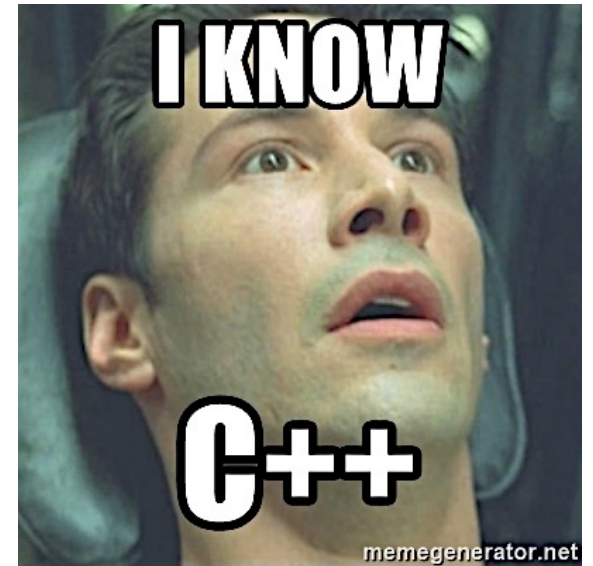
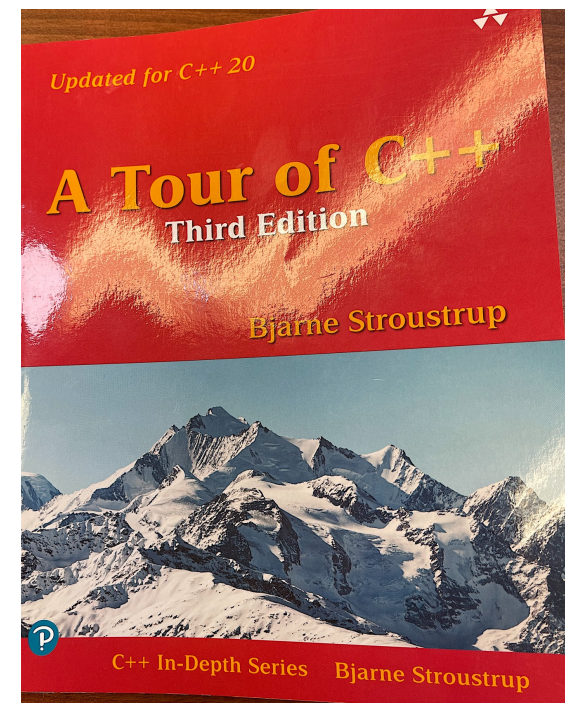


Lecture 23: C++ part VIII



Suggested Literature



More IO (including FILES)

C++ IO

```
#include <sstream> // stringstream
#include <iostream>
#include <string>
#include <iomanip> // setw, setfill, setprecision
#include <fmt/core.h>
using namespace std;

int main ()
{
    std::stringstream ss;

    ss << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << 12.3 << " and ";
    ss << left << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << 15.6;

    string s1 = ss.str(); // get the string
    cout << s1 << endl;

    // via FMT library (and C++23 standard)
    string s2 = fmt::format(
        "{:*>21.10f} and {:*<21.10f}", 12.3, 15.6);
    cout << s2 << endl;

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io12.c -o io12 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io12
*****12.3000000000 and 15.6000000000*****
*****12.3000000000 and 15.6000000000*****
```

C++ offers streams
for strings (C++20
offers **std::format**)

Works like any
other stream, but it
does not output to
terminal/file
(can work as buffer)

C++ IO

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name;
    int age;
    cin >> name >> age;

    cout << "Name : " << name << endl;
    cout << "Age : " << age << endl;

    for (int x; !(cin>>x).fail();)
    {
        cout << "x : " << x << endl;
    }

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ io10.c -o io10 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io10
VivianMiranda 37
Name : VivianMiranda
Age : 37
1
x : 1
2
x : 2
3
x : 3
XW
```

How to read from the terminal (and files)?

Invert from **<<** & **cout**
(output) to **>>** & **cin**
(input)

C++ IO

How to write to files?

```
#include <fstream>
#include <iostream>
#include <list>
#include <fmt/core.h>
#include <iomanip> // setw, setfill, setprecision
using namespace std;

int main () {
    list<double> v = {1.0, 2.0, 3.0, 4.0, 5.0};

    fstream file;
    file.open("test_io9.txt", ios::out);
    if(!file.is_open()) {
        cout << "Error in creating file!!!" << endl;
        return 1;
    }

    for (auto x : v) { // only change is cout <-> file
        file << right << setfill('*') <<
            setw(21) << setprecision(10) <<
            fixed << x << endl;
    }
    file.close();

    // via FMT library (part I using format)
    list<double> v2 = {20.0, 40.0, 60.0, 80.0, 100.0};

    file.open("test_io9.txt", ios::app); // append
    if(!file.is_open()) {
        cout << "Error opening file" << endl;
        return 1;
    }

    for (auto x : v2) {
        file << fmt::format("{:*>21.10f}", x) << endl;
    }
    file.close();
}
```

```
whovian@phy504:~/host/CPP$ g++ io9.c -o io9 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io9
whovian@phy504:~/host/CPP$ more test_io9.txt
*****1.0000000000
*****2.0000000000
*****3.0000000000
*****4.0000000000
*****5.0000000000
*****20.0000000000
*****40.0000000000
*****60.0000000000
*****80.0000000000
*****100.0000000000
```

streams: users only need to
replace **std::cout** with
std::fstream

Remember that streams are
compatible with
fmt::format

C++ IO

How to read from the terminal (and files)?

Invert from << (output) to >> (input)

Don't forget the **ios::in** flag

```
#include <fstream>
#include <iostream>
#include <string>
#include <iomanip> //setw, setfill, setprecision
using namespace std;

int main ()
{
    fstream file;
    file.open("test_io11.txt", ios::in);
    if(!file.is_open()) {
        cout << "Error reading the file!!!" << endl;
        return 1;
    }

    string x;
    int y;
    file >> x >> y;

    cout << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << x << endl;

    cout << right << setfill('*') <<
        setw(21) << setprecision(10) <<
        fixed << y << endl;

    return 0;
}
```

```
whovian@phy504:~/host/CPP$ more test_io11.txt
VivianMiranda 37
whovian@phy504:~/host/CPP$ g++ io11.c -o io11 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./io11
*****VivianMiranda
*****37
```

C++ IO

```
#include <sstream> // stringstream
#include <fstream>
#include <iostream>
#include <string>
#include <iomanip> //setw, setfill, setprecision
```

```
using namespace std;
```

```
whovian@phy504:~/host/CPP$ g++ io14.c -o io14 -std=c++20 -lfmt
```

```
whovian@phy504:~/host/CPP$ ./io14
```

```
VivianMiranda 37
```

```
int main ()
{
    ifstream file("test_io11.txt");
    if(!file.is_open()) {
        cout << "Error reading the file!!!" << endl;
        return 1;
    }

    // read the entire file into memory
    std::stringstream buffer;
    buffer << file.rdbuf(); // read buffer
    file.close();

    std::string file_content = buffer.str();
    if (file_content.empty()) {
        cout << "file empty" << endl;
    }
    cout << file_content << endl;

    return 0;
}
```

C++ offers streams for strings (C++20 offers **std::format**)

We can use C++ **std::stringstream** to read entire file on RAM buffer

ENUM CLASSES

C++ Enum Classes

```
#include <string>
#include <iostream>

using namespace std;

enum class Human : int
{
    Vivian,
    Beyonce,
    Rihanna
};

enum class Professor : int
{
    Vivian, // Not a problem to have same name
    Wayne,
    Ioav
};

int main ()
{
    Human x = Human::Rihanna;

    if (x == Human::Rihanna)
        cout << "Human is Rihanna" << endl;
    else
        cout << "Human is not Rihanna" << endl;

    Professor y = Professor::Vivian;

    if (y == Professor::Vivian)
        cout << "Professor is Vivian" << endl;
    else
        cout << "Professor is not Vivian" << endl;

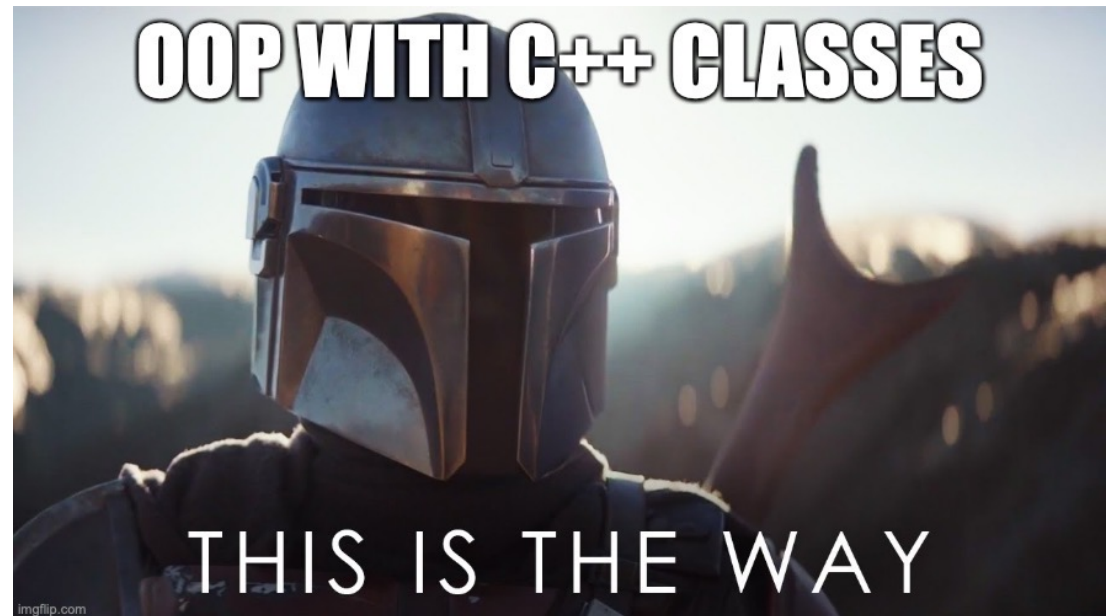
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ enumclass1.cpp -o enumclass1 -std=c++20
whovian@phy504:~/host/CPP$ ./enumclass1
Human is Rihanna
Professor is Vivian
```

enum class solves the main problem of enumerators in C. Variables inside C enumerators are global. Big problem: you can't have two enumerators with variables that share the same name

C++ Classes Part I

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



C++ Classes

C++ Class v0.1

public: / **private:**
access control

C-Style classes are only possible if all variables are set to **public** access

In C++, the default for **class** is **private** access

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

// constexpr = compile time constant (C++11 feature)
constexpr int NDigitsSSN = 9;
constexpr int NBirthday = 3; // birthday moth, day, year
constexpr int NDigitsYear = 4; // Force Year to be 4 digits
constexpr int NCountries = 195; // number of countries
constexpr int CurrentYear = 2023;

enum class Genders : int
{ // C++ version of enumerators (safer than C version)
    Female,
    Male,
    NB
};

class Human // Simples C-Style class
{
public:
    int age_;
    int ssn_ = -1; // default value (flag not set)
    Genders gender_;
    double height_ = -1.0; // default value (flag not set)
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_; // We may have multiple nationalities
    list<string> name_; // We have name, surname
private:
    // none
};

int main() {
    Human x;
    x.age_ = 37;
    x.name_ = {"Vivian", "Miranda"};
    cout << x.age_ << endl;
    return 0;
}
```

C++ Classes

C++ Class v0.2

Constructors

Tells the compiler how an instance of the class should be constructed

E.g., given our explicit constructor, Human instances cannot be created without a defined name

```
class Human
{
public:
    // Constructor - we require name when
    // creating an instance of the class
    // No Human without defined name
    Human(list<string> name) :
        name_(name)
    {
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    // Human x; // This line does not compile
    Human x({"Vivian", "Miranda"});
    x.age_ = 37;
    cout << x.age_ << endl;
    return 0;
}
```

Constructor has the same name of the class



C++ Classes

C++ Class v0.3

We can have multiple **constructors** in C++.

Use a new **constructor** to convert from **std::string** to **std::list<string>** w/ code from the previous lecture.

this-> is optional (but should always include it)

```
class Human {
public:
    Human(list<string> name) :
        name_(name)
    {
    }
    Human(const string name)
    {
        boost::split(this->name_, name,
            boost::is_any_of(" \\t"),
            boost::token_compress_on);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthdays> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    // Human x; // This line does not compile
    // Human x({"Vivian", "Miranda"}); // This line doesn't
    // compile (ambiguous)
    Human x(list<string>{"Vivian", "Miranda"});
    x.age_ = 37;
    cout << x.age_ << endl;

    Human y(string{"Vivian Miranda"});
    y.age_ = 37;
    cout << y.age_ << endl;

    return 0;
}
```

C++ Classes

C++ Class v0.3

We can have multiple **constructors** in C++, but more is not always better (choose the easier API)

```
int main() {  
    // Human x; // This line does not compile  
    // Human x({"Vivian", "Miranda"}); // This line doesn't  
                                        // compile (ambiguous)  
    Human x(list<string>{"Vivian", "Miranda"});  
    x.age_ = 37;  
    cout << x.age_ << endl;  
  
    Human y(string{"Vivian Miranda"});  
    y.age_ = 37;  
    cout << y.age_ << endl;  
  
    return 0;  
}
```

```
whovian@phy504:~/host/CPP$ g++ class3.cpp -o class3 -std=c++20  
class3.cpp: In function 'int main()':  
class3.cpp:48:32: error: call of overloaded 'Human(<brace-enclosed initializer list>)' is ambiguous  
  48 |     Human x({"Vivian", "Miranda"});  
      |                               ^  
class3.cpp:29:3: note: candidate: 'Human::Human(std::string)'  
  29 |     Human(const string name)  
      |     ~~~~~  
class3.cpp:25:3: note: candidate: 'Human::Human(std::__cxx11::list<std::__cxx11::basic_string<char> >>)'  
  25 |     Human(list<string> name) :  
      |     ~~~~~
```

C++ Classes

C++ Class v0.4

Choose the
easier API
when designing
constructors

```
class Human {
public:
    Human(const string name)
    {
        boost::split(this->name_, name,
            boost::is_any_of(" \t"),
            boost::token_compress_on);
    }
    Human(const string name, array<int, NBirthday> birthday) :
        Human(name) // Construtor delegation avoids code repetition
    {
        this->birthday_ = birthday;
        this->age_ = CurrentYear - this->birthday_.at(2);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    Human x("Vivian Miranda", {02,28,1986});
    cout << x.age_ << endl;
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ class4.cpp -o class4 -std=c++20
whovian@phy504:~/host/CPP$ ./class4
37
```

C++ Classes

C++ Class v0.5

In this example, coding multiple **constructors** was not really needed

Use default arguments instead

```
class Human {
public:
    Human(const string name,
          array<int, NBirthday> birthday = {0,0,0})
    {
        boost::split(this->name_, name,
                     boost::is_any_of(" \t"),
                     boost::token_compress_on);

        this->birthday_ = birthday;
        this->age_ = CurrentYear - this->birthday_.at(2);
    }
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    // none
};

int main() {
    Human x("Vivian Miranda", {02,28,1986});
    cout << x.age_ << endl;

    Human y("Vivian Miranda");
    y.age_ = 37;
    cout << y.age_ << endl;
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ nano class5.cpp
whovian@phy504:~/host/CPP$ g++ class5.cpp -o class5 -std=c++20
whovian@phy504:~/host/CPP$ ./class5
37
37
```

```

class Human {
public:
    Human(const string name,
           array<int, NBirthday> birthday = {0,0,-1}) :
        birthday_(birthday),
        is_birthday_set_(false),
        age_(CurrentYear - birthday.at(2))
    {
        boost::split(this->name_, name,
                     boost::is_any_of(" \t"),
                     boost::token_compress_on);

        if(!this->age_ > CurrentYear) {
            this->is_birthday_set_ = true;
        }
    }

    void PrintBirthday () { // member function
        auto& x = this->birthday_; // just variable alias (& = ref)

        if(!this->is_birthday_set_) {
            spdlog::critical("Birthday not set");
            exit(1);
        }
        fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
    }

    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
private:
    bool is_birthday_set_; // can only be accessed by member function
};

int main() {
    Human x("Vivian Miranda", {02,28,1986});
    x.PrintBirthday();
    // Problem: if you change birthday_ manually, you can't
    // use PrintBirthday() because we can't access is_birthday_set_
    Human y("Vivian Miranda");
    y.birthday_ = {02,28,1986};
    y.PrintBirthday();
}

```

C++ Classes

C++ Class v0.6

C++ class allows the definition of **member functions**

public: / **private:** access makes more sense. Private variables can only be accessed via **member functions**

```

whovian@phy504:~/host/CPP$ g++ class6.cpp -o class6 -std=c++20 -lfmt
whovian@phy504:~/host/CPP$ ./class6
Birthday: 02,28,1986
[2023-04-21 15:42:43.684] [critical] Birthday not set

```

C++ Classes

C++ Class v0.7

Better to strictly
control access to

member variables (via
member functions)

Also, it is better for
readability to declare
member functions
inside the C++ **class**

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

    void PrintBirthday();
private: // Lets move all variables to be private
    bool is_birthday_set_;
    int age_;
    int ssn_ = -1;
    Genders gender_;
    double height_ = -1.0;
    array<int, NBirthday> birthday_ = {0, 0, 0};
    list<int> nationality_;
    list<string> name_;
};

Human::Human(const string name, array<int, NBirthday> birthday) :
    birthday_(birthday),
    is_birthday_set_(false),
    age_(CurrentYear - birthday.at(2))
{
    boost::split(this->name_, name,
                 boost::is_any_of(" \t"),
                 boost::token_compress_on);

    if(! (this->age_ > CurrentYear)) {
        this->is_birthday_set_ = true;
    }
}

void Human::PrintBirthday () {
    auto& x = this->birthday_;
    if(!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
}
```

```

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++ Classes

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)

C++ Classes

C++ Class v0.8

Note that for member function bodies outside the class, the **Human::** keyword must precede the function name

Member functions don't exist in isolation. They are intrinsically tied to the class

```
Human::Human(const string name, array<int, 3> birthday) :  
    birthday_(birthday),  
    age_(CurrentYear - birthday.at(2)) {  
    boost::split(this->name_, name, boost::is_any_of(" \\t"),  
        boost::token_compress_on);  
    if(!(this->age_ > CurrentYear)) {  
        this->is_birthday_set_ = true;  
    }  
}  
  
void Human::SetBirthday(const int month, const int day,  
                        const int year) {  
    this->SetBirthday(array<int, 3>{month, day, year});  
}  
  
void Human::SetBirthday(array<int, 3> x) {  
    if (x[0] < 0 || x[0] > 12) {  
        spdlog::critical("Birthday Month Invalid");  
        exit(1);  
    }  
    if (x[2] > CurrentYear) {  
        spdlog::critical("Birthday Year Invalid");  
        exit(1);  
    }  
    if (x[1] < 0 || x[1] > 31) {  
        spdlog::critical("Birthday Day Invalid");  
        exit(1);  
    }  
    this->birthday_ = x;  
    this->is_birthday_set_ = true;  
    this->age_ = CurrentYear - x[2];  
}  
  
void Human::SetGender(const Genders x) {  
    this->gender_ = x;  
    this->is_gender_set_ = true;  
}  
  
void Human::AddNationality(const int x) {  
    this->nationality_.push_back(x);  
    this->is_min_one_nationality_set_ = true;  
}
```

C++ Classes

C++ Class v0.8

One advantage of **get/set** type of member functions is they provide opportunities for the developer to include extra checks

Function overloading via the number (and/or type) of arguments works for member functions as well

```
array<int, 3> Human::GetBirthday() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->birthday_;
}

Genders Human::GetGender() const {
    if (!this->is_gender_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->gender_;
}

int Human::GetAge() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->age_;
}

list<int> Human::GetNationality() const {
    if (!this->is_min_one_nationality_set_) {
        spdlog::critical("No Nationality set");
        exit(1);
    }
    return this->nationality_;
}

void Human::Print() const {
    fmt::print("Name: ");
    for (auto x : this->name_) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
    auto& x = this->birthday_; // just variable alias (& = ref)
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
}
```

C++ Classes

C++ Class v0.8

Const correctness.

const copies and const references to Human class can only access functions that have the **const** keyword in their declaration (except constructor, of course)

```
array<int, 3> Human::GetBirthday() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->birthday_;
}

Genders Human::GetGender() const {
    if (!this->is_gender_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->gender_;
}

int Human::GetAge() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->age_;
}

list<int> Human::GetNationality() const {
    if (!this->is_min_one_nationality_set_) {
        spdlog::critical("No Nationality set");
        exit(1);
    }
    return this->nationality_;
}

void Human::Print() const {
    fmt::print("Name: ");
    for (auto x : this->name_) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
    auto& x = this->birthday_; // just variable alias (& = ref)
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
}
```

C++ Classes

C++ Class v0.8 - Const correctness.



C++ class methods have an implicit `this` parameter which comes before all the explicit ones. So a function declared within a class like this:

102



```
class C {  
    void f(int x);  
}
```



You can imagine really looks like this:

Question score (upvotes - downvotes)



```
void f(C* this, int x);
```

Now, if you declare it this way:

```
void f(int x) const;
```

It's as if you wrote this:

```
void f(const C* this, int x);
```

That is, the trailing `const` makes the `this` parameter `const`, meaning that you can invoke the method on `const` objects of the class type, and that the method cannot modify the object on which it was invoked (at least, not via the normal channels).

Share Edit Follow Flag

answered Apr 14, 2013 at 12:38



John Zwinck

250k ● 43 ● 342 ● 454

C++ Classes

C++ Class v0.8

Const Correctness.

const copies and const references to Human class can only access functions that have keyword **const** (except constructor)

Const member functions can still affect mutable variables

```
array<int, 3> Human::GetBirthday() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->birthday_;
}

Genders Human::GetGender() const {
    if (!this->is_gender_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->gender_;
}

int Human::GetAge() const {
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    return this->age_;
}

list<int> Human::GetNationality() const {
    if (!this->is_min_one_nationality_set_) {
        spdlog::critical("No Nationality set");
        exit(1);
    }
    return this->nationality_;
}

void Human::Print() const {
    fmt::print("Name: ");
    for (auto x : this->name_) {
        fmt::print("{0} ", x);
    }
    fmt::print("\n");
    auto& x = this->birthday_; // just variable alias (& = ref)
    if (!this->is_birthday_set_) {
        spdlog::critical("Birthday not set");
        exit(1);
    }
    fmt::print("Birthday: {:02d},{:02d},{:04d}\n",x[0],x[1],x[2]);
}
```

C++ Classes

Const mem functions can still change mutables

// Source: <https://stackoverflow.com/a/15999152/2472169>

```
struct X
{
    void foo() const // <== The implicit "this" pointer is const-qualified!
    {
        _x = 42; // ERROR! The "this" pointer is implicitly const
        _y = 42; // OK (_y is mutable)
    }

    void bar(X& obj) const // <== The implicit "this" pointer is const-qualified!
    {
        obj._x = 42; // OK! obj is a reference to non-const
        _x = 42; // ERROR! The "this" pointer is implicitly const
    }

    void bar(X const& obj) // <== The implicit "this" pointer is NOT const-qualified!
    {
        obj._x = 42; // ERROR! obj is a reference to const
        obj._y = 42; // OK! obj is a reference to const, but _y is mutable
        _x = 42; // OK! The "this" pointer is implicitly non-const
    }

    int _x;
    mutable int _y;
};
```

C++ Classes

```
int main() {  
    const Human x("Vivian Miranda", {02,28,1986});  
  
    // Here you can't use set functions!!!  
  
    cout << x.GetAge() << endl; // ok  
  
    x.Print(); // ok  
  
    //x.AddNationality(55); // compilation error  
}
```

```
whovian@phy504:~/host/CPP$ g++ class8.cpp -o class8 -std=c++20 -lfmt  
whovian@phy504:~/host/CPP$ ./class8  
37  
Name: Vivian Miranda  
Birthday: 02,28,1986
```

```
whovian@phy504:~/host/CPP$ g++ class8.cpp -o class8 -std=c++20 -lfmt  
class8.cpp: In function 'int main()':  
class8.cpp:154:19: error: passing 'const Human' as 'this' argument discards qualifiers [-fpermissive]  
  154 |     x.AddNationality(55); // compilation error  
      |     ~~~~~^~~~~  
class8.cpp:94:6: note: in call to 'void Human::AddNationality(int)'  
   94 | void Human::AddNationality(const int x) {  
      |     ~~~~~
```

Static Functions

Static Functions belong to the class itself, not to a particular object. Static func can't access **this**

E.g., can create a **Factory** function that makes sure that Mom's age is >0 (with the use of C++26 **std::expected**)

```
#include <iostream>
#include <expected>
#include <string>
#include <print>

class Mom {
    int x_;
    // Yes we can have private constructor. Force
    // the factory API (we can't call Mom m{5} directly
    explicit Mom(int v) : x_{v} {}
public:
    // Static factory function: returns a Mom on success,
    // or a string describing what went wrong on failure.
    static std::expected<Mom, std::string> create(int v) {
        if (v < 0)
            return std::unexpected("Mom age must be non-negative");
        if (v > 120)
            return std::unexpected("Mom age too large");
        return Mom{v}; // success: build and return the object
    }
    void print(double x) const {
        std::println("PrintMom: {}", x - x_);
    }
};

int main() {
    auto a = Mom::create(7);
    auto b = Mom::create(-5);
    auto c = Mom::create(9999);

    if (a) a->print(100);
    else std::println("a failed: {}", a.error());

    if (b) b->print(100);
    else std::println("b failed: {}", b.error());

    if (c) c->print(100);
    else std::println("c failed: {}", c.error());
}
```

::create not associated with a class instance

```
whovian@phy504:~/host/Lecture23$ nano class38.cpp
whovian@phy504:~/host/Lecture23$ /opt/gcc-latest/bin/g++ class38.cpp -o c38 -std=c++26
whovian@phy504:~/host/Lecture23$ ./c38
PrintMom: 93
b failed: Mom age must be non-negative
c failed: Mom age too large
```

Class Inheritance

Classes can have parents, grandparents, and so on...

Public inheritance: the child class cannot access private member functions of the parent class. E.g., **int x_**

One child class can have multiple parents

```
#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);
}
```

```
whovian@phy504:~/host/CPP$ g++ class9.cpp -o class9 -std=c++20
whovian@phy504:~/host/CPP$ ./class9
PrintBase: 98
PrintDerived: 90
```

Class Inheritance

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

```
#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);
}
```

```
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

Class Inheritance

If member functions have different names, then no explicit casting is needed

```
#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 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++ class11.cpp -o class11 -std=c++20
whovian@phy504:~/host/CPP$ ./class11
PrintBase: 98
PrintDerived: 90
```

```

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

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

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

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

int main ()
{
    Derived x;
    x.print(100);
    x.print2(100);
    x.print3(100);
}

```

```

whovian@phy504:~/host/CPP$ g++ class12.cpp -o class12 -std=c++20
whovian@phy504:~/host/CPP$ ./class12
PrintBase: 98
PrintBase: 90
PrintDerived: 50

```

Class Inheritance

One child class can have multiple parents

Again, if member functions have different names, then no explicit casting is needed

All **private** variables are independent

```

#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)

Homework I

1. Create a base class `Vehicle` with a `move()` function. Create `Car` and `Bike` derived classes. Implement `move()` differently in each derived class (but not virtual – next class topic). Call the `move()` function directly from objects (not pointers)
2. Create a base class `Account` with `deposit()` and `balance` member. Derive `SavingsAccount` and `CheckingAccount`, each with their own `deposit()` method (non-virtual). Call `deposit()` from both types and observe function resolution.
3. Define a class `Printer` with multiple `print()` overloads. One takes `int`, one takes `std::string`. Create objects and call different `print()` overloads.

Homework II

1. Initializer list is a type of construction that allows users to construct an object using curly brackets **{}** — research about **initializer lists (constructor)**. Create a Book class with title, author, and year. Use a constructor initializer list. Add a **display()** function. Create a few Book objects in main().

Homework III

Create a class **DataSet** with a private **vector<double>**. Add a member function **apply(auto fn) const** that takes any lambda and transforms the data. Test the function with two different lambdas.

- a) one that squares each element
- b) one that normalizes the data, i.e., it produces a new vector where every element is divided by the maximum value.

Hints:

- Apply uses auto for the lambda parameter. This makes it a template member function. The compiler deduces the lambda type.
- Inside Apply, pipe the data through **std::views::transform** and **std::ranges::to**
- For normalization, you'll need **std::ranges::max** to find the max value, then divide each element by it.
- Apply returns a new vector, so it doesn't modify the internal data.

Homework IV

Create a class **Counter** with a private **int count_** and **mutable int count2_**. Write **increment()** (not const), **get_count() const** (increments **count2** each time it's called), and **get_access_count() const**. In **main()**, create both a **Counter x** and a **const Counter y**. Show which member functions work on each and explain why. Then write a function **void print_counter(const Counter& c)**, what can it call?

Hints:

- A const object can only call member functions marked const.
- mutable is the escape hatch: it lets a const member function modify specific variables. Use it for bookkeeping that doesn't change the "logical" state of the.
- **void print_counter(const Counter& x)** receives a const reference. Inside that function, x behaves exactly like a const object.
- Think about what the compiler does behind the scenes: **get_count() const** means the hidden this pointer is **const Counter***, so you can't modify **count_** through it. But **mutable** members are exempt.
- Use the constructor's initializer list to set both **count_** and **count2_** to 0.

Homework V

Create a class **Base** with private **int x_ = 10** and a **public print() const**. Create **Derived : public Base** with its own private **int x_ = 20** and its own **print() const**. In main, create a **Derived d**. Call **d.print()**, which **x_** does it use? Then cast to **Base&** and call **print()** again, which **x_** now? Add a **print_both() const** to **Derived** that prints both its own **x_** and **Base's x_**.

Hints:

- **Derived** doesn't replace **Base's x_**; it shadows it. Both **x_** variables exist inside the **Derived** object. **this->x_** refers to whichever **x_** is closest in scope.
- When you cast to **Base&**, the compiler forgets about **Derived** and calls **Base's print()**, which sees **Base's x_**.
- You can't access **Base's** private **x_** from **Derived** directly; that's what private means. The solution: add a *public getter* in **Base** like **get_x() const**. Then **Derived** can call **Base::get_x()** to read the parent's value.
- **static_cast<Base&>(d)** creates a reference to the **Base** part of **d**; no copy!. But **static_cast<Base>(d)** would copy only the **Base** part (this is known as object slicing).

Homework VI

Create a class **Builder** with a private `std::vector<std::string> parts_`. Add `add(std::string)` that returns **Builder&**, so calls can be chained: `x.add("x").add("y").add("z")`. Add `build() const` that joins all parts with " " (space). Then create a second version `add_broken(std::string)` that returns **Builder** by value instead of by reference. In `main()`, chain `add_broken` calls and show that it doesn't work. Why doesn't it work?

Hints:

- return `*this` returns the current object.
- `build() const` is a good place to use `fmt::join(parts_, " ")`.
- The broken version is a great way to understand the difference between value and reference semantics.
- Think about it: `b.add("x")` returns `b` itself. `b.add_broken("x")` returns a stranger that looks like `b`. Elaborate on that using more technical wording

Homework VII - Static Members and Scope (hard)

Create a class **UniqueID** with a **static int next_id_** and a **non-static const int id_** (research about static members inside a class). The constructor assigns **next_id_** to **id_** and increments **next_id_**. Since **id_** is **const**, it can only be set in the member initializer list, never reassigned. Add a **static int get_next_id()** (research about static functions inside a class). Note that static member functions have **NO this pointer**, so they cannot access non-static members. Add **get_id() const**. In **main()**, create several objects and show that each gets a unique ID. Then create objects inside a loop and show the IDs keep incrementing. Try calling **get_next_id()** without creating any object to show it works on the class itself.

Hints:

- static members belong to the class, not to any instance. **next_id_** is shared, i.e., there's only one copy, no matter how many objects exist.
- **const int id_** means once set, it never changes. The only place to set it is the member class initializer list.
- static member functions can be called on the class directly: **UniqueID::get_next_id()**. They don't receive this, so they can only access other static members.
- The static member must be defined outside the class: **int UniqueID::next_id_ = 1;**. Without this line, the linker will complain about an undefined symbol.
- Think about what **const int id_** prevents: you can't write **this->id_ = 5;** anywhere, definitely not in member functions, not anywhere. The object's ID is frozen at birth.

Additional Reading

- Discussion about Constructor delegation and member initialization
(<https://stackoverflow.com/a/12190126/2472169>)
- Discussion about mutable variables
(<https://stackoverflow.com/q/105014/2472169>)