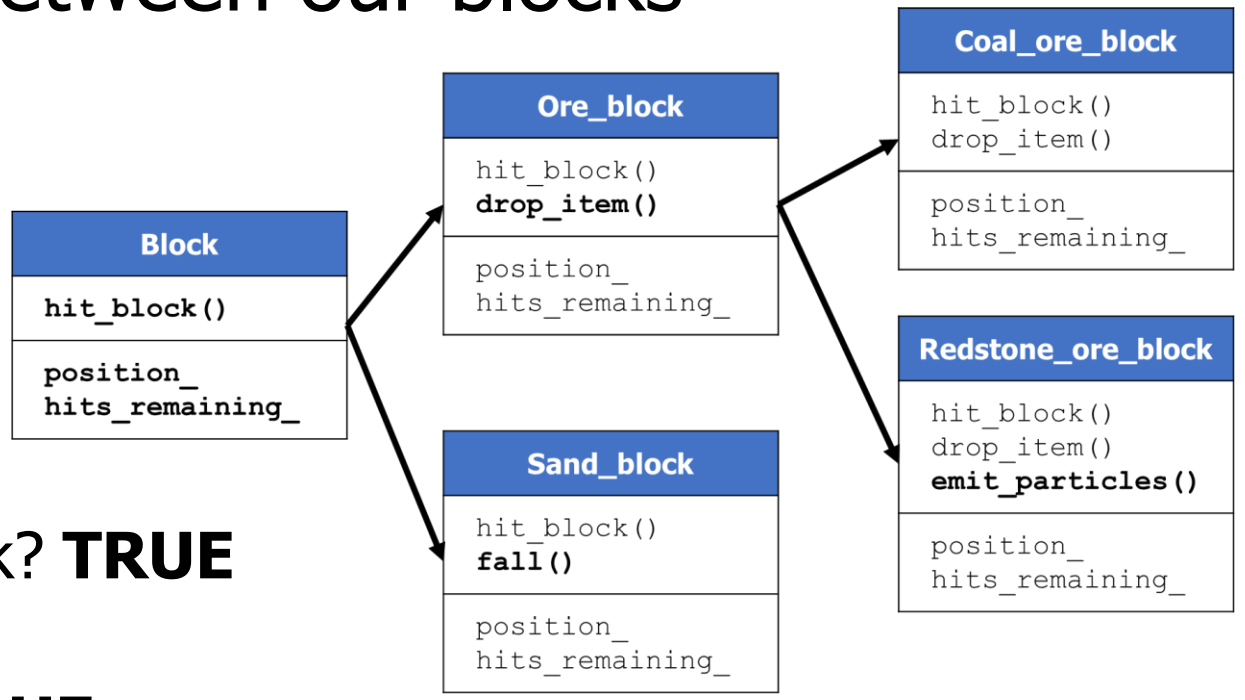
- Determine if the following is-a relationships exist



- True or False:
  - Redstone_ore_block is-a Ore_block?
  - Coal_ore_block is-a Ore_block?
  - Coal_ore_block is-a Block?
  - Coal_ore_block is-a Redstone_ore_block?
  - Ore_block is-a Redstone_ore_block?

Break + Quiz: Relationships between our blocks

- Determine if the following is-a relationships exist



- True or False:
  - 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**

Outline

- Concept of Inheritance
- **Inheritance in C++**
  - Overriding Functions
  - Storing Inherited Classes
- 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
{ };
```

- It is possible to inherit from any number of classes
  - Can add some difficulties outside the scope of this class (Diamond problem)

- `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
    - Which really defeats the whole purpose

Derived class 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_;
};
```

Break + Open Question

- How do you decide whether a given member should be `private` or `protected`?

Break + Open Question

- How do you decide whether a given member should be `private` or `protected`?
  - No always-correct answer here, but some thoughts:
  - If your class will never be inherited from: make it `private`
  - If your class will be inherited from: likely make it `protected`
    - Unless it's special to this implementation and won't be reused
    - Or further if inheriting classes *should not* modify it directly

Outline

- Concept of Inheritance
- **Inheritance in C++**
  - **Overriding Functions**
  - Storing Inherited Classes
- GE211 Inheritance

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 doesn’t know in advance which specific 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 written 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 is 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

- Concept of Inheritance
- **Inheritance in C++**
  - Overriding Functions
  - **Storing Inherited Classes**
- GE211 Inheritance

Storing a collection of inherited objects

- We can make a vector of generic "`Block`" and fill it with specific types of blocks
  - Although we have to do some extra work: using pointers in this example
  - More on this later

```
std::vector<Block*> blocks;
blocks.push_back(&Coal_ore_block());
blocks.push_back(&Redstone_ore_block());
blocks.push_back(&Coal_ore_block());
blocks.push_back(&Sand_block());

blocks[1]->hit_block(); // calls Redstone hit_block()
```

The simple thing is broken

vector_of_base.hxx
vector_of_base.cxx

```
Position p1 {1, 2};
Position3D p2 {-1, -2, -3};

std::vector<Printable> broken;
broken.push_back(p1);
broken.push_back(p2);

for (Printable const& p: broken) {
  print_position(p); // prints the wrong thing!
}
```

Object slicing

- `std::vector<Printable>` only allocates enough space to hold the "Printable" class, not the extra stuff for the other classes

- So, you put a child class in and it "slices" off all the special parts
  - Only holds whatever was in the base class

- In terms of memory:
  - If each `Printable` needed 10 bytes
  - And each `Position` needed 30 bytes
  - The vector only hangs on to the first 10 bytes of each `Position`

Fixing object slicing

- To solve this problem, we just need to make sure there's actually memory available
  - `std::vector<Position>` has enough memory for each `Position`
  - `std::vector<Position3D>` has enough memory for each `Position3D`

- But we really want to mix objects of different inherited types
  - So the solution is to hang on to pointers instead!

Storing pointers fixes object slicing

vector_of_base.hxx
vector_of_base.cxx

```
Position p1 {1, 2};
Position3D p2 {-1, -2, -3};

std::vector<Printable*> fixed;
fixed.push_back(&p1);
fixed.push_back(&p2);

for (Printable* const& p: fixed) {
  print_position(*p); // prints the right thing!
}
```

Warning: now we're worried about scoping and liftetimes!

- The new vector just hangs on to pointers, not to the objects themselves!
  - That means we need to make sure that the objects are actually stored somewhere too

- Common solutions
  - Keep each object as a member of a class
    - We do this with sprites in GE211!
  - Keep an array of each individual object type (likely still as a member)
    - And a mixed array of pointers to all of them
  - Dynamic memory (we'll talk about this next week)

Smart pointers example

vector_of_base.hxx
vector_of_base.cxx

```
std::vector<std::unique_ptr<Printable>> heap;
heap.push_back(std::make_unique<Position>(1, 2));
heap.push_back(std::make_unique<Position3D>(-7, -6, -5);

for (std::unique_ptr<Printable> const& p: heap) {
  print_position(*p); // prints the right thing!
}
```

- More on this next week

Break + Practice

```
class Shape {
public:
  Shape(std::string col)
    : color_(col)
  {}

protected:
  std::string color_ = "purple";
};
```

```
class Rectangle: public Shape {
public:
  Rectangle(float x, float y,
            std::string col)
    : Shape(col), height_(x), width_(y)
  { }

  void print_color() {
    std::cout << color_ << "\n";
  }

private:
  float height_;
  float width_;
};
```

What does this print?

```
Rectangle rect(3, 4, "red");
rect.print_color();
```

Break + Practice

```
class Shape {
public:
  Shape(std::string col)
    : color_(col)
  {}

protected:
  std::string color_ = "purple";
};
```

```
class Rectangle: public Shape {
public:
  Rectangle(float x, float y,
            std::string col)
    : Shape(col), height_(x), width_(y)
  { }

  void print_color() {
    std::cout << color_ << "\n";
  }

private:
  float height_;
  float width_;
};
```

What does this print?

```
Rectangle rect(3, 4, "red");
rect.print_color();
```

It prints:
"red"

Outline

- Concept of Inheritance
- Inheritance in C++
  - Overriding Functions
  - Storing Inherited Classes
- **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

Break + Open Question

- Do you need to use inheritance in your Final Project?
  - Technically yes: Controller inherits from Abstract_game
  - Otherwise no, you could make everything as a part of the model
  - Situations where inheritance *could* help
    - Multiple pieces that have some shared behaviors and some unique behaviors
    - Could still manage this manually, or could use classes/inheritance

Outline

- Concept of Inheritance
- Inheritance in C++
  - Overriding Functions
  - Storing Inherited Classes
- GE211 Inheritance