Inheritance

Bachelor of Science - École polytechnique

gael.thomas@inria.fr

Key concepts

- If a class B inherits from a class A
  - B contains the fields and methods of A
  - The type B is compatible with A (but not vice-versa)

- A class can
  - Inherits from multiple classes
  - Uses the constructors of its direct parents in its constructor

- Static versus dynamic dispatch
  - By default C++ uses static dispatch
  - A `virtual` method uses dynamic dispatch
  - A pure `virtual` method is a `virtual` method without body

Summary

Principle of inheritance

- Principle: a child class can inherit a parent class
  - The child class has the fields and methods of the parent
  - And can add new one to specialize the parent
  - The child defines a new type
  - The child type is compatible with the type of the parent

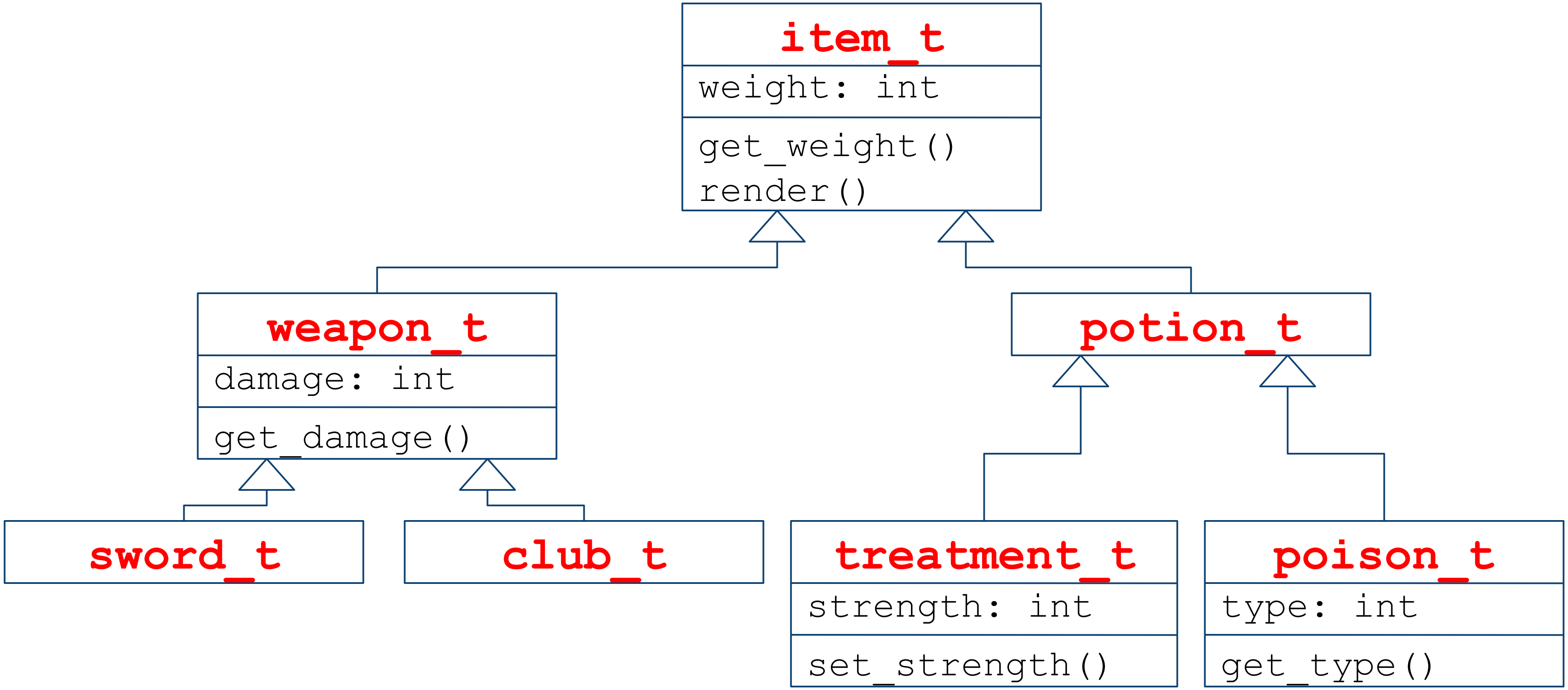
- Inheritance is a transitive relationship
  - If C inherits B and B inherits A, then C inherits A

Goal of inheritance

- Inheritance is useful to specialize a class
  - A generic item in a game, specialized to a weapon or a potion
  - A generic output stream, specialized for the terminal or for the file system
  - …

- Improve code reusability by manipulating the parent
  - The inventory of a character holds items
  - A code that prints to a stream
  - …

Inheritance by example



Inheritance by example

```
struct item_t {
  int weight;
  int get_weight() { return weight; }
  void render()  { ... }
};

struct weapon_t : item_t {
  int damage;
  int get_damage() { return damage; }
};

struct potion_t : item_t {
};

int main(int argc, char* argv[]) {
  weapon_t w { 42, 666 };
  std::cout << w.get_weight() << std::endl; // 42
  std::cout << w.get_damage() << std::endl; // 666
  return 0;
}
```

“ : ” means “inherits”

Summary

1. Principle of inheritance
2. Inheritance and typing
3. Multiple inheritance
4. Inheritance and visibility
5. Inheritance and constructors
6. Static versus dynamic dispatch
7. Dynamic cast

Type compatibility: upcast

- The type of a child is compatible with the type of the parent
  - We can upcast a child to its parent

```
int main(int argc, char* argv[]) {
  item_t* item = new sword_t { 5 };
  // a sword_t is a kind of item_t
  // here use the sword as an item_t
  // => can access the item_t fields and methods
  std::cout << item->get_weight() << std::endl;
  return 0;
}
```

In this example, we can only use the `item_t` fields and methods by using the `item` variable, not the ones of `sword_t`

Type compatibility: downcast

- C++ signals an error if we downcast a parent to one of its children

```
int main(int argc, char* argv[]) {
  item_t* item = new sword_t { 5 };
  // sword_t* sword = item;
  //   => error because an item is not necessarily a sword
  return 0;
}
```

Type compatibility: downcast

- You can however downcast a parent to one of its children by using an explicit static cast
  - `static_cast<destination_type_t>(value)`

```
int main(int argc, char* argv[]) {
  item_t* item = new sword_t { 5, 666 };
  sword_t* sword = static_cast<sword_t*>(item); // ok
  club_t* club = static_cast<club_t*>(item);    // bug!!!
  return 0;
}
```

A `static_cast` is dangerous: `item` is a `sword_t`, but not a `club_t`
=> the cast to `club_t` will lead to bugs at runtime

(more about casts later)

Upcast and generic code

- Thanks to an upcast, you can write a generic code
  - By only considering the fields and methods of a parent class

```
int main(int argc, char* argv[]) {
  item_t* items[] = {
    new sword_t  { 3 }, // weight = 3
    new club_t   { 2 }, // weight = 5
    new poison_t { 7 }  // weight = 7
  };

  int tot_weight = 0;
  for(int i=0; i<3; i++)
    tot_weight += items[i]->get_weight();

  std::cout << tot_weight << std::endl; // 14

  return 0;
}
```

Summary

1. Principle of inheritance
2. Inheritance and typing
3. Multiple inheritance
4. Inheritance and visibility
5. Inheritance and constructors
6. Static versus dynamic dispatch
7. Dynamic cast

Multiple inheritance

- A structure or a class can inherit one or multiple types

```
struct a_t { ... };
struct b_t { ... };
struct c_t : a_t, b_t { };
```

`c_t` inherits the fields and methods of both `a_t` and `b_t`

Summary

1. Principle of inheritance
2. Inheritance and typing
3. Multiple inheritance
4. Inheritance and visibility
5. Inheritance and constructors
6. Static versus dynamic dispatch
7. Dynamic cast

Inheritance and visibility

- The default visibility of a parent is given by the keyword used to define the class
  - `struct` => by default public
    => fields and methods of parents visible everywhere
  - `class` => by default private
    => fields and methods of parents only visible from the child

- You can change the default visibility with `public` and `private`

Inheritance and visibility

```
struct a_t { int x; };
struct b_t { int y; };

class c_t : public a_t, b_t {
  void f() { x = 1; y = 2; } // parent always visible from child
};

struct d_t : private a_t, b_t {
  void f() { x = 1; y = 2; } // parent always visible from child
};

int main(int argc, char* argv[]) {
  c_t c; d_t d;
  std::cout << c.x << std::endl;
  //std::cout << c.y << std::endl; hidden (c_t defined with class)
  //std::cout << d.x << std::endl; hidden (a_t is private)
  std::cout << d.y << std::endl;
  return 0;
}
```

Inheritance and visibility

- C++ also introduces the `protected` visibility
  - A parent defined as `protected` is transitively visible in all inherited classes
  - While a `private` parent is visible only in a direct child

```
struct a_t {  int x;  };

class b_t : protected a_t {  };

class c_t : b_t {
  void f() { x = 42; } // visible through protected
};
```

Note: a field can also be `protected`

Summary

1. Principle of inheritance
2. Inheritance and typing
3. Multiple inheritance
4. Inheritance and visibility
5. Inheritance and constructors
6. Static versus dynamic dispatch
7. Dynamic cast

Inheritance and constructors

- A child can use a constructor of a parent in its constructors
  - By considering a field named as the parent in the constructor

```
struct item_t {
  int weight;
  item_t(int weight) : weight { weight } { }
};

struct weapon_t : item_t {
  int damage;
  weapon_t(int weight) : item_t { weight }, damage { 100 } { }
};

int main(int argc, char* argv[]) {
  weapon_t w { 33 };
  return 0;
}
```

Summary

1. Principle of inheritance
2. Inheritance and typing
3. Multiple inheritance
4. Inheritance and visibility
5. Inheritance and constructors
6. Static versus dynamic dispatch
7. Dynamic cast

Static dispatch

- If a method is redefined in a child the static type of an object is used to identify the method that is called

```
struct weapon_t : item_t {
  std::string render() {
    return "weapon";
  }
};
```

```
struct item_t {
  std::string render() {
    return "item";
  }
};
```

```
struct potion_t : item_t {
  std::string render() {
    return "potion";
  }
};
```

```
int main(int argc, char* argv[]) {
  item_t* it[] = { new weapon_t {}, new potion_t {} };

  std::cout << it[0]->render() << " " << it[1]->render() << std::endl;
  // => item item
  return 0;
}
```

Dynamic dispatch (virtual)

- The **virtual** keyword changes this behavior: the method of the actual type is used

```
struct weapon_t : item_t {
  std::string render() {
    return "weapon";
  }
};
```

```
struct item_t {
  virtual std::string render() {
    return "item";
  }
};
```

```
struct potion_t : item_t {
  std::string render() {
    return "potion";
  }
};
```

```
int main(int argc, char* argv[]) {
  item_t* it[] = { new weapon_t {}, new potion_t {} };

  std::cout << it[0]->render() << " " << it[1]->render() << std::endl;
  // => weapon potion
  return 0;
}
```

Dynamic dispatch (`virtual`)

- The `virtual`/non `virtual` behavior
  - Is fixed by the first method in a class hierarchy
  - And cannot be modified in children classes

Pure `virtual` methods

- A pure `virtual` method is defined at `0`
  - Does not have a body
  - Its class cannot be instantiated, but the type can be used
  - Instantiable children have to implement the method
  - Useful to force overriding

```
struct item_t {
  virtual std::string render() = 0;
};
```

```
struct weapon_t : item_t {
  std::string render() {
    return "weapon";
  }
};
```

```
struct potion_t : item_t {
  std::string render() {
    return "potion";
  }
};
```

Pure virtual methods

```
int main(int argc, char* argv[]) {
  item_t* it[] = { new weapon_t {}, new potion_t {} };

  std::cout << it[0]->render() << " " << it[1]->render() << std::endl;
  // => weapon potion
  return 0;
}
```

```
struct item_t {
  virtual std::string render() = 0;
};
```

```
struct weapon_t : item_t {
  std::string render() {
    return "weapon";
  }
};
```

```
struct potion_t : item_t {
  std::string render() {
    return "potion";
  }
};
```

Summary

1. Principle of inheritance
2. Inheritance and typing
3. Multiple inheritance
4. Inheritance and visibility
5. Inheritance and constructors
6. Static versus dynamic dispatch
7. Dynamic cast

dynamic_cast

- `dynamic_cast`: as `static_cast`
  - But return `nullptr` if the type is incompatible
  - Only usable with polymorphic classes
    (i.e., at least one virtual method)

```
int main(int argc, char* argv[]) {
  item_t* item = new sword_t { 5 };
  sword_t* sword = dynamic_cast<sword_t*>(item); // ok
  club_t* club = dynamic_cast<club_t*>(item);    // nullptr

  if(sword == nullptr)
    std::cout << "this is not a sword" << std::endl;
  if(club == nullptr)
    std::cout << "this is not a club" << std::endl;

  return 0;
}
```

Key concepts

- If a class B inherits from a class A
  - B contains the fields and methods of A
  - The type B is compatible with A (but not vice-versa)
- A class can
  - Inherits from multiple classes
  - Uses the constructors of its direct parents in its constructor
- Static versus dynamic dispatch
  - By default C++ uses static dispatch
  - A `virtual` method uses dynamic dispatch
  - A pure `virtual` method is a `virtual` method without body