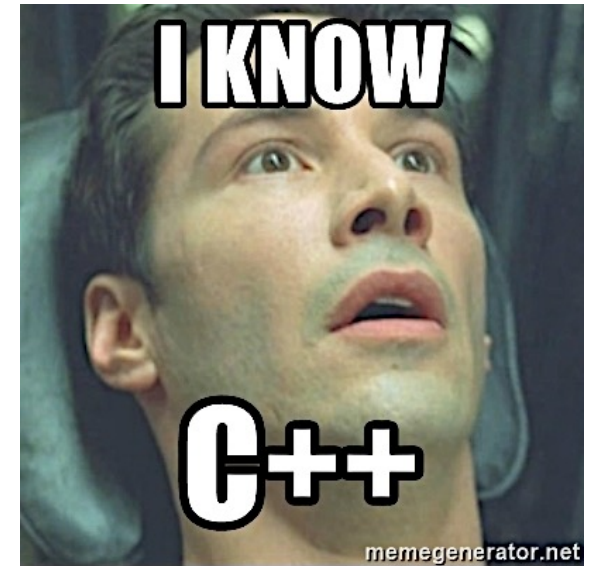
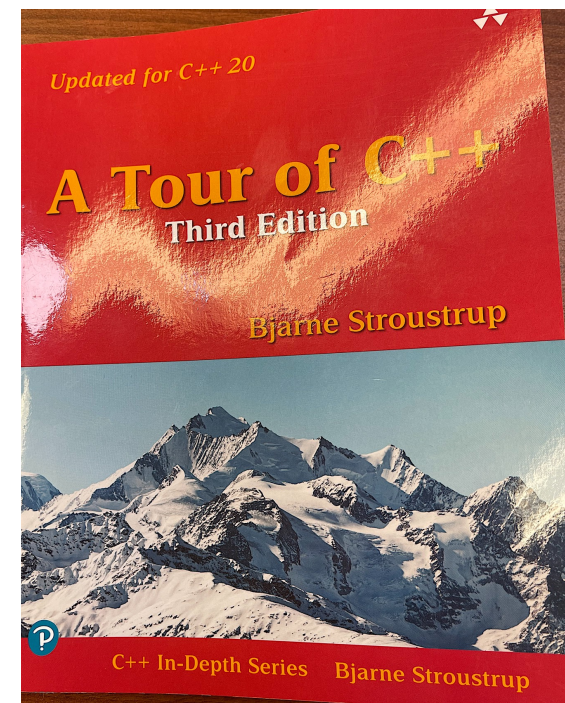


Lecture 24: C++ part IX



Suggested Literature



RECAP: C++ Classes

```
class Human {
public:
    Human(const string name, array<int, 3> bd = {0,0,-1});

    void SetBirthday(const int month, const int day,
                    const int year);

    void SetBirthday(array<int, 3> x);

    void SetGender(const Genders x);

    void AddNationality(const int x);

    array<int, 3> GetBirthday() const;

    Genders GetGender() const;

    int GetAge() const;

    list<int> GetNationality() const;

    void Print() const;
private:
    bool is_birthday_set_ = false;
    bool is_gender_set_ = false;
    bool is_min_one_nationality_set_ = false;
    int age_ = 0;
    Genders gender_;
    array<int, 3> birthday_ = {0, 0, -1};
    list<int> nationality_;
    list<string> name_;
};
```

C++ Class v0.8

Getting closer to the
v1.0 release of the
Human class!

Functions that do not
change member
variables should have
the keyword **const**
(const correctness)

RECAP: Class Inheritance

```
#include <string>
#include <iostream>
using namespace std;

class Base {
    int x_ = 2; // default visibility is private
public:
    void print(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10; // default visibility is private
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main () {
    Derived x;
    x.print(100);

    // const here is mandatory (compiler error)
    const Base& y = static_cast<Base>(x);
    y.print(100);

    Base* z = static_cast<Base*>(&x);
    z->print(110); // pointers use -> instead of .

    Base w = static_cast<Base>(x); // copy (only parent)!
    w.print(120);
}
```

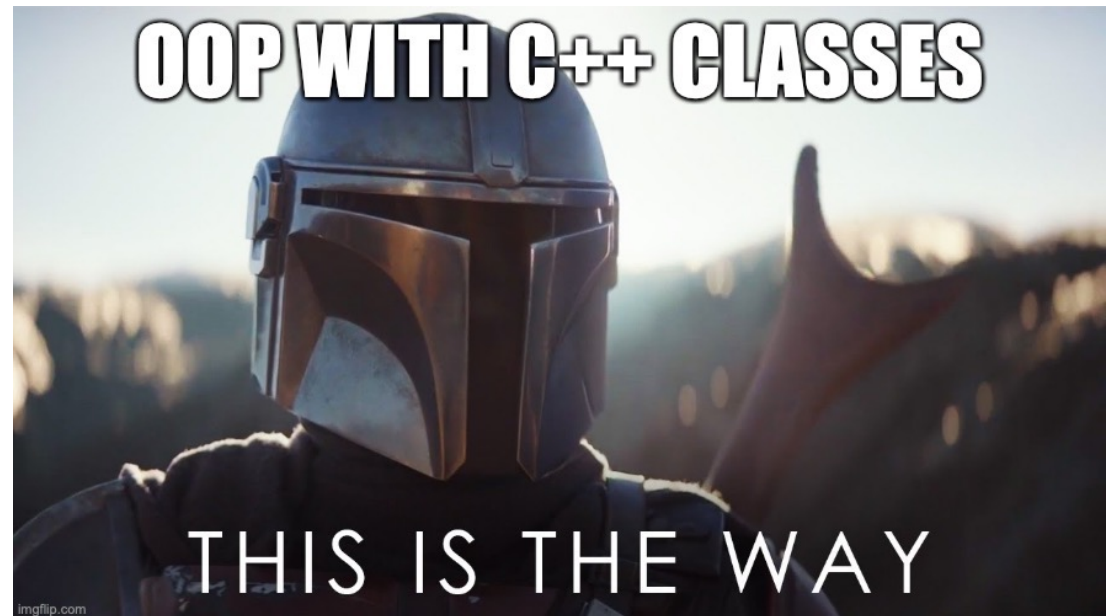
Child and parent classes
can have member
functions with the same
name and arg. types (no
overloading).

```
whovian@phy504:~/host/CPP$ g++ class10.cpp -o class10 -std=c++20
whovian@phy504:~/host/CPP$ ./class10
PrintDerived: 90
PrintBase: 98
PrintBase: 108
PrintBase: 118
```

Access to the parent's
version happens via
casting

C++ Classes Part II

```
class Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
class Derived : public Base {  
    public:  
        void print() {  
            // code  
        }  
};  
  
int main() {  
    Derived derived1;  
    derived1.print();  
    return 0;  
}
```



```

#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase: " << tmp << endl;
    }
private:
    int x_ = 2;
};

class Derived : public Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived: " << tmp << endl;
    }
private:
    int x_ = 10;
};

int main ()
{
    Base y;
    y.print(100);

    Derived x;
    x.print(100);
}

```

Class Inheritance

Constructors (non-default)

What happens if the user defines a **non-default constructor** (constructor w/ no args/parameters)?

Remember that if the user defines a constructor that requires an argument, the **default constructor** is not accessible (unless explicitly coded)

Class Inheritance

Remember: you can't access **Base** private members from **Derived**

Therefore, we need to delegate constructors (call the **Base** constructor from the **Derived** constructor)

```
#include <string>
#include <iostream>
using namespace std;

class Base {
    int x_ = 2;
public:
    Base(const double x) :
        x_(x)
    {}
    void print(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10;
public:
    Derived(const double x, const double y) :
        Base(x), ← Call Base from Derived
        x_(y)
    {}
    void print2(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main ()
{
    Derived x(10, 30);
    x.print(100);
    x.print2(100);
}
```

```
whovian@phy504:~/host/CPP$ nano class13.cpp
whovian@phy504:~/host/CPP$ g++ class13.cpp -o class13 -std=c++20
whovian@phy504:~/host/CPP$ ./class13
PrintBase: 90
PrintDerived: 70
```

Class Inheritance

Remember: you can't
access **Base** private
members from **Derived**

Therefore, we need to
delegate constructors (call
the **Base** constructor from
the **Derived** constructor)

```
#include <string>
#include <iostream>
using namespace std;

class Mom {
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
    Mom(const double x) :
        x_(x) {}
};

class Dad {
    int x_ = 10;
public:
    void print2(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
    Dad(const double x) :
        x_(x) {}
};

class Derived : public Mom, public Dad {
    int x_ = 50;
public:
    Derived(const double x, const double y, const double z) :
        Mom(x),
        Dad(y),
        x_(z) {}
    void print3(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main () {
    Derived x(10, 20, 30);
    x.print(100);
    x.print2(100);
    x.print3(100);
}
```

```
whovian@phy504:~/host/CPP$ g++ class14.cpp -o class14 -std=c++20
whovian@phy504:~/host/CPP$ ./class14
PrintBase: 90
PrintBase: 80
PrintDerived: 70
```

Class Inheritance

What if you want to cast **Derived** to **base** (at the pointer level) but continue to use the **Derived** version of the member function?

Then, you need to make the member functions **virtual**

```
#include <iostream>
using namespace std;

class Base {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
    void print2(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
    void print2(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main () {
    Derived x;

    // first time we use dynamic cast
    Base* y = dynamic_cast<Base*>(&x);

    // 1) Virtual function (runtime binding)
    y->print(100);
    // 2) Non-virtual function (compilation binding)
    y->print2(100);
}
```



```
whovian@phy504:~/host/CPP$ g++ class15.cpp -o class15 -std=c++20
whovian@phy504:~/host/CPP$ ./class15
PrintDerived: 90
PrintBase: 98
```

```

#include <iostream>
using namespace std;

class Base {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

class Derived2 : public Base {
    int x_ = 50;
public:
    void print(const double x) const {
        cout << "PrintDerived2: " << x-this->x_ << endl;
    }
};

int main () {
    Derived x;

    Base* y = dynamic_cast<Base*>(&x); // good cast
    if (y == nullptr)
        cout << "dynamic_cast<Base*> failed" << endl;

    Derived2* z = dynamic_cast<Derived2*>(&x); // bad cast
    if (z == nullptr)
        cout << "dynamic_cast<Derived2*> failed" << endl;
}

```

```

whovian@phy504:~/host/CPP$ g++ class16.cpp -o class16 -std=c++20
class16.cpp: In function 'int main()':
class16.cpp:33:17: warning: 'dynamic_cast<class Derived2*>(Derived x)' can never succeed
   33 |     Derived2* z = dynamic_cast<Derived2*>(&x); // bad cast
       |                               ^~~~~~
whovian@phy504:~/host/CPP$ ./class16
dynamic_cast<Derived2*> failed

```

Class Inheritance

What happens when
dynamic_cast fails
 (illegal casting attempt)?
 returns **nullptr**

Compiler may catch it as
 a warning (see above)

Class Inheritance

What happens with
dynamic_cast when
you have child and
grandchild classes?

Only the **Mom** class
needs the word **virtual**

```
#include <iostream>
using namespace std;

class Mom {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintMom: " << x-this->x_ << endl;
    }
    virtual ~Mom() = default; // required
};

class Child : public Mom {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintChild: " << x-this->x_ << endl;
    }
};

class GrandChild : public Child {
    int x_ = 20;
public:
    void print(const double x) const {
        cout << "PrintGrandChild: " << x-this->x_ << endl;
    }
};

int main () {
    Child x;
    Mom* y = dynamic_cast<Mom*>(&x);
    y->print(100);

    GrandChild x2;
    Mom* y2 = dynamic_cast<Mom*>(&x2);
    y2->print(100);
}
```

```
whovian@phy504:~/host/CPP$ g++ class17.cpp -o class17 -std=c++20
whovian@phy504:~/host/CPP$ ./class17
PrintChild: 90
PrintGrandChild: 80
```

```

#include <iostream>
using namespace std;

class Mom {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintMom: " << x-this->x_ << endl;
    }
    virtual ~Mom() = default; // required
};

class Child : public Mom {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintChild: " << x-this->x_ << endl;
    }
};

int main () {
    Child x;
    // Explicit dynamic upcasting
    Mom* y = dynamic_cast<Mom*>(&x);
    y->print(100);
    // Upcasting can be implicit with pointers (100% ok)
    Mom* y2 = &x;
    y2->print(100);
    // Reference dynamic upcasting also ok
    Mom& y3 = x;
    y3.print(100);
    // downcasting needs to be explicit
    Child* y4 = dynamic_cast<Child*>(y);
    y4->print(100);
    // Copy: where slicing happens
    // (lost information about derived)
    Mom y5 = x;
    y5.print(100);
}

```

Class Inheritance

Biggest worry: object slicing

You make a **copy** while also casting from derived to base

You lose all information stored in the derived class.

All the **dynamic_cast** game happens with pointer casting (not copy)

```

whovian@phy504:~/host/Lecture24$ nano class17v2.cpp
whovian@phy504:~/host/Lecture24$ g++ class17v2.cpp -o c17v2 -std=c++26
whovian@phy504:~/host/Lecture24$ ./c17v2
PrintChild: 90
PrintChild: 90
PrintChild: 90
PrintChild: 90
PrintMom: 98

```

Class Inheritance

Biggest worry: object slicing

You make a **copy** while also casting from derived to base

Is there a way to copy and cast without slicing (so the output is still 90 even though we cast to **Mom** base during copy? Yes (in **C++26**)

```
#include <iostream>
#include <memory>      // for std::polymorphic in C++26
#include <vector>
using namespace std;

class Mom {
    int x_ = 2;
public:
    virtual void print(const double x) const {
        cout << "PrintMom: " << x-this->x_ << endl;
    }
    virtual ~Mom() = default;
};

class Child : public Mom {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintChild: " << x-this->x_ << endl;
    }
};

int main () {
    Child x;
    // Copy: where slicing happens
    Mom y1 = x;
    y1.print(100);

    // Copy with without slicing
    std::polymorphic<Mom> y2{x};
    y2->print(100);

    // interesting fact y2 = x not allowed
    // in C++26 (constructor is "explicit")
    // but the code below is allow (explicit conversion)
    std::polymorphic<Mom> y3 = std::polymorphic<Mom>{x};
    y3->print(100);

    // polymorphism preserved
    std::vector<std::polymorphic<Mom>> v1;
    v1.emplace_back(Child{});
    v1.emplace_back(Mom{});
    for (const auto& a : v1) a->print(100);
}
```

```
whovian@phy504:~/host/Lecture24$ /opt/gcc-latest/bin/g++ class17v3.cpp -o c17v3 -std=c++26
whovian@phy504:~/host/Lecture24$ ./c17v3
PrintMom: 98
PrintChild: 90
PrintChild: 90
PrintChild: 90
PrintChild: 90
PrintMom: 98
```

```

#include <string>
#include <iostream>

using namespace std;

class Base {
protected:
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintBase: " << x-this->x_ << endl;
    }
};

class Derived : public Base {
public:
    void print2(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main ()
{
    Derived x;
    x.print(100);
    x.print2(100);
}

```

```

whovian@phy504:~/host/CPP$ g++ class19.cpp -o class19 -std=c++20
whovian@phy504:~/host/CPP$ ./class19
PrintBase: 98
PrintDerived: 98

```

Class Inheritance

Between **public:** and **private:** - is there a middle ground?

Yes! **protected:** (not public, but accessible by **Derived** class)

```

#include <string>
#include <iostream>
using namespace std;

class God {
protected:
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintGod: " << x-this->x_ << endl;
    }
};

class Mom : public God {
public:
    void print(const double x) const {
        cout << "PrintMom: " << x-this->x_ << endl;
    }
};

class Dad : public God {
public:
    void print(const double x) const {
        cout << "PrintDad: " << x-this->x_ << endl;
    }
};

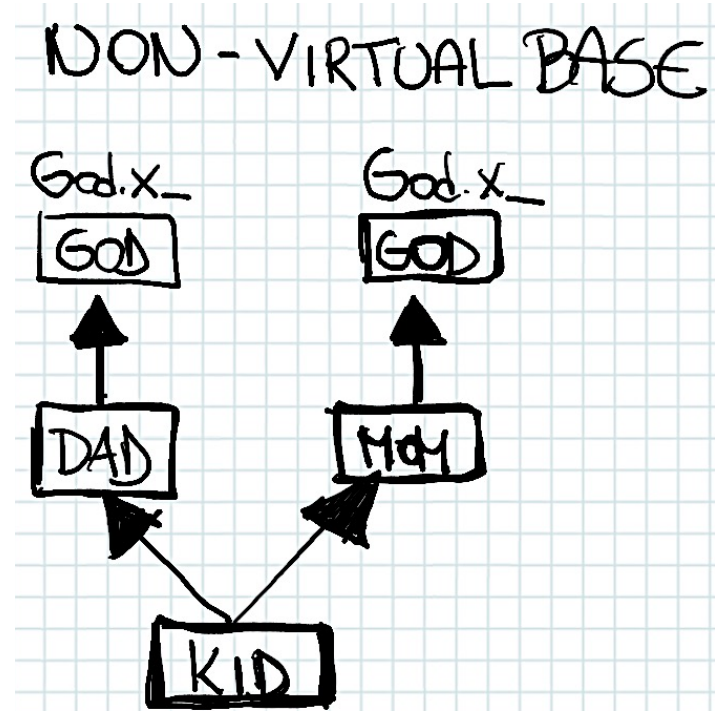
class Kid : public Mom, public Dad {
public:
    void print(const double x) const {
        cout << "PrintKid: " << x-this->x_ << endl;
    }
};

int main () {
    Kid x; // Error: with Mom and Dad having
           // independent God classes, which x_
           // should kids.print() prints?
    x.print(100);
}

```

Class Inheritance

Non-virtual Base Class



Code doesn't compile

```

whovian@phy504:~/host/CPP$ g++ class20.cpp -o class20 -std=c++20
class20.cpp: In member function 'void Kid::print(double) const':
class20.cpp:28:37: error: request for member 'x_' is ambiguous
   28 |         cout << "PrintKid: " << x-this->x_ << endl;
      |                                     ^~
class20.cpp:7:7: note: candidates are: 'int God::x_'
    7 |     int x_ = 2;
      |     ^~
class20.cpp:7:7: note:         'int God::x_'

```

Class Inheritance

Virtual Base Class

```
#include <string>
#include <iostream>
using namespace std;

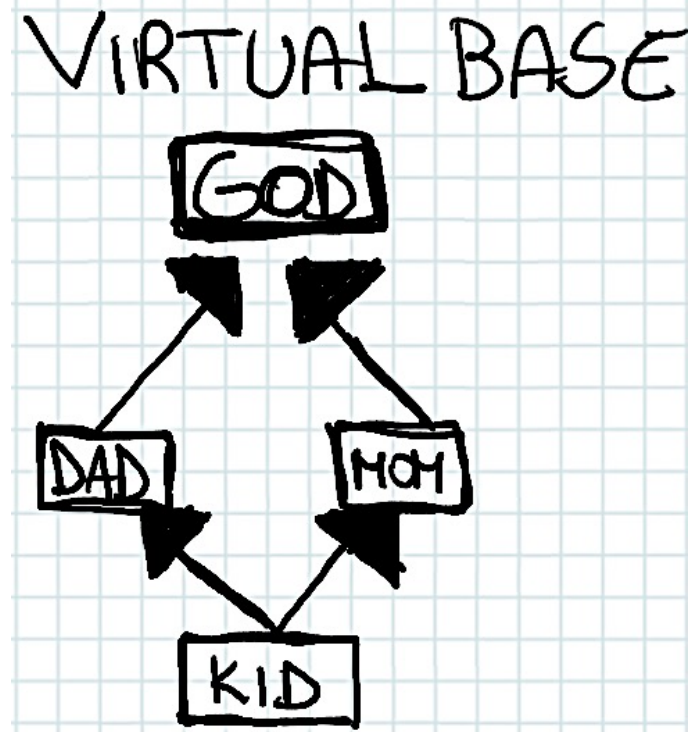
class God {
protected:
    int x_ = 2;
public:
    void print(const double x) const {
        cout << "PrintGod: " << x-this->x_ << endl;
    }
};

class Mom : public virtual God {
public:
    void print(const double x) const {
        cout << "PrintMom: " << x-this->x_ << endl;
    }
};

class Dad : public virtual God {
public:
    void print(const double x) const {
        cout << "PrintDad: " << x-this->x_ << endl;
    }
};

class Kid : public Mom, public Dad {
public:
    void print(const double x) const {
        cout << "PrintKid: " << x-this->x_ << endl;
    }
};

int main () {
    Kid x;
    x.print(100);
}
```



Code now runs no problem

```
whovian@phy504:~/host/CPP$ g++ class21.cpp -o class21 -std=c++20
whovian@phy504:~/host/CPP$ ./class21
PrintKid: 98
```

Class Inheritance

Abstract **Base** Class

Abstract base class
cannot be created,
only **dynamically**
cast from **derived**
class

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
    y->print(100);
    // Base x2; Illegal line of C++
}
```

```
whovian@phy504:~/host/CPP$ g++ class18.cpp -o class18 -std=c++20
whovian@phy504:~/host/CPP$ ./class18
PrintDerived: 90
```

Class Inheritance

Abstract **Base** Class

Abstract base class
cannot be created, only
dynamically cast from
derived class

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    void print(const double x) const {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

void myglobal(Base* y) {
    y->print(100);
}

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);

    myglobal(y);
    // you can cast directly from new
    Base* y2 = new Derived();
    myglobal(y2);
}
```

```
whovian@phy504:~/host/CPP$ g++ class18v2.cpp -o class18v2 -std=c++20
whovian@phy504:~/host/CPP$ ./class18v2
PrintDerived: 90
PrintDerived: 90
```

Class Inheritance

If you are 100% sure the derived class member function must override a virtual function on Base, you can use the optional keyword **override** for additional checks

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    // Error: no print2 function on Base (override)
    void print2(const double x) const override {
        cout << "PrintDerived: " << x-this->x_ << endl;
    }
};

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
}
```

```
whovian@phy504:~/host/CPP$ g++ class18v3.cpp -o class18v3 -std=c++20
class18v3.cpp:14:8: error: 'void Derived::print2(double) const' marked 'override', but does not override
   14 |     void print2(const double x) const override {
      |         ~~~~~
class18v3.cpp: In function 'int main()':
class18v3.cpp:20:11: error: cannot declare variable 'x' to be of abstract type 'Derived'
   20 |     Derived x;
      |         ^
class18v3.cpp:10:7: note: because the following virtual functions are pure within 'Derived':
   10 | class Derived : public Base {
      |         ~~~~~
class18v3.cpp:7:16: note: 'virtual void Base::print(double) const'
    7 |     virtual void print(const double x) const = 0;
      |         ~~~~~
```

Class Inheritance

If you are 100% sure the derived class member function must override a virtual function on Base, you can use the optional keyword **override** for additional checks

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    // Error: wrong overloading (base version has arg)
    void print() const override {
        cout << "PrintDerived: " << 10-this->x_ << endl;
    }
};

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
}
```

```
whovian@phy504:~/host/CPP$ g++ class18v4.cpp -o class18v4 -std=c++20
class18v4.cpp:14:8: error: 'void Derived::print() const' marked 'override', but does not override
 14 |     void print() const override {
    |         ~~~~~
class18v4.cpp: In function 'int main()':
class18v4.cpp:20:11: error: cannot declare variable 'x' to be of abstract type 'Derived'
 20 |     Derived x;
    |         ^
class18v4.cpp:10:7: note: because the following virtual functions are pure within 'Derived':
 10 |     class Derived : public Base {
    |         ~~~~~
class18v4.cpp:7:16: note:     'virtual void Base::print(double) const'
   7 |     virtual void print(const double x) const = 0;
    |         ~~~~~
```

Class Inheritance

If you are 100% sure the derived class member function must override a virtual function on Base, you can use the optional keyword **override** for additional checks

```
#include <iostream>
using namespace std;

class Base {
public:
    // = 0 required to indicate abstract function
    virtual void print(const double x) const = 0;
};

class Derived : public Base {
    int x_ = 10;
public:
    // Error: wrong overloading again (forgot: const)
    void print(const double x) override {
        cout << "PrintDerived: " << x->x_ << endl;
    }
};

int main () {
    Derived x;
    Base* y = dynamic_cast<Base*>(&x);
}
```

```
whovian@phy504:~/host/CPP$ g++ class18v5.cpp -o class18v5 -std=c++20
class18v5.cpp:14:8: error: 'void Derived::print(double)' marked 'override', but does not override
 14 |     void print(const double x) override {
    |           ^~~~~~
class18v5.cpp: In function 'int main()':
class18v5.cpp:20:11: error: cannot declare variable 'x' to be of abstract type 'Derived'
 20 |     Derived x;
    |           ^
class18v5.cpp:10:7: note: because the following virtual functions are pure within 'Derived':
 10 | class Derived : public Base {
    |       ^~~~~~
class18v5.cpp:7:16: note: 'virtual void Base::print(double) const'
  7 |     virtual void print(const double x) const = 0;
    |           ^~~~~~
```

Function overloading

Member function
overloading rules may
surprise you.

They don't work
automatically across scopes
(from base to derived)

```
#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const int x) const {
        const int tmp = x - this->x_;
        cout << "PrintBase (int): " << tmp << endl;
    }

    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase (double): " << tmp << endl;
    }
protected:
    int x_ = 2;
};

class Derived : public Base {
public:
    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived (double): " << tmp << endl;
    }
};

int main ()
{
    Base y;
    y.print(100.1);
    y.print(5);

    Derived x;
    x.print(100.1);
    x.print(5);
}
```

```
whovian@phy504:~/host/CPP$ g++ class28.cpp -o class28 -std=c++20
whovian@phy504:~/host/CPP$ ./class28
PrintBase (double): 98.1
PrintBase (int): 3
PrintDerived (double): 98.1
PrintDerived (double): 3
```

```

#include <string>
#include <iostream>
using namespace std;

class Base {
public:
    void print(const int x) const {
        const int tmp = x - this->x_;
        cout << "PrintBase (int): " << tmp << endl;
    }

    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintBase (double): " << tmp << endl;
    }
protected:
    int x_ = 2;
};

class Derived : public Base {
public:
    using Base::print;

    void print(const double x) const {
        const double tmp = x - this->x_;
        cout << "PrintDerived (double): " << tmp << endl;
    }
};

int main ()
{
    Base y;
    y.print(100.1);
    y.print(5);

    Derived x;
    x.print(100.1);
    x.print(5);
}

```

Function overloading

Member function

overloading rules: they don't work across scopes (from base to derived)

Workaround? This behavior can be modified by the **using** statement

```

whovian@phy504:~/host/CPP$ g++ class29.cpp -o class29 -std=c++20
whovian@phy504:~/host/CPP$ ./class29
PrintBase (double): 98.1
PrintBase (int): 3
PrintDerived (double): 98.1
PrintBase (int): 3

```

Operators in C++

Classes can have operators
(unary and binary operators)

Many operators can be
defined / overloaded

```
#include <iostream>
#include <spdlog/spdlog.h>
#include <fmt/core.h>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() // overloading default constructor
    {
        // this ensures proper array init
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }

    double operator[](const int i) {
        return this->x_[i];
    }

    double operator()(const int i) { // safer version
        if (i < 0) {
            spdlog::critical("negative index i");
            exit(1);
        }
        else if (i > maxsz) {
            spdlog::critical(
                fmt::format(
                    "index out of bounds (max={:3d})", maxsz));
            exit(1);
        }
        return this->x_[i];
    }
};

int main () {
    Base x;
    cout << x[50] << endl;
    cout << x(200) << endl;
    cout << x(2000) << endl;
}
```

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

```
whovian@phy504:~/host/CPP$ g++ class22.cpp -o class22 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./class22
50
200
[2023-04-07 15:58:59.955] [critical] index out of bounds (max=1000)
```

```

#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    Base operator++() { // prefix operator ++x
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return *this;
    }
    Base operator++(int) { // postfix operator (x++)
        // int is a dummy argument that tells the compiler
        // this is the postfix term. Weird, but that was
        // a choice of C++ original designers. See
        // https://stackoverflow.com/a/12740593/2472169
        Base tmp = *this;
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return tmp; //copy of the object before increment
    }
};

int main () {
    Base x;
    Base y = x++;
    cout << y[50] << endl;

    Base x2;
    Base y2 = ++x2;
    cout << y2[50] << endl;
}

```

Operators in C++

Prefix and postfix operators
are interesting (and a bit
weird) to overload

Why does the postfix need an
integer dummy variable?

Choice of the original C++
designers

```

whovian@phy504:~/host/CPP$ g++ class23.cpp -o class23 -std=c++20
whovian@phy504:~/host/CPP$ ./class23
50
51

```

```

#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    Base operator++() { // prefix operator ++x
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return *this;
    }
    // args by reference (explanation next class!)
    friend bool operator==(Base& l, Base& r);
};

// operator is a normal function (not member func)
bool operator==(Base& l, Base& r) {
    bool res = true;
    for(int i=0; i<maxsz; i++) {
        if (l.x_[i] != r.x_[i]) {
            res = false;
            break;
        }
    }
    return res;
}

int main () {
    Base x;
    ++x;
    Base y;

    bool res = (x == y);
    cout << std::boolalpha << res << endl;
}

```

Class Inheritance

It is best for some operators to be a normal global function (not member functions).

But how can a normal function access the private data of a class? Typically, it can't.

However, this limitation can be overridden with the **friend** keyword

```

whovian@phy504:~/host/CPP$ g++ class24.cpp -o class24 -std=c++20
whovian@phy504:~/host/CPP$ ./class24
false

```

```

#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    Base operator++() {
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += 1;
        }
        return *this;
    }
    friend Base operator+(Base& l, Base& r);
};

Base operator+(Base& l, Base& r) {
    Base res;
    for(int i=0; i<maxsz; i++)
        res.x_[i] = l.x_[i] + r.x_[i];
    return res;
}

int main () {
    Base x;
    Base y;
    Base z = x + y;
    cout << x[50] << " " << y[50]
         << " " << z[50] << endl;
}

```

Class Inheritance

It is best for some operators to be a normal global function (not member functions).

But how can a normal function access the private data of a class? Typically, it can't.

However, this limitation can be overridden with the **friend** keyword

```

whovian@phy504:~/host/CPP$ g++ class25.cpp -o class25 -std=c++20
whovian@phy504:~/host/CPP$ ./class25
50 50 100

```

```

#include <iostream>
using namespace std;
constexpr int maxsz = 1000;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    // return reference = optimization
    Base& operator+=(Base& other) {
        for(int i=0; i<maxsz; i++) {
            this->x_[i] += other.x_[i];
        }
        return *this;
    }
};

int main () {
    Base x;
    Base y;
    cout << x[50] << " " << y[50];
    x += y;
    cout << " " << x[50] << endl;
}

```

Class Inheritance

How do you decide between a member and a global operator?

+ **-** ***** **/** should be global operators

++, **+=**, **-=**, ***=**, **/=** should be member operators

```

whovian@phy504:~/host/CPP$ g++ class26.cpp -o class26 -std=c++20
whovian@phy504:~/host/CPP$ ./class26
50 50 100

```

```

#include <iostream>
using namespace std;
constexpr int maxsz = 10;

class Base {
    double x_[maxsz];
public:
    Base() {
        for(int i=0; i<maxsz; i++)
            this->x_[i] = i;
    }
    double operator[](const int i) {
        return this->x_[i];
    }
    friend ostream& operator<<(ostream& os, Base x);
};

ostream& operator<<(ostream& os, Base x)
{
    os << "Printing Base \n";
    for(int i=0; i<maxsz; i++)
        os << x.x_[i] << " ";
    return os;
}

int main () {
    Base x;
    cout << x << endl;
}

```

Operators in C++

How do we overload the output stream << (an alternative way to print the class instead of a **print()** method)?

Output stream << should be a global operator

```

whovian@phy504:~/host/CPP$ g++ class27.cpp -o class27 -std=c++20
whovian@phy504:~/host/CPP$ ./class27
Printing Base
0 1 2 3 4 5 6 7 8 9

```

Homework I

// Define an abstract base class in a `.cpp` file:

```
class Person {  
protected:  
    std::string name;  
    int age;  
public:  
    Person(std::string n, int a);  
    virtual ~Person() = default;  
    virtual std::string role() const = 0;  
    virtual void speak() const = 0;  
};
```

/*

Then derive `Dad`, `Mother`, and `Student`—each must:

1. Call the `Person` constructor to initialize `name` and `age`.
2. Implement `role()` returning `"Dad"`, `"Mother"`, or `"Student"`.
3. Implement `speak()` to print a short phrase (e.g. `Dad says hi!`).

Homework II

```
// 2. Virtual Functions & Polymorphism
```

```
// Add to your same `.cpp` (from HW1):
```

```
virtual void introduce() const {  
    std::cout << "Hi, I'm " << name << ", a " << role() << "\n";  
}
```

```
/*
```

```
then override `introduce()` in at least two derived classes  
(e.g. `Mother` adds "...and I love gardening."). In `main()`, store a  
`std::vector<Person*>` with mixed instances, loop over it, and call `introduce()`  
to demonstrate dynamic dispatch.
```

Homework III

//3. Constructor Delegation

Task

In your `Student` **class** (all in `.cpp`), implement three constructors:

```cpp

// 1) Just a name

Student(std::string name);

// 2) Name + age

Student(std::string name, **int** age);

// 3) Name + age + GPA

Student(std::string name, **int** age, **double** gpa);

```

// Use C++11 constructor delegation so (1) → (2) with default age `18`, and (2) → (3) with
// default GPA `4.0`. Add `std::cout` in each constructor to show which runs.

Homework IV

// 4. Operator Overloading: Member vs. Free Functions

Task

In your `.cpp``, **for** ``Person`` implement:

1. ****Member**** ``bool operator<``(`const Person& other``) `const``;` (compare ages)■
2. ****Free**** ``bool operator!=``(`const Person& lhs``, `const Person& rhs``);` (negates ``==``)■
3. ****Friend**** ``std::ostream& operator<<``(`std::ostream& os``, `const Person& p``);` (prints ``Name(age): Role``)

In ``main()``, create two different ``Person``-derived objects and:

- Use ``<`` and ``!=``.■
- Stream them to ``std::cout``.

Homework V (virtual functions)

Create a class **Shape** with a non-virtual **area() const** that returns 0.0, and a **print() const** that calls **area()** internally. Derive **Circle** with its own **area() const**. In **main()**

- Create **Circle c(5.0)** and call **c.area()**. What value does it return? Why?
- Now call **c.print()**. What value does it return? Why?
- Now create a **Shape* ptr = &c** and call **ptr->area()**. What value does it return? Why?
- Fix the bug by adding **virtual** to **.area()**. Now, which values do **c.print()** and **ptr->area()** return? Why??

Hints:

- Without a virtual keyword, function dispatch is static and determined by the declared type.
- With a virtual keyword, dispatch is dynamic; the runtime checks what the object actually is.