Invoking the base constructor

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

class Base {
public:
    Base() { i = 21; d = 22.2; }
    Base(int i):i(i),d(22.2){}
    Base(double d):i(21),d(d){}
    void display() { cout << "i: " << i << ", d: " << d << endl; }
private:
    int i;
    double d;
};
class Derived : public Base {
public:
    Derived(int i): Base(i){}
    Derived(double d): Base(d){}
};

int main() {
    Derived d(0);
    d.display();
}
```

```
i: 0, d: 22.2
```

Invoking the base constructor

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

class Base {
public:
    Base() { i = 21; d = 22.2; }
    Base(int i):i(i),d(22.2){}
    Base(double d):i(21),d(d){}
    void display() { cout << "i: " << i << ", d: " << d << endl; }
private:
    int i;
    double d;
};
class Derived : public Base {
public:
    Derived(int i): Base(i){}
    Derived(double d): Base(d){}
};

int main() {
    Derived d;
    d.display();
}
```

Invoking the base constructor – cont.

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

class Base {
public:
    Base() { i = 21; d = 22.2; }
    Base(int i):i(i),d(22.2){}
    Base(double d):i(21),d(d){}
    void display() { cout << "i: " << i << ", d: " << d << endl; }
private:
    int i;
    double d;
};
class Derived : public Base {
public:
    Derived(int i): Base(i){}
    Derived(double d): Base(d){}
};

int main() {
    Derived d;
    d.display();
}
```

```
Build started...
1>------ Build started: Project: cs2124, Configuration: Debug x64 ------
1>C:\MyPrograms\MicrosoftVisualStudio_2022_Community\MSBuild\Microsoft\VC\v170\Microsoft.CppBuild.targets(531,5): warning MSB8028: The intermediate directory (x64\Debug\) contains files shared from another project (Lect_01_B.vcxproj).  This can lead to incorrect clean and rebuild behavior.
1>temp.cpp
1>C:\MyDropbox\Dropbox\CS2124_OOP_24S\temp.cpp(23,13): error C2512: 'Derived': no appropriate default constructor available
1>C:\MyDropbox\Dropbox\CS2124_OOP_24S\temp.cpp(15,7): message : see declaration of 'Derived'
1>Done building project "cs2124.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
========== Build started at 12:03 PM and took 00.567 seconds ==========
```

- Compilation error – no default constructor

Constructor Inheritance

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

class Base {
public:
    Base() { i = 21; d = 22.2; }
    Base(int i):i(i),d(22.2){}
    Base(double d):i(21),d(d){}
    void display() { cout << "i: " << i << ", d: " << d << endl; }
private:
    int i;
    double d;
};
class Derived : public Base {
public:
    using Base::Base;
};

int main() {
    Derived d;
    d.display();
}
```

```
i: 21, d: 22.2
```

- The `using` statement causes a class to inherit its base class's constructors

Redefinition and hiding

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

class Base {
public:
    void foo(int n) const { cout << "Base::foo(n)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    //der.foo(17); // fails to compile now. Base version is "hidden"
    der.Base::foo(17); // can work around it with a direct call
}
```

- Call to der.foo(17) now fails.
- Definition of Derived::foo() "hides" any definition of foo in Base
- Can work around it, calling the Base version directly

Redefinition and hiding – cont.

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

class Base {
public:
    void foo(int n) const { cout << "Base::foo(n)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
};

int main() {
    Derived der;
    //der.foo(17); // fails to compile now. Base version is "hidden"
    der.Base::foo(17); // can work around it with a direct call
}
```

- A method in the derived class can **NEVER** overload a method in the base class
    - It hides it!
    - Work around that by scoping (using base class, i.e. `der.Base::(17);`)

Redefinition and hiding – cont.

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

class Base {
public:
    void foo(int n) const { cout << "Base::foo(n)\n"; }
};

class Derived : public Base {
public:
    void foo() const { cout << "Derived::foo()\n"; }
    void foo(int n) const { Base::foo(n); }
};

int main() {
    Derived der;
    der.foo(17); // Works again! Calls version in derived, which
    // "wraps" call to base version.
}
```

- Call to der.foo(17) now succeeds by having Derived::foo(int) call Base version

Overloading with inheritance

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

class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

void func(Parent& base) { cout << "func(Parent)\n"; }
void func(Child& derived) {cout << "func(Child)\n"; }

void otherFunc(Parent& base) {
    func(base);
}

int main() {
    Parent parent;
    func(parent);
    Child child;
    func(child);
    Grandchild gc;
    func(gc);

    otherFunc(child);
}
```

??

- How does the compiler make the choice for overloading, when the parameters are related by inheritance?

Overloading with inheritance – cont.

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

class Parent {};
class Child : public Parent {};
class Grandchild : public Child {};

void func(Parent& base) { cout << "func(Parent)\n"; }
void func(Child& derived) {cout << "func(Child)\n"; }

void otherFunc(Parent& base) {
    func(base);
}

int main() {
    Parent parent;
    func(parent);
    Child child;
    func(child);
    Grandchild gc;
    func(gc);

    otherFunc(child);
}
```

```
func(Parent)
func(Child)
func(Child)
func(Parent)
```

- How does the compiler make the choice for overloading based, when the paramters are related by inheritance?

Overloading vs overriding

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

class Parent {
public:
    virtual void whereami() const {
        cout << "Parent" << endl;
    }
};
class Child : public Parent {
public:
    void whereami() const {
        cout << "Child!!!" << endl;
    }
};
class Grandchild : public Child {
public:
    void whereami() const {
        cout << "Grandchild!!!" << endl;
    }
};

void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) {cout << "func(Child)\n"; }
void otherFunc(Parent& base) {
    func(base);
    base.whereami();
}

int main() {
    Parent parent;
    otherFunc(parent); // obvious, hopefully

    Child child;
    otherFunc(child); // less obvious?

    Grandchild gc;
    otherFunc(gc); // ok, by now you know what happens
}
```

??

Overloading vs overriding

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

class Parent {
public:
    virtual void whereami() const {
        cout << "Parent" << endl;
    }
};
class Child : public Parent {
public:
    void whereami() const {
        cout << "Child!!!" << endl;
    }
};
class Grandchild : public Child {
public:
    void whereami() const {
        cout << "Grandchild!!!" << endl;
    }
};

void func(const Parent& base) { cout << "func(Parent)\n"; }
void func(const Child& derived) {cout << "func(Child)\n"; }
void otherFunc(Parent& base) {
    func(base);
    base.whereami();
}

int main() {
    Parent parent;
    otherFunc(parent); // obvious, hopefully

    Child child;
    otherFunc(child); // less obvious?

    Grandchild gc;
    otherFunc(gc); // ok, by now you know what happens
}
```

```
func(Parent)
Parent
func(Parent)
Child!!!
func(Parent)
Grandchild!!!
```

- Compiler makes the choice for **overloading at compile time**
  - Based on declared type
- And for **overriding at run time**.
  - Based on actual type

Polymorphic free functions

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

class Base {
public:
    friend ostream& operator<<(ostream& os, const Base& rhs);
};
class Derived : public Base {
public:
    friend ostream& operator<<(ostream& os, const Derived& rhs);
private:
    int x=17;
};

ostream& operator<<(ostream& os, const Base& rhs) {
    os << "Base";
    return os;
}
ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived, x := " << rhs.x;
    return os;
}

void func(const Base& base) {
    cout << base << endl;
}

int main() {
    Derived der;
    func(der);
}
```

- What will this print?

Polymorphic free functions

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

class Base {
public:
    friend ostream& operator<<(ostream& os, const Base& rhs);
};
class Derived : public Base {
public:
    friend ostream& operator<<(ostream& os, const Derived& rhs);
private:
    int x=17;
};

ostream& operator<<(ostream& os, const Base& rhs) {
    os << "Base";
    return os;
}
ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived, x := " << rhs.x;
    return os;
}

void func(const Base& base) {
    cout << base << endl;
}

int main() {
    Derived der;
    func(der);
}
```

Base

- Not a polymorphic calls
- Overloading is resolved at compile time
  - Doesn't care about **"actual" (dynamic)** type of object
  - Only cares about **"declared" (static)** type of the object)
- **How can we call a non-member function and get it to behave polymorphically**?

Polymorphic free functions

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

class Base {
public:
    friend ostream& operator<<(ostream& os, const Base& rhs);
};
class Derived : public Base {
public:
    friend ostream& operator<<(ostream& os, const Derived& rhs);
private:
    int x=17;
};

ostream& operator<<(ostream& os, const Base& rhs) {
    os << "Base";
    return os;
}
ostream& operator<<(ostream& os, const Derived& rhs) {
    os << "Derived, x := " << rhs.x;
    return os;
}

void func(const Base& base) {
    cout << base << endl;
}

int main() {
    Derived der;
    func(der);
}
```

Base

- **How can we call a non-member function and get it to behave polymorphically?**
- **In other words, how can we make this a polymorphic call?**
  - **Can we create a "virtual" free function?**

Polymorphic free functions

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

class Base {
public:
    friend ostream& operator<<(ostream& os, const Base& rhs);
    virtual void display(ostream& os) const { os << "Base"; }
};
class Derived : public Base {
public:
    friend ostream& operator<<(ostream& os, const Derived& rhs);
    virtual void display(ostream& os) const { os << "Derived, x := "
<< x; }
private:
    int x=17;
};

ostream& operator<<(ostream& os, const Base& rhs) {
    rhs.display(os);
//    os << "Base";
    return os;
}
//ostream& operator<<(ostream& os, const Derived& rhs) {
//    rhs.display(os);
//    os << "Derived, x := " << rhs.x;
//    return os;
//}

void func(const Base& base) {
    cout << base << endl;
}

int main() {
    Derived der;
    func(der);
}
```

Derived, x := 17

- Now, that's a polymorphic call!

Can we have polymorphic member functions ?

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

class Base {
public:
    Base() {}
    virtual void foo() const { cout << "Base\n"; }
    void display() { foo(); }
};

class Derived : public Base {
public:
    Derived(int n) : x(n) {}
    void foo() const { cout << "Derived: x == " << x << endl; }
private:
    int x;
};

int main() {
    Derived der(17);
    der.display();
}
```

What will be displayed here?

- "Base"

OR

- "Derived: x== 17"

Polymorphic member functions

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

class Base {
public:
    Base() {}
    virtual void foo() const { cout << "Base\n"; }
    void display() { foo(); }
};

class Derived : public Base {
public:
    Derived(int n) : x(n) {}
    void foo() const { cout << "Derived: x == " << x << endl; }
private:
    int x;
};

int main() {
    Derived der(17);
    der.display();
}
```

```
Derived: x == 17
```

- display() is defined in Base
- But when it invokes foo(), the first parameter is implicit and it is a pointer to the object involved

- Every call to member functions (e.g. foo() in the display() method) is invoked as this->foo()
  - foo() is a polymorphic call
  - display() is a polymorphic member function

Inheritance with copy control

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout << "Base()\n"; }
    Base(const Base& rhs) {
        cout << "Base(const Base&)\n";
    }
    virtual ~Base() { cout << "~Base()\n"; }
    Base& operator=(const Base& rhs) {
        cout << "Base::operator=(const Base&)\n";
        return *this;
    }
};
class Member {
public:
    Member() { cout << "Member()\n"; }
    Member(const Member& rhs) { cout << "Member(const Member&)\n"; }
    Member& operator=(const Member& rhs) {
        cout << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() { cout << "~Member()\n"; }
};
class Derived : public Base {
public:
    Derived() { cout << "Derived()\n"; }
    ~Derived() {
        cout << "~Derived()\n";
    }
    Derived(const Derived& rhs) : Base(rhs), member(rhs.member) {
        cout << "Derived(const Derived&)\n";
    }
    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        member = rhs.member;
        cout << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member member;
};
```

Base destructor:

- We need to mark the **Base class destructor as virtual** for the delete bp; line in the test code to use polymorphism

Derived destructor:

- User code **then**
- Implicitly calls the destructors for the non-primitive fields, **then**
- Implicitly calls the base class destructor

Inheritance with copy control

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout << "Base()\n"; }
    Base(const Base& rhs) {
        cout << "Base(const Base&)\n";
    }
    virtual ~Base() { cout << "~Base()\n"; }
    Base& operator=(const Base& rhs) {
        cout << "Base::operator=(const Base&)\n";
        return *this;
    }
};
class Member {
public:
    Member() { cout << "Member()\n"; }
    Member(const Member& rhs) { cout << "Member(const Member&)\n"; }
    Member& operator=(const Member& rhs) {
        cout << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() { cout << "~Member()\n"; }
};
class Derived : public Base {
public:
    Derived() { cout << "Derived()\n"; }
    ~Derived() {
        cout << "~Derived()\n";
    }
    Derived(const Derived& rhs) : Base(rhs), member(rhs.member) {
        cout << "Derived(const Derived&)\n";
    }
    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        member = rhs.member;
        cout << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member member;
};
```

Derived copy constructor:

- Like any derived constructor, **first**, it implicitly calls the base class constructor
  - If we don't specify which base class constructor then it will call the base class default constructor here → Wrong choice!
  - We need to invoke the base class copy constructor
    - Works due to the principle of substitutability, i.e. the "this" pointer is pointing to a derived object, but the base constructor is invoked
- **Then**, non-primitive member constructors. We also call the non-primitive member variable copy constructors in our initialization list.

Inheritance with copy control

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout << "Base()\n"; }
    Base(const Base& rhs) {
        cout << "Base(const Base&)\n";
    }
    virtual ~Base() { cout << "~Base()\n"; }
    Base& operator=(const Base& rhs) {
        cout << "Base::operator=(const Base&)\n";
        return *this;
    }
};
class Member {
public:
    Member() { cout << "Member()\n"; }
    Member(const Member& rhs) { cout << "Member(const Member&)\n"; }
    Member& operator=(const Member& rhs) {
        cout << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() { cout << "~Member()\n"; }
};
class Derived : public Base {
public:
    Derived() { cout << "Derived()\n"; }
    ~Derived() {
        cout << "~Derived()\n";
    }
    Derived(const Derived& rhs) : Base(rhs), member(rhs.member) {
        cout << "Derived(const Derived&)\n";
    }
    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        member = rhs.member;
        cout << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member member;
};
```

Derived copy constructor:

- Order of non-primitive member constructors does NOT follow order in the initialization list
- It rather **follows the order of declaration**
    - This is the source of MANY BUGS, which I have personally witnessed!

Inheritance with copy control

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout << "Base()\n"; }
    Base(const Base& rhs) {
        cout << "Base(const Base&)\n";
    }
    virtual ~Base() { cout << "~Base()\n"; }
    Base& operator=(const Base& rhs) {
        cout << "Base::operator=(const Base&)\n";
        return *this;
    }
};
class Member {
public:
    Member() { cout << "Member()\n"; }
    Member(const Member& rhs) { cout << "Member(const Member&)\n"; }
    Member& operator=(const Member& rhs) {
        cout << "Member::operator=(const Member&)\n";
        return *this;
    }
    ~Member() { cout << "~Member()\n"; }
};
class Derived : public Base {
public:
    Derived() { cout << "Derived()\n"; }
    ~Derived() {
        cout << "~Derived()\n";
    }
    Derived(const Derived& rhs) : Base(rhs), member(rhs.member) {
        cout << "Derived(const Derived&)\n";
    }
    Derived& operator=(const Derived& rhs) {
        Base::operator=(rhs);
        member = rhs.member;
        cout << "Derived::operator=(const Derived&)\n";
        return *this;
    }
private:
    Member member;
};
```

<u>Derived op=</u>

- Since there is no special syntax for invoking the base op=, we just call it
- Similarly, we have to assign all of the member variables.