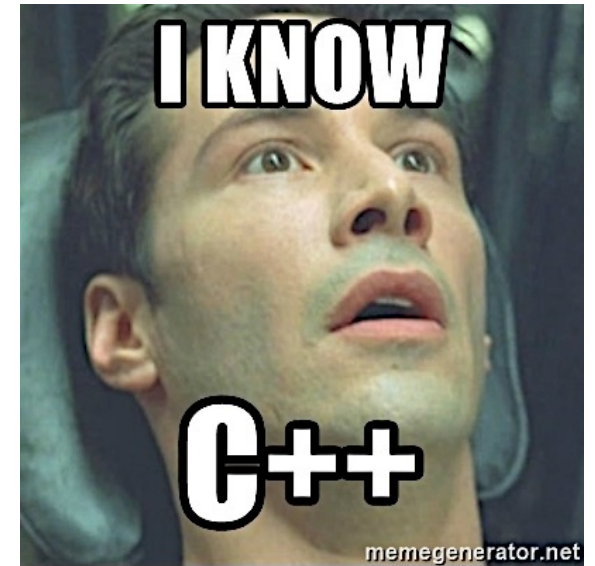
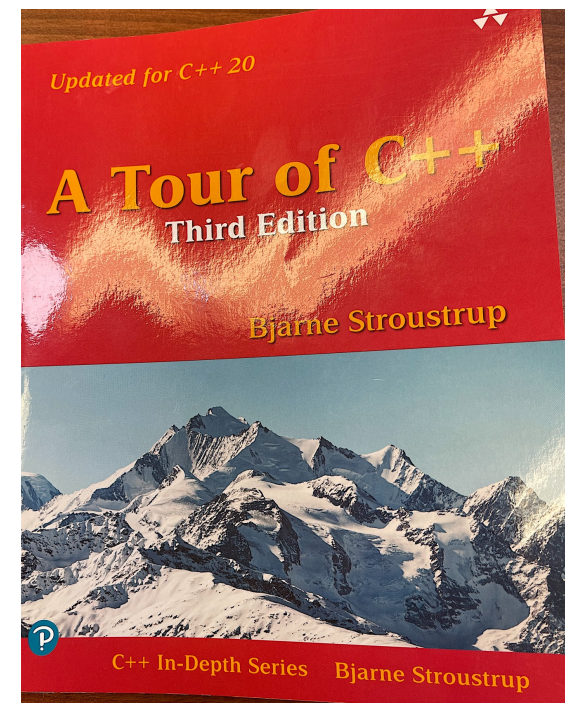


Lecture 25: C++ part X



Suggested Literature



RECAP: Class Inheritance

```
#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;
    }
};

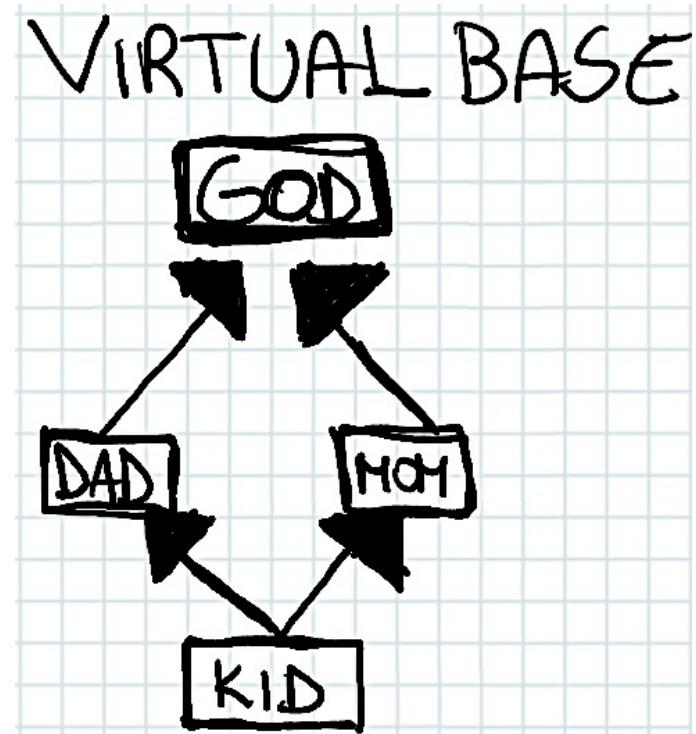
class Mom : public virtual God {
public:
    void print(const double x) const {
        cout << "PrintMom: " << x-this->x_ << endl;
    }
};

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);
}
```

Virtual Base Class



Code now runs no problem

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

RECAP: Class Inheritance

```
#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++
}
```

Abstract **Base** Class

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

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

RECAP: Operators in C++

```
#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;
}
```

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
```

RECAP: Class Inheritance

```
#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;
}
```

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
```

r-value references (brief introduction)

```

#include <iostream>

using namespace std;

void Print(double& x)
{
    cout << "Print (l-value ref): " << x << endl;
}

void Print(double&& x)
{
    cout << "Print (r-value ref): " << x << endl;
}

int main()
{
    double x = 10;
    Print(x);

    Print(20);

    return 0;
}

```

r-value references

r-value references

bind to **r-values**

On intrinsic types, it makes little sense to use **l-value** of **r-value** references

Just pass by value and use tuples to have multiple returns

```

whovian@phy504:~/host/CPP$ g++ ref12.cpp -o ref12 -std=c++20
whovian@phy504:~/host/CPP$ ./ref12
Print (l-value ref): 10
Print (r-value ref): 20

```

```

#include <iostream>

using namespace std;

void Print(double& x)
{
    cout << "Print (l-value ref): " << x << endl;
    x = 2; // this move will propagate to main()
}

void Print(double&& x)
{
    cout << "Print (r-value ref): " << x << endl;
}

int main()
{
    double x = 10;
    Print(x);

    double y = 20;
    Print(move(y)); // data was moved(!!) from y
                    // to the r-value reference

    // Here y does not hold the data 20 anymore

    // What y holds after move? Undefined Behavior!

    // However, y is still a valid l-value variable
    y = 30; // 100% ok code!
    cout << "Print (main): " << y << endl; // 100% ok!

    return 0;
}

```

r-value references

But on expensive types
(classes) to be copied,
there are good
applications for **l-value
references, r-value
references, and copy-by-
value**

Example: **std::move()**

```

whovian@phy504:~/host/CPP$ g++ ref13.cpp -o ref13 -std=c++20
whovian@phy504:~/host/CPP$ ./ref13
Print (l-value ref): 10
Print (r-value ref): 20
Print (main): 30

```

```

#include <iostream>

using namespace std;

//void Print(double& x)
//{
//  cout << "Print (l-value ref): " << x << endl;
//}

void Print(double&& x) {
  cout << "Print (r-value ref): " << x << endl;
}

int main()
{
  Print(20); // ok - 20 is an r-value

  // Can r-value references deal with l-values?
  // Not automatically!
  double x = 10;
  //Print(x); // Compiler error:
  //           // error: cannot bind rvalue
  //           // reference of type 'double&&'
  //           // to lvalue of type 'double'

  Print(move(x)); // that is ok
                  // we moved the data

  // x does not hold 10 anymore
  // Again: what x holds? undefined!
  // but x is still a perfectly normal var
  x = 2; // 100% valid code
  cout << "Print (main): " << x << endl;
  return 0;
}

```

r-value references

Can **r-value references**
bind to **l-values**? No.

We need to convert **l-value**
to **r-value** with **std::move()**

This (**std::move()**) has real-
life consequences. The **l-
value** variable does not
store the data anymore

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ ref18.cpp -o ref18 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./ref18
Print (r-value ref): 20
Print (r-value ref): 10
Print (main): 2

```

r-value references

When should the
different
function
overloading
types be used?

[Nice explanation](#)
[from](#)
[StackOverflow](#)

```
//Hello. I want my own local copy of your Widget that I will manipulate,  
//but I don't want my changes to affect the one you have. I may or may not  
//hold onto it for later, but that's none of your business.
```

```
void foo(Widget w);
```

```
//Hello. I want to take your Widget and play with it. It may be in a  
//different state than when you gave it to me, but it'll still be yours  
//when I'm finished. Trust me!
```

```
void foo(Widget& w);
```

```
//Hello. Can I see that Widget of yours? I don't want to mess with it;  
//I just want to check something out on it. Read that one value from it,  
//or observe what state it's in. I won't touch it and I won't keep it.
```

```
void foo(const Widget& w);
```

```
//Hello. Ooh, I like that Widget you have. You're not going to use it  
//anymore, are you? Please just give it to me. Thank you! It's my  
//responsibility now, so don't worry about it anymore, m'kay?
```

```
void foo(Widget&& w);
```

For another way of looking at it:

```
//Here, let me buy you a new car just like mine. I don't care if you wreck  
//it or give it a new paint job; you have yours and I have mine.
```

```
void foo(Car c);
```

```
//Here are the keys to my car. I understand that it may come back...  
//not quite the same... as I lent it to you, but I'm okay with that.
```

```
void foo(Car& c);
```

```
//Here are the keys to my car as long as you promise to not give it a  
//paint job or anything like that
```

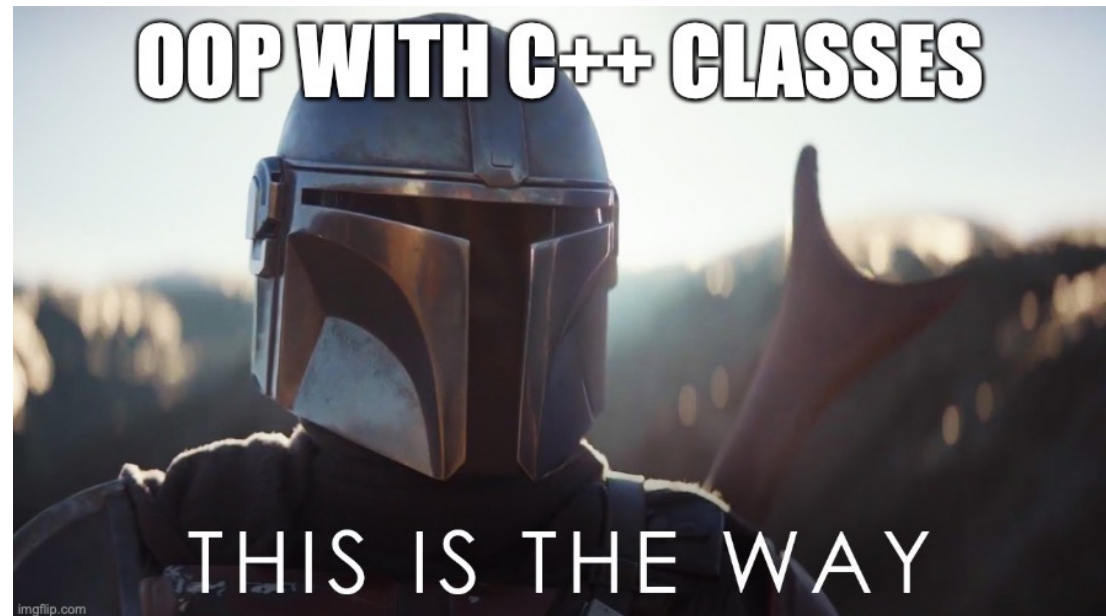
```
void foo(const Car& c);
```

```
//I don't need my car anymore, so I'm signing the title over to you now.  
//Happy birthday!
```

```
void foo(Car&& c);
```

C++ Classes Part III

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



new/delete

C and C++ have different ways to allocate memory to the heap

malloc / free deal with bytes

new / delete deal with objects (so no need to tell byte size)

```
#include <cstdlib> // the C++ wrapper for C's <stdlib.h>
                    // the C funcs are placed into std:: namespace
#include <print>

int main() {
    std::println("malloc / free");

    int* p1 = (int*) std::malloc(sizeof(int)); // p1 points to 4 bytes
                                                // of raw memory.
                                                // The value is undefined.

    if (p1 == nullptr)
        return 1; // malloc returns nullptr on failure
    *p1 = 42;
    std::println("*p1 = {}", *p1);

    std::free(p1); // release the memory

    std::println("new / delete: no initialization");

    p1 = new int; // Value is also undefined?
                 // Why? 'int' has no constructor to initialize it.
                 // new on classes is a different story

    *p1 = 99;
    std::println("*p1 = {}", *p1);

    delete p1;

    std::println("new / delete: w/ initialization");

    p1 = new int(7); // allocate and initialize to 7
    std::println("*p1 = {}", *p1);
    delete p1;

    p1 = new int{}; // allocate and initialize to 0
    std::println("*p1 = {}", *p1);
    delete p1;
}
```

```
whovian@phy504:~/host/Lecture25$ nano sm6.cpp
whovian@phy504:~/host/Lecture25$ g++ sm6.cpp -o sm6 -std=c++26
whovian@phy504:~/host/Lecture25$ ./sm6
malloc / free
*p1 = 42
new / delete: no initialization
*p1 = 99
new / delete: w/ initialization
*p1 = 7
*p1 = 0
```

```

#include <iostream>
#include <unistd.h>
using namespace std;
constexpr int maxsz = 600000000;

class Base {
    double* x_;
public:
    Base(); // Class Constructor
    ~Base(); // Class Destructor
};

Base::Base() {
    this->x_ = new double[maxsz];
}

Base::~~Base() { // Class Destructor
    cout << "Destructor called" << endl;
    if(this->x_ != nullptr) {
        delete [] this->x_;
        cout << "Ptr deleted" << endl;
    }
}

int main () {
    for (int i=0; i<5; i++) {
        Base y;
        sleep(2);
    }
}

```

Memory Management

If a member variable is allocated via **new** operator (C++ version of **malloc**), how can we avoid memory leaks?

Answer: RAII (Resource Acquisition is Initialization)

```

whovian@phy504:~/host/CPP$ g++ class30.cpp -o class30 -std=c++20 -O0
whovian@phy504:~/host/CPP$ ./class30
Destructor called
Ptr deleted
Destructor called
Ptr deleted
Destructor called
Ptr deleted
Destructor called
Ptr deleted
Destructor called
Ptr deleted

```

RAII - An Aside

RAII is a bad name for the concept.

"The best example of why I shouldn't be in marketing"

"I didn't have a good day when I named that"



Bjarne Stroustrup, still unhappy
with the name RAII in 2012

RAII - An Aside

A better name is probably:

Constructor **A**cquires, **D**estructor **R**eleases

or

Scope **B**ased **R**esource **M**anagement

```

#include <iostream>
#include <unistd.h>
using namespace std;
constexpr int maxsz = 600000000;

class Base {
    double* x_;
public:
    Base(); // Class Constructor
    ~Base(); // Class Destructor
};

Base::Base() {
    this->x_ = new double[maxsz];
}

Base::~~Base() { // Class Destructor
    cout << "Destructor called" << endl;
    if(this->x_ != nullptr) {
        delete [] this->x_;
        cout << "Ptr deleted" << endl;
    }
}

int main () {
    for (int i=0; i<5; i++) {
        Base y;
        sleep(2);
    }
}

```

Memory Management

If a member variable is allocated via the **new** operator (C++ **malloc**), how can we avoid memory leaks?

Answer: RAII (Resource Acquisition is Initialization)

```

whovian@phy504:~/host/CPP$ g++ class30.cpp -o class30 -std=c++20 -O0
whovian@phy504:~/host/CPP$ ./class30
Destructor called
Ptr deleted
Destructor called
Ptr deleted
Destructor called
Ptr deleted
Destructor called
Ptr deleted
Destructor called
Ptr deleted

```

```

#include <iostream>
#include <unistd.h>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base(); // Class Constructor
    ~Base(); // Class Destructor
};

Base::Base() {
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++) {
        this->x_[i] = i;
    }
}

Base::~~Base() { // Class Destructor
    cout << "Destructor called" << endl;
    if(this->x_ != nullptr) {
        delete [] this->x_;
        cout << "Ptr deleted" << endl;
    }
}

int main ()
{
    Base a;
    Base b(a); // copy constructor called
    a.x_[50] = -1;
    // two bugs!
    // Bug 1: a.x_[50] == b.x_[50]
    // Why? C++ copied the pointer not the data
    // Bug 2: at the end of the program, we will
    // see a double deletion of the same data array
    cout << a.x_[50] << " " << b.x_[50] << endl;
}

```

Memory Management

Adoption of the **new** operator creates difficulties in copying class instantiations (objects)

Answer: copying the pointer is not the same as copying the underlying data

```

whovian@phy504:~/host/CPP$ g++ class31.cpp -o class31 -std=c++20 -O0
whovian@phy504:~/host/CPP$ ./class31
-1 -1
Destructor called
Ptr deleted
Destructor called
double free or corruption (!prev)
qemu: uncaught target signal 6 (Aborted) - core dumped
Aborted

```

```

#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base(); // Class Default Constructor
    ~Base(); // Class Destructor
    Base(const Base& other); // Copy Constructor
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main ()
{
    Base a;
    Base b(a); // copy constructor called
    a.x_[50] = -1;
    cout << a.x_[50] << " " <<
        b.x_[50] << endl;
}

```

Memory Management

If using **new** / **delete** operators explicitly (bad idea!), then we need to teach the class how to copy (and move) the data

How? Answer: **copy** and **move** (new concept) **constructors** (and **assignment operators**).

```

whovian@phy504:~/host/CPP$ g++ class32.cpp -o class32 -std=c++20
whovian@phy504:~/host/CPP$ ./class32
Default Constructor
Copy Constructor
-1  50
Destructor
Destructor

```

Memory Management

There are nuances in defining assignment operators.

Usually adopted strategies based on the **copy constructor** (copy-and-swap idiom)

```
#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base();      // Default Constructor
    ~Base();     // Class Destructor
    Base(const Base& other); // Copy Constructor
    Base& operator=(Base other); // Assignment operator
                                // implemented via copy
                                // and swap idiom
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) {
    // On copy-and-swap idiom, arg is passed
    // by value (it will trigger the copy constructor)
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b;
    b = a;
    a.x_[50] = -1;
    cout << a.x_[50] << " " << b.x_[50] << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ class33.cpp -o class33 -std=c++20
whovian@phy504:~/host/CPP$ ./class33
Default Constructor
Default Constructor
Copy Constructor
operator = (assignment)
Destructor
-1 50
Destructor
Destructor
```

Memory Management

There are nuances in defining assignment operators.

If you code instead **Base b = a;** (assignment on initialization), the compiler may call the copy constructor instead

```
#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base();      // Default Constructor
    ~Base();     // Class Destructor
    Base(const Base& other); // Copy Constructor
    Base& operator=(Base other); // Assignment operator
                                // implemented via copy
                                // and swap idiom
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) {
    // On copy-and-swap idiom, arg is passed
    // by value (it will trigger the copy constructor)
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b = a;
    a.x_[50] = -1;
    cout << a.x_[50] << " " << b.x_[50] << endl;
}
```

```
whovian@phy504:~/host/CPP$ g++ class34.cpp -o class34 -std=c++20
whovian@phy504:~/host/CPP$ ./class34
Default Constructor
Copy Constructor
-1 50
Destructor
Destructor
```

Memory Management

Finally, can I prevent a class from having an instance copied? Yes

We can delete the **copy constructor** and **assignment operator**

```
#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base();
    ~Base();
    Base(const Base& other) = delete;
    Base& operator=(Base other) = delete;
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b = a;
}
```

```
whovian@phy504:~/host/CPP$ g++ class35.cpp -o class35 -std=c++20
class35.cpp: In function 'int main()':
class35.cpp:29:12: error: use of deleted function 'Base::Base(const Base&)'
   29 |     Base b = a;
      |           ^
class35.cpp:10:3: note: declared here
   10 |     Base(const Base& other) = delete;
      |     ~~~~~
```

Memory Management

Finally, can I prevent a class from having an instance copied? Yes

We can delete the **copy constructor** and **assignment operator**

```
#include <iostream>
using namespace std;
constexpr int maxsz = 6000;

class Base {
public:
    double* x_; // made public
    Base();
    ~Base();
    Base(const Base& other) = delete;
    Base& operator=(Base other) = delete;
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

int main () {
    Base a;
    Base b;
    b = a;
}
```

```
whovian@phy504:~/host/CPP$ g++ class36.cpp -o class36 -std=c++20
class36.cpp: In function 'int main()':
class36.cpp:30:7: error: use of deleted function 'Base& Base::operator=(Base)'
   30 |     b = a;
      |         ^
class36.cpp:11:9: note: declared here
   11 |     Base& operator=(Base other) = delete;
      |         ~~~~~
```

```

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

class Base {
public:
    double* x_;
    // The rule of 5: Either don't declare them, or declare them all
    Base(); // Default Constructor (1)
    ~Base(); // Class Destructor (2)
    Base(const Base& other); // Copy Constructor (3)
    Base& operator=(Base other); // Assignment operator (4)
                                // implemented via copy
                                // and swap idiom
    Base(Base&& other); // Move Constructor (5)
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) { // copy-and-swap idiom
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

Base::Base(Base&& other) {
    cout << "Move Constructor" << endl;
    this->x_ = other.x_; // you copy the pointer to the data
    other.x_ = nullptr; // you nullify the other.x_ pointer
}

int main () {
    Base a;
    Base b;
    cout << "\n";
    b = std::move(a);
    cout << "\n";
    a = Base();
    cout << "\n";
}

```

Move constructors

How do you teach a class how to move their data to another object?

Move constructors

```

whovian@phy504:~/host/CPP$ g++ ref14.cpp -o ref14 -std=c++20
whovian@phy504:~/host/CPP$ ./ref14
Default Constructor
Default Constructor

Move Constructor ← No copy to arg Base other
operator = (assignment)
Destructor

Default Constructor
operator = (assignment)
Destructor

Destructor
Destructor

```

Why the rule of 5?

If you don't define the constructors, the compiler does it for you.

If you use STL containers, no memory leaks (safe).

*If you define any constructor (even with the **default** keyword), that will cause the compiler to stop creating other default constructors.*

E.g., user-defined copy constructor disables the default move constructor. If you don't understand move semantics at a high level, this may be a good

Legend

- **defaulted** = implicitly generated by the compiler
- **undeclared** = not implicitly declared
- **deleted** = implicitly declared as deleted
- **user declared** = explicitly provided by the user
- **fallback enabled** = move falls back to copy
- **fallback disabled** = no fallback move behavior

user declaration	default constructor	copy constructor	copy assignment	move constructor	move assignment	destructor
nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
any constructor	undeclared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
copy constructor	undeclared	user declared	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
copy assignment	defaulted	defaulted	user declared	undeclared (fallback enabled)	undeclared (fallback enabled)	defaulted
move constructor	undeclared	deleted	deleted	user declared	undeclared (fallback disabled)	defaulted
move assignment	defaulted	deleted	deleted	undeclared (fallback disabled)	user declared	defaulted
destructor	defaulted	defaulted	defaulted	undeclared (fallback enabled)	undeclared (fallback enabled)	user declared

Why
the rule
of 5?

The Rule of 5

If there is no move constructor, then **std::move** is equivalent to copy (expensive)

Remember: no default move constructor with user-defined copy constructor (Rule of 5)

```
#include <iostream>
#include <memory> // swap
using namespace std;
constexpr int maxsz = 1000;

class Base {
public:
    double* x_;
    Base();
    ~Base();
    Base(const Base& other);
    Base& operator=(Base other);
    // no move constructor (won't be defined by compiler)
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = i;
}

Base::~~Base() {
    cout << "Destructor" << endl;
    if(this->x_ != nullptr)
        delete [] this->x_;
}

Base::Base(const Base& other) {
    cout << "Copy Constructor" << endl;
    this->x_ = new double[maxsz];
    for (int i=0; i<maxsz; i++)
        this->x_[i] = other.x_[i];
}

Base& Base::operator=(Base other) { // copy-and-swap idiom
    cout << "operator = (assignment)" << endl;
    swap(this->x_, other.x_);
    return *this;
}

int main () {
    Base a;
    Base b;
    cout << "\n";
    b = std::move(a); // this will be done via copy
    cout << "\n";
    a = Base();
    cout << "\n";
}
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ ref15.cpp -o ref15 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./ref15
Default Constructor
Default Constructor

Copy Constructor
operator = (assignment)
Destructor

Default Constructor
operator = (assignment)
Destructor

Destructor
Destructor
```

```

#include <iostream>

using namespace std;

class Base {
public:
    double x_; // just a normal double
    Base();
    ~Base() = default;
    Base(const Base& other) = default;
    // no move constructor (won't be defined by the
    // compiler). Because the user typed explicitly
    // the copy constructor (even by setting to
    // default)
};

Base::Base() {
    cout << "Default Constructor" << endl;
    this->x_ = 10;
}

int main () {
    Base a;
    Base b;
    b = std::move(a); // this will be done via copy
    a = Base();
}

```

The Rule of 5

If there is no move constructor, then **std::move** is equivalent to copy (expensive)

If this is what you want, use the default keyword even when the copy constructor is trivial

Lambda function (again)

```

#include <iostream>
using namespace std;

class MyClass {
    int x_ = 0;
public:
    void f() {
        // capture this pointer by value, x is reference
        auto pf = [this]() {
            this->x_++;
        };

        pf();
        cout << this->x_ << endl;
        pf();
        cout << this->x_ << endl;

        // capture *this by value, x is value
        // need mutable word to not be read-only
        auto pf2 = [*this]() mutable {
            this->x_++;
        };
        this->x_ = 0;
        pf2();
        cout << this->x_ << endl;
        pf2();
        cout << this->x_ << endl;
    }
};

int main() {
    MyClass a;
    a.f();
}

```

C++ Lambdas

Capture **this** pointer vs
***this**

First case: copy the pointer **this**, a lambda can change the state of all variables in the surrounding class

Second case (***this**): copy the object. Lambda cannot change the state of the surrounding class

```

whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda7.cpp -o lambda7 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda7
1
2
0
0

```

C++ Lambdas

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

class MyClass
{
public:
    int value {123};

    auto get_value_copy()
    {
        // Capture *this by value is C++17
        return [*this] { return value; };
    }

    auto get_value_ref()
    {
        return [this] { return value; };
    }
};

int main ()
{
    MyClass a;

    auto value_copy = a.get_value_copy();
    auto value_ref = a.get_value_ref();

    a.value = 321;

    cout << "by value: " << value_copy() << endl;

    cout << "by reference: " << value_ref() <<
        endl; //(dangerous - hard to keep track)
}
```

```
whovian@phy504:~/host/Phy504Docker/CPP$ g++ lambda8.cpp -o lambda8 -std=c++20
whovian@phy504:~/host/Phy504Docker/CPP$ ./lambda8
by value: 123
by reference: 321
```

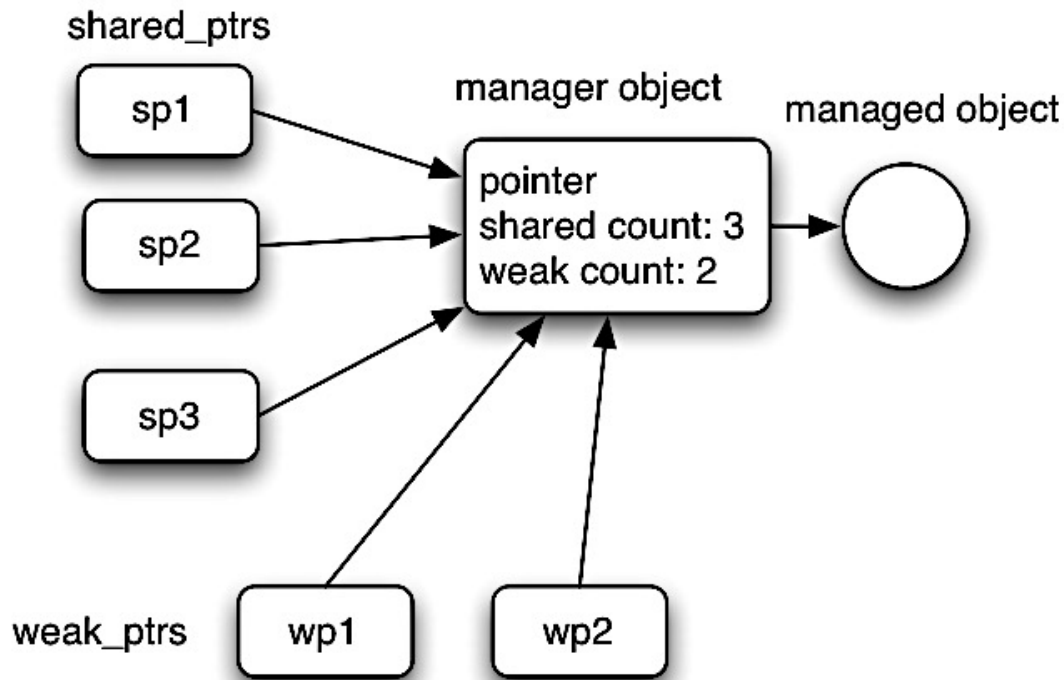
Capture **this** pointer vs
***this**

Careful when capturing by
reference, especially if you
return the lambda from a
member function

smart pointers

C++ Smart Pointers

When using **C pointers**, we learned that we need to employ **RAII** to make sure memory is deleted



Smart pointers are an upgraded version (more powerful and equally convenient) of **RAII** trick.

There are 3 types of **smart pointers**: **`unique_ptr`** (unique pointer), **`shared_ptr`** (shared pointer), **`weak_ptr`** (weak pointer)

C++ Smart Pointers

Unique Pointer
(**unique_ptr**)
uniquely own its
resource and
deletes it when the
object is destroyed

```
#include <fmt/core.h>
#include <chrono> // We only covered chrono superficially
#include <thread> // We won't have time to cover threads
#include <memory> // smart pointers
constexpr int sz = 50;
constexpr int sz2 = 30'000'000;

using namespace std::this_thread; // sleep_for, sleep_until
using namespace std::chrono_literals; // ns, us, ms, s, h, etc.
using std::chrono::system_clock;

int main()
{
    for(int i=0; i<sz; i++)
    {
        std::unique_ptr<int[]> p(new int[sz2]);
        for(int j=0; j<sz2; j++)
        {
            p[j] = j;
        }
        // Here, the std::unique_ptr gets out of scope
        // std::unique_ptr automatically deletes the
        // raw C pointer it holds.

        // How does it know how to delete the raw C pointer?
        // The template argument <int []> indicates the
        // type of C pointer (in this case something it knows
        // how to delete it

        // No memory leak!!!!!!
        sleep_until(system_clock::now() + 1s);
        fmt::print("\nIteration {0} completed\n", i);
    }
}
```

No memory leak with **unique_ptr**. We have tried a similar exercise in C and got disastrous memory leak

C++ Smart Pointers

Why **unique_ptr** is called unique pointer?

Because only one **unique_ptr** can hold the data (so no copies allowed! Move is ok!)

```
#include <fmt/core.h>
#include <chrono> // We only covered chrono superficially
#include <thread> // We won't have time to cover threads
#include <memory> // smart pointers
constexpr int sz = 50;
constexpr int sz2 = 30'000'000;

using namespace std::this_thread; // sleep_for, sleep_until
using namespace std::chrono_literals; // ns, us, ms, s, h, etc.
using std::chrono::system_clock;

int main()
{
    for(int i=0; i<sz; i++)
    {
        std::unique_ptr<int[]> p(new int[sz2]);
        for(int j=0; j<sz2; j++)
        {
            p[j] = j;
        }
        // Only one unique_ptr can hold the data
        // Move is ok (remember how move works!)
        std::unique_ptr<int[]> p2 = std::move(p);

        // here p doesn't hold the data anymore
        if (!p) {
            fmt::print("Confirmation p doesn't hold "
                       "the data anymore\n");
        }
        if (p2) {
            fmt::print("Confirmation p2 holds the data\n");
        }

        // No memory leak!!!!!!
        sleep_until(system_clock::now() + 1s);
        fmt::print("\nIteration {0} completed\n", i);
    }
}
```

C++ Smart Pointers

```
com.docker.cli
whovian@phy504:~/host/CPP$ ./sm2
Confirmation p doesn't hold the data anymore
Confirmation p2 holds the data

Iteration 0 completed
^XConfirmation p doesn't hold the data anymore
Confirmation p2 holds the data
^C
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$ nano sm2.cpp
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$
whovian@phy504:~/host/CPP$ nano sm2.cpp
whovian@phy504:~/host/CPP$ g++ sm2.cpp -o sm2 -std=c++20 -lfmt
```

only one **unique_ptr** can hold the data (so no copy allowed! Move is ok!)

C++ Smart Pointers (preview classes)

```
#include <fmt/core.h>
#include <memory> // smart pointers
using std::unique_ptr;
typedef gsl_integration_workspace gsl_int_work; // nickname

// This is a template Class (C++ metaprogramming)
// We will overview template next class (Lecture 14)
template<typename T> class Deleter;

// Template specialization. This class will work as a functor
template<> class Deleter<gsl_int_work> {
public:
    void operator()(gsl_int_work* ptr) {
        gsl_integration_workspace_free(ptr);
        return;
    }
};

// unique_ptr<A,B> has two template arguments
// A = type of the raw C pointer it holds
// B = functor that teaches the smart pointer how
//     to delete the raw C pointer
typedef unique_ptr<gsl_int_work, Deleter<gsl_int_work>>
        cpp_gsl_integration_workspace;
```

Smart Pointers
and the GSL
library

Can smart
pointers wrap C
pointers with a
complicated API
for deletion?

YES, with a few
caveats (and help
from **Templates**)

C++ Smart Pointers

Can smart pointers wrap
C pointers with a
complicated API for
deletion?

YES, with a few caveats
(and help from
Templates)

```
#include <gsl/gsl_integration.h>
#include <fmt/core.h>
#include <memory> // smart pointers
using std::unique_ptr;
typedef gsl_integration_workspace gsl_int_work; // nickname

template<typename T> class Deleter;
template<> class Deleter<gsl_int_work> {
public:
    void operator()(gsl_int_work* ptr) {
        gsl_integration_workspace_free(ptr);
        return;
    }
};
typedef unique_ptr<gsl_int_work, Deleter<gsl_int_work>>
    cpp_gsl_integration_workspace;

double f (double x, void* params) {
    double alpha = *(double *) params;
    return log(alpha*x) / sqrt(x);
}

int main (void) {
    // unique_ptr constructor can have as an arg the raw pointer
    cpp_gsl_integration_workspace
        w(gsl_integration_workspace_alloc (1000));

    double result, error, alpha = 1.0;
    gsl_function F; F.function = &f; F.params = &alpha;
    const double int_min = 0.0, int_max = 1.0;
    const double rel_error = 1e-7, abs_error = 0.0;

    // .get() returns the raw C pointer. That defeats
    // the encapsulation of unique_ptr (no copy allowed)
    // But it works, as long as you make sure that w is in scope
    gsl_integration_qags (&F, int_min, int_max, abs_error,
        rel_error, 1000, w.get(), &result, &error);

    fmt::print ("result      = {0}\n", result);
    fmt::print ("exact result  = {0}\n", -4.0);
    return 0;
}
```

```
whovian@phy504:~/host/CPP$ g++ sm5.cpp -o sm5 -std=c++20 -lfmt -lgsl -lgslcblas
whovian@phy504:~/host/CPP$ ./sm5
result      = -4.00000000000000085
exact result = -4
```

Homework I

The goal: turn a C-style linked list into C++-style range-based for loops.

1. Create a **LinkedList** class with a private inner struct **Node** (holds **int** data and **Node*** next). Add **head_**, **size_**, a constructor, and **push_front(int)**. Use **malloc** to allocate (remember it returns **void***, so you need **static_cast<Node*>**). Unlike **new**, **malloc** doesn't call constructors, so manually set the fields after allocation (if you are comfortable with the **new** operator, you can use **new** instead of **malloc**).
2. Write a destructor. Every **malloc** needs a **free**. Walk the list and free every.
3. Create an inner class **Iterator** that wraps a **Node***. This is the core of the exercise. Remember how you traverse in C: **p->data** gets the value, **p = p->next** advances, **p != NULL** checks if you're done. Your iterator does the exact same three things, just spelled as C++ operators
 1. Create **operator*()** that returns the current node's data. Same as **p->data**.
 2. Create **operator++()** that advances to the next node and returns ***this**.
 3. Create **operator++(int)** the **int** is a dummy parameter that distinguishes post-increment (**it++**) from pre-increment. Save the old state, advance, return the old state. Think about why: ***it++** should give you the current value and THEN move forward.
 4. Create **operator!=()**: compares two iterators by comparing their internal pointers.
4. Add **begin()** and **end()** to **LinkedList**. **begin()** returns an **Iterator** wrapping **head_**. **end()** returns an **Iterator** wrapping **nullptr**. These two functions are what the range-based for loop calls behind the scenes.
5. Test it. Create a list, push some values, and use **for (int x : list)**. Also, try using explicit iterators with both **++it** and **it++** to see the difference between pre- and post-increment.

Homework II

1. Create a class `Sensor` with a constructor that prints "Sensor created", a destructor that prints "Sensor destroyed", and a **`read() const`** that returns a **`double`**. Allocate one with **`new`**, call **`read()`**, then **`delete`** it. Verify the destructor runs.
2. Write a function **`process(Sensor* s)`** that returns **`true`** or false depending on whether **`s->read()`** is above a threshold. In **`main()`**, allocate a **`Sensor`** with **`new`**, call `process`, and if it returns **`false`**, print an error and return 1 early. Run the program with a value that triggers the early return. Observe: the destructor never prints. The memory leaked.
3. Fix it by adding **`delete`** before every **`return`**. Now add a second early return (e.g., another validation check). And a third. Notice how every exit path needs its own `delete`. This is the fundamental problem: the more complex your function gets, the easier it is to forget a `delete` on one path.

Homework III

1. Write your own class **SensorGuard** that takes a **Sensor*** (from HW II) in its constructor and calls delete on it in its destructor.
2. In **main()**, write **SensorGuard guard(new Sensor());**. Call the same read process that has an early exit. Verify that the Sensor is destroyed automatically when the guard goes out of scope.
3. Replaces the **SensorGuard** wrapper by **std::unique_ptr**