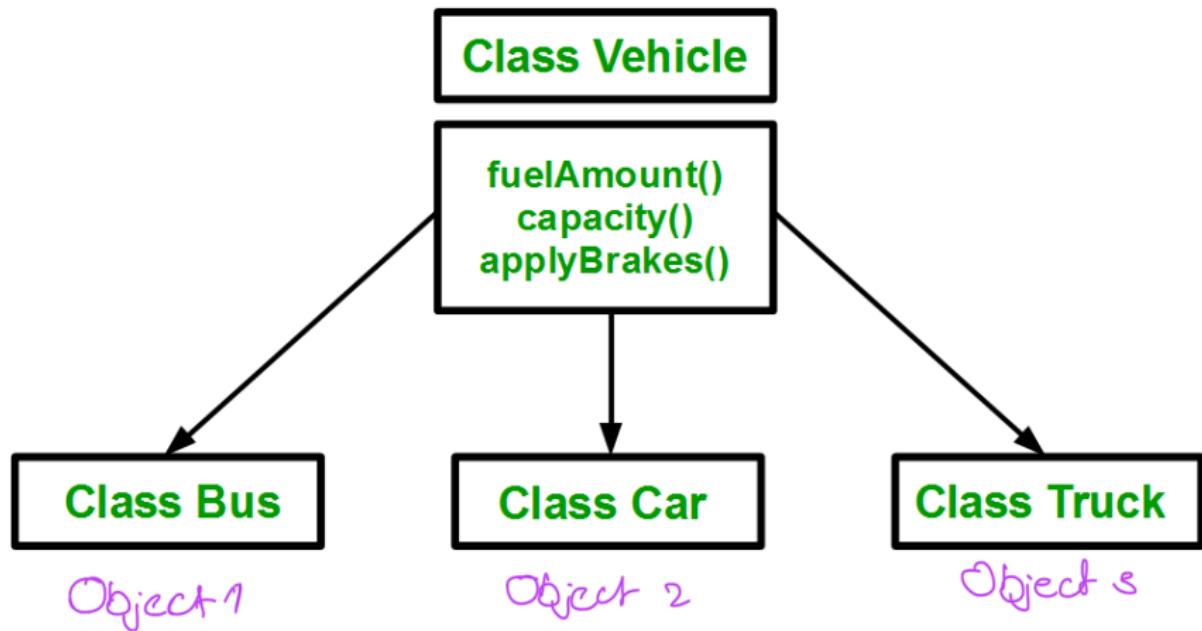


Modern C++ for Computer Vision and Image Processing

Lecture 7: Object Oriented Design

Ignacio Vizzo and Cyrill Stachniss

Inheritance



C vs C++ Inheritance Example

C Code

```
1 // "Base" class, Vehicle
2 typedef struct vehicle {
3     int seats_;          // number of seats on the vehicle
4     int capacity_;       // amount of fuel of the gas tank
5     char* brand_;        // make of the vehicle :-)
6 } vehicle_t;
```

C++ Code

```
1 class Vehicle {
2     private:
3         int seats_ = 0;          // number of seats on the vehicle
4         int capacity_ = 0;       // amount of fuel of the gas tank
5         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

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`

```
1 class Derived : public Base {  
2     // Contents of the derived class.  
3 };
```

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

```
1 #include <iostream>
2 using std::cout; using std::endl;
3 class Rectangle {
4 public:
5     Rectangle(int w, int h) : width_{w}, height_{h} {}
6     int width() const { return width_; }
7     int height() const { return height_; }
8 protected:
9     int width_ = 0;
10    int height_ = 0;
11 };
12 class Square : public Rectangle {
13 public:
14     explicit Square(int size) : Rectangle{size, size} {}  
15 };           → no type convert
16 int main() {
17     Square sq(10); // Short name to save space.
18     cout << sq.width() << " " << sq.height() << endl;
19     return 0;
20 }
```

Function overriding

- A function can be declared virtual ✓ Base

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

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

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

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

```
1 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

ಹಾಡು: Language ಯೂರೋಪನ್ ವೆಸ್ಟ್‌ಬ್ರೇಡ್ ಭಾಷೆಗಳಲ್ಲಿ
C++ ಕ್ವಾಡೆನ್

- **Abstract class:** ಏರಿಯಂತೆ pure virtual function ಹಾಡಿ
ಅಂದಿನ ಪಿಂಡ Class ನ್ನಿಂದಾಗಿ.
- **Interface:** Class ಹಿನ್ನ ಅರ್ಥದಲ್ಲಿ ಎಂಬುದಕ್ಕೆ pure
virtual functions ರೂಪ ಅರ್ಥಿಸಿ: ಹಾಡು: Data Members ರೂಪ
ಅಂದಿ, ಎಲ್ಲಿ ಇರುತ್ತಿರುವುದಕ್ಕೆ Interface ನೀಡಿರುತ್ತಾಗೆ.

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

enforce message Interface

- 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

Calls may function of call \in Implementation
నీ ఉత్సవాలు

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 struct Printable { // Saving space. Should be a class.
5     virtual void Print() const = 0;
6 };
7 struct A : public Printable {
8     void Print() const override { cout << "A" << endl; }
9 };
10 struct B : public Printable {
11     void Print() const override { cout << "B" << endl; }
12 };
13 void Print(const Printable& var) { var.Print(); }
14 int main() {
15     Print(A());
16     Print(B());
17     return 0;
18 }
```

int var

Geometry2D and Image

Open3D::Geometry::Geometry2D

```
1 class Geometry2D {  
2     public:  
3         Geometry& Clear() = 0;  
4         bool IsEmpty() const = 0;  
5         virtual Eigen::Vector2d GetMinBound() const = 0;  
6         virtual Eigen::Vector2d GetMaxBound() const = 0;  
7 };
```

return

only

must override

Open3D::Geometry::Image

```
1 class Image : public Geometry2D {  
2     public:  
3         Geometry& Clear() override;  
4         bool IsEmpty() const override;  
5         virtual Eigen::Vector2d GetMinBound() const override;  
6         virtual Eigen::Vector2d GetMaxBound() const override;  
7 };
```

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

```
1 class Rectangle {
2     public:
3         Rectangle(int w, int h) : width_{w}, height_{h} {}
4         int width() const { return width_; }
5         int height() const { return height_; }
6
7     protected:
8         int width_ = 0;
9         int height_ = 0;
10    };
11
12 class Square : public Rectangle {
13     public:
14         explicit Square(int size) : Rectangle{size, size} {}
15    };

```

Polymorphism Example 1

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

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4 int main() {
5     ✓ Square sq(10);
6     cout << "Sq:" << sq.width() << " " << sq.height();
7
8     ✓ Rectangle rec(10, 15);
9     cout << "Rec:" << sq.width() << " " << sq.height();
10    return 0;
11 }
```

Polymorphism මෙයින් Object නෑරුව සඳහා පෙන්වනු ලබයි

Polymorphism Example 2

```
1 class Rectangle {
2 public:
3     Rectangle(int w, int h) : width_{w}, height_{h} {}
4     int width() const { return width_; }
5     int height() const { return height_; }
6
7     ✓ void Print() const {
8         cout << "Rec:" << width_ << " " << height_ << endl;
9     }

```

```
1 class Square : public Rectangle {
2 public:
3     explicit Square(int size) : Rectangle{size, size} {}
4     ✓ void Print() const {
5         cout << "Sq:" << width_ << " " << height_ << endl;
6     }
7 };

```

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**

```
1 int main() {
2     Square sq(10);
3     sq.Print();
4
5     Rectangle rec;
6     rec.Print();
7
8     return 0;
9 }
```

ମୋରଗୁଡ଼ିକ ରବ୍‌ନାର୍ଫ୍ରିଆ
con: ଶେରି.

```
1 virtual void Rectangle::Print() const {
2     cout << "Rec:" << width_ << " " << height_ << endl;
3 }
```



```
1 void Square::Print() const override {
2     cout << "Sq:" << width_ << " " << height_ << endl;
3 }
```

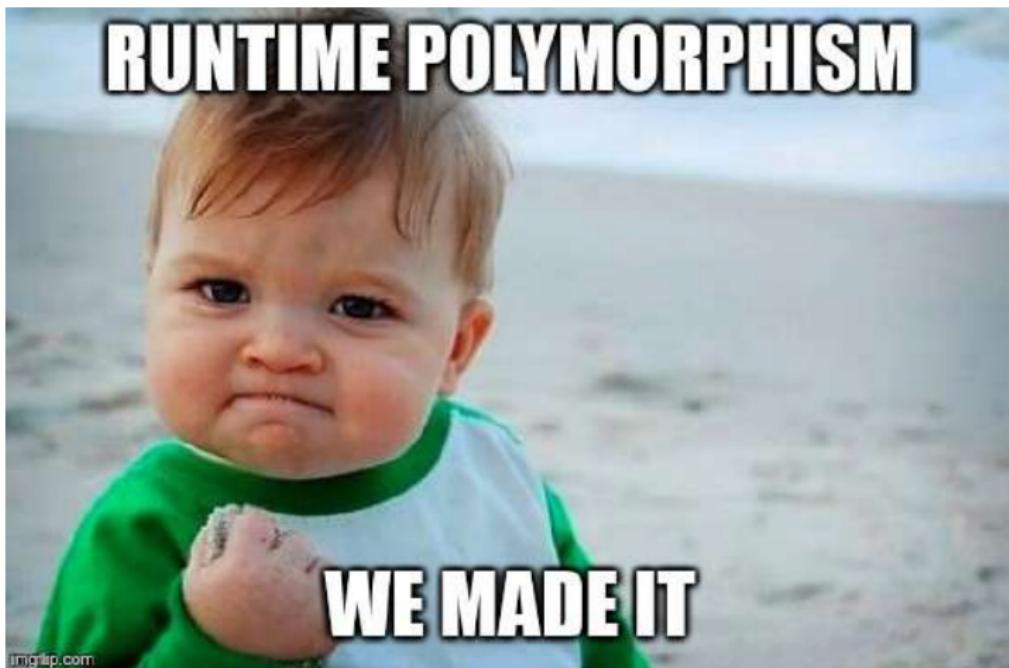


```
1 void PrintShape(const Rectangle& rec) { rec.Print(); }
```



```
1 int main() {
2     Square sq(10);
3     Rectangle rec(10, 15);
4
5     PrintShape(rec);
6     PrintShape(sq);
7
8     return 0;
9 }
```

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>

```
1 #include <memory>
2 #include <vector>
3 using std::make_unique;
4 using std::unique_ptr;
5 using std::vector;
6
7 int main() {
8     vector<unique_ptr<Rectangle>> shapes;
9     shapes.emplace_back(make_unique<Rectangle>(10, 15));
10    shapes.emplace_back(make_unique<Square>(10));
11
12    for (const auto &shape : shapes) {
13        shape->Print();
14    }
15
16    return 0;
17 }
```

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.

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 cast to `int` or `float`
-  Full specification is complex!

Object Pointer may Base Class Obj Pointc may upcast

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

15:50:45

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

Derived Class Object Pointer ~~not~~ Base Class Object Pointer ~~to~~
cast ~~never~~. ~~knows.~~ runtime ~~problem~~

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

ಸಂಖ್ಯೆಗಳನ್ನು ಬೇರೆ ರೀತಿಯಲ್ಲಿ ಪ್ರತಿಕ್ರಿಯಾಪಿಸಬಹುದು.

Binary Data ಯಾಗ್ನಿ ಕಾಣಬೇಕು.

ಹಿಂದಿನ ವರ್ಣನೆಯನ್ನು ಮಾಡಬಹುದು.

Binary Data ಯಾಗ್ನಿ ಹಿಂದಿನ ವರ್ಣನೆ.

const_cast

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

→ Constant data type \Rightarrow Non Constant datatype
→ Non Constant \Rightarrow Constant \Rightarrow const_cast

↳ same way as \rightarrow

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.

explicit \Rightarrow Compiler rejects type conversion if conversion is ambiguous.

Brace Initialization \Rightarrow no conversion, conversion is safe & unambiguous.

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

```
1 class Strategy {  
2     public:  
3         virtual void Print() const = 0;  
4 };
```



```
1 class StrategyA : public Strategy {  
2     public:  
3         void Print() const override { cout << "A" << endl; }  
4 };  
5  
6 class StrategyB : public Strategy {  
7     public:  
8         void Print() const override { cout << "B" << endl; }  
9 };
```

Interface

(2) derived class

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.

```
1 class MyClass {  
2     public:  
3         explicit MyClass(const Strategy& s) : strategy_(s) {}  
4         void Print() const { strategy_.Print(); }  
5  
6     private:  
7         const Strategy& strategy_;  
8 };
```

- ② ■ 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

```
1  StrategyA strategy_a = StrategyA();  
2  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.

```
1  MyClass obj_1(strategy_a);  
2  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.

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



Do not overuse it

မရှိနေမည်။

- 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

避免複製複雜的〈Code〉

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.

```
1 class Singleton {  
2     private:  
3         Singleton() = default;  
4         ~Singleton() = default;  
5     };
```

Private හිඳුවේ Hide නිෂ්පාදන.

]

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

එකු මූල්‍ය නි ප්‍රමාණ තැබුමෙන් දෙන
සුත්‍ර ප්‍රතිඵලීය ප්‍රමාණ තැබුමෙන් දෙන

Singleton Pattern: How?

- And now let's delete any copy capability:

- Copy Constructor.

- Copy Assignment Operator.

Copy ആവശ്യമുള്ള കോണ്ടലറ്റ്
ഡേൽ.

```
1 class Singleton {  
2     public:  
3         Singleton(const Singleton&) = delete;  
4         void operator=(const Singleton&) = delete;  
5     };
```

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

Singleton Pattern: What now?

Create Instance

- 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** !.

```
1 class Singleton {  
2     public:  
3         static Singleton& GetInstance() {  
4             static Singleton instance;  
5             return instance;  
6         }  
7     };  
8 }
```



Singleton Pattern: Completed

```
1 class Singleton {  
2     private:  
3         Singleton() = default; ✓  
4         ~Singleton() = default; ✓  
5  
6     public:  
7         Singleton(const Singleton&) = delete; ✓  
8         void operator=(const Singleton&) = delete; ✓  
9         static Singleton& GetInstance() {  
10             static Singleton instance; ✓  
11             return instance;  
12         }  
13     };
```

Singleton Pattern: Usage



```
1 #include "Singleton.hpp"
2
3 int main() {
4     auto& singleton = Singleton::GetInstance();
5     // ...
6     // do stuff with singleton, the only instance.
7     // ...
8
9     Singleton s1;                      // Compiler Error!
10    Singleton s2(singleton);          // Compiler Error!
11    Singleton s3 = singleton;         // Compiler Error! +
```

TP ~~CRPT~~ Pattern

Curiously Recurring template pattern
(CRTP)

```
1 #include <boost/core/demangle.hpp>
2 using boost::core::demangle;
3
4 template <typename T>
5 class Printable {
6 public:
7     explicit Printable() {
8         // Always print its type when created
9         cout << demangle(typeid(T).name()) << " created\n";
10    }
11 };
12
13 class Example1 : public Printable<Example1> {};
14 class Example2 : public Printable<Example2> {};
15 class Example3 : public Printable<Example3> {};
```

compile time polymorphism

Usage:

```
1 int main() {  
2     const Example1 obj1;  
3     const Example2 obj2;  
4     const Example3 obj3;  
5     return 0;  
6 }
```

Output:

```
1 Example1 Created  
2 Example2 Created  
3 Example3 Created
```