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;
}
```

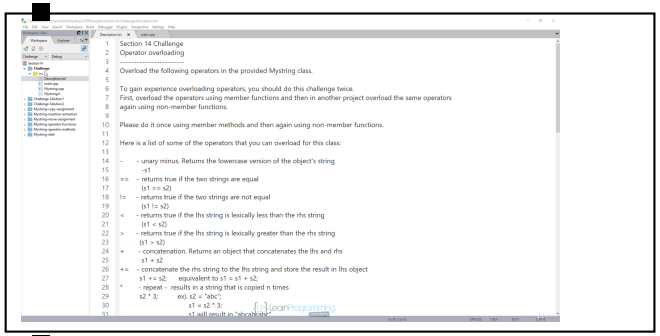
- Return a reference to the `ostream` so we can keep inserting
- Don't return `ostream` by value!

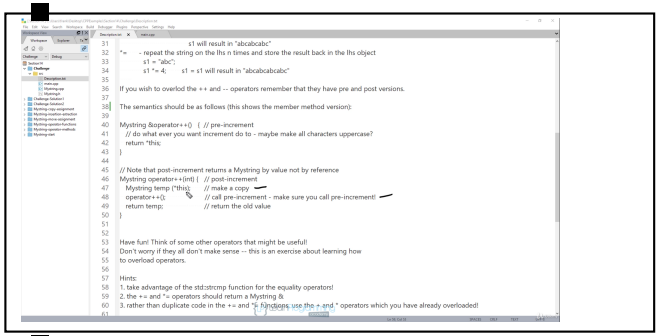
Operator Overloading

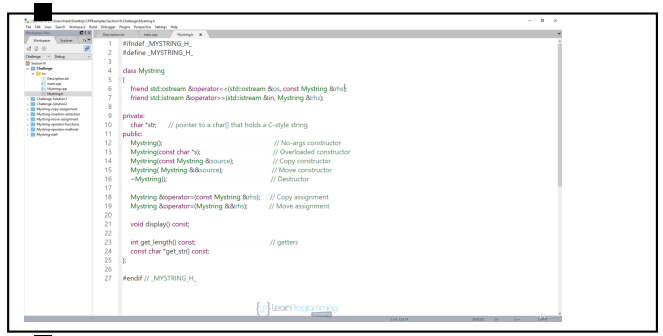
stream extraction operator (>>)

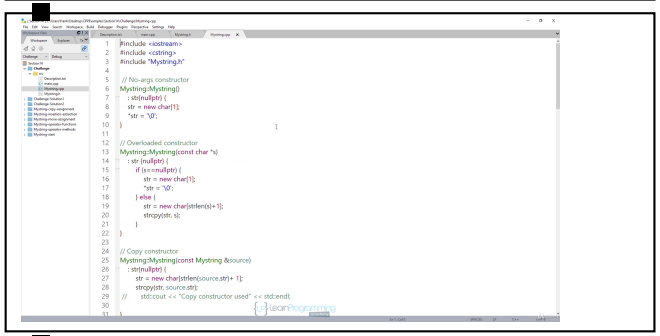
```
std::istream &operator>>(std::istream &is, Mystring &obj) {
    char *buff = new char[1000];
    is >> buff;
    obj = Mystring{buff};// If you have copy or move assignment
    delete [] buff;
    return is;
}
```

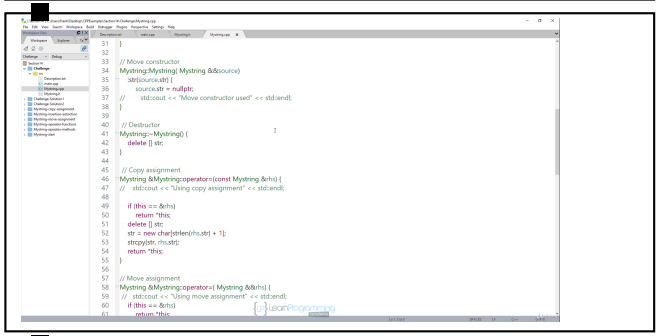
- Return a reference to the `istream` so we can keep inserting
- Update the object passed in

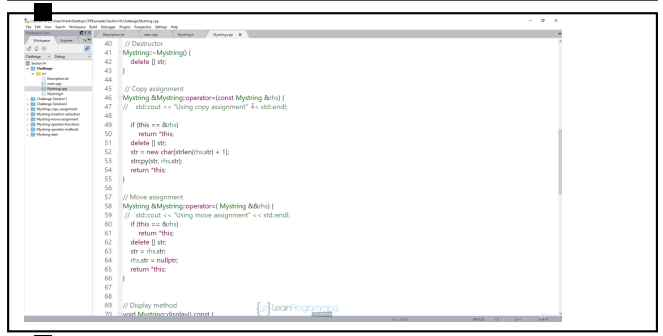


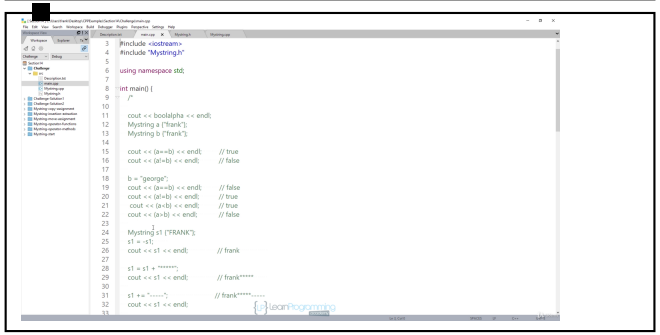


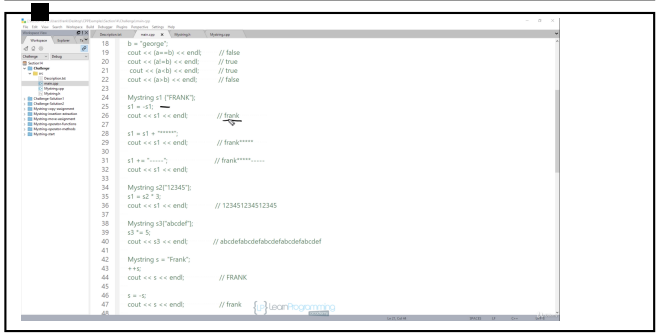












Inheritance

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

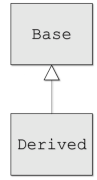
Terminology

- 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

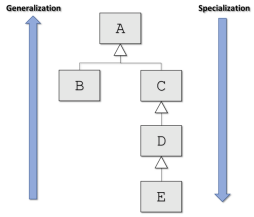
- "Is-A" relationship
  - Public inheritance
  - Derived classes are sub-types of their Base classes
  - Can use a derived class object wherever we use a base class object
- Generalization
  - Combining similar classes into a single, more general class based on common attributes
- Specialization
  - 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

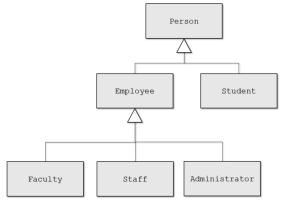


Inheritance

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



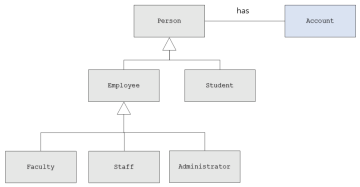
Inheritance

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

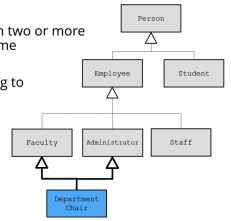
```
Base b;
b.deposit(1000.0);              // Base::deposit

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