1 Operator Overloading

Operator Overloading

What is Operator Overloading?

- Using traditional operators such as +, =, *, etc. with user-defined types
- Allows user defined types to behave similar to built-in types
- Can make code more readable and writable
- Not done automatically (except for the assignment operator)
  They must be explicitly defined

Operator Overloading

What is Operator Overloading?

Suppose we have a Number class that models any number

- Using functions:

```
Number result = multiply(add(a,b),divide(c,d));
```

- Using member methods:

```
Number result = (a.add(b)).multiply(c.divide(d));
```


Operator Overloading

What is Operator Overloading?

Suppose we have a Number class that models any number

- Using overloaded operators

```
Number result = (a+b)*(c/d);
```


Operator Overloading

What operators can be overloaded?

- The majority of C++'s operators can be overloaded
- The following operators cannot be overload

operator
::
:?
.*
.
sizeof

Operator Overloading

Some basic rules

- Precedence and Associativity cannot be changed
- 'arity' cannot be changed (i.e. can't make the division operator unary)
- Can't overload operators for primitive type (e.g. `int`, `double`, `etc.`)
- Can't create new operators
- `[]`, `()`, `->`, and the assignment operator (`=`) **must** be declared as member methods
- Other operators can be declared as member methods or global functions

Operator Overloading

Some examples

- **int**
  ```
  a = b + c
  a < b
  std::cout << a
  ```
- **double**
  ```
  a = b + c
  a < b
  std::cout << a
  ```
- **long**
  ```
  a = b + c
  a < b
  std::cout << a
  ```
- **std::string**
  ```
  s1 = s2 + s3
  S1 < s2
  std::cout << s1
  ```
- **Mystring**
  ```
  s1 = s2 + s3
  s1 < s2
  s1 == s2
  std::cout << s1
  ```
- **Player**
  ```
  p1 < p2
  p1 == p2
  std::cout << p1
  ```


Operator Overloading

Mystring class declaration

```
class Mystring {

private:
    char *str; // C-style string

public:
    Mystring();
    Mystring(const char *s);
    Mystring(const Mystring &source);
    ~Mystring();
    void display() const;
    int get_length() const;
    const char *get_str() const;
};
```


Operator Overloading

Copy assignment operator (=)

- C++ provides a default assignment operator used for assigning one object to another

```
Mystring s1 {"Frank"};
Mystring s2 = s1;        // NOT assignment
                         // same as Mystring s2{s1};

s2 = s1;                 // assignment
```

- Default is memberwise assignment (shallow copy)
  - If we have raw pointer data member we must deep copy

Operator Overloading

Overloading the copy assignment operator (deep copy)

```
Type &Type::operator=(const Type &rhs);


Mystring &Mystring::operator=(const Mystring &rhs);

s2 = s1;                 // We write this

s2.operator=(s1);        // operator= method is called
```


Operator Overloading

Overloading the copy assignment operator (deep copy)

```
Mystring &Mystring::operator=(const Mystring &rhs) {
   if (this == &rhs)
      return *this;

   delete [] str;
   str = new char[std::strlen(rhs.str) + 1];
   std::strcpy(str, rhs.str);

   return *this;
}
```


Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Check for self assignment

```
if (this == &rhs)      // p1 = p1;
   return *this;       // return current object
```

- Deallocate storage for `this->str` since we are overwriting it

```
delete [] str;
```


Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Allocate storage for the deep copy

```
str = new char[std::strlen(rhs.str) + 1];
```

Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Perform the copy

```
std::strcpy(str, rhs.str);
```

- Return the current by reference to allow chain assignment

```
return *this;


// s1 = s2 = s3;
```


Operator Overloading

Move assignment operator (=)

- You can choose to overload the move assignment operator
  - C++ will use the copy assignment operator if necessary

```
Mystring s1;

s1 = Mystring {"Frank"}; // move assignment
```

- If we have raw pointer we should overload the move assignment operator for efficiency

Operator Overloading

Overloading the Move assignment operator

```
Type &Type::operator=(Type &&rhs);


Mystring &Mystring::operator=(Mystring &&rhs);

s1 = Mystring("Joe"); // move operator= called

s1 = "Frank";         // move operator= called
```


Operator Overloading

Overloading the Move assignment operator – steps

- Check for self assignment

```
if (this == &rhs)
   return *this;         // return current object
```

- Deallocate storage `for this->str` since we are overwriting it

```
delete [] str;
```


Operator Overloading

Overloading the Move assignment operator – steps for deep copy

- Steal the pointer from the rhs object and assign it to this->str

```
str = rhs.str;
```

- Null out the rhs pointer

```
rhs.str = nullptr;
```

- Return the current object by reference to allow chain assignment

```
return *this;
```


Operator Overloading

Unary operators as member methods (++, --, -, !)

```
ReturnType Type::operatorOp();


Number Number::operator-() const;
Number Number::operator++();        // pre-increment
Number Number::operator++(int);     // post-increment
bool Number::operator!() const;

Number n1 {100};
Number n2 = -n1;                    // n1.operator-()
n2 = ++n1;                          // n1.operator()
n2 = n1++;                          // n1.operator++(int)
```


Operator Overloading

Mystring operator- make lowercase

```
Mystring larry1 {"LARRY"};
Mystring larry2;

larry1.display();          // LARRY

larry2 = -larry1;          // larry1.operator-()

larry1.display();          // LARRY
larry2.display();          // larry
```


Operator Overloading

Mystring operator- make lowercase

```
Mystring Mystring::operator-() const {
   char *buff = new char[std::strlen(str) + 1];
   std::strcpy(buff, str);
   for (size_t i=0; i<std::strlen(buff); i++)
      buff[i] = std::tolower(buff[i]);
   Mystring temp {buff};
   delete [] buff;
   return temp;
}
```


Operator Overloading

Binary operators as member methods (+,-,==,!=,<,>, etc.)

```
ReturnType Type::operatorOp(const &Type rhs);


Number Number::operator+(const &Number rhs) const;
Number Number::operator-(const &Number rhs) const;
bool Number::operator==(const &Number rhs) const;
bool Number::operator<(const &Number rhs) const;

Number n1 {100}, n2 {200};
Number n3 = n1 + n2;                // n1.operator+(n2)
n3 = n1 - n2;                       // n1.operator-(n2)
if (n1 == n2) . . .                 // n1.operator==(n2)
```


Operator Overloading

Mystring `operator==`

```
bool Mystring::operator==(const Mystring &rhs) const {
   if (std::strcmp(str, rhs.str) == 0)
      return true;
   else
      return false;
}

// if (s1 == s2)        // s1 and s2 are Mystring objects
```


Operator Overloading

Mystring `operator+ (concatenation)`

```
Mystring larry {"Larry"};
Mystring moe{"Moe"};
Mystring stooges {" is one of the three stooges"};

Mystring result = larry + stooges;
                  // larry.operator+(stooges);

result = moe +  " is also a stooge";
                  // moe.operator+("is also a stooge");

result = "Moe" + stooges; // "Moe".operator+(stooges) ERROR
```


Operator Overloading

Mystring `operator+ (concatenation)`

```
Mystring Mystring::operator+(const Mystring &rhs) const {
   size_t buff_size = std::strlen(str) +
                      std::strlen(rhs.str) + 1;
   char *buff = new char[buff_size];
   std::strcpy(buff, str);
   std::strcat(buff, rhs.str);
   Mystring temp {buff};
   delete [] buff;
   return temp;
}
```

Operator Overloading

Unary operators as global functions (++, --, -, !)

```
ReturnType operatorOp(Type &obj);

Number operator-(const Number &obj);
Number operator++(Number &obj);          // pre-increment
Number operator++(Number &obj, int);     // post-increment
bool   operator!(const Number &obj);

Number n1 {100};
Number n2 = -n1;                 // operator-(n1)
n2 = ++n1;                       // operator++(n1)
n2 = n1++;                       // operator++(n1,int)
```


Operator Overloading

Mystring `operator-` make lowercase

```
Mystring larry1 {"LARRY"};
Mystring larry2;

larry1.display();                // LARRY

larry2 = -larry1;                // operator-(larry1)

larry1.display();                // LARRY
larry2.display();                // larry
```


Operator Overloading

Mystring `operator-`

- Often declared as **friend** functions in the class declaration

```
Mystring operator-(const Mystring &obj) {
   char *buff = new char[std::strlen(obj.str) + 1];
   std::strcpy(buff, obj.str);
   for (size_t i=0; i<std::strlen(buff); i++)
      buff[i] = std::tolower(buff[i]);
   Mystring temp {buff};
   delete [] buff;
   return temp;
}
```


Operator Overloading

Binary operators as global functions (+,-,==,!=,<,>, etc.)

```
ReturnType operatorOp(const &Type lhs, const &Type rhs);

Number operator+(const &Number lhs, const &Number rhs);
Number operator-(const &Number lhs, const &Number rhs);
bool   operator==(const &Number lhs, const &Number rhs);
bool   operator<(const &Number lhs, const &Number rhs);

Number n1 {100}, n2 {200};
Number n3 = n1 + n2;             // operator+(n1,n2)
n3 = n1 - n2;                    // operator-(n1,n2)
if (n1 == n2) . . .              // operator==(n1,n2)
```


Operator Overloading

Mystring `operator==`

```
bool operator==(const Mystring &lhs, const Mystring &rhs){
   if (std::strcmp(lhs.str, rhs.str) == 0)
      return true;
   else
      return false;
}
```

- If declared as a friend of `Mystring` can access private `str` attribute
- Otherwise must use getter methods

Operator Overloading

Mystring `operator+ (concatenation)`

```
Mystring larry {"Larry"};
Mystring moe{"Moe"};
Mystring stooges {" is one of the three stooges"};

Mystring result = larry + stooges;
                  // operator+(larry, stooges);

result = moe +  " is also a stooge";
                  // operator+(moe, "is also a stooge");

result = "Moe" + stooges; // OK with non-member functions
```


Operator Overloading

Mystring `operator+ (concatenation)`

```
Mystring operator+(const Mystring &lhs, const myString &rhs) {
   size_t buff_size = std::strlen(lhs.str) +
                      std::strlen(rhs.str) + 1;
   char *buff = new char[buff_size];
   std::strcpy(buff, lhs.str);
   std::strcat(buff, rhs.str);
   Mystring temp {buff};
   delete [] buff;
   return temp;
}
```


Operator Overloading

stream insertion and extraction operators (<<, >>)

```
Mystring larry {"Larry"};

cout << larry << endl;   // Larry


Player hero {"Hero", 100, 33};

cout << hero << endl;    // {name: Hero, health: 100, xp:33}
```


Operator Overloading

stream insertion and extraction operators (<<, >>)

```
Mystring larry;

cin >> larry;


Player hero;

cin >> hero;
```


Operator Overloading

stream insertion and extraction operators (<<, >>)

- Doesn't make sense to implement as methods
  - Left operand must be a user-defined class
  - Not the way we normally use these operators

```
Mystring larry;
larry << cout;          // huh?

Player hero;
hero >> cin;            // huh?
```


Operator Overloading

stream insertion operator (<<)

```
std::ostream &operator<<(std::ostream &os, const Mystring &obj) {
    os << obj.str;        // if friend function
    // os << obj.get_str();  // if not friend function
    return os;
}
```

- Return a reference to the `ostream` so we can keep inserting
- Don't return `ostream` by value!

Operator Overloading

stream extraction operator (>>)

```
std::istream &operator>>(std::istream &is, Mystring &obj) {
    char *buff = new char[1000];
    is >> buff;
    obj = Mystring{buff};// If you have copy or move assignment
    delete [] buff;
    return is;
}
```

- Return a reference to the `istream` so we can keep inserting
- Update the object passed in

2 Inheritance

Inheritance

What is it and why is it used?

- Provides a method for creating new classes from existing classes
- The new class contains the data and behaviors of the existing class
- Allow for reuse of existing classes
- Allows us to focus on the common attributes among a set of classes
- Allows new classes to modify behaviors of existing classes to make it unique
  - Without actually modifying the original class

Inheritance

Related classes

- Player, Enemy, Level Boss, Hero, Super Player, etc.
- Account, Savings Account, Checking Account, Trust Account, etc.
- Shape, Line, Oval, Circle, Square, etc.
- Person, Employee, Student, Faculty, Staff, Administrator, etc.

Inheritance

Accounts

- Account
  - **balance**, **deposit**, **withdraw**, . . .
- Savings Account
  - **balance**, **deposit**, **withdraw**, interest rate, . . .
- Checking Account
  - **balance**, **deposit**, **withdraw**, minimum balance, per check fee, . . .
- Trust Account
  - **balance**, **deposit**, **withdraw**, interest rate, . . .

Inheritance

Accounts – without inheritance – code duplication

```
class Account {
    // balance, deposit, withdraw, . . .
};

class Savings_Account {
    // balance, deposit, withdraw, interest rate, . . .
};

class Checking_Account {
    // balance, deposit, withdraw, minimum balance, per check fee, . . .
};

class Trust_Account {
    // balance, deposit, withdraw, interest rate, . . .
};
```


Inheritance

Accounts – with inheritance – code reuse

```
class Account {
    // balance, deposit, withdraw, . . .
};

class Savings_Account : public Account {
    // interest rate, specialized withdraw, . . .
};

class Checking_Account : public Account {
    // minimum balance, per check fee, specialized withdraw, . . .
};

class Trust_Account : public Account {
    // interest rate, specialized withdraw, . . .
};
```


Inheritance

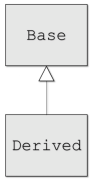
Terminology

- Inheritance
  - Process of creating new classes from existing classes
  - Reuse mechanism
- Single Inheritance
  - A new class is created from another 'single' class
- Multiple Inheritance
  - A new class is created from two (or more) other classes

Inheritance

Terminology

- Base class (parent class, super class)
  - The class being extended or inherited from
- Derived class (child class, sub class)
  - The class being created from the Base class
  - Will inherit attributes and operations from Base class



Inheritance

Terminology

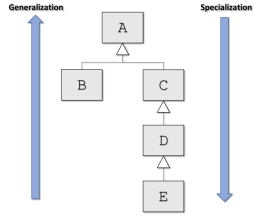
- "Is-A" relationship
  - Public inheritance
  - Derived classes are sub-types of their Base classes
  - Can use a derived class object wherever we use a base class object
- Generalization
  - Combining similar classes into a single, more general class based on common attributes
- Specialization
  - Creating new classes from existing classes proving more specialized attributes or operations
- Inheritance or Class Hierarchies
  - Organization of our inheritance relationships

Inheritance

Class hierarchy

Classes:

- A
- B is derived from A
- C is derived from A
- D is derived from C
- E is derived from D

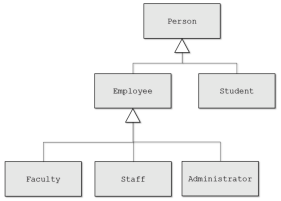


Inheritance

Class hierarchy

Classes:

- Person
- Employee is derived from Person
- Student is derived from Person
- Faculty is derived from Employee
- Staff is derived from Employee
- Administrator is derived from Employee



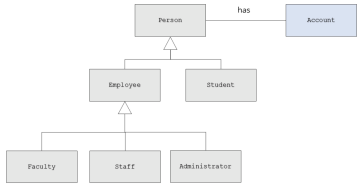
Inheritance

Public Inheritance vs. Composition

- Both allow reuse of existing classes
- Public Inheritance
  - "is-a" relationship
    - Employee 'is-a' Person
    - Checking Account 'is-a' Account
    - Circle "is-a" Shape
- Composition
  - "has-a" relationship
    - Person "has a" Account
    - Player "has-a" Special Attack
    - Circle "has-a" Location

Inheritance

Public Inheritance vs. Composition



Inheritance

Composition

```
class Person {
private:
   std::string name;    // has-a name
   Account account; // has-a account
};
```


Deriving classes from exiting classes

C++ derivation syntax

```
class Base {
  // Base class members . . .
};

class Derived: access-specifier Base {
   // Derived class members . . .
};
```

Access-specifier can be: `public`, `private`, or `protected`

Deriving classes from exiting classes

Types of inheritance in C++

- `public`
  - Most common
  - Establishes '**is-a**' relationship between Derived and Base classes
- `private` and `protected`
  - Establishes "derived class **has a** base class" relationship
  - "Is implemented in terms of" relationship
  - Different from composition

Deriving classes from exiting classes

C++ derivation syntax

```
class Account {
  // Account class members . . .
};

class Savings_Account: public Account {
   // Savings_Account class members . . .
};
```

`Savings_Account` 'is-a' `Account`

Deriving classes from exiting classes

C++ creating objects

```
Account account {};
Account *p_account = new Account();

account.deposit(1000.0);
p_account->withdraw(200.0);

delete p_account;
```


Deriving classes from exiting classes

C++ creating objects

```
Savings_Account sav_account {};
Savings_Account *p_sav_account = new Savings_Account();

sav_account.deposit(1000.0);
p_sav_account->withdraw(200.0);

delete p_sav_account;
```


Protected Members and Class Access

The protected class member modifier

```
class Base {

  protected:
    // protected Base class members . . .
};
```

- Accessible from the Base class itself
- Accessible from classes Derived from Base
- Not accessible by objects of Base or Derived

Protected Members and Class Access

The protected class member modifier

```
class Base {
   public:
     int a;   // public Base class members . . .

   protected:
     int b;   // protected Base class members . . .

   private:
     int c;   // private Base class members . . .

};
```


Deriving classes from exiting classes

Access with **`public`** inheritance

Base Class		Access in Derived Class

`public: a`
`protected: b`
`private: c`
	public inheritance	`public: a`
		`protected: b`
		`c : no access`

Deriving classes from exiting classes

Access with **`protected`** inheritance

Base Class		Access in Derived Class

`public: a`
`protected: b`
`private: c`
	protected inheritance	**`protected`**`: a`
		`protected: b`
		`c : no access`

Constructors and Destructors

Constructors and class initialization

- A Derived class inherits from its Base class
- The Base part of the Derived class MUST be initialized BEFORE the Derived class is initialized
- When a Derived object is created
  - Base class constructor executes then
  - Derived class constructor executes

Constructors and Destructors

Constructors and class initialization

```
class Base {
public:
   Base(){ cout << "Base constructor" << endl; }
};

class Derived : public Base {
public:
   Derived(){ cout << "Derived constructor " << endl; }
};
```

Constructors and Destructors

Constructors and class initialization

```
                                Output
Base base;                      Base constructor

Derived derived;                Base constructor
                                Derived constructor
```


Constructors and Destructors

Destructors

- Class destructors are invoked in the reverse order as constructors
- The Derived part of the Derived class MUST be destroyed BEFORE the Base class destructor is invoked
- When a Derived object is destroyed
  - Derived class destructor executes then
  - Base class destructor executes
  - Each destructor should free resources allocated in it's own constructors

Constructors and Destructors

Destructors

```
class Base {
public:
   Base(){ cout << "Base constructor" << endl; }
   ~Base(){ cout << "Base destructor" << endl; }
};

class Derived : public Base {
public:
   Derived(){ cout << "Derived constructor " << endl; }
   ~Derived(){ cout << "Derived destructor " << endl; }
};
```


Constructors and Destructors

Constructors and class initialization

```
                                Output
Base base;                      Base constructor
                                Base destructor

Derived derived;                Base constructor
                                Derived constructor
                                Derived destructor
                                Base destructor
```


Constructors and Destructors

Constructors and class initialization

- A Derived class does NOT inherit
  - Base class constructors
  - Base class destructor
  - Base class overloaded assignment operators
  - Base class friend functions
- However, the base class constructors, destructors, and overloaded assignment operators can invoke the base-class versions
- C++11 allows explicit inheritance of base 'non-special' constructors with
  - `using Base::Base;` anywhere in the derived class declaration
  - Lots of rules involved, often better to define constructors yourself

Inheritance

Passing arguments to base class constructors

- The Base part of a Derived class must be initialized first
- How can we control exactly which Base class constructor is used during initialization?
- We can invoke the whichever Base class constructor we wish in the initialization list of the Derived class

Inheritance

Passing arguments to base class constructors

```
class Base {
public:
   Base();
   Base(int);
   . . .
};

Derived::Derived(int x)
   : Base(x), {optional initializers for Derived} {
   // code
}
```


Constructors and Destructors

Constructors and class initialization

```
class Base {
   int value;
public:
   Base(): value{0} {
      cout << "Base no-args constructor" << endl;
   }
   Base(int x) : value{x} {
      cout << "int Base constructor" << endl;
   }
};
```


Constructors and Destructors

Constructors and class initialization

```
class Derived : public Base {
   int doubled_value;
public:
   Derived(): Base{}, doubled_value{0} {
      cout << "Derived no-args constructor " << endl;
   }
   Derived(int x) : Base{x}, doubled_value {x*2} {
      cout << "int Derived constructor " << endl;
   }
};
```


Constructors and Destructors

Constructors and class initialization

```
                                Output
Base base;                      Base no-args constructor

Base base{100};                 int Base constructor

Derived derived;                Base no-args constructor
                                Derived no-args constructor

Derived derived{100};           int Base constructor
                                int Derived constructor
```


Inheritance

Copy/Move constructors and overloaded operator=

- Not inherited from the Base class
- You may not need to provide your own
  - Compiler-provided versions may be just fine
- We can explicitly invoke the Base class versions from the Derived class

Inheritance

Copy constructor

- Can invoke Base copy constructor explicitly
  - Derived object '*other*' will be **sliced**

```
Derived::Derived(const Derived &other)
    : Base(other), {Derived initialization list}
{
     // code
}
```

Constructors and Destructors

Copy Constructors

```
class Base {
   int value;
public:
   // Same constructors as previous example

   Base(const Base &other) :value{other.value} {
   cout << "Base copy constructor" << endl;
   }
};
```


Constructors and Destructors

Copy Constructors

```
class Derived : public Base {
   int doubled_value;
public:
   // Same constructors as previous example

   Derived(const Derived &other)
   : Base(other), doubled_value {other.doubled_value } {
         cout << "Derived copy constructor " << endl;
   }
};
```


Constructors and Destructors

operator=

```
class Base {
   int value;
public:
   // Same constructors as previous example
   Base &operator=(const Base &rhs) {
      if (this != &rhs) {
         value = rhs.value;   // assign
      }
      return *this;
   }
};
```


Constructors and Destructors

operator=

```
class Derived : public Base {
   int doubled_value;
public:
   // Same constructors as previous example
   Derived &operator=(const Derived &rhs) {
      if (this != &rhs) {
         Base::operator=(rhs);          // Assign Base part
         doubled_value = rhs.doubled_value; // Assign Derived part
      }
      return *this;
   }
};
```


Inheritance

Copy/Move constructors and overloaded operator=

- Often you do not need to provide your own
- If you **DO NOT** not define them in Derived
  - then the compiler will create them and automatically and call the base class's version
- If you **DO** provide Derived versions
  - then **YOU** must invoke the Base versions **explicitly** yourself
- Be careful with raw pointers
  - Especially if Base and Derived each have raw pointers
  - Provide them with deep copy semantics

Inheritance

Using and redefining Base class methods

- Derived class can directly invoke Base class methods
- Derived class can **override** or **redefine** Base class methods
- Very powerful in the context of polymorphism (next section)

Inheritance

Using and redefining Base class methods

```
class Account {
public:
   void deposit(double amount) { balance += amount; }
};

class Savings_Account: public Account {
public:
   void deposit(double amount) { // Redefine Base class method
      amount += some_interest;
      Account::deposit(amount);  // invoke call Base class method
   }
};
```


Inheritance

Static binding of method calls

- Binding of which method to use is done at compile time
  - Default binding for C++ is static
  - Derived class objects will use Derived::deposit
  - But, we can explicitly invoke Base::deposit from Derived::deposit
  - OK, but limited – much more powerful approach is dynamic binding which we will see in the next section

Inheritance

Static binding of method calls

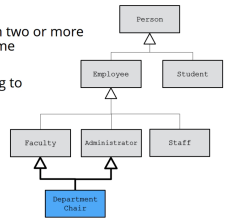
```
Base b;
b.deposit(1000.0);              // Base::deposit

Derived d;
d.deposit(1000.0);              // Derived::deposit

Base *ptr = new Derived();
ptr->deposit(1000.0);           // Base::deposit ????
```


Multiple Inheritance

- A derived class inherits from two or more Base classes at the same time
- The Base classes may belong to unrelated class hierarchies
- A Department Chair
  - Is-A Faculty and
  - Is-A Administrator



Multiple Inheritance

C++ Syntax

```
class Department_Chair:
      public Faculty, public Administrator {
   . . .
};
```

- Beyond the scope of this course
- Some compelling use-cases
- Easily misused
- Can be very complex

3 Polymorphism

What is Polymorphism?

Fundamental to Object-Oriented Programming

Polymorphism
- Compile-time / early binding / static binding
- Run-time / late binding / dynamic binding

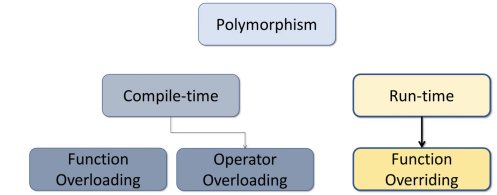
Runtime polymorphism
- Being able to assign different meanings to the same function at run-time

Allows us to program more abstractly
- Think general vs. specific
- Let C++ figure out which function to call at run-time

Not the default in C++, run-time polymorphism is achieved via
- Inheritance
- Base class pointers or references
- virtual functions

What is Polymorphism?

An non-polymorphic example – Static Binding



What is Polymorphism?

An non-polymorphic example – Static Binding

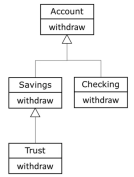
```
Account a;
a.withdraw(1000);   // Account::withdraw()

Savings b;
b.withdraw(1000);   // Savings::withdraw()

Checking c;
c.withdraw(1000);   // Checking::withdraw()

Trust d;
d.withdraw(1000);   // Trust::withdraw()

Account *p = new Trust();
P->withdraw(1000);  // Account::withdraw()
                    // should be
                    // Trust::withdraw()
```



What is Polymorphism?

An non-polymorphic example – Static Binding

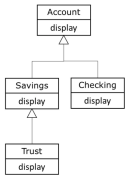
```
void display_account(const Account &acc)  {
   acc.display();
   // will always use Account::display
}

Account a;
display_account(a);

Savings b;
display_account(b);

Checking c;
display_account(c);

Trust d;
display_account(d);
```



What is Polymorphism?

A polymorphic example – Dynamic Binding

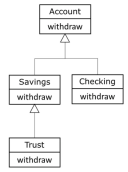
```
Account a;
a.withdraw(1000);   // Account::withdraw()

Savings b;
b.withdraw(1000);   // Savings::withdraw()

Checking c;
c.withdraw(1000);   // Checking::withdraw()

Trust d;
d.withdraw(1000);   // Trust::withdraw()

Account *p = new Trust();
P->withdraw(1000);  // Trust::withdraw()
```



`withdraw` method is virtual in Account

What is Polymorphism?

A polymorphic example – Dynamic Binding

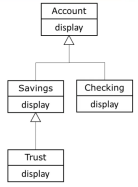
```
void display_account(const Account &acc)
{
   // will always the display method
   // depending on the object's type
   // at RUN-TIME!
}

Account a;
display_account(a);

Savings b;
display_account(b);

Checking c;
display_account(c);

Trust d;
display_account(d);
```



`display` method is virtual in Account

Polymorphism

Using a Base class pointer

For dynamic polymorphism we must have:

- Inheritance
- Base class pointer or Base class reference
- virtual functions

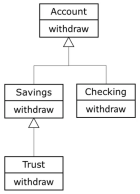
Polymorphism

Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

p1->withdraw(1000);       //Account::withdraw
p2->withdraw(1000);       //Savings::withdraw
p3->withdraw(1000);       //Checking::withdraw
p4->withdraw(1000);       //Trust::withdraw

// delete the pointers
```



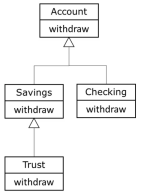
Polymorphism

Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

Account *array [] = {p1,p2,p3,p4};

for (auto i=0; i<4; ++i)
   array[i]->withdraw(1000);
```



Polymorphism

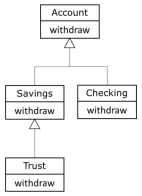
Using a Base class pointer

```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

vector<Account *> accounts
      {p1, p2, p3, p4 };

for (auto acc_ptr: accounts)
   acc_ptr->withdraw();

// delete the pointers
```



Polymorphism

Virtual functions
- Redefined functions are bound statically
- Overridden functions are bound dynamically
- Virtual functions are overridden
- Allow us to treat all objects generally as objects of the Base class

Polymorphism

Declaring virtual functions
- Declare the function you want to override as virtual in the Base class
- Virtual functions are virtual all the way down the hierarchy from this point
- Dynamic polymorphism only via Account class pointer or reference

```
class Account {
public:
    virtual void withdraw(double amount);
    . . .
};
```

Polymorphism

Declaring virtual functions

- Override the function in the Derived classes
- Function signature and return type must match EXACTLY
- Virtual keyword not required but is best practice
- If you don't provide an overridden version it is inherited from it's base class

```
class Checking : public Account {
public:
    virtual void withdraw(double amount);
    . . .
};
```


Polymorphism

Virtual Destructors

- Problems can happen when we destroy polymorphic objects
- If a derived class is destroyed by deleting its storage via the base class pointer and the class a non-virtual destructor. Then the behavior is undefined in the C++ standard.
- Derived objects must be destroyed in the correct order starting at the correct destructor

Polymorphism

Virtual Destructors

- Solution/Rule:
  - If a class has virtual functions
  - ALWAYS provide a public virtual destructor
  - If base class destructor is virtual then all derived class destructors are also virtual

```
class Account {
public:
    virtual void withdraw(double amount);
    virtual ~Account();
    . . .
};
```


Polymorphism

The override specifier

- We can override Base class virtual functions
- The function signature and return must be EXACTLY the same
- If they are different then we have redefinition NOT overriding
- Redefinition is statically bound
- Overriding is dynamically bound
- C++11 provides an override specifier to have the compiler ensure overriding

Polymorphism

The override specifier

```
class Base {
public:
    virtual void say_hello() const {
        std::cout << "Hello - I'm a Base class object" << std::endl;
    }
    virtual ~Base() {}
};

class Derived: public Base {
public:
    virtual void say_hello()   {   // Notice I forgot the const - NOT OVERRIDING
        std::cout << "Hello - I'm a Derived class object" << std::endl;
    }
    virtual ~Derived() {}
};
```


Polymorphism

The override specifier

```
Base:

    virtual void say_hello() const;


Derived:

    virtual void say_hello();
```


Polymorphism

The override specifier

```
Base *p1 = new Base();
p1->say_hello();          // "Hello - I'm a Base class object"

Base *p2 = new Derived();
p2->say_hello();          // "Hello - I'm a Base class object"
```

- Not what we expected
- say_hello method signatures are different
- So Derived **redefines** say_hello instead of overriding it!

Polymorphism

The override specifier

```
class Base {
public:
    virtual void say_hello() const {
        std::cout << "Hello - I'm a Base class object" << std::endl;
    }
    virtual ~Base() {}
};

class Derived: public Base {
public:
    virtual void say_hello() override  {   // Produces compiler error
                                           // Error: marked override but does not override
        std::cout << "Hello - I'm a Derived class object" << std::endl;
    }
    virtual ~Derived() {}
};
```


Polymorphism

The `final` specifier

- C++11 provides the final specifier
  - When used at the class level
  - Prevents a class from being derived from
  - When used at the method level
  - Prevents virtual method from being overridden in derived classes

Polymorphism

`final` class

```
class My_class final {
    . . .
};

class Derived final: public Base {
    . . .
};
```


Polymorphism

`final` method

```
class A {
public:
    virtual void do_something();
};

class B: public A {
    virtual void do_something() final;    // prevent futher overriding
};

class C: public B {
    virtual void do_something();          // COMPILER ERROR - Can't override
};
```


Polymorphism

Using Base class references

- We can also use Base class references with dynamic polymorphism
- Useful when we pass objects to functions that expect a Base class reference

Polymorphism

Using Base class references

```
Account a;
Account &ref = a;
ref.withdraw(1000); // Account::withdraw

Trust t;
Account &ref1 = t;
ref1.withdraw(1000);// Trust::withdraw
```


Polymorphism

Using Base class references

```
void do_withdraw(Account &account, double amount) {
   account.withdraw(amount);
}

Account a;
do_withdraw(a, 1000);  // Account::withdraw

Trust t;
do_withdraw(t, 1000);  // Trust::withdraw
```


Polymorphism

Pure virtual functions and abstract classes

- Abstract class
  - Cannot instantiate objects
  - These classes are used as base classes in inheritance hierarchies
  - Often referred to as Abstract Base Classes
- Concrete class
  - Used to instantiate objects from
  - All their member functions are defined
    - `Checking Account, Savings Account`
    - `Faculty, Staff`
    - `Enemy, Level Boss`

Polymorphism

Pure virtual functions and abstract classes

- Abstract base class
  - Too generic to create objects from
    - `Shape, Employee, Account, Player`
  - Serves as parent to Derived classes that may have objects
  - Contains at least one pure virtual function

Polymorphism

Pure virtual functions and abstract classes

- Pure virtual function
  - Used to make a class abstract
  - Specified with "=0" in its declaration

```
virtual void function() = 0; // pure virtual function
```

  - Typically do not provide implementations

Polymorphism

Pure virtual functions and abstract classes

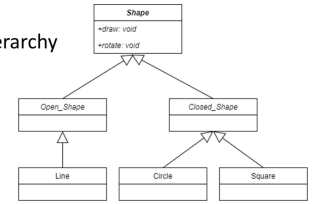
- Pure virtual function
  - Derived classes MUST override the base class
  - If the Derived class does not override then the Derived class is also abstract
  - Used when it doesn't make sense for a base class to have an implementation
    - But concrete classes must implement it

```
virtual void draw() = 0;      // Shape
virtual void defend() = 0;          // Player
```


Polymorphism

Pure virtual functions and abstract classes

Shape Class Hierarchy



Polymorphism

Pure virtual functions and abstract classes

```
class Shape {                  // Abstract
private:
      // attributes common to all shapes
public:
      virtual void draw() = 0;       // pure virtual function
      virtual void rotate() = 0;     // pure virtual function
      virtual ~Shape();
      . . .
};
```


Polymorphism

Pure virtual functions and abstract classes

```
class Circle: public Shape {               // Concrete
private:
      // attributes for a circle
public:
      virtual void draw() override {
          // code to draw a circle
      }
      virtual void rotate() override {
          // code to rotate a circle
      }
      virtual ~Circle();
      . . .
};
```


Polymorphism

Pure virtual functions and abstract classes

Abstract Base class

- Cannot be instantiated

```
Shape shape;              // Error
Shape *ptr = new Shape();  // Error
```

- We can use pointers and references to dynamically refer to concrete classes derived from them

```
Shape *ptr = new Circle();
ptr->draw();
ptr->rotate();
```


Polymorphism

What is using a class as an interface?

- An abstract class that has only pure virtual functions
- These functions provide a general set of services to the user of the class
- Provided as public
- Each subclass is free to implement these services as needed
- Every service (method) must be implemented
- The service type information is strictly enforced

Polymorphism

A Printable example

- C++ does not provide true interfaces
- We use abstract classes and pure virtual functions to achieve it
- Suppose we want to be able to provide Printable support for any object we wish without knowing it's implementation at compile time

```
std::cout << any_object << std::endl;
```

  - `any_object` must conform to the `Printable` interface

Polymorphism

An Printable example

```
class Printable {
   friend ostream &operator<<(ostream &, const Printable &obj);
public:
   virtual void print(ostream &os) const = 0;
   virtual ~Printable() {};
   . . .
};

ostream &operator<<(ostream &os, const Printable &obj) {
   obj.print(os);
   return os;
}
```


Polymorphism

An Printable example

```
class Any_Class : public Printable {
public:
   // must override Printable::print()
   virtual void print(ostream *os) override {
      os << "Hi from Any_Class" ;
   }
   . . .
};
```


Polymorphism

An Printable example

```
Any_Class *ptr= new Any_Class();
cout << *ptr << endl;

void function1 (Any_Class &obj) {
   cout << obj << endl;
}

void function2 (Printable &obj) {
   cout << obj << endl;
}
function1(*ptr);          // "Hi from Any_Class"
function2(*ptr);          // "Hi from Any_Class"
```


Polymorphism

A Shapes example

```
class Shape {
public:
   virtual void draw() = 0;
   virtual void rotate() = 0;
   virtual ~Shape() {};
   . . .
};
```


Polymorphism

A Shapes example

```
class Circle : public Shape {
public:
   virtual void draw() override { /* code */ };
   virtual void rotate() override { /* code */ };
   virtual ~Circle() {};
   . . .
};
```


Polymorphism

A Shapes example

```
class I_Shape {
public:
   virtual void draw() = 0;
   virtual void rotate() = 0;
   virtual ~I_Shape() {};
   . . .
};
```


Polymorphism

A Shapes example

```
class Circle : public I_Shape {
public:
   virtual void draw() override   { /* code */ };
   virtual void rotate() override { /* code */ };
   virtual ~Circle() {};
   . . .
};
```

- Line and Square classes would be similar

Polymorphism

A Shapes example

```
vector< I_Shape *> shapes;

I_Shape *p1 = new Circle();
I_Shape *p2 = new Line();
I_Shape *p3 = new Square();

for (auto const &shape: shapes) {
   shape->rotate();
   shape->draw();
}
// delete the pointers
```

4 Smart Pointers

Smart Pointers

Issues with Raw Pointers

- C++ provides absolute flexibility with memory management
  - Allocation
  - Deallocation
  - Lifetime management
- Some potentially serious problems
  - Uninitialized (wild) pointers
  - Memory leaks
  - Dangling pointers
  - Not exception safe
- Ownership?
  - Who owns the pointer?
  - When should a pointer be deleted?

Smart Pointers

What are they?

- Objects
- Can only point to heap-allocated memory
- Automatically call delete when no longer needed
- Adhere to RAII principles
- C++ Smart Pointers
  - Unique pointers (`unique_ptr`)
  - Shared pointers (`shared_ptr`)
  - Weak pointers (`weak_ptr`)
  - Auto pointers (`auto_ptr`) **Deprecated** – we will not discuss

Smart Pointers

What are they?

- `#include <memory>`
- Defined by class templates
  - Wrapper around a raw pointer
  - Overloaded operators
    - Dereference (`*`)
    - Member selection (`->`)
    - Pointer arithmetic not supported (`++`, `--`, etc.)
  - Can have custom deleters

Smart Pointers

A simple example

```
{
   std::smart_pointer<Some_Class> ptr = . . .

   ptr->method();
   cout << (*ptr) << endl;
}

// ptr will be destroyed automatically when
// no longer needed
```


Smart Pointers

RAII – Resource Acquisition Is Initialization

- Common idiom or pattern used in software design based on container object lifetime
- RAII objects are allocated on the stack
- Resource Acquisition
  - Open a file
  - Allocate memory
  - Acquire a lock
- Is Initialization
  - The resource is acquired in a constructor
- Resource relinquishing
  - Happens in the destructor
    - Close the file
    - Deallocate the memory
    - Release the lock

Smart Pointers

`unique_ptr`

- Simple smart pointer – very efficient!
- `unique_ptr<T>`
  - Points to an object of type T on the heap
  - It is unique – there can only be one `unique_ptr<T>` pointing to the object on the heap
  - Owns what it points to
  - Cannot be assigned or copied
  - CAN be moved
  - When the pointer is destroyed, what it points to is automatically destroyed

Smart Pointers

`unique_ptr` – creating, initializing and using

```
{
   std::unique_ptr<int> p1 {new int {100} };

   std::cout << *p1 << std::endl;     // 100

   *p1 = 200;

   std::cout << *p1 << std::endl;     // 200

}// automatically deleted
```


Smart Pointers

`unique_ptr` – some other useful methods

```
{
   std::unique_ptr<int> p1 {new int {100} };

   std::cout << p1.get() << std::endl;  // 0x564388

   p1.reset(); // p1 is now nullptr

   if (p1)
     std::cout << *p1 << std::endl;     // won't execute
}// automatically deleted
```


Smart Pointers

`unique_ptr` – user defined classes

```
{
   std::unique_ptr<Account> p1 {new Account{"Larry"}};
   std::cout << *p1 << std::endl;  // display account

   p1->deposit(1000);
   p1->withdraw(500);


}// automatically deleted
```


Smart Pointers

`unique_ptr` – vectors and move

```
{
   std::vector<std::unique_ptr<int>> vec;

   std::unique_ptr<int> ptr {new int{100}};

   vec.push_back(ptr);   // Error – copy not allowed

   vec.push_back(std::move(ptr));

}// automatically deleted
```


Smart Pointers

`unique_ptr` – make_unique (C++14)

```
{
   std::unique_ptr<int> p1 = make_unique<int>(100);

   std::unique_ptr<Account> p2  = make_unique<Account>("Curly", 5000);

   auto p3 = make_unique<Player>("Hero", 100, 100);

}// automatically deleted
```

More efficient – no calls to `new` or `delete`

Smart Pointers

`shared_ptr`

- Provides shared ownership of heap objects
- `shared_ptr<T>`
  - Points to an object of type T on the heap
  - It is not unique – there can many shared_ptrs pointing to the same object on the heap
  - Establishes shared ownership relationship
  - CAN be assigned and copied
  - CAN be moved
  - Doesn't support managing arrays by default
  - When the use count is zero, the managed object on the heap is destroyed

Smart Pointers

`shared_ptr` – creating, initializing and using

```
{
   std::shared_ptr<int> p1 {new int {100} };

   std::cout << *p1 << std::endl;     // 100

   *p1 = 200;

   std::cout << *p1 << std::endl;     // 200

}// automatically deleted
```


Smart Pointers

`shared_ptr` – some other useful methods

```
{
   // use_count - the number of shared_ptr objects managing the heap object
   std::shared_ptr<int> p1 {new int {100} };
   std::cout << p1.use_count () << std::endl;         // 1

   std::shared_ptr<int> p2 { p1 };                    // shared ownership
   std::cout << p1.use_count () << std::endl;         // 2

   p1.reset();      // decrement the use_count; p1 is nulled out
   std::cout << p1.use_count() << std::endl;          // 0
   std::cout << p2.use_count() << std::endl;          // 1
}  // automatically deleted
```


Smart Pointers

`shared_ptr` – vectors and move

```
{
   std::vector<std::shared_ptr<int>> vec;

   std::shared_ptr<int> ptr {new int{100}};

   vec.push_back(ptr);   // OK - copy IS allowed

   std:: cout << ptr.use_count() << std::endl;     // 2
}// automatically deleted
```


Smart Pointers

`shared_ptr` – make_shared (C++11)

```
{
   std::shared_ptr<int> p1 = make_shared<int>(100); // use_count : 1
   std::shared_ptr<int> p2 { p1 };                  // use_count : 2
   std::shared_ptr<int> p3;
   p3 = p1;                                         // use_count : 3

}// automatically deleted
```

- Use make_shared – it's more efficient!
- All 3 pointers point to the SAME object on the heap!
- When the use_count becomes 0 the heap object is deallocated

Smart Pointers

`weak_ptr`

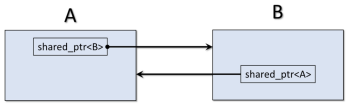
- Provides a non-owning "weak" reference

- `weak_ptr<T>`
  - Points to an object of type T on the heap
  - Does not participate in owning relationship
  - Always created from a `shared_ptr`
  - Does NOT increment or decrement reference use count
  - Used to prevent strong reference cycles which could prevent objects from being deleted

Smart Pointers

`weak_ptr` – circular or cyclic reference

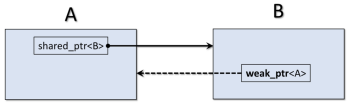
- A refers to B
- B refers to A
- Shared strong ownership prevents heap deallocation



Smart Pointers

`weak_ptr` – circular or cyclic reference

- Solution – make one of the pointers non-owning or 'weak'
- Now heap storage is deallocated properly



Smart Pointers

Custom deleters

- Sometimes when we destroy a smart pointer we need more than to just destroy the object on the heap
- These are special use-cases
- C++ smart pointers allow you to provide custom deleters
- Lots of way to achieve this
  - Functions
  - Lambdas
  - Others. . .

Smart Pointers

Custom deleters - function

```
void my_deleter(Some_Class *raw_pointer) {
   // your custom deleter code
   delete raw_pointer;
}

shared_ptr<Some_Class> ptr { new Some_class{}, my_deleter };
```


Smart Pointers

Custom deleters - lambda

```
shared_ptr<Test> ptr (new Test{100}, [] (Test *ptr)  {
      cout << "\tUsing my custom deleter" << endl;
      delete ptr;
});
```