

14. 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

BEGINNING C++ PROGRAMMING
Operator Overloading



by Simey

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



by Simey

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



by Simey

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



by Simey

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));
```

BEGINNING C++ PROGRAMMING
Operator Overloading



by Simey

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

BEGINNING C++ PROGRAMMING
Operator Overloading



by Simey

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



by Simey

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
- Mystring
s1 = s2 + s3
s1 < s2
std::cout << s1
- Player
p1 < p2
p1 == p2
std::cout << p1

BEGINNING C++ PROGRAMMING
Operator Overloading



by Simey

Operator Overloading

Copy assignment operator (=)

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

BEGINNING C++ PROGRAMMING
Copy Assignment Operator



by Simey

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



by Simey

Operator Overloading

Overloading the copy assignment operator (deep copy)

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



by Simey

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



by Simey

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

LearnProgramming



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

LearnProgramming



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

LearnProgramming



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

LearnProgramming



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



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



Operator Overloading

Mystring operator- make lowercase

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

BEGINNING C++ PROGRAMMING

Operator as Member Function



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



Operator Overloading

Overloading the Move assignment operator

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

```
Mystring &Mystring::operator=(Mystring &&rhs);
sl = Mystring("Joe"); // move operator= called
sl = "Frank"; // move operator= called
```

BEGINNING C++ PROGRAMMING

Move Assignment Operator



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



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



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;
}
```

BEGINNING C++ PROGRAMMING

Operator as Member Function



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

15. 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

BEGINNING C++ PROGRAMMING
What is Inheritance?

Learn Programming

30 mins

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
What is Inheritance?

Learn Programming

30 mins

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
What is Inheritance?

Learn Programming

30 mins

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, ...  
};
```

BEGINNING C++ PROGRAMMING
What is Inheritance?

Learn Programming

30 mins

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?

Learn Programming

30 mins

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

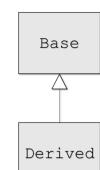
Learn Programming

30 mins

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

Learn Programming

30 mins

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

Learn Programming

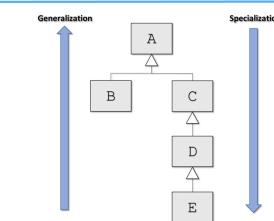
30 mins

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

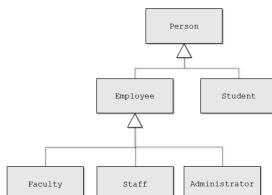
Learn Programming

30 mins

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

Learn Programming

30 mins

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

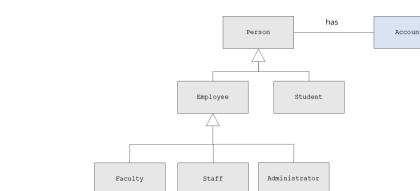
BEGINNING C++ PROGRAMMING
Inheritance vs Composition

Learn Programming

30 mins

Inheritance

Public Inheritance vs. Composition



BEGINNING C++ PROGRAMMING
Inheritance vs Composition

Learn Programming

30 mins

Inheritance

Composition

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

BEGINNING C++ PROGRAMMING
Inheritance vs Composition

LearnProgramming

13 min

Deriving classes from existing 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

13 min

Deriving classes from existing 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

13 min

Deriving classes from existing 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

BEGINNING C++ PROGRAMMING
Deriving Classes from Existing Classes

LearnProgramming

13 min

Deriving classes from existing 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

13 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

13 min

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

13 min

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

13 min

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

13 min

Deriving classes from existing classes

Access with **public** inheritance

Base Class

```
public: a  
protected: b  
private: c
```

public inheritance

Derived Class

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

public inheritance

Deriving classes from existing classes

Access with **protected** inheritance

Base Class

```
public: a  
protected: b  
private: c
```

protected inheritance

Access in
Derived Class

```
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 first
 - Derived class constructor executes

BEGINNING C++ PROGRAMMING
Constructors and Destructors

LearnProgramming

13 min

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

13 min

Constructors and Destructors

Constructors and class initialization

Output

```
Base base;
```

Base constructor

```
Derived derived;
```

Base constructor
Derived constructor

BEGINNING C++ PROGRAMMING
Constructors and Destructors



11 mins

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



11 mins

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



11 mins

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



11 mins

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 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{} {  
        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 base(100);  
  
Derived derived;  
Derived derived(100);
```

```
int Base constructor  
Base no-args constructor  
Derived no-args constructor  
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



10 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



10 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



10 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



10 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



10 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



10 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



10 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



10 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 ????
```

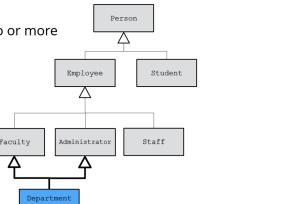
BEGINNING C++ PROGRAMMING
Using And Refining Base Methods



10 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



10 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



10 mins

16. 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

BEGINNING C++ PROGRAMMING
What is Polymorphism?



What is Polymorphism?

An non-polymorphic example - Static Binding

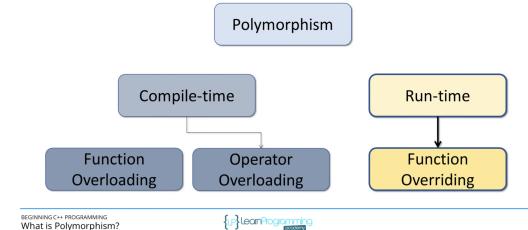
```
void display_account(const Account &acc) {  
    acc.display();  
    // will always use Account::display  
}  
  
Account a;  
display_account(a);  
  
Savings b;  
display_account(b);  
  
Checking c;  
display_account(c);  
  
Trust d;  
display_account(d);
```

BEGINNING C++ PROGRAMMING
What is Polymorphism?



What is Polymorphism?

An non-polymorphic example - Static Binding



BEGINNING C++ PROGRAMMING
What is Polymorphism?



What is Polymorphism?

An non-polymorphic example - Static Binding

```
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()
```

BEGINNING C++ PROGRAMMING
What is Polymorphism?



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

BEGINNING C++ PROGRAMMING
What is Polymorphism?



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);
```

BEGINNING C++ PROGRAMMING
What is Polymorphism?



Polymorphism

Using a Base class pointer

For dynamic polymorphism we must have:

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

BEGINNING C++ PROGRAMMING
Polymorphism



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
```

BEGINNING C++ PROGRAMMING
Polymorphism



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);
```

BEGINNING C++ PROGRAMMING
Polymorphism



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
```

BEGINNING C++ PROGRAMMING
Polymorphism



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

BEGINNING C++ PROGRAMMING
Virtual Functions



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);  
    . . .  
};
```

BEGINNING C++ PROGRAMMING
Virtual Functions



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 its base class

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

BEGINNING C++ PROGRAMMING
Virtual Functions

Learn Programming
C++

15 min

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

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

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();  
};
```

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

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

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

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 OVERWRITING  
        std::cout << "Hello - I'm a Derived class object" << std::endl;  
    }  
    virtual ~Derived() {}  
};
```

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

Polymorphism

The override specifier

```
Base:  
  
    virtual void say_hello() const;  
  
Derived:  
  
    virtual void say_hello();
```

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

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!
```

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

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  
        std::cout << "Hello - I'm a Derived class object" << std::endl;  
    }  
    virtual ~Derived() {}  
};
```

BEGINNING C++ PROGRAMMING
Virtual Destructor

Learn Programming
C++

15 min

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

BEGINNING C++ PROGRAMMING
The Final Specifier

Learn Programming
C++

15 min

Polymorphism

final class

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

BEGINNING C++ PROGRAMMING
The Final Specifier

Learn Programming
C++

15 min

Polymorphism

final method

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

BEGINNING C++ PROGRAMMING
The Final Specifier

Learn Programming
C++

15 min

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

BEGINNING C++ PROGRAMMING
Using Base Class References

Learn Programming
C++

15 min

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
```

BEGINNING C++ PROGRAMMING

Using Base Class References

Learn Programming

15 min

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

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

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
```

BEGINNING C++ PROGRAMMING

Using Base Class References

Learn Programming

15 min

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

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

Polymorphism

Pure virtual functions and abstract classes

- Pure virtual function
 - Used to make a class abstract
 - Specified with "=0" in its declaration
 - Typically do not provide implementations

Polymorphism

Pure virtual functions and abstract classes

- Pure virtual function
 - Used to make a class abstract
 - Specified with "=0" in its declaration
 - Typically do not provide implementations

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

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
```

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

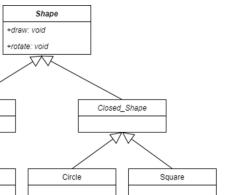
Learn Programming

15 min

Polymorphism

Pure virtual functions and abstract classes

Shape Class Hierarchy



BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

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();  
};
```

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

Polymorphism

Pure virtual functions and abstract classes

```
class Circle: public Shape {  
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();  
};
```

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

Polymorphism

Pure virtual functions and abstract classes

Abstract Base class

- Cannot be instantiated

```
Shape shape; // Error
```

```
ptr->draw(); // Error
```

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

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

BEGINNING C++ PROGRAMMING

Pure Virtual and Abstract Classes

Learn Programming

15 min

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

BEGINNING C++ PROGRAMMING

Abstract Classes As Interfaces

Learn Programming

15 min

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 its implementation at compile time

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

• any_object must conform to the Printable interface

BEGINNING C++ PROGRAMMING

Abstract Classes As Interfaces

Learn Programming

15 min

Polymorphism

A Printable example

```
class Printable {
    friend ostream &operator<<(ostream &s, 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;
}
```

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

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" ;
    }
    ...
};
```

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

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"
```

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

Polymorphism

A Shapes example

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

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

Polymorphism

A Shapes example

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

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

Polymorphism

A Shapes example

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

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

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
```

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

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
```

BEGINNING C++ PROGRAMMING
Abstract Classes As Interfaces



33 min

17. 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?

BEGINNING C++ PROGRAMMING

Issues with Raw Pointers

learnProgramming

Smart Pointers

What are they?

- Objects
- Can only point to heap-allocated memory
- Automatically call delete when no longer needed
- Adhere to RAIIs 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

A simple example

```
{  
    std::smart_pointer<Some_Class> ptr = ...  
  
    ptr->method();  
    cout << (*ptr) << endl;  
}  
  
// ptr will be destroyed automatically when  
// no longer needed
```

BEGINNING C++ PROGRAMMING

What are Smart Pointers

learnProgramming

Smart Pointers

RAII – Resource Acquisition Is Initialization

- Common idiom or pattern used in software design based on container object lifetime
- RAIIs 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 - 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
```

BEGINNING C++ PROGRAMMING

Unique Pointers

learnProgramming

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
```

BEGINNING C++ PROGRAMMING

Unique Pointers

learnProgramming

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

BEGINNING C++ PROGRAMMING

What are Smart Pointers

learnProgramming

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

BEGINNING C++ PROGRAMMING

Unique Pointers

learnProgramming

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
```

BEGINNING C++ PROGRAMMING

Unique Pointers

learnProgramming

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
```

BEGINNING C++ PROGRAMMING

Unique Pointers

learnProgramming

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

BEGINNING C++ PROGRAMMING

Unique Pointers

learnProgramming

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

BEGINNING C++ PROGRAMMING

Shared Pointers

learnProgramming

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
```

BEGINNING C++ PROGRAMMING
Shared Pointers



33 min

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
```

BEGINNING C++ PROGRAMMING
Shared Pointers



33 min

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
```

BEGINNING C++ PROGRAMMING
Shared Pointers



33 min

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
```

BEGINNING C++ PROGRAMMING
Shared Pointers



33 min

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

BEGINNING C++ PROGRAMMING
Weak Pointers

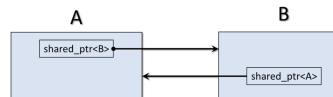


33 min

Smart Pointers

weak_ptr - circular or cyclic reference

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



BEGINNING C++ PROGRAMMING
Weak Pointers

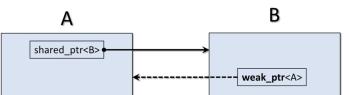


33 min

Smart Pointers

weak_ptr - circular or cyclic reference

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



BEGINNING C++ PROGRAMMING
Weak Pointers



33 min

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 ways to achieve this
 - Functions
 - Lambdas
 - Others...

BEGINNING C++ PROGRAMMING
Custom Deleters



33 min

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 };
```

BEGINNING C++ PROGRAMMING
Custom Deleters



33 min

Smart Pointers

Custom deleters - lambda

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

BEGINNING C++ PROGRAMMING
Custom Deleters



33 min

18. Exception Handling

Exception Handling

Basic concepts

- Exception handling
 - dealing with extraordinary situations
 - indicates that an extraordinary situation has been detected or has occurred
 - program can deal with the extraordinary situations in a suitable manner

- What causes exceptions?
 - insufficient resources
 - missing resources
 - invalid operations
 - range violations
 - underflows and overflows
 - illegal data and many others

- Exception safe
 - when your code handles exceptions

BEGINNING C++ PROGRAMMING
Basic Concepts and Example



15 min

Exception Handling

Divide by zero example

- What happens if `total` is zero?
 - crash, overflow?
 - it depends

```
double average {};
average = sum / total;
```

BEGINNING C++ PROGRAMMING
Basic Concepts and Example



15 min

Exception Handling

Terminology

- Exception
 - an object or primitive type that signals that an error has occurred
- Throwing an exception (raising an exception)
 - your code detects that an error has occurred or will occur
 - the place where the error occurred may not know how to handle the error
 - code can throw an exception describing the error to another part of the program that knows how to handle the error
- Catching an exception (handle the exception)
 - code that handles the exception
 - may or may not cause the program to terminate

BEGINNING C++ PROGRAMMING
Basic Concepts and Example



15 min

Exception Handling

C++ Syntax

- `try { code that may throw an exception }`
 - you place code that may throw an exception in a try block
 - if the code throws an exception the try block is exited
 - the thrown exception is handled by a catch handler
 - if no catch handler exists the program terminates
- `catch(Exception ex) { code to handle the exception }`
 - code that handles the exception
 - can have multiple catch handlers
 - may or may not cause the program to terminate

BEGINNING C++ PROGRAMMING
Basic Concepts and Example



15 min

Exception Handling

Divide by zero example

- What happens if `total` is zero?
 - crash, overflow?
 - it depends

Exception Handling

Divide by zero example

- What happens if `total` is zero?
 - crash, overflow?
 - it depends

```
double average {};
if (total == 0)
    // what to do?
else
    average = sum / total;
```

BEGINNING C++ PROGRAMMING
Basic Concepts and Example



15 min

Exception Handling

Divide by zero example

```
double average {};
try {
    if (total == 0)
        throw 0;           // throw the exception
    average = sum / total; // won't execute if total == 0
    // use average here
}
catch (int &ex) {          // exception handler
    std::cerr << "can't divide by zero" << std::endl;
}
std::cout << "program continues" << std::endl;
```

BEGINNING C++ PROGRAMMING
Basic Concepts and Example



15 min

Exception Handling

Throwing an exception from a function

What do we return if `total` is zero?

```
double calculate_avg(int sum, int total) {
    return static_cast<double>(sum) / total;
}
```

BEGINNING C++ PROGRAMMING
Throwing from a Function



15 min

Exception Handling

Throwing an exception from a function

Throw an exception if we can't complete successfully

```
double calculate_avg(int sum, int total) {
    if (total == 0)
        throw 0;
    return static_cast<double>(sum) / total;
}
```

BEGINNING C++ PROGRAMMING
Throwing from a Function



15 min

Exception Handling

Catching an exception thrown from a function

```
double average {};
try {
    average = calculate_avg(sum, total);
    std::cout << average << std::endl;
}
catch (int &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}

std::cout << "Bye" << std::endl;
```

BEGINNING C++ PROGRAMMING
Throwing from a Function



15 min

Exception Handling

Throwing multiple exceptions from a function

What if a function can fail in several ways

- gallons is zero

- miles or gallons is negative

```
double calculate_mpg(int miles, int gallons) {
    return static_cast<double>(miles) / gallons;
}
```

BEGINNING C++ PROGRAMMING
Handling Multiple Exceptions



15 min

Exception Handling

Throwing an exception from a function

Throw different type exceptions for each condition

```
double calculate_mpg(int miles, int gallons) {
    if (gallons == 0)
        throw 0;
    if (miles < 0 || gallons < 0)
        throw std::string("Negative value error");
    return static_cast<double>(miles) / gallons;
}
```

BEGINNING C++ PROGRAMMING
Handling Multiple Exceptions



15 min

Exception Handling

Catching an exception thrown from a function

```
double miles_per_gallon ();
try {
    miles_per_gallon = calculate_mpg(miles, gallons);
    std::cout << miles_per_gallon << std::endl;
}
catch (int &ex) {
    std::cerr << "You can't divide by zero" << std::endl;
}
catch (std::string &ex) {
    std::cerr << ex << std::endl;
}

std::cout << "Bye" << std::endl;
```

BEGINNING C++ PROGRAMMING
Handling Multiple Exceptions



15 min

Exception Handling

Catching any type of exception

```
catch ( int &ex) {  
}  
catch (std::string &ex) {  
}  
catch (...) {  
    std::cerr << "Unknown exception" << std::endl;  
}
```

BEGINNING C++ PROGRAMMING
Handling Multiple Exceptions



15 min

Exception Handling

Creating exception classes

```
class DivideByZeroException {  
};  
  
class NegativeValueException {  
};
```

BEGINNING C++ PROGRAMMING
User Defined Exceptions



15 min

Exception Handling

Stack unwinding

- If an exception is thrown but not caught in the current scope C++ tries to find a handler for the exception by unwinding the stack
- Function in which the exception was not caught terminates and is removed from the call stack
- If a try block was used then catch blocks are checked for a match
- If no try block was used or the catch handler doesn't match stack unwinding occurs again
- If the stack is unwound back to main and no catch handler handles the exception the program terminates

BEGINNING C++ PROGRAMMING
Stack Unwinding



15 min

Exception Handling

User-defined exceptions

We can create exception classes and throw instances of those classes

Best practice:

- throw an object not a primitive type
- throw an object by value
- catch an object by reference (or const reference)

BEGINNING C++ PROGRAMMING
User Defined Exceptions



15 min

Exception Handling

Creating user-defined exception classes

```
double calculate_mpg(int miles, int gallons) {  
  
    if (gallons == 0)  
        throw DivideByZeroException();  
    if (miles < 0 || gallons < 0)  
        throw NegativeValueException();  
  
    return static_cast<double>(miles) / gallons;  
}
```

BEGINNING C++ PROGRAMMING
User Defined Exceptions



15 min

Exception Handling

Throwing user-defined exception classes

```
try {  
    miles_per_gallon = calculate_mpg(miles, gallons);  
    std::cout << miles_per_gallon << std::endl;  
}  
catch (const DivideByZeroException &ex) {  
    std::cerr << "You can't divide by zero" << std::endl;  
}  
catch (const NegativeValueException &ex) {  
    std::cerr << "Negative values aren't allowed" << std::endl;  
}  
  
std::cout << "Bye" << std::endl;
```

BEGINNING C++ PROGRAMMING
User Defined Exceptions



15 min

Exception Handling

Catching user-defined exceptions

```
try {  
    miles_per_gallon = calculate_mpg(miles, gallons);  
    std::cout << miles_per_gallon << std::endl;  
}  
catch (const DivideByZeroException &ex) {  
    std::cerr << "You can't divide by zero" << std::endl;  
}  
catch (const NegativeValueException &ex) {  
    std::cerr << "Negative values aren't allowed" << std::endl;  
}  
  
std::cout << "Bye" << std::endl;
```

BEGINNING C++ PROGRAMMING
User Defined Exceptions



15 min

Exception Handling

Class-level exceptions

Exceptions can also be thrown from within a class:

- Method
 - These work the same way as they do for functions as we've seen
- Constructor
 - Constructors may fail
 - Constructors do not return any value
 - Throw an exception in the constructor if you cannot initialize an object
- Destructor
 - Do NOT throw exceptions from your destructor

BEGINNING C++ PROGRAMMING
Class Level Exceptions



15 min

Exception Handling

Class-level exceptions

```
Account::Account(std::string name, double balance)  
: name(name), balance(balance) {  
  
    if (balance < 0.0)  
        throw IllegalBalanceException();  
  
}
```

BEGINNING C++ PROGRAMMING
Class Level Exceptions



15 min

Exception Handling

Class-level exceptions

```
try {  
    std::unique_ptr<Account> moes_account =  
        std::make_unique<Checking_Account>("Moe", -10.0);  
    // use moes_account  
}  
catch (const IllegalBalanceException &ex) {  
    std::cerr << "Couldn't create account" << std::endl;  
}
```

BEGINNING C++ PROGRAMMING
Class Level Exceptions



15 min

Exception Handling

The C++ standard library exception class hierarchy

C++ provides a class hierarchy of exception classes

- `std::exception` is the base class
- all subclasses implement the `what()` virtual function
- we can create our own user-defined exception subclasses

```
virtual const char *what() const noexcept;
```

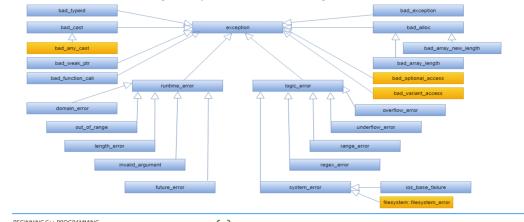
BEGINNING C++ PROGRAMMING
Standard Exceptions



15 min

Exception Handling

The C++ standard library exception class hierarchy



BEGINNING C++ PROGRAMMING
Standard Exceptions



15 min

Exception Handling

Deriving our class from `std::exception`

```
class IllegalBalanceException: public std::exception  
{  
public:  
    IllegalBalanceException() noexcept = default;  
    ~IllegalBalanceException() = default;  
    virtual const char* what() const noexcept {  
        return "Illegal balance exception";  
    }  
};
```

BEGINNING C++ PROGRAMMING
Standard Exceptions



15 min

Exception Handling

Our modified Account class constructor

```
Account::Account(std::string name, double balance)
: name{name}, balance{balance} {
    if (balance < 0.0)
        throw IllegalBalanceException{};
}
```

BEGINNING C++ PROGRAMMING
Standard Exceptions



15 min

Exception Handling

Creating an Account object

```
try {
    std::unique_ptr<Account> moes_account =
        std::make_unique<Checking_Account>("Moe", -100.0);
    std::cout << "Use moes_account" << std::endl;
} catch (const IllegalBalanceException &ex)
{
    std::cerr << ex.what() << std::endl;
    // displays "Illegal balance exception"
}
```

BEGINNING C++ PROGRAMMING
Standard Exceptions



15 min

19. File, Streams and I/O

Files, Streams and I/O

- C++ uses streams as an interface between the program and input and output devices
- Independent of the actual device
- Sequence of bytes
- Input stream provides data to the program
- Output stream receives data from the program

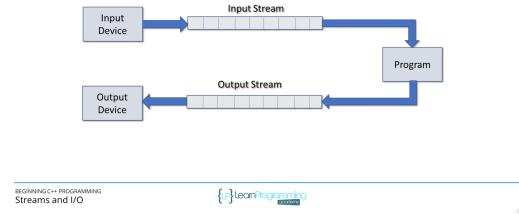
BEGINNING C++ PROGRAMMING

Streams and I/O

LearnProgramming

13 min

Files, Streams and I/O



Files, Streams and I/O

Commonly used stream classes

Class	Description
ios	Provides basic support for formatted and unformatted I/O operations. Base class most other classes
ifstream	Provides for high-level input operations on file based streams
ofstream	Provides for high-level output operations on file based streams
fstream	Provides for high-level I/O operations on file based streams Derived from ofstream and ifstream
stringstream	Provides for high-level I/O operations on memory based strings Derived from istream and ostream

BEGINNING C++ PROGRAMMING

Streams and I/O

LearnProgramming

13 min

Files, Streams and I/O

Global stream objects

Object	Description
cin	Standard input stream – by default ‘connected’ to the standard input device (keyboard) Instance of istream
cout	Standard output stream – by default ‘connected’ to the standard output device (console) Instance of ostream
cerr	Standard error stream – by default ‘connected’ to the standard error device (console) Instance of ostream (unbuffered)
clog	Standard log stream – by default ‘connected’ to the standard log device (console) Instance of ostream (unbuffered)

- Global objects – initialized before main executes
- Best practice is to use cerr for error messages and clog for log messages.

BEGINNING C++ PROGRAMMING

Streams and I/O

LearnProgramming

13 min

Files, Streams and I/O

Common header files

Header File	Description
iostream	Provides definitions for formatted input and output from/to streams
fstream	Provides definitions for formatted input and output from/to file streams
iomanip	Provides definitions for manipulators used to format stream I/O

BEGINNING C++ PROGRAMMING

Streams and I/O

LearnProgramming

13 min

Files, Streams and I/O

Common stream manipulators

- Boolean
 - boolalpha, noboolalpha
- Integer
 - dec, hex, oct, showbase, noshowbase, showpos, noshowpos, uppercase, nouppercase
- Floating point
 - fixed, scientific, setprecision, showpoint, noshowpoint, showpos, noshowpos
- Field width, justification and fill
 - setw, left, right, internal, setfill
- Others
 - endl, flush, skipws, noskipws, ws

BEGINNING C++ PROGRAMMING

Manipulators

LearnProgramming

13 min

Files, Streams and I/O

Formatting boolean types

- Default when displaying boolean values is 1 or 0
- Sometimes the strings true or false are more appropriate

BEGINNING C++ PROGRAMMING

Manipulators Boolean

LearnProgramming

13 min

Files, Streams and I/O

Formatting boolean types

```
std::cout << (10 == 10) << std::endl;  
std::cout << (10 == 20) << std::endl;
```

// Will display

```
1  
0
```

BEGINNING C++ PROGRAMMING

Manipulators Boolean

LearnProgramming

13 min

Files, Streams and I/O

Formatting boolean types

```
std::cout << std::boolalpha;  
  
std::cout << (10 == 10) << std::endl;  
std::cout << (10 == 20) << std::endl;  
  
// Will display  
  
true  
false
```

BEGINNING C++ PROGRAMMING

Manipulators Boolean

LearnProgramming

13 min

Files, Streams and I/O

Formatting boolean types

- All further boolean output will be affected
- ```
std::cout << std::noboolalpha;// 1 or 0
std::cout << std::boolalpha; // true or false
```

BEGINNING C++ PROGRAMMING

Manipulators Boolean

LearnProgramming

13 min

## Files, Streams and I/O

Formatting boolean types

```
std::cout.setf(std::ios::boolalpha);
std::cout.setf(std::ios::noboolalpha);
```

• Reset to default

```
std::cout << std::resetiosflags(std::ios::boolalpha);
```

BEGINNING C++ PROGRAMMING

Manipulators Boolean

LearnProgramming

13 min

## Files, Streams and I/O

### Formatting integer types

- Default when displaying integer values is:
  - dec (base 10)
  - noshowbase - prefix used to show hexadecimal or octal
  - nouppercase - when displaying a prefix and hex values it will be lower case
  - noshowpos - no '+' is displayed for positive numbers
- These manipulators affect all further output to the stream

BEGINNING C++ PROGRAMMING  
Manipulators Integer



33 min

## Files, Streams and I/O

### Formatting integer types - display hex in uppercase

```
int num {255};

std::cout << std::showbase << std::uppercase;
std::cout << std::hex << num << std::endl;

// Will display

0xFF // note capitalized XFF
```

BEGINNING C++ PROGRAMMING  
Manipulators Integer



33 min

## Files, Streams and I/O

### Formatting integer types - setting base

```
int num {255};

std::cout << std::dec << num << std::endl;
std::cout << std::hex << num << std::endl;
std::cout << std::oct << num << std::endl;

// Will display
255
ff
277
```

BEGINNING C++ PROGRAMMING  
Manipulators Integer



33 min

## Files, Streams and I/O

### Formatting integer types - showing the base

```
int num {255};

std::cout << std::showbase; // std::noshowbase
std::cout << std::dec << num << std::endl;
std::cout << std::hex << num << std::endl;
std::cout << std::oct << num << std::endl;

// Will display
255
0xff // note the 0x prefix for hexadecimal
0377 // note the 0 prefix for octal
```

BEGINNING C++ PROGRAMMING  
Manipulators Integer



33 min

## Files, Streams and I/O

### Formatting integer types - displaying the positive sign

```
int num1 {255};
int num2 {-255};

std::cout << num1 << std::endl; // 255
std::cout << num2 << std::endl; // -255

std::cout << std::showpos; // std::noshowpos

std::cout << num1 << std::endl; // +255
std::cout << num2 << std::endl; // -255
```

BEGINNING C++ PROGRAMMING  
Manipulators Integer



33 min

## Files, Streams and I/O

### Setting/resetting integer types

- Set using setf
  - std::cout.setf(std::ios::showbase);
  - std::cout.setf(std::ios::uppercase);
  - std::cout.setf(std::ios::showpos);
- Reset to defaults
  - std::cout << std::resetiosflags(std::ios::basefield);
  - std::cout << std::resetiosflags(std::ios::showbase);
  - std::cout << std::resetiosflags(std::ios::showpos);
  - std::cout << std::resetiosflags(std::ios::uppercase);

BEGINNING C++ PROGRAMMING  
Manipulators Integer



33 min

## Files, Streams and I/O

### Formatting floating point types

- Default when displaying floating point values is:
  - setprecision - number of digits displayed (6)
  - fixed - not fixed to a specific number of digits after the decimal point
  - noshowpoint - trailing zeroes are not displayed
  - nouppercase - when displaying in scientific notation
  - noshowpos - no '+' is displayed for positive numbers
- These manipulators affect all further output to the stream

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

### Formatting floating point types - precision

```
double num {1234.5678};

std::cout << num << std::endl;

// Will display

1234.57 // Notice precision is 6 and rounding
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

### Formatting floating point types - precision

```
double num {123456789.987654321};

std::cout << num << std::endl;

// Will display

1.234567e+008 // Notice precision is 6
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

### Formatting floating point types - precision

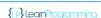
```
double num {123456789.987654321};

std::cout << std::setprecision(9);
std::cout << num << std::endl;

// Will display

123456790 // Note that rounding occurs
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

### Formatting floating point types - fixed

```
double num {123456789.987654321};

std::cout << std::fixed;
std::cout << num << std::endl;

// Will display precision 6 from the decimal

123456789.987654
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

### Formatting floating point types - fixed

```
double num {123456789.987654321};

std::cout << std::setprecision(3) << std::fixed;
std::cout << num << std::endl;

// Will display precision 3 from the decimal

123456789.988
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

Formatting floating point types - scientific

```
double num {123456789.987654321};

std::cout << std::setprecision(3)
 << std::scientific;
std::cout << num << std::endl;

// Will display precision 3
1.23e+008
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

Formatting floating point types - scientific uppercase

```
double num {123456789.987654321};

std::cout << std::setprecision(3)
 << std::scientific
 << std::uppercase;
std::cout << num << std::endl;

// Will display precision 3
1.23E+008 // Note the capital 'E'
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

Formatting floating point types - displaying the positive sign

```
double num {123456789.987654321};

std::cout << std::setprecision(3)
 << std::fixed
 << std::showpos;
std::cout << num << std::endl;

// Will display
+123456789.988 // Note the leading '+'
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

Formatting floating point types - trailing zeroes

```
double num {12.34};

std::cout << num << std::endl; // 12.34

std::cout << std::showpoint;
std::cout << num << std::endl; // 12.3400

// Will display
12.34 // Note no trailing zeroes (default)
12.3400 // Note trailing zeroes up to precision
```

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

Returning to general settings

- unsetf  
std::cout.unsetf(std::ios::scientific | std::ios::fixed);
- or  
std::cout << std::resetiosflags(std::ios::floatfield);
- Refer to the docs for other set/reset flags

BEGINNING C++ PROGRAMMING  
Manipulators Floating Point



33 min

## Files, Streams and I/O

Field width, align and fill

- Default when displaying floating point values is:
  - setw - not set by default
  - left - when no field width, right - when using field width
  - fill - not set by default - blank space is used
- Some of these manipulators affect only the next data element put on the stream

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Defaults

```
double num {1234.5678};
std::string hello("Hello");

std::cout << num << hello << std::endl;

// Will display
1234.57Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Defaults

```
double num {1234.5678};
std::string hello("Hello");

std::cout << num << std::endl;
std::cout << hello << std::endl;

// Will display
1234.57
Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Field width - setw

```
double num {1234.5678};
std::string hello("Hello");

std::cout << std::setw(10) << num
 << std::setw(10) << hello
 << std::setw(10) << hello << std::endl;

// Will display
1234567890123456789012345678901234567890
1234.57Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Field width - setw

```
double num {1234.5678};
std::string hello("Hello");

std::cout << std::setw(10) << num
 << std::setw(10) << hello
 << std::setw(10) << hello << std::endl;

// Will display
1234567890123456789012345678901234567890
1234.57 Hello Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Field width - setw

```
double num {1234.5678};
std::string hello("Hello");

std::cout << std::setw(10) << num
 << std::setw(10) << std::right << hello
 << std::setw(15) << std::right << hello
 << std::endl;

// Will display
1234567890123456789012345678901234567890
1234.57 Hello Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Filling fixed width - setfill

```
double num {1234.5678};
std::string hello("Hello");

std::cout << std::setfill('-')
 << std::setw(10) << num
 << std::setw(10) << hello
 << std::endl;

// Will display
1234567890123456789012345678901234567890
---1234.57Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



33 min

## Files, Streams and I/O

Field width - setw

```
double num (1234.5678);
std::string hello("Hello");

std::cout << std::setfill('*');
std::cout << std::setw(10) << num
<< std::setfill('-') << std::setw(10) << hello
<< std::setw(15) << hello
<< std::endl;

// Will display
123456789012345678901234567890
***1234.57-----Hello-----Hello
```

BEGINNING C++ PROGRAMMING  
Manipulators Align and Fill



## Files, Streams and I/O

Opening a file for reading with (ifstream)

```
std::ifstream in_file ("../myfile.txt",
 std::ios::in);

std::ifstream in_file ("../myfile.txt");

// Open for reading in binary mode
std::ifstream in_file ("../myfile.txt",
 std::ios::binary);
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Check if file opened successfully – test the stream object

```
if (in_file) { // just check the object
 // read from it
} else {
 // file could not be opened
 // does it exist?
 // should the program terminate?
}
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Reading from files using getline

- We can use getline to read the file one line at a time

```
std::string line(); // This is a line
std::getline(in_file, line);
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Input files (fstream and ifstream)

fstream and ifstream are commonly used for input files

- #include <fstream>
- Declare an ifstream or ifstream object
- Connect it to a file on your file system (opens it for reading)
- Read data from the file via the stream
- Close the stream

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Opening a file for reading with (fstream)

```
std::fstream in_file ("../myfile.txt",
 std::ios::in);
```

• Open for reading in binary mode

```
std::fstream in_file ("../myfile.txt",
 std::ios::in | std::ios::binary);
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Opening a file for reading with open

```
std::ifstream in_file;
std::string filename;
std::cin >> filename; // get the file name

in_file.open(filename);
// or
in_file.open(filename, std::ios::binary);
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Check if file opened successfully (is\_open)

```
if (in_file.is_open()) {
 // read from it
} else {
 // file could not be opened
 // does it exist?
 // should the program terminate?
}
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Closing a file

## Files, Streams and I/O

Closing a file

- Always close any open files to flush out any unwritten data

```
in_file.close();
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Reading from files using (>>)

- We can use the extraction operator for formatted read
- Same way we used it with cin

```
int num {};
double total {};
std::string name {};
```

|        |
|--------|
| 100    |
| 255.67 |
| Larry  |

```
in_file >> num;
in_file >> total >> name;
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Reading text file one line at a time

## Files, Streams and I/O

Reading text file one line at a time

```
std::ifstream in_file("../myfile.txt"); // open file
std::string line {};

if (!in_file) { // check if file is open
 std::cerr << "File open error" << std::endl;
 return 1; // exit the program (main)
}
while (!in_file.eof()) { // while not at the end
 std::getline(in_file, line); // read a line
 cout << line << std::endl; // display the line
}
in_file.close(); // close the file
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Reading text file one line at a time

```
std::ifstream in_file("../myfile.txt"); // open file
std::string line {};

if (!in_file) { // check if file is open
 std::cerr << "File open error" << std::endl;
 return 1; // exit the program (main)
}
while (std::getline(in_file, line)) // read a line
 cout << line << std::endl; // display the line
in_file.close(); // close the file
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File



## Files, Streams and I/O

Reading text file one character at a time (get)

```
std::ifstream in_file("../myfile.txt"); // open file
char c;

if (!in_file) { // check if file is open
 std::cerr << "File open error" << std::endl;
 return 1; // exit the program (main)
}
while (in_file.get(c)) // read a character
 cout << c; // display the character

in_file.close(); // close the file
```

BEGINNING C++ PROGRAMMING  
Reading from a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Opening a file for writing with (fstream)

```
std::fstream out_file ("../myfile.txt",
 std::ios::out);

// Open for writing in binary mode
std::fstream out_file ("../myfile.txt",
 std::ios::out | std::ios::binary);
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Output files (fstream and ofstream)

fstream and ofstream are commonly used for output files

1. #include <fstream>
2. Declare an fstream or ofstream object
3. Connect it to a file on your file system (opens it for writing)
4. Write data to the file via the stream
5. Close the stream

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Output files (fstream and ofstream)

fstream and ofstream are commonly used for output files

- Output files will be created if they don't exist
- Output files will be overwritten (truncated) by default
- Can be opened so that new writes append
- Can be open in text or binary modes

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Opening a file for writing with (ofstream)

## Files, Streams and I/O

Opening a file for writing with (ofstream)

```
std::ofstream out_file ("../myfile.txt",
 std::ios::out);

std::ofstream out_file ("../myfile.txt");

// Open for writing in binary mode
std::ofstream out_file ("../myfile.txt",
 std::ios::binary);
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Opening a file for writing with (ofstream)

```
// truncate (discard contents) when opening
std::ofstream out_file ("../myfile.txt",
 std::ios::trunc);

// append on each write
std::ofstream out_file ("../myfile.txt",
 std::ios::app);

// seek to end of stream when opening
std::ofstream out_file ("../myfile.txt",
 std::ios::ate);
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Opening a file for writing with open

```
std::ofstream out_file;
std::string filename;
std::cin >> filename; // get the file name

out_file.open(filename);
// or
out_file.open(filename, std::ios::binary);
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Check if file opened successfully (is\_open)

```
if (out_file.is_open()) {
 // read from it
} else {
 // file could not be created or opened
 // does it exist?
 // should the program terminate?
}
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Check if file opened successfully – test the stream object

```
if (out_file) { // just check the object
 // read from it
} else {
 // file could not be opened
 // does it exist?
 // should the program terminate?
}
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Closing a file

- Always close any open files to flush out any unwritten data

```
out_file.close();
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Writing to files using (<<)

- We can use the insertion operator for formatted write
- Same way we used it with cout

```
int num (100);
double total (255.67);
std::string name {"Larry"};

out_file << num << "\n"
 << total << "\n"
 << name << std::endl;
```

100  
255.67  
Larry

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Copying a text file one line at a time

```
std::ifstream in_file("../myfile.txt"); // open file
std::ofstream out_file("../copy.txt");

if (!in_file) { // check if file is open
 std::cerr << "File open error" << std::endl;
 return 1; // exit the program (main)
}
if (!out_file) { // check if file is open
 std::cerr << "File create error" << std::endl;
 return 1; // exit the program (main)
}
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
C++

15 min

## Files, Streams and I/O

Copying a text file one line at a time

```
std::string line {};\n\nwhile (std::getline(in_file, line)) // read a line\n out_file << line << std::endl; // write a line\n\nin_file.close(); // close the files\nout_file.close();
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Using string streams

- Allow us to read or write from strings in memory much as we would read and write to files
- Very powerful
- Very useful for data validation

BEGINNING C++ PROGRAMMING  
Using String Streams

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Copying a text file one character at a time (get/put)

```
std::ifstream in_file("../myfile.txt"); // open file\nstd::ofstream out_file("../copy.txt");\n\nif (!in_file) { // check if file is open\n std::cerr << "File open error" << std::endl;\n return 1; // exit the program (main)\n}\nif (!out_file) { // check if file is open\n std::cerr << "File create error" << std::endl;\n return 1; // exit the program (main)\n}
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Copying a text file one character at a time (get/put)

```
char c;\n\nwhile (in_file.get(c)) // read a character\n out_file.put(c); // write the character\n\nin_file.close(); // close the files\nout_file.close();
```

BEGINNING C++ PROGRAMMING  
Writing to a Text File

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Using string streams

- Allow us to read or write from strings in memory much as we would read and write to files
- Very powerful
- Very useful for data validation

BEGINNING C++ PROGRAMMING  
Using String Streams

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Using string streams

stringstream, istringstream and ostringstream

- #include <sstream>
- Declare an stringstream, istringstream or ostringstream object
- Connect it to a std::string
- Read/write data from/to the string stream using formatted I/O

BEGINNING C++ PROGRAMMING  
Using String Streams

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Reading from a stringstream

```
#include <sstream>\n\nint num {};\ndouble total {};\nstd::string name {};\nstd::string info {"Moe 100 1234.5"};\n\nstd::istringstream iss(info);\niss >> name >> num >> total;
```

BEGINNING C++ PROGRAMMING  
Using String Streams

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Writing to a stringstream

```
#include <sstream>\n\nint num {100};\ndouble total {1234.5};\nstd::string name {"Moe"};\n\nstd::ostringstream oss{};\nostringstream << name << " " << num << " " << total;\nstd::cout << oss.str() << std::endl;
```

BEGINNING C++ PROGRAMMING  
Using String Streams

Learn Programming  
Tutorial

33 min

## Files, Streams and I/O

Validating input with stringstream

```
int value {};\nstd::string input{};\n\nstd::cout << "Enter an integer: ";\nstd::cin >> input;\n\nstd::stringstream ss(input);\nif (ss >> value) {\n std::cout << "An integer was entered";\n} else\n std::cout << "An integer was NOT entered";
```

BEGINNING C++ PROGRAMMING  
Using String Streams

Learn Programming  
Tutorial

33 min

## 20. Standard Template Library (STL)

