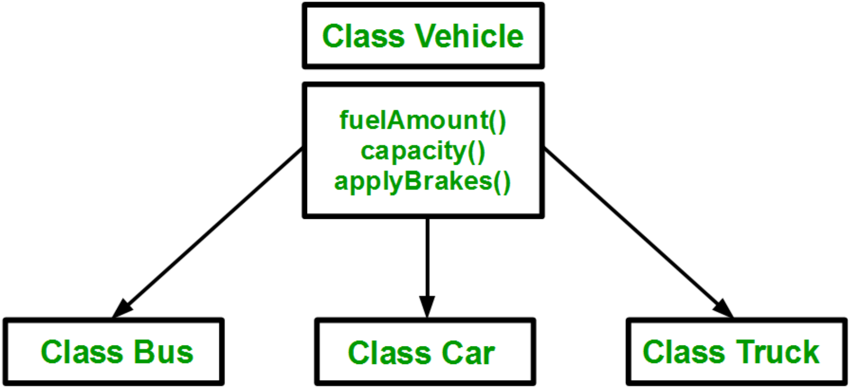
Modern C++ for Computer Vision and Image Processing

Lecture 7: Object Oriented Design

Ignacio Vizzo and Cyrill Stachniss

UNIVERSITÄT BONN

Inheritance



https://www.geeksforgeeks.org/inheritance-in-c/

C vs C++ Inheritance Example

C Code

```
// "Base" class, Vehicle
typedef struct vehicle {
  int seats_;      // number of seats on the vehicle
  int capacity_;   // amount of fuel of the gas tank
  char* brand_;    // make of the vehicle
} vehicle_t;
```

C++ Code

```
class Vehicle {
 private:
  int seats_ = 0;       // number of seats on the vehicle
  int capacity_ = 0;    // amount of fuel of the gas tank
  string brand_;        // make of the vehicle
```

Inheritance

- Class and struct can **inherit data and functions** from other classes
- There are 3 types of inheritance in C++:
  - public `[used in this course]` GOOGLE-STYLE
  - protected
  - private
- `public` inheritance keeps all access specifiers of the base class

https://google.github.io/styleguide/cppguide.html#Inheritance

Public inheritance

- Public inheritance stands for **"is a"** relationship, i.e. if class `Derived` inherits publicly from class `Base` we say, that `Derived` **is a kind of** `Base`

```
class Derived : public Base {
  // Contents of the derived class.
};
```

- Allows `Derived` to use all `public` and `protected` members of `Base`
- `Derived` still gets its own special functions: constructors, destructor, assignment operators

```cpp
#include <iostream>
using std::cout; using std::endl;
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }
 protected:
  int width_ = 0;
  int height_ = 0;
};
class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
};
int main() {
  Square sq(10);  // Short name to save space.
  cout << sq.width() << " " << sq.height() << endl;
  return 0;
}
```

Function overriding

- A function can be declared `virtual`

```
virtual Func(<PARAMS>);
```

- If function is virtual in `Base` class it can be overridden in `Derived` class:

```
Func(<PARAMS>) override;
```

- `Base` can force all `Derived` classes to override a function by making it **pure virtual**

```
virtual Func(<PARAMS>) = 0;
```

Overloading vs overriding

- Do not confuse function **overloading** and **overriding**
- **Overloading:**
  - Pick from all functions with the **same name**, but **different parameters**
  - Pick a function **at compile time**
  - Functions don’t have to be in a class
- **Overriding:**
  - Pick from functions with the **same arguments and names** in different classes of **one class hierarchy**
  - Pick **at runtime**

Abstract classes and interfaces

- **Abstract class:** class that has at least one pure virtual function
- **Interface:** class that has only pure virtual functions and no data members

How virtual works

- A class with virtual functions has a virtual table
- When calling a function the class checks which of the virtual functions that match the signature should be called
- Called **runtime polymorphism**
- Costs some time but is very convenient

Using interfaces

- Use interfaces when you must **enforce** other classes to implement some functionality
- Allow thinking about classes in terms of **abstract functionality**
- **Hide implementation** from the caller
- Allow to easily extend functionality by simply adding a new class

```cpp
#include <iostream>
using std::cout;
using std::endl;
struct Printable {  // Saving space. Should be a class.
  virtual void Print() const = 0;
};
struct A : public Printable {
  void Print() const override { cout << "A" << endl; }
};
struct B : public Printable {
  void Print() const override { cout << "B" << endl; }
};
void Print(const Printable& var) { var.Print(); }
int main() {
  Print(A());
  Print(B());
  return 0;
}
```

Geometry2D and Image

Open3D::Geometry::Geometry2D

```
class Geometry2D {
 public:
  Geometry& Clear() = 0;
  bool IsEmpty() const = 0;
  virtual Eigen::Vector2d GetMinBound() const = 0;
  virtual Eigen::Vector2d GetMaxBound() const = 0;
};
```

Open3D::Geometry::Image

```
class Image : public Geometry2D {
 public:
  Geometry& Clear() override;
  bool IsEmpty() const override;
  virtual Eigen::Vector2d GetMinBound() const override;
  virtual Eigen::Vector2d GetMaxBound() const override;
};
```

Polymorphism

> From Greek **polys**, "many, much"
> and **morphē**, "form, shape"

-Wiki

- Allows morphing derived classes into their base class type:

```
const Base& base = Derived(…)
```

Polymorphism Example 1

```
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }

 protected:
  int width_ = 0;
  int height_ = 0;
};

class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
};
```

Polymorphism Example 1

No real **Polymorphism**, just use all the objects as they are

```
#include <iostream>
using std::cout;
using std::endl;
int main() {
  Square sq(10);
  cout << "Sq:" << sq.width() << " " << sq.height();

  Rectangle rec(10, 15);
  cout << "Rec:" << sq.width() << " " << sq.height();
  return 0;
}
```

Polymorphism Example 2

```
class Rectangle {
 public:
  Rectangle(int w, int h) : width_{w}, height_{h} {}
  int width() const { return width_; }
  int height() const { return height_; }

  void Print() const {
    cout << "Rec:" << width_ << " " << height_ << endl;
  }
```

```
class Square : public Rectangle {
 public:
  explicit Square(int size) : Rectangle{size, size} {}
  void Print() const {
    cout << "Sq:" << width_ << " " << height_ << endl;
  }
};
```

Polymorphism Example 2

Better than manually calling the getter methods, but still need to explicitly call the `Print()` function for each type of object. Again, no real **Polymorphism**

```
int main() {
  Square sq(10);
  sq.Print();

  Rectangle rec(10, 15);
  rec.Print();

  return 0;
}
```

```cpp
  virtual void Rectangle::Print() const {
    cout << "Rec:" << width_ << " " << height_ << endl;
  }
```

```cpp
  void Square::Print() const override {
    cout << "Sq:" << width_ << " " << height_ << endl;
  }
```

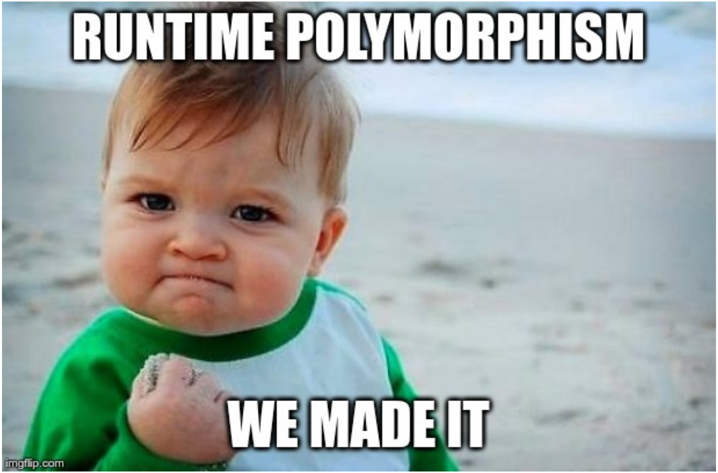
```cpp
void PrintShape(const Rectangle& rec) { rec.Print(); }
```

```cpp
int main() {
  Square sq(10);
  Rectangle rec(10, 15);

  PrintShape(rec);
  PrintShape(sq);

  return 0;
}
```

Now we are using **Runtime Polymorphism**, we are printing shapes to the `std::cout` and deciding at runtime with type of `shape` it is



std::vector<Rectangle>

```
#include <memory>
#include <vector>
using std::make_unique;
using std::unique_ptr;
using std::vector;

int main() {
  vector<unique_ptr<Rectangle>> shapes;
  shapes.emplace_back(make_unique<Rectangle>(10, 15));
  shapes.emplace_back(make_unique<Square>(10));

  for (const auto &shape : shapes) {
    shape->Print();
  }

  return 0;
}
```

When is it useful?

- Allows encapsulating the implementation inside a class only asking it to conform to a common interface
- Often used for:
  - Working with all children of some Base class in unified manner
  - Enforcing an interface in multiple classes to force them to implement some functionality
  - In **strategy** pattern, where some complex functionality is outsourced into separate classes and is passed to the object in a modular fashion

Creating a class hierarchy

- Sometimes classes must form a hierarchy
- Distinguish between **is a** and **has a** to test if the classes should be in one hierarchy:
  - Square **is a** Shape: can inherit from Shape
  - Student **is a** Human: can inherit from Human
  - Car **has a** Wheel: should **not** inherit each other
- GOOGLE-STYLE **Prefer composition**, i.e. including an object of another class as a member of your class
- NACHO-STYLE **Don't get too excited**, use it only when improves code performance/readability.

https://google.github.io/styleguide/cppguide.html#Inheritance

Casting type of variables

- Every variable has a type
- Types can be converted from one to another
- Type conversion is called **type casting**

Casting type of variables

- There are 5 ways of type casting:
  - `static_cast`
  - `reinterpret_cast`
  - `const_cast`
  - `dynamic_cast`
  - `C`-style cast(unsafe), will try to:
    - `const_cast`
    - `static_cast`
    - `static_cast`, then `const_cast` (change type + remove const)
    - `reinterpret_cast`
    - `reinterpret_cast`, then `const_cast` (change type + remove const)

static_cast

- Syntax: static_cast<NewType>(variable)
- Convert type of a variable at compile time
- **Rarely needed to be used explicitly**
- Can happen implicitly for some types, e.g. `float` can be cast to `int`
- Pointer to an object of a Derived class can be **upcast** to a pointer of a Base class
- Enum value can be caster to `int` or `float`
- Full specification is complex!

Full specs: http://en.cppreference.com/w/cpp/language/static_cast

dynamic_cast

- Syntax: `dynamic_cast<Base*>(derived_ptr)`
- Used to convert a pointer to a variable of `Derived` type to a pointer of a `Base` type
- Conversion happens at runtime
- If `derived_ptr` cannot be converted to `Base*` returns a `nullptr`
- GOOGLE-STYLE Avoid using dynamic casting

Full specs: http://en.cppreference.com/w/cpp/language/dynamic_cast

reinterpret_cast

- Syntax:
  `reinterpret_cast<NewType>(variable)`
- Reinterpret the bytes of a variable as another type
- We must know what we are doing!
- Mostly used when writing binary data

Full specs: `http://en.cppreference.com/w/cpp/language/reinterpret_cast`

const_cast

- Syntax: `const_cast<NewType>(variable)`
- Used to “constify” objects
- Used to “de-constify” objects
- Not widely used

Full specs: http://en.cppreference.com/w/cpp/language/const_cast

Google Style

- `GOOGLE-STYLE` Do not use C-style casts. Instead, use these C++-style casts when explicit type conversion is necessary.
- `GOOGLE-STYLE` Use brace initialization to convert arithmetic types (e.g. `int64{x}`). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.

Google Style

- GOOGLE-STYLE Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.

Google Style

- `GOOGLE-STYLE` Use `const_cast` to remove the const qualifier (see const).
- `GOOGLE-STYLE` Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

Using strategy pattern

- If a class relies on complex external functionality use strategy pattern
- Allows to **add/switch functionality** of the class without changing its implementation
- All strategies must conform to one strategy interface

```cpp
class Strategy {
 public:
  virtual void Print() const = 0;
};
```

```cpp
class StrategyA : public Strategy {
 public:
  void Print() const override { cout << "A" << endl; }
};

class StrategyB : public Strategy {
 public:
  void Print() const override { cout << "B" << endl; }
};
```

So far, nothing is new with this source code. We just defined an `interface` and then we derived 2 classes from this `interface` and implemented the virtual methods.

```cpp
class MyClass {
 public:
  explicit MyClass(const Strategy& s) : strategy_(s) {}
  void Print() const { strategy_.Print(); }

 private:
  const Strategy& strategy_;
};
```

- `MyClass` holds a `const reference` to an object of type `Strategy`.
- The strategy will be “picked” when we create an object of the class `MyClass`.
- We don’t need to hold a reference to all the `types` of available strategies.
- The `Print` method has nothing to do with the one we’ve defined in `Strategy`.

- Create two different `strategies` objects

```
StrategyA strategy_a = StrategyA();
StrategyB strategy_b = StrategyB();
```

- Create 2 objects that will use the `Strategy` pattern. We pick which `Print` strategy to use when we construct these objects.

```
MyClass obj_1(strategy_a);
MyClass obj_2(strategy_b);
```

- Use the objects in a " polymorphic" fashion. Both objects will have a `Print` method but they will call different functions according to the `Strategy` we picked when we build the objects.

```
obj_1.Print();
obj_2.Print();
```

Do not overuse it

- Only use these patterns when you need to
- If your class should have a single method for some functionality and will never need another implementation don’t make it virtual
- Used mostly to **avoid copying code** and to **make classes smaller** by moving some functionality out

Singleton Pattern

- We want only one instance of a given `class`.
- Without C++ this would be a `if/else` mess.
- C++ has a powerfull compiler, we can use it.
- We can make sure that nobody creates more than 1 instance of a given class, **at compile time**.
- Don't over use it, it's easy to learn, but usually hides a **design** error in your code.
- Sometimes is still necessary, and makes your code better.
- You need to use it in homework_7.

Singleton Pattern: How?

- We can `delete` any `class` member functions.
- This also holds true for the special functions:
  - `MyClass()`
  - `MyClass(const MyClass& other)`
  - `MyClass& operator=(const MyClass& other)`
  - `MyClass(MyClass&& other)`
  - `MyClass& operator=(MyClass&& other)`
  - `~MyClass()`
- Any `private` function can only be accessed by member of the class.

Singleton Pattern: How?

- Let's **hide** the default `Constructor` and also the `destructor`.

```
class Singleton {
 private:
  Singleton() = default;
  ~Singleton() = default;
};
```

- This completely **disable** the possibility to create a `Singleton` object or destroy it.

Singleton Pattern: How?

- And now let's delete any copy capability:
  - `Copy Constructor`.
  - `Copy Assigment Operator`.

```
class Singleton {
 public:
  Singleton(const Singleton&) = delete;
  void operator=(const Singleton&) = delete;
};
```

- This completely **disable** the possibility to copy any existing `Singleton` object.

Singleton Pattern: What now?

- Now we need to create at least **one** instance of the `Singleton` class.
- **How?** Compiler to the rescue:
  - We can create **one unique** instance of the class.
  - At compile time …
  - Using `static` !.

```
class Singleton {
 public:
  static Singleton& GetInstance() {
    static Singleton instance;
    return instance;
  }
};
```

Singleton Pattern: Completed

```cpp
class Singleton {
 private:
  Singleton() = default;
  ~Singleton() = default;

 public:
  Singleton(const Singleton&) = delete;
  void operator=(const Singleton&) = delete;
  static Singleton& GetInstance() {
    static Singleton instance;
    return instance;
  }
};
```

Singleton Pattern: Usage

```cpp
#include "Singleton.hpp"

int main() {
  auto& singleton = Singleton::GetInstance();
  // ...
  // do stuff with singleton, the only instance.
  // ...

  Singleton s1;                // Compiler Error!
  Singleton s2(singleton);     // Compiler Error!
  Singleton s3 = singleton;    // Compiler Error!

  return 0;
}
```

CRPT Pattern

```cpp
#include <boost/core/demangle.hpp>
using boost::core::demangle;

template <typename T>
class Printable {
 public:
  explicit Printable() {
    // Always print its type when created
    cout << demangle(typeid(T).name()) << " created\n";
  }
};

class Example1 : public Printable<Example1> {};
class Example2 : public Printable<Example2> {};
class Example3 : public Printable<Example3> {};
```

CRPT Pattern

Usage:

```cpp
int main() {
  const Example1 obj1;
  const Example2 obj2;
  const Example3 obj3;
  return 0;
}
```

Output:

```
Example1 Created
Example2 Created
Example3 Created
```

Suggested Video

Object Oriented Design

https://youtu.be/pTB0EiLXUC8

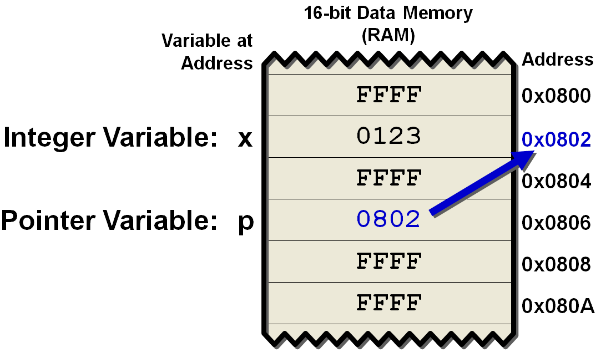
Suggested Video

Polymorphism

https://youtu.be/bP-Trkf8hNA

Must Watch

Raw Pointers: Skip min 30



https://www.youtube.com/watch?v=mIrOcFf2crk&t=1729s

Image Courtesy of Micriochip

References

- **Object Oriented Design**
  - https://en.cppreference.com/w/cpp/language/derived_class
  - https://en.cppreference.com/w/cpp/language/virtual
  - https://en.cppreference.com/w/cpp/language/abstract_class
  - https://en.cppreference.com/w/cpp/language/override
  - https://en.cppreference.com/w/cpp/language/final
  - https://en.cppreference.com/w/cpp/language/friend
- **Type Conversion**
  - https://en.cppreference.com/w/cpp/language/static_cast
  - https://en.cppreference.com/w/cpp/language/dynamic_cast
  - https://en.cppreference.com/w/cpp/language/reinterpret_cast
  - https://en.cppreference.com/w/cpp/language/const_cast

References: Patterns

- **Strategy Pattern**
  - https://en.wikipedia.org/wiki/Strategy_pattern
  - https://refactoring.guru/design-patterns/strategy/cpp/example
  - https://stackoverflow.com/a/1008289/11525517
- **Singleton Pattern**
  - https://en.wikipedia.org/wiki/Singleton_pattern
  - https://refactoring.guru/design-patterns/singleton/cpp/example
- **CRPT Pattern**
  - https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern