

## 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

BEGINNING C++ PROGRAMMING  
Operator Overloading

LearnProgramming  
YouTube

## 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 operators can be overloaded?

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

operator
::
:?
.*
.
sizeof

BEGINNING C++ PROGRAMMING  
Operator Overloading

LearnProgramming  
YouTube

## 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

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;  
};
```

BEGINNING C++ PROGRAMMING  
Operator Overloading

LearnProgramming  
YouTube

## 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)

```
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;  
}
```

BEGINNING C++ PROGRAMMING  
Copy Assignment Operator

LearnProgramming  
YouTube

## 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;
```

BEGINNING C++ PROGRAMMING  
Copy Assignment Operator

LearnProgramming  
YouTube

## 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);
```

BEGINNING C++ PROGRAMMING  
Operator Overloading

LearnProgramming  
YouTube

## 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  
s1 == s2  
std::cout << s1
- **MyString**  
s1 = s2 + s3  
s1 < s2  
s1 == s2  
std::cout << s1
- **Player**  
p1 < p2  
p1 == p2  
std::cout << p1

BEGINNING C++ PROGRAMMING  
Operator Overloading

LearnProgramming  
YouTube

## 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
```

BEGINNING C++ PROGRAMMING  
Copy Assignment Operator

LearnProgramming  
YouTube

## 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];
```

BEGINNING C++ PROGRAMMING  
Copy Assignment Operator

LearnProgramming  
YouTube

## 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;
```

BEGINNING C++ PROGRAMMING

Copy Assignment Operator



by Udemy

## 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;
```

BEGINNING C++ PROGRAMMING

Move Assignment Operator



by Udemy

## Operator Overloading

Mystring operator- make lowercase

```
Mystring larry1 ("LARRY");
Mystring larry2;

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

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

BEGINNING C++ PROGRAMMING

Operator as Member Function



by Udemy

## 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
```

BEGINNING C++ PROGRAMMING

Operator as Member Function



by Udemy

## 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

BEGINNING C++ PROGRAMMING

Move Assignment Operator



by Udemy

## 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;
```

BEGINNING C++ PROGRAMMING

Move Assignment Operator



by Udemy

## 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
```

BEGINNING C++ PROGRAMMING

Move Assignment Operator



by Udemy

## Operator Overloading

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

```
ReturnType Type::operatorOp();
```

```
Number Number::operator-() const;
Number Number::operator++();
Number Number::operator++(int); // 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)
```

BEGINNING C++ PROGRAMMING

Operator as Member Function



by Udemy

## 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)
```

BEGINNING C++ PROGRAMMING

Operator as Member Function



by Udemy

## 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
```

BEGINNING C++ PROGRAMMING

Operator as Member Function



by Udemy

```
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;
}
```

BEGINNING C++ PROGRAMMING

Operator as Member Function



by Udemy

## 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)
```

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## Operator Overloading

Mystring operator- make lowercase

```
Mystring larry1 ("LARRY");  
Mystring larry2;  
  
larry1.display(); // LARRY  
  
larry2 = -larry1; // operator-(larry1)  
  
larry1.display(); // LARRY  
larry2.display(); // larry
```

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## 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;  
}
```

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## 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)
```

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## 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

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## 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
```

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## 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;  
}
```

BEGINNING C++ PROGRAMMING  
Operator as Global Function

LearnProgramming

## 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)
```

BEGINNING C++ PROGRAMMING  
Overloading Insertion and Extraction

LearnProgramming

## Operator Overloading

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

```
Mystring larry;  
  
cin >> larry;  
  
Player hero;  
  
cin >> hero;
```

BEGINNING C++ PROGRAMMING  
Overloading Insertion and Extraction

LearnProgramming

## Operator Overloading

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

- Doesn't make sense to implement as member 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?
```

BEGINNING C++ PROGRAMMING  
Overloading Insertion and Extraction

LearnProgramming

## 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!

BEGINNING C++ PROGRAMMING  
Overloading Insertion and Extraction

LearnProgramming

## 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;  
}
```

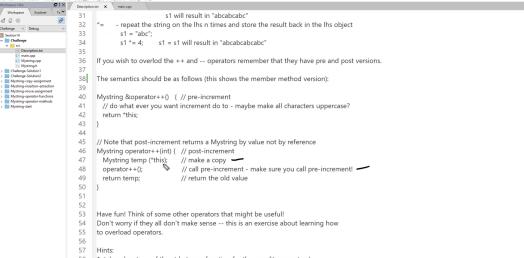
- Return a reference to the istream so we can keep inserting
- Update the object passed in

BEGINNING C++ PROGRAMMING  
Overloading Insertion and Extraction

LearnProgramming

The screenshot shows the Java IDE interface with the following details:

- Title Bar:** MyStringChallenge
- Left Sidebar:** Navigator, Search, Help, and a context menu.
- Central Area:**
  - Section 14 Challenge
  - Operator overloading
  - Overload the following operators in the provided MyString class.
  - 
  - 
  - To gain experience overloading operators, you should do this challenge twice.
  - First, overload the operators using member functions and then in another project overload the same operators again using non-member functions.
  - Please do it once using member methods and then again using non-member functions.
  - 
  - Here is a list of some of the operators that you can overload for this class:
  - `<unary minus`. Returns the lowercase version of the object's string
  - `=` - returns true if the two strings are equal
  - `==` - returns true if the two strings are not equal
  - `!=` - returns true if the two strings are not equal
  - `[1+1 < 2]` - returns true if the lhs string is lexically less than the rhs string
  - `(1 < 2) >` - returns true if the lhs string is lexically greater than the rhs string
  - `[1+1 < 2] + [3+3 < 4]` - concatenation. Returns an object that concatenates the lhs and rhs
  - `s1 + s2`
  - `[s1 = s2]` - concatenate the rhs string to the lhs string and store the result in the object
  - `s1 + s2 =` - equivalent to `s1 + s2 + "`
  - `* repeat` - results in a string that is copied n times
  - `"2";` - the character '2'
  - `s1 + s2 + "` - `s1 will result as "abcde"`
- Bottom Status Bar:** MyStringChallenge



```
 31     s1 += "c"; // s1 will result in "abcabc"
 32     // repeat the string on the lns n times and store the result back in the lns object
 33     s1 *= 3; // s1 = "abc";
 34     s1 *= 4; // s1 will result in "abcababc"
 35
 36 // If you wish to overload the + and -- operators remember that they have pre and post versions.
 37
 38 // The methods should be as follows (this shows the member method version):
 39
 40 MyString Resistor() { // pre-increment
 41     // do what ever you want increment do to - maybe make all characters uppercase?
 42     return "This";
 43 }
 44
 45 // Note that post-increment returns a MyString by value not by reference
 46 MyString operator++(int) { // post-increment
 47     MyString str = ToString();
 48     operator++(0); // call pre-increment - make sure you call pre-increment
 49     return temp; // return the old value
 50 }
 51
 52
 53 // Now fun! Think of some other operators that might be useful!
 54 // Don't worry if they all don't make sense - this is an exercise about learning how
 55 // to overload operators
 56
 57 Hints:
 58 // take advantage of the returning function for the equality operators!
 59 // the ++ and -- operators should return a MyString &
 60 // rather than duplicate code in the ++ and -- functions use the + and - operators which you have already overloaded!
```

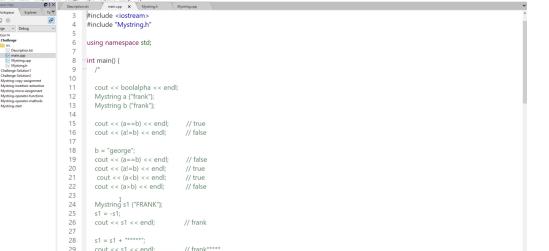


```
1 //ifndef _MYSTRING_H_
2 #define _MYSTRING_H_
3
4 class MyString {
5 public:
6     friend std::ostream &operator<<(std::ostream &os, const MyString &rhs);
7     friend std::istream &operator>>(std::istream &in, MyString &rhs);
8
9     private:
10     char *str; // pointer to a char[] that holds a C-style string
11
12     MyString(); // No-param constructor
13     MyString(const char *s); // Overloaded constructor
14     MyString(const MyString &source); // Copy constructor
15     MyString( MyString &source); // Move constructor
16     ~MyString(); // Destructor
17
18     MyString &operator=(const MyString &rhs); // Copy assignment
19     MyString &operator=(MyString &rhs); // Move assignment
20
21     void display() const;
22
23     int getLength() const; // getters
24     const char *getStr() const;
25 };
26
27 #endif // _MYSTRING_H_
```



```
1 //include <iostream>
2 //include <string>
3 //include "MyString.h"
4
5 //No-args constructor
6 MyString::MyString()
7 {
8     str[0] = '\0';
9     str[1] = '\0';
10 }
11
12 //Overloaded constructor
13 MyString::MyString(const char *s)
14 {
15     str = input(s);
16     if(str == '\0')
17     {
18         str = new char[1];
19         *str = '\0';
20     }
21     else
22     {
23         str = new char[strlen(s)+1];
24         strcpy(str, s);
25     }
26 }
27
28 //Copy constructor
29 MyString::MyString(const MyString &source)
30 {
31     str = source.str;
32     for(int i=0; str[i] != '\0'; i++)
33     {
34         str[i] = source.str[i];
35     }
36 }
37
38 //Copy assignment operator
39 MyString &MyString::operator=(const MyString &source)
40 {
41     if(this == &source)
42     {
43         return *this;
44     }
45     else
46     {
47         delete[] str;
48         str = source.str;
49     }
50     return *this;
51 }
```

```
40 // Destructor
41 MyString::~MyString() {
42     delete [] m;
43 }
44
45 // Copy assignment
46 MyString &MyString::operator=(const MyString &rhs) {
47     std::cout << "Using copy assignment" << std::endl;
48     if (this == &rhs)
49         return *this;
50     delete [] m;
51     m = new char[strlen(rhs.m) + 1];
52     strcpy(m, rhs.m);
53     return *this;
54 }
55
56
57 // Move assignment
58 MyString &MyString::operator=(MyString &&rhs) {
59     std::cout << "Using move assignment" << std::endl;
60     if (this == &rhs)
61         return *this;
62     delete [] m;
63     m = rhs.m;
64     rhs.m = nullptr;
65     return *this;
66 }
67
68
69 // Display method
70 void MyString::display() const {
```



The screenshot shows a Microsoft Visual Studio Code window with the following code:

```
1 #include <iostream>
2 #include "MyString.h"
3
4 using namespace std;
5
6 int main() {
7     cout << endl;
8
9     cout << "bodilakka" << endl;
10    MyString a ("Frank");
11    MyString b ("Frank");
12
13    cout << (a+b) << endl; // true
14    cout << (a+b) << endl; // false
15
16    b = "george";
17    cout << (a+b) << endl; // false
18    cout << (a+b) << endl; // true
19    cout << (a+b) << endl; // true
20    cout << (a+b) << endl; // false
21
22    cout << (a+b) << endl; // false
23
24    MyString s1 ("FRANK");
25    s1 += s1;
26    cout << s1 << endl; // frank
27
28    s1 += s1 + "*****";
29    cout << s1 << endl; // frank*****+
30
31    s1 += "*****";
32    cout << s1 << endl; // frank*****+
```

### 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.

---

BEGINNING C++ PROGRAMMING

{UP} LearnProgramming

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

---

BEGINNING C++ PROGRAMMING

{LP} LearnProgramming

## Inheritance

## Accounts – without inheritance – code duplication

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

---

63

63

---

BEGINNING C++ PROGRAMMING

{up} Learn Programming

## 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, . . .  
};
```

BEGINNING C++ PROGRAMMING

What is Inheritance?

LearnProgramming

11 min

## 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

BEGINNING C++ PROGRAMMING

Terminology and Notation

LearnProgramming

11 min

## 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

BEGINNING C++ PROGRAMMING

Terminology and Notation

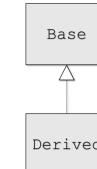
LearnProgramming

11 min

## 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



BEGINNING C++ PROGRAMMING

Terminology and Notation

LearnProgramming

11 min

## 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

BEGINNING C++ PROGRAMMING

Terminology and Notation

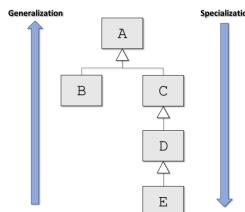
LearnProgramming

11 min

## 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



BEGINNING C++ PROGRAMMING

Terminology and Notation

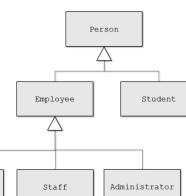
LearnProgramming

11 min

## 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



BEGINNING C++ PROGRAMMING

Terminology and Notation

LearnProgramming

11 min

## 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

BEGINNING C++ PROGRAMMING

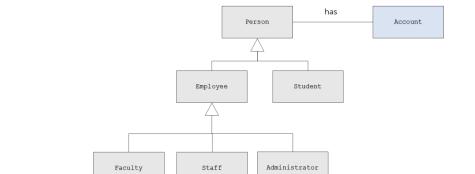
Inheritance vs Composition

LearnProgramming

11 min

## Inheritance

Public Inheritance vs. Composition



BEGINNING C++ PROGRAMMING

Inheritance vs Composition

LearnProgramming

11 min

## Inheritance

Composition

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

BEGINNING C++ PROGRAMMING

Inheritance vs Composition

LearnProgramming

11 min

## 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
```

BEGINNING C++ PROGRAMMING

Deriving Classes from Existing Classes

LearnProgramming

11 min

## 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 "is-a"
  - Different from composition

BEGINNING C++ PROGRAMMING

Deriving Classes from Existing Classes

LearnProgramming

11 min

## 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
```

BEGINNING C++ PROGRAMMING

Deriving Classes from Existing Classes

LearnProgramming

11 min

## Deriving classes from existing classes

C++ creating objects

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

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

delete p_account;
```

BEGINNING C++ PROGRAMMING  
Deriving Classes from Existing Classes

LearnProgramming

1 lesson

## 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 . . .
};
```

BEGINNING C++ PROGRAMMING  
Protected Members and Class Access

LearnProgramming

1 lesson

## Deriving classes from existing 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;
```

BEGINNING C++ PROGRAMMING  
Deriving Classes from Existing Classes

LearnProgramming

1 lesson

## 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
```

BEGINNING C++ PROGRAMMING  
Protected Members and Class Access

LearnProgramming

1 lesson

## 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

BEGINNING C++ PROGRAMMING  
Constructors and Destructors

LearnProgramming

1 lesson

## 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; }
```

BEGINNING C++ PROGRAMMING  
Constructors and Destructors

LearnProgramming

1 lesson

## Constructors and Destructors

Constructors and class initialization

Output  
  
**Base** base; Base constructor  
  
**Derived** derived; Base constructor  
Derived constructor

BEGINNING C++ PROGRAMMING  
Constructors and Destructors

LearnProgramming

1 lesson

## 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 its own constructors

BEGINNING C++ PROGRAMMING  
Constructors and Destructors

LearnProgramming

1 lesson

## 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; }
```

BEGINNING C++ PROGRAMMING  
Constructors and Destructors

LearnProgramming

1 lesson

## Constructors and Destructors

Constructors and class initialization

Output  
  
**Base** base; Base constructor  
Base destructor  
  
**Derived** derived; Base constructor  
Derived constructor  
Derived destructor  
Base destructor

BEGINNING C++ PROGRAMMING  
Constructors and Destructors

LearnProgramming

1 lesson

## 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

BEGINNING C++ PROGRAMMING Constructors and Destructors



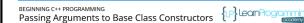
11 mins

## 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

BEGINNING C++ PROGRAMMING Passing Arguments to Base Class Constructors



11 mins

## Inheritance

### Passing arguments to base class constructors

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

BEGINNING C++ PROGRAMMING Passing Arguments to Base Class Constructors



11 mins

## 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;  
    }  
};
```

BEGINNING C++ PROGRAMMING Passing Arguments to Base Class Constructors



11 mins

## 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;  
    }  
};
```

BEGINNING C++ PROGRAMMING Passing Arguments to Base Class Constructors



11 mins

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

BEGINNING C++ PROGRAMMING Passing Arguments to Base Class Constructors



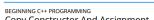
11 mins

## 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

BEGINNING C++ PROGRAMMING Copy Constructor And Assignment



11 mins

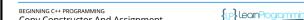
## Inheritance

### 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  
}
```

BEGINNING C++ PROGRAMMING Copy Constructor And Assignment



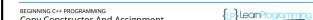
11 mins

## 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;  
    }  
};
```

BEGINNING C++ PROGRAMMING Copy Constructor And Assignment



11 mins

## 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;  
    }  
};
```

BEGINNING C++ PROGRAMMING Copy Constructor And Assignment



11 mins

## 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;  
    }  
};
```

BEGINNING C++ PROGRAMMING Copy Constructor And Assignment



11 mins

## 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;  
    }  
};
```

BEGINNING C++ PROGRAMMING Copy Constructor And Assignment



11 mins

## Inheritance

- Copy/Move constructors and overloaded operator=
- Often you do not need to provide your own
- If you DO NOT define them in Derived
  - then the compiler will create them and automatically 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

BEGINNING C++ PROGRAMMING  
Copy Constructor And Assignment



13 mins

## 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

BEGINNING C++ PROGRAMMING  
Using And Refining Base Methods



13 mins

## 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)

BEGINNING C++ PROGRAMMING  
Using And Refining Base Methods



13 mins

## 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  
    }  
};
```

BEGINNING C++ PROGRAMMING  
Using And Refining Base Methods



13 mins

## 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

BEGINNING C++ PROGRAMMING  
Using And Refining Base Methods



13 mins

## 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 ????
```

BEGINNING C++ PROGRAMMING  
Using And Refining Base Methods



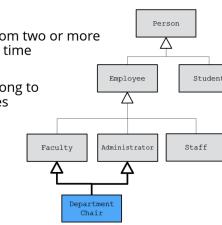
13 mins

## 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



BEGINNING C++ PROGRAMMING  
Multiple Inheritance



13 mins

## 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

BEGINNING C++ PROGRAMMING  
Multiple Inheritance



13 mins