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

Operator Overloading

What is Operator Overloading?

Suppose we have a Number class that models any number

- Using functions:

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

- Using member methods:

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


Operator Overloading

What is Operator Overloading?

Suppose we have a Number class that models any number

- Using overloaded operators

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


Operator Overloading

What operators can be overloaded?

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

operator
::
:?
.*
.
sizeof

Operator Overloading

Some basic rules

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

Operator Overloading

Some examples

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


Operator Overloading

Mystring class declaration

```
class Mystring {

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

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


Operator Overloading

Copy assignment operator (=)

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

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

s2 = s1;                 // assignment
```

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

Operator Overloading

Overloading the copy assignment operator (deep copy)

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


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

s2 = s1;                 // We write this

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


Operator Overloading

Overloading the copy assignment operator (deep copy)

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

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

   return *this;
}
```


Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Check for self assignment

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

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

```
delete [] str;
```


Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Allocate storage for the deep copy

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

Operator Overloading

Overloading the copy assignment operator – steps for deep copy

- Perform the copy

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

- Return the current by reference to allow chain assignment

```
return *this;


// s1 = s2 = s3;
```


Operator Overloading

Move assignment operator (=)

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

```
Mystring s1;

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

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

Operator Overloading

Overloading the Move assignment operator

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


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

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

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


Operator Overloading

Overloading the Move assignment operator – steps

- Check for self assignment

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

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

```
delete [] str;
```


Operator Overloading

Overloading the Move assignment operator – steps for deep copy

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

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

- Null out the rhs pointer

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

- Return the current object by reference to allow chain assignment

```
return *this;
```


Operator Overloading

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

```
ReturnType Type::operatorOp();


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

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


Operator Overloading

Mystring operator- make lowercase

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

larry1.display();          // LARRY

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

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


Operator Overloading

Mystring operator- make lowercase

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


Operator Overloading

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

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


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

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


Operator Overloading

Mystring `operator==`

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

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


Operator Overloading

Mystring `operator+ (concatenation)`

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

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

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

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


Operator Overloading

Mystring `operator+ (concatenation)`

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

Operator Overloading

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

```
ReturnType operatorOp(Type &obj);

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

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


Operator Overloading

Mystring `operator-` make lowercase

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

larry1.display();                // LARRY

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

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


Operator Overloading

Mystring `operator-`

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

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


Operator Overloading

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

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

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

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


Operator Overloading

Mystring `operator==`

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

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

Operator Overloading

Mystring `operator+ (concatenation)`

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

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

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

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


Operator Overloading

Mystring `operator+ (concatenation)`

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


Operator Overloading

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

```
Mystring larry {"Larry"};

cout << larry << endl;   // Larry


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

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


Operator Overloading

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

```
Mystring larry;

cin >> larry;


Player hero;

cin >> hero;
```


Operator Overloading

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

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

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

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


Operator Overloading

stream insertion operator (<<)

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

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

Operator Overloading

stream extraction operator (>>)

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

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

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

Inheritance

Related classes

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

Inheritance

Accounts

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

Inheritance

Accounts – without inheritance – code duplication

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

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

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

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


Inheritance

Accounts – with inheritance – code reuse

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

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

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

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


Inheritance

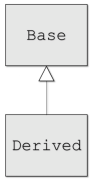
Terminology

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

Inheritance

Terminology

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



Inheritance

Terminology

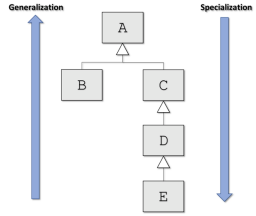
- "Is-A" relationship
  - Public inheritance
  - Derived classes are sub-types of their Base classes
  - Can use a derived class object wherever we use a base class object
- Generalization
  - Combining similar classes into a single, more general class based on common attributes
- Specialization
  -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

Inheritance

Class hierarchy

Classes:

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

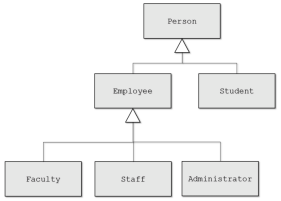


Inheritance

Class hierarchy

Classes:

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



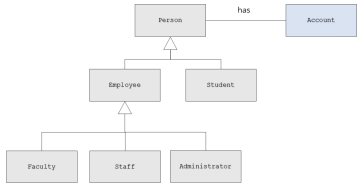
Inheritance

Public Inheritance vs. Composition

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

Inheritance

Public Inheritance vs. Composition



Inheritance

Composition

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


Deriving classes from exiting classes

C++ derivation syntax

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

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

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

Deriving classes from exiting classes

Types of inheritance in C++

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

Deriving classes from exiting classes

C++ derivation syntax

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

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

`Savings_Account` 'is-a' `Account`

Deriving classes from exiting classes

C++ creating objects

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

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

delete p_account;
```


Deriving classes from exiting classes

C++ creating objects

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

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

delete p_sav_account;
```


Protected Members and Class Access

The protected class member modifier

```
class Base {

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

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

Protected Members and Class Access

The protected class member modifier

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

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

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

};
```


Deriving classes from exiting classes

Access with **`public`** inheritance

Base Class		Access in Derived Class

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

Deriving classes from exiting classes

Access with **`protected`** inheritance

Base Class		Access in Derived Class

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

Constructors and Destructors

Constructors and class initialization

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

Constructors and Destructors

Constructors and class initialization

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

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

Constructors and Destructors

Constructors and class initialization

```
                                Output
Base base;                      Base constructor

Derived derived;                Base constructor
                                Derived constructor
```


Constructors and Destructors

Destructors

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

Constructors and Destructors

Destructors

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

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


Constructors and Destructors

Constructors and class initialization

```
                                Output
Base base;                      Base constructor
                                Base destructor

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


Constructors and Destructors

Constructors and class initialization

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

Inheritance

Passing arguments to base class constructors

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

Inheritance

Passing arguments to base class constructors

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

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


Constructors and Destructors

Constructors and class initialization

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


Constructors and Destructors

Constructors and class initialization

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


Constructors and Destructors

Constructors and class initialization

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

Base base{100};                 int Base constructor

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

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


Inheritance

Copy/Move constructors and overloaded operator=

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

Inheritance

Copy constructor

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

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

Constructors and Destructors

Copy Constructors

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

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


Constructors and Destructors

Copy Constructors

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

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


Constructors and Destructors

operator=

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


Constructors and Destructors

operator=

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


Inheritance

Copy/Move constructors and overloaded operator=

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

Inheritance

Using and redefining Base class methods

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

Inheritance

Using and redefining Base class methods

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

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


Inheritance

Static binding of method calls

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

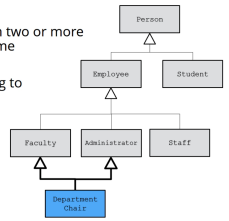
Inheritance

Static binding of method calls

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

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

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


Multiple Inheritance

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



Multiple Inheritance

C++ Syntax

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

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

16. Polymorphism

What is Polymorphism?

Fundamental to Object-Oriented Programming

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

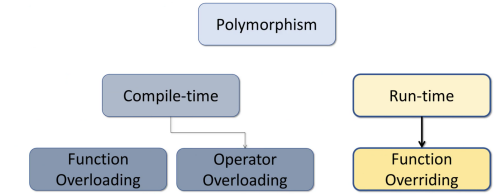
Runtime polymorphism
- Being able to assign different meanings to the same function at run-time

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

What is Polymorphism?

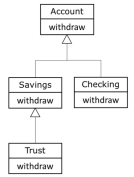
An non-polymorphic example – Static Binding



What is Polymorphism?

An non-polymorphic example – Static Binding

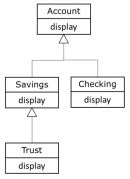
```
Account a;
a.withdraw(1000);   // Account::withdraw()

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

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

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


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



What is Polymorphism?

An non-polymorphic example – Static Binding

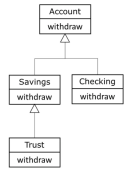
```
void display_account(const Account &acc)  {
   acc.display();
   // will always use Account::display
}

Account a;
display_account(a);

Savings b;
display_account(b);

Checking c;
display_account(c);

Trust d;
display_account(d);
```



What is Polymorphism?

A polymorphic example – Dynamic Binding

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

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

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

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


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

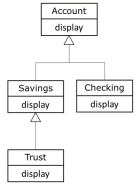


`withdraw` method is virtual in Account

What is Polymorphism?

A polymorphic example – Dynamic Binding

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

Account a;
display_account(a);

Savings b;
display_account(b);

Checking c;
display_account(c);

Trust d;
display_account(d);
```



`display` method is virtual in Account

Polymorphism

Using a Base class pointer

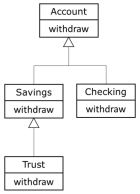
For dynamic polymorphism we must have:

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

Polymorphism

Using a Base class pointer

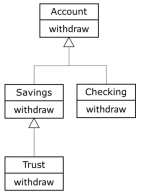
```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

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

// delete the pointers
```



Polymorphism

Using a Base class pointer

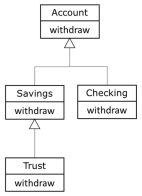
```
Account *p1 = new Account();
Account *p2 = new Savings();
Account *p3 = new Checking();
Account *p4 = new Trust();

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

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



Polymorphism

Using a Base class pointer

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

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

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

// delete the pointers
```



Polymorphism

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

Polymorphism

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

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

Polymorphism

Declaring virtual functions

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

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


Polymorphism

Virtual Destructors

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

Polymorphism

Virtual Destructors

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

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


Polymorphism

The override specifier

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

Polymorphism

The override specifier

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

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


Polymorphism

The override specifier

```
Base:

    virtual void say_hello() const;


Derived:

    virtual void say_hello();
```


Polymorphism

The override specifier

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

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

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

Polymorphism

The override specifier

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

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


Polymorphism

The `final` specifier

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

Polymorphism

`final` class

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

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


Polymorphism

`final` method

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

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

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


Polymorphism

Using Base class references

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

Polymorphism

Using Base class references

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

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


Polymorphism

Using Base class references

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

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

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


Polymorphism

Pure virtual functions and abstract classes

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

Polymorphism

Pure virtual functions and abstract classes

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

Polymorphism

Pure virtual functions and abstract classes

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

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

  - Typically do not provide implementations

Polymorphism

Pure virtual functions and abstract classes

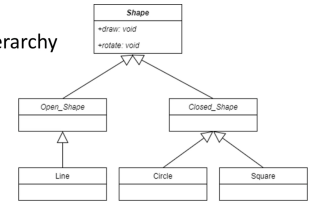
- Pure virtual function
  - Derived classes MUST override the base class
  - If the Derived class does not override then the Derived class is also abstract
  - Used when it doesn't make sense for a base class to have an implementation
    - But concrete classes must implement it

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


Polymorphism

Pure virtual functions and abstract classes

Shape Class Hierarchy



Polymorphism

Pure virtual functions and abstract classes

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


Polymorphism

Pure virtual functions and abstract classes

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


Polymorphism

Pure virtual functions and abstract classes

Abstract Base class

- Cannot be instantiated

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

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

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


Polymorphism

What is using a class as an interface?

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

Polymorphism

A Printable example

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

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

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

Polymorphism

An Printable example

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

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


Polymorphism

An Printable example

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


Polymorphism

An Printable example

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

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

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


Polymorphism

A Shapes example

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


Polymorphism

A Shapes example

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


Polymorphism

A Shapes example

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


Polymorphism

A Shapes example

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

- Line and Square classes would be similar

Polymorphism

A Shapes example

```
vector< I_Shape *> shapes;

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

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

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?

Smart Pointers

What are they?

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

Smart Pointers

What are they?

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

Smart Pointers

A simple example

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

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

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


Smart Pointers

RAII – Resource Acquisition Is Initialization

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

Smart Pointers

`unique_ptr`

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

Smart Pointers

`unique_ptr` – creating, initializing and using

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

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

   *p1 = 200;

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

}// automatically deleted
```


Smart Pointers

`unique_ptr` – some other useful methods

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

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

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

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


Smart Pointers

`unique_ptr` – user defined classes

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

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


}// automatically deleted
```


Smart Pointers

`unique_ptr` – vectors and move

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

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

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

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

}// automatically deleted
```


Smart Pointers

`unique_ptr` – make_unique (C++14)

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

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

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

}// automatically deleted
```

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

Smart Pointers

`shared_ptr`

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

Smart Pointers

`shared_ptr` – creating, initializing and using

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

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

   *p1 = 200;

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

}// automatically deleted
```


Smart Pointers

`shared_ptr` – some other useful methods

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

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

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


Smart Pointers

`shared_ptr` – vectors and move

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

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

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

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


Smart Pointers

`shared_ptr` – make_shared (C++11)

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

}// automatically deleted
```

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

Smart Pointers

`weak_ptr`

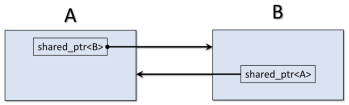
- Provides a non-owning "weak" reference

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

Smart Pointers

`weak_ptr` – circular or cyclic reference

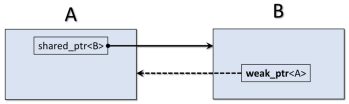
- A refers to B
- B refers to A
- Shared strong ownership prevents heap deallocation



Smart Pointers

`weak_ptr` – circular or cyclic reference

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



Smart Pointers

Custom deleters

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

Smart Pointers

Custom deleters - function

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

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


Smart Pointers

Custom deleters - lambda

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

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

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

Exception Handling

C++ Syntax

- `throw`
  - throws an exception
  - followed by an argument
- `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

Exception Handling

Divide by zero example

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

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


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;
```


Exception Handling

Divide by zero example

```
double average {};
try {                          // try block
   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;
```


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;
}
```


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;
}
```


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;
```


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;
}
```


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;
}
```


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;
```

Exception Handling

Catching any type of exception

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


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 to 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

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)

Exception Handling

Creating exception classes

```
class DivideByZeroException {
};

class NegativeValueException {
};
```


Exception Handling

Throwing 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;
}
```


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;
```


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

Exception Handling

Class-level exceptions

```
Account::Account(std::string name, double balance)
    : name{name}, balance{balance} {

    if (balance < 0.0)
        throw IllegalBalanceException{};

}
```


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;
}
```


Exception Handling

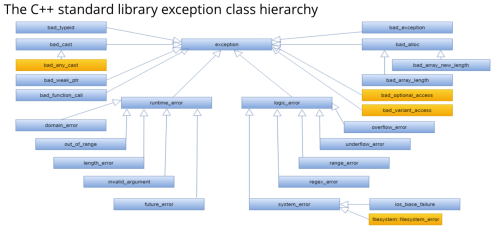
The C++ standard library exception class hierarchy

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;
```


Exception Handling

The C++ standard library exception class hierarchy



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";
    }
};
```

Exception Handling

Our modified Account class constructor

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


Exception Handling

Creating an Account object

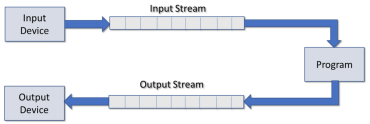
```
try {
    std::unique_ptr<Account> moes_account =
        std::make_unique<Checking_Account>("Moe", -100.0);

    std::cout << "Use moes_account" << std::endl;
}
catch (const IllegalBalanceException &ex)
{
    std::cerr << ex.what() << std::endl;
          // displays "Illegal balance exception"
}
```

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

Files, Streams and I/O



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

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 `istringstream` and `ostringstream`

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.

Files, Streams and I/O

Stream manipulators

- Streams have useful member functions to control formatting
- Can be used on input and output streams
- The time of the effect on the stream varies
- Can be used as member functions or as a manipulato

```
std::cout.width(10);            // member function
std::cout << std::setw(10); // manipulator
```

- We'll focus on manipulator usage

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

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

Files, Streams and I/O

Formatting boolean types

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

// Will display

1
0
```


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
```


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
```


Files, Streams and I/O

Formatting boolean types

- Method version

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

- Reset to default

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

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

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
```


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
```


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
```


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
```


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);
```


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

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
```


Files, Streams and I/O

Formatting floating point types - precision

```
double num {123456789.987654321};

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

// Will display

1.23457e+008      // Notice precision is 6
```


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
```


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
```


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
```

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
```


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'
```


Files, Streams and I/O

Formatting floating point types - displaying the positive sign

```
double num {123456789.987654321};

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

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

// Will display

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


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
```


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

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

Files, Streams and I/O

Defaults

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

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

// Will display

1234.57Hello
```


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
```


Files, Streams and I/O

Field width - setw

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

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

// Will display
12345678901234567890123456789012345678901234567890
   1234.57Hello
```


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
12345678901234567890123456789012345678901234567890
   1234.57     Hello     Hello
```


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
12345678901234567890123456789012345678901234567890
   1234.57     Hello          Hello
```


Files, Streams and I/O

Filling fixed width - setfill

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

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

// Will display
12345678901234567890123456789012345678901234567890
---1234.57Hello
```

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
12345678901234567890123456789012345678901234567890
***1234.57-----Hello---------Hello
```


Files, Streams and I/O

Input files (fstream and ifstream)

`fstream` and `ifstream` are commonly used for input files

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

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};
```


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};
```


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);
```


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?
}
```


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?
}
```


Files, Streams and I/O

Closing a file

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

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


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 {};

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

```
100
255.67
Larry
```


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{};

std::getline(in_file, line);
```

```
This is a line
```


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
```


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
```

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
```


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

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

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};
```


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};
```


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};
```


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);
```


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?
}
```


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?
}
```


Files, Streams and I/O

Closing a file

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

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


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
```


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)
}
```

Files, Streams and I/O

Copying a text file one line at a time

```
std::string line {};

while (std::getline(in_file, line)) // read a line
   out_file << line << std::endl;   // write a line

in_file.close();    // close the files
out_file.close();
```


Files, Streams and I/O

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

```
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)
}
```


Files, Streams and I/O

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

```
char c;

while (in_file.get(c)) // read a character
   out_file.put(c);    // write the character

in_file.close();    // close the files
out_file.close();
```


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

Files, Streams and I/O

Using string streams

`stringstream, istringstream and ostringstream`

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

Files, Streams and I/O

Reading from a `stringstream`

```
#include <sstream>

int num {};
double total {};
std::string name {};
std::string info {"Moe  100  1234.5"};

std::istringstream iss{info};
iss >> name >> num >> total;
```


Files, Streams and I/O

Writing to a `stringstream`

```
#include <sstream>

int num {100};
double total {1234.5};
std::string name {"Moe"};

std::ostringstream oss{};
oss << name << " " << num << " " << total;
std::cout << oss.str() << std::endl;
```


Files, Streams and I/O

Validating input with `stringstream`

```
int value {};
std::string input{};

std::cout << "Enter an integer: ";
std::cin >> input;

std::stringstream ss{input};
if (ss >> value) {
   std::cout << "An integer was entered";
else
   std::cout << "An integer was NOT entered";
```

20. Standard Template Library (STL)

What is the STL?

- A library of powerful, reusable, adaptable, generic classes and functions
- Implemented using C++ templates
- Implements common data structures and algorithms
- Huge class library!!
- Alexander Stepanov (1994)

Why use the STL?

- Assortment of commonly used containers
- Known time and size complexity
- Tried and tested – Reusability!!!
- Consistent, fast, and type-safe
- Extensible

Elements of the STL

- Containers
  - Collections of objects or primitive types (array, vector, deque, stack, set, map, etc.)
- Algorithms
  - Functions for processing sequences of elements from containers (find, max, count, accumulate, sort, etc.)
- Iterators
  - Generate sequences of element from containers (forward, reverse, by value, by reference, constant, etc.)

Elements of the STL

A simple example

```
#include <vector>
#include <algorithm>

std::vector<int> v {1,5,3};
```


Elements of the STL

A simple example – sort a vector

```
std::sort(v.begin(), v.end());

for (auto elem: v)
   std::cout << elem << std::endl;

1
3
5
```


Elements of the STL

A simple example – reverse a vector

```
std::reverse(v.begin(), v.end());

for (auto elem: v)
   std::cout << elem << std::endl;

5
3
1
```


Elements of the STL

A simple example - accumulate

```
int sum{};

sum = std::accumulate(v.begin(), v.end(), 0);
std::cout << sum << std::endl;

9    // 1+3+5
```


Types of Containers

- Sequence containers
  - array, vector, list, forward_list, deque
- Associative containers
  - set, multi set, map, multi map
- Container adapters
  - stack, queue, priority queue

Types of Iterators

- Input iterators – from the container to the program
- Output iterators – from the program to the container
- Forward iterators – navigate one item at a time in one direction
- Bi-directional iterators – navigate one item at a time both directions
- Random access iterators – directly access a container item

Types of Algorithms

- About 60 algorithms in the STL
- Non-modifying
- Modifying

The Standard Template Library

Generic Programming with macros

- Generic programming
  "Writing code that works with a variety of types as arguments, as long as those argument types meet specific syntactic and semantic requirements", Bjarne Stroustrup
- Macros ***** beware *****
- Function templates
- Class templates

The Standard Template Library

Macros (`#define`)

- C++ preprocessor directives
- No type information
- Simple substitution

```
#define MAX_SIZE 100

#define PI  3.14159
```

The Standard Template Library

Macros (#define)

```
//#define MAX_SIZE 100      // removed

//#define PI 3.14159        // removed

if (num > 100)
    std::cout << "Too big");

double area = 3.14159 * r * r;
```


The Standard Template Library

max function

- Suppose we need a function to determine the max of 2 integers

```
int max(int a, int b) {
   return (a > b) ? a : b;
}

int x = 100;
int y = 200;
std::cout << max(x, y);      // displays 200
```


The Standard Template Library

max function

- Now suppose we need to determine the max of 2 doubles, and 2 chars

```
int max(int a, int b) {
   return (a > b) ? a : b;
}

double max(double a, double b) {
   return (a > b) ? a : b;
}

char max(char a, char b) {
   return (a > b) ? a : b;
}
```


The Standard Template Library

Macros with arguments (#define)

- We can write a generic macro with arguments instead

```
#define MAX(a, b) ((a > b) ? a : b)

std::cout << MAX(10,20)    << std::endl;   // 20
std::cout << MAX(2.4, 3.5) << std::endl;   // 3.5
std::cout << MAX('A', 'C') << std::endl;   // C
```


The Standard Template Library

Macros with arguments (#define)

- We have to be careful with macros

```
#define SQUARE(a) a*a

result = SQUARE(5);          // Expect 25
result = 5*5;                // Get 25

result = 100/SQUARE(5);      // Expect 4
result = 100/5*5             // Get 100!
```


The Standard Template Library

Macros with arguments (#define)

```
#define SQUARE(a) ((a)*(a)) // note the parenthesis

result = SQUARE(5);          // Expect 25
result = ((5)*(5));          // Still Get 25

result = 100/SQUARE(5);      // Expect 4
result = 100/((5)*(5));      // Now we get 4!!
```


The Standard Template Library

Generic Programming with function templates

What is a C++ Template?

- Blueprint
- Function and class templates
- Allow **plugging-in** any data type
- Compiler generates the appropriate function/class from the blueprint
- Generic programming / meta-programming

The Standard Template Library

Generic Programming with function templates

- Let's revisit the max function from the last lecture

```
int max(int a, int b) {
   return (a > b) ? a : b;
}

int x = 100;
int y = 200;
std::cout << max(x, y);      // displays 200
```


The Standard Template Library

max function

- Now suppose we need to determine the max of 2 doubles, and 2 chars

```
int max(int a, int b) {
   return (a > b) ? a : b;
}

double max(double a, double b) {
   return (a > b) ? a : b;
}

char max(char a, char b) {
   return (a > b) ? a : b;
}
```


The Standard Template Library

max function as a template function

- We can replace type we want to generalize with a name, say T
- But now this won't compile

```
T max(T a, T b) {
   return (a > b) ? a : b;
}
```


The Standard Template Library

max function as a template function

- We need to tell the compiler this is a template function
- We also need to tell it that T is the template parameter

```
template <typename T>
T max(T a, T b) {
   return (a > b) ? a : b;
}
```


The Standard Template Library

max function as a template function

- We may also use **class** instead of **typename**

```
template <class T>
T max(T a, T b) {
   return (a > b) ? a : b;
}
```

The Standard Template Library

max function as a template function

- Now the compiler can generate the appropriate function from the template
- Note, this happens at compile-time!

```
int a {10};
int b {20};

std::cout << max<int>(a, b);
```


The Standard Template Library

max function as a template function

- Many times the compiler can deduce the type and the template parameter is not needed
- Depending on the type of a and b, the compiler will figure it out

```
std::cout << max<double>(c, d);

std::cout << max(c, d);
```


The Standard Template Library

max function as a template function

- And we can use **almost** any type we need

```
char a {'A'};
char b {'Z'};

std::cout << max(a, b) << std::endl;
```


The Standard Template Library

max function as a template function

- Notice the type MUST support the `>` operator either natively or as an overloaded operator **(operator>)**

```
template <typename T>
T max(T a, T b) {
   return (a > b) ? a : b;
}
```


The Standard Template Library

max function as a template function

- The following will not compile unless `Player` overloads `operator>`

```
Player p1{"Hero", 100, 20};
Player p2{"Enemy", 99, 3};

std::cout << max<Player>(p1, p2);
```


The Standard Template Library

multiple types as template parameters

- We can have multiple template parameters
- And their types can be different

```
template <typename T1, typename T2>
void func(T1 a, T2 b) {
   std::cout << a << " " << b;
}
```


The Standard Template Library

multiple types as template parameters

- When we use the function we provide the template parameters
- Often the compiler can deduce them

```
func<int,double>(10, 20.2);

func('A', 12.4);
```


The Standard Template Library

Generic Programming with class templates

What is a C++ **Class** Template?

- Similar to function template, but at the class level
- Allows **plugging-in** any data type
- Compiler generates the appropriate class from the blueprint

The Standard Template Library

Generic Programming with class templates

- Let's say we want a class to hold Items where the item has a name and an integer

```
class Item {
private:
    std::string name;
    int value;
public:
    Item(std::string name, int value)
       : name{name}, value{value}
    {}
    std::string get_name() const {return name; }
    int get_value() const { return value; }
};
```


The Standard Template Library

Generic Programming with class templates

- But we'd like our `Item` class to be able to hold any type of data in addition to the string
- We can't overload class names
- We don't want to use dynamic polymorphism

The Standard Template Library

Generic Programming with class templates

```
class Item {
private:
    std::string name;
    T value;
public:
    Item(std::string name, T value)
       : name{name}, value{value}
    {}
    std::string get_name() const {return name; }
    T get_value() const { return value; }
};
```


The Standard Template Library

Generic Programming with class templates

```
template <typename T>
class Item {
private:
    std::string name;
    T value;
public:
    Item(std::string name, T value)
       : name{name}, value{value}
    {}
    std::string get_name() const {return name; }
    T get_value() const { return value; }
};
```

The Standard Template Library

Generic Programming with class templates

```
Item<int> item1 {"Larry", 1};

Item<double> item2 {"House", 1000.0};

Item<std::string> item3 {"Frank", "Boss"};

std::vector<Item<int>> vec;
```


The Standard Template Library

Multiple types as template parameters

- We can have multiple template parameters
- An their types can be different

```
template <typename T1, typename T2>
struct My_Pair {
    T1 first;
    T2 second;
};
```


The Standard Template Library

Multiple types as template parameters

```
My_Pair <std::string, int> p1 {"Frank", 100};

My_Pair <int, double> p2 {124, 13.6};

std::vector<My_Pair<int, double>> vec;
```


The Standard Template Library

std::pair

```
#include <utility>

std::pair<std::string, int> p1 {"Frank", 100};

std::cout << p1.first;      // Frank
std::cout << p1.second;     // 100
```


The Standard Template Library

Containers

- Data structures that can store object of *almost* any type
  - Template-based classes
- Each container has member functions
  - Some are specific to the container
  - Others are available to all containers
- Each container has an associated header file
  - `#include <container_type>`

The Standard Template Library

Containers – common

Function	Description
`Default constructor`	Initializes an empty container
`Overloaded constructors`	Initializes containers with many options
`Copy constructor`	Initializes a container as a copy of another container
`Move constructor`	Moves existing container to new container
`Destructor`	Destroys a container
`Copy assignment (operator=)`	Copy one container to another
`Move assignment (operator=)`	Move one container to another
`size`	Returns the number of elements in the container
`empty`	Returns boolean – is the container empty?
`insert`	Insert an element into the container

The Standard Template Library

Containers – common

Function	Description
`operator< and operator<=`	Returns boolean - compare contents of 2 containers
`operator> and operator>=`	Returns boolean - compare contents of 2 containers
`operator== and operator!=`	Returns boolean - are the contents of 2 containers equal or not
`swap`	Swap the elements of 2 containers
`erase`	Remove element(s) from a container
`clear`	Remove all elements from a container
`begin and end`	Returns iterators to first element or end
`rbegin and rend`	Returns reverse iterators to first element or end
`cbegin and cend`	Returns constant iterators to first element or end
`crbegin and crend`	Returns constant reverse iterators to first element or end

The Standard Template Library

Container elements

What types of elements can we store in containers?

- A **copy** of the element will be stored in the container
  - All primitives OK
- Element should be
  - Copyable and assignable (copy constructor / copy assignment)
  - Moveable for efficiency (move Constructor / move Assignment)
- Ordered associative containers must be able to compare elements
  - `operator<, operator==`

The Standard Template Library

Iterators

- Allows abstracting an arbitrary container as a sequence of elements
- They are objects that work like pointers by design
- Most container classes can be traversed with iterators

The Standard Template Library

Declaring iterators

- Iterators must be declared based on the container type they will iterate over

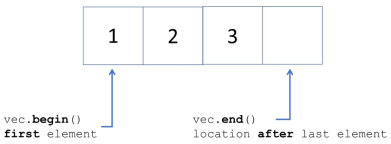
container_type::iterator_type *iterator_name*;

```
std::vector<int>::iterator it1;

std::list<std::string>::iterator it2;

std::map<std::string, std::string>::iterator it3;

std::set<char>::iterator it4;
```


The Standard Template Library

iterator begin and end methods

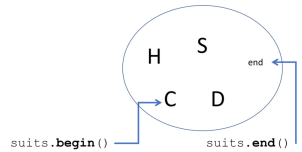
```
std::vector<int> vec {1,2,3};
```



The Standard Template Library

Declaring iterators

```
std::set<char> suits {'C','H','S','D'};
```



The Standard Template Library

Initializing iterators

```
std::vector<int> vec {1,2,3};

std::vector<int>::iterator it = vec.begin();

or

auto it = vec.begin();
```


The Standard Template Library

Operations with iterators `(it)`

Operation	Description	Type of Iterator
`++it`	Pre-increment	All
`it++`	Post-increment	All
`it = it1`	Assignment	All
`*it`	Dereference	Input and Output
`it->`	Arrow operator	Input and Output
`it == it1`	Comparison for equality	Input
`it != it1`	Comparison for inequality	Input
`--it`	Pre-decrement	Bidirectional
`it--`	Post-decrement	Bidirectional
`it + i, it += i` `it - i, it -= i`	Increment and decrement	Random Access
`it < it1, it <= it1` `it > it1, it >= it1`	Comparison	Random Access

The Standard Template Library

Using iterators – std::vector

```
std::vector<int> vec {1,2,3};

std::vector<int>::iterator it = vec.begin();

while (it != vec.end()) {
   std::cout << *it << " ";
   ++it;
}
// 1 2 3
```


The Standard Template Library

Using iterators – std::vector

```
std::vector<int> vec {1,2,3};

for (auto it = vec.begin(); it != vec.end(); it++) {
   std::cout << *it << " ";
}
// 1 2 3
```

- This is how the range-based for loop works

The Standard Template Library

Using iterators – std::set

```
std::set<char> suits {'C', 'H', 'S', 'D'};

auto it = suits.begin();

while (it != suits.end()) {
   std::cout << *it << " " << std::end;
   ++it;
}
// C H S D
```


The Standard Template Library

Reverse iterators

- Works in reverse
- Last element is the first and first is the last
- ++ moves backward, -- moves forward

```
std::vector<int> vec {1,2,3};
std::vector<int>::reverse_iterator it = vec.begin();
while (it != vec.end()) {
   std::cout << *it << " ";
   ++it;
}
// 3 2 1
```


The Standard Template Library

Other iterators

- `begin()` and `end()`
  - `iterator`
- `cbegin()` and `cend()`
  - `const_iterator`
- `rbegin()` and `rend()`
  - `reverse_iterator`
- `crbegin()` and `crend()`
  - `const_reverse_iterator`

The Standard Template Library

Algorithms

- STL algorithms work on sequences of container elements provided to them by an iterator
- STL has many common and useful algorithms
- Too many to describe in this section
  - http://en.cppreference.com/w/cpp/algorithm
- Many algorithms require extra information in order to do their work
  - Functors (function objects)
  - Function pointers
  - Lambda expressions (C++11)

The Standard Template Library

Algorithms and iterators

- `#include <algorithm>`
- Different containers support different types of iterators
  - Determines the types of algorithms supported
- All STL algorithms expect iterators as arguments
  - Determines the sequence obtained from the container

The Standard Template Library

Iterator invalidation

- Iterators point to container elements
- It's possible iterators become invalid during processing
- Suppose we are iterating over a vector of 10 elements
  - And we clear() the vector while iterating? What happens?
  - Undefined behavior – our iterators are pointing to invalid locations

The Standard Template Library

Example algorithm – `find` with primitive types

- The `find` algorithm tries to locate the first occurrence of an element in a container
- Lots of variations
- Returns an iterator pointing to the located element or `end()`

```
std::vector<int> vec {1,2,3};

auto loc = std::find(vec.begin(), vec.end(), 3);

if (loc != vec.end())              // found it!
   std::cout << *loc << std::endl; // 3
```


The Standard Template Library

Example algorithm – `find` with user-defined types

- Find needs to be able to compare object
- `operator==` is used and must be provided by your class

```
std::vector<Player> team { /* assume initialized */ }
Player p {"Hero", 100, 12};

auto loc = std::find(team.begin(), team.end(), p);

if (loc != vec.end())                 // found it!
   std::cout << *loc << std::endl;    // operator<< called
```

The Standard Template Library

Example algorithm – `for_each`

- `for_each` algorithm applies a function to each element in the iterator sequence

- Function must be provided to the algorithm as:
  - Functor (function object)
  - Function pointer
  - Lambda expression (C++11)

- Let's square each element

The Standard Template Library

`for_each` - using a functor

```
struct Square_Functor {
      void operator()(int x)      // overload () operator
            std::cout << x * x << " ";
      }
};
Square_Functor square;      // Function object

std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(), square);
// 1 4 9 16
```


The Standard Template Library

`for_each` - using a function pointer

```
void square(int x) {              // function
      std::cout << x * x << " ";
}

std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(), square);
// 1 4 9 16
```


The Standard Template Library

`for_each` - using a lambda expression

```
std::vector<int> vec {1, 2, 3, 4};

std::for_each(vec.begin(), vec.end(),
      [](int x) { std::cout << x * x << " "; }) // lambda

// 1 4 9 16
```


The Standard Template Library

`std::array` (C++11)

```
#include <array>
```

- Fixed size
  - Size must be known at compile time
- Direct element access
- Provides access to the underlying raw array
- Use instead of raw arrays when possible
- All iterators available and do not invalidate

The Standard Template Library

`std::array` – initialization and assignment

```
std::array<int, 5> arr1 { {1,2,3,4,5} };   C++11 vs. C++14

std::array<std::string, 3> stooges {
      std::string("Larry"},
      "Moe",
      std::string("Curly"}
};

arr1 = {2,4,6,8,10};
```


The Standard Template Library

`std::array` – common methods

```
std::array<int, 5> arr {1,2,3,4,5};

std::cout << arr.size();       // 5

std::cout << arr.at(0);        // 1
std::cout << arr[1];           // 2

std::cout << arr.front();      // 1
std::cout << arr.back();       // 5
```


The Standard Template Library

`std::array` – common methods

```
std::array<int, 5> arr {1,2,3,4,5};
std::array<int, 5> arr1 {10,20,30,40,50};

std::cout << arr.empty();      // 0 (false)
std::cout << arr.max_size();   // 5

std::cout << arr.fill(10);     // fills all to 10
arr.swap(arr1);                // swaps the 2 arrays
int *data = arr.data();        // get raw array address
```


The Standard Template Library

`std::vector`

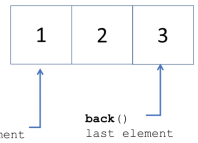
```
#include <vector>
```

- Dynamic size
  - Handled automatically
  - Can expand and contract as needed
  - Elements are stored in contiguous memory as an array
- Direct element access (constant time)
- Rapid insertion and deletion at the back (constant time)
- Insertion or removal of elements (linear time)
- All iterators available and may invalidate

The Standard Template Library

`std::vector`

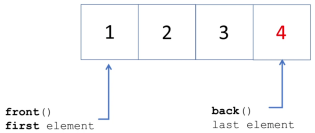
```
std::vector<int> vec {1,2,3};
```



The Standard Template Library

`std::vector`

```
vec.push_back(4)
```



The Standard Template Library

`std::vector` – initialization and assignment

```
std::vector<int> vec {1,2,3,4,5};
std::vector<int> vec1 (10,100);  // ten 100s

std::vector<std::string> stooges {
      std::string("Larry"},
      "Moe",
      std::string("Curly"}
};

vec1 = {2,4,6,8,10};
```

The Standard Template Library

`std::vector` – initialization and assignment

```
std::vector<int> vec {1,2,3,4,5};

std::cout << vec.size();         // 5
std::cout << vec.capacity();     // 5
std::cout << vec.max_size;       // a very large number

std::cout << vec.at(0);          // 1
std::cout << vec[1];             // 2

std::cout << vec.front();        // 1
std::cout << vec.back();         // 5
```


The Standard Template Library

`std::vector` – initialization and assignment

```
Person p1 {"Larry", 18};
std::vector<Person> vec;

vec.push_back(p1);        // add p1 to the back
vec.pop_back();           // remove p1 from the back

vec.push_back(Person{"Larry", 18});

vec.emplace_back("Larry", 18};      // efficient!!
```


The Standard Template Library

`std::vector` – common methods

```
std::vector<int> vec1 {1,2,3,4,5};
std::vector<int> vec2 {10,20,30,40,50};

std::cout << vec1.empty();     // 0 (false)
vec1.swap(vec2);               // swaps the 2 vector

std::sort(vec1.begin(), vec1.end());
```


The Standard Template Library

`std::vector` – common methods

```
std::vector<int> vec1 {1,2,3,4,5};
std::vector<int> vec2 {10,20,30,40,50};

auto it = std::find(vec1.begin(), vec1.end(), 3);
vec1.insert(it, 10);      // 1,2,10,3,4,5

it = std::find(vec1.begin(), vec1.end(), 4);
std::insert(it, vec2.begin(), vec2.end());
     // 1,2,10,3,10,20,30,40,50,4,5
```


The Standard Template Library

`std::deque (double ended queue)`

```
#include <deque>
```

- Dynamic size
  - Handled automatically
  - Can expand and contract as needed
  - Elements are NOT stored in contiguous memory
- Direct element access (constant time)
- Rapid insertion and deletion at the front **and** back (constant time)
- Insertion or removal of elements (linear time)
- All iterators available and may invalidate

The Standard Template Library

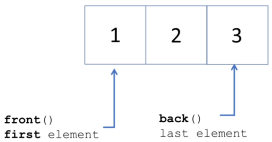
`std::deque` – initialization and assignment

```
std::deque<int> d{1,2,3,4,5};
std::deque<int> d1(10,100);      // ten 100s

std::deque<std::string> stooges {
      std::string("Larry"),
      "Moe",
      std::string("Curly")
};

d = {2,4,6,8,10};
```


The Standard Template Library

`std::deque`

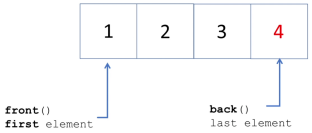
```
std::deque<int> d{1,2,3};
```



The Standard Template Library

`std::deque`

```
d.push_back(4)
```



The Standard Template Library

`std::deque`

The Standard Template Library

`std::deque` – common methods

```
std::deque<int> d {1,2,3,4,5};

std::cout << d.size();          // 5
std::cout << d.max_size;        // a very large number

std::cout << d.at(0);           // 1
std::cout << d[1];              // 2

std::cout << d.front();         // 1
std::cout << d.back();          // 5
```


The Standard Template Library

`std::deque` – common methods

```
Person p1 {"Larry", 18};
std::deque<Person> d;

d.push_back(p1);            // add p1 to the back
d.pop_back();               // remove p1 from the  back

d.push_front(Person{"Larry", 18});
d.pop_front();              // remove element from the front

d.emplace_back("Larry", 18};     // add to back efficient!!
d.emplace_front("Moe", 24};      // add to front
```


The Standard Template Library

`std::list and std::forward_list`

- Sequence containers
- Non-contiguous in memory
- No direct access to elements
- Very efficient for inserting and deleting elements once an element is found

The Standard Template Library

`std::list`

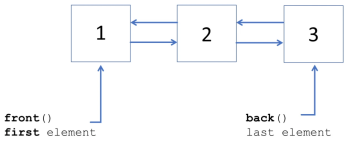
```
#include <list>
```

- Dynamic size
  - Lists of elements
  - list is bidirectional (doubly-linked)
- Direct element access is NOT provided
- Rapid insertion and deletion of elements anywhere in the container (constant time)
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

`std::list`

```
std::list<int> l{1,2,3};
```



front()
first element

back()
last element

The Standard Template Library

`std::list` – initialization and assignment

```
std::list<int> l {1,2,3,4,5};
std::list<int> l1(10,100); // ten 100s

std::list<std::string> stooges {
      std::string("Larry"),
      "Moe",
      std::string("Curly")
};

l = {2,4,6,8,10};
```


The Standard Template Library

`std::list` – common methods

```
std::list<int> l {1,2,3,4,5};

std::cout << l.size();        // 5
std::cout << l.max_size;      // a very large number

std::cout << l.front();       // 1
std::cout << l.back();        // 5
```


The Standard Template Library

`std::list` – common methods

```
Person p1 {"Larry", 18};
std::list<Person> l;

l.push_back(p1);            // add p1 to the back
l.pop_back();               // remove p1 from the  back

l.push_front(Person("Larry", 18));
l.pop_front();              // remove element from the front

l.emplace_back("Larry", 18);     // add to back efficient!!
l.emplace_front("Moe", 24);      // add to front
```


The Standard Template Library

`std::list` – methods that use iterators

```
std::list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

l.insert(it, 10);           // 1 2 10 3 4 5

l.erase(it);                // erases the 3,  1 2 10 4 5

l.resize(2);                // 1 2

l.resize(5);                // 1 2 0 0 0
```


The Standard Template Library

`std::list` – common methods

```
// traversing the list (bi-directional)

std::list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

std::cout << *it;             // 3
it++;
std::cout << *it;             // 4
it--;
std::cout << *it;             // 3
```


The Standard Template Library

`std::forward_list`

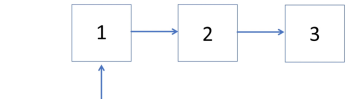
```
#include <forward_list>
```

- Dynamic size
  - Lists of elements
  - list uni-directional (singly-linked)
  - Less overhed than a std::list
- Direct element access is NOT provided
- Rapid insertion and deletion of elements anywhere in the container (constant time)
- Reverse iterators not available. Iterators invalidate when corresponding element is deleted

The Standard Template Library

`std::forward_list`

```
std::forward_list<int> l{1,2,3};
```



front()
first element

The Standard Template Library

`std::forward_list` – common methods

```
std::forward_list<int> l {1,2,3,4,5};

~~std::cout << l.size();~~        // Not available
std::cout << l.max_size;      // a very large number

std::cout << l.front();       // 1
~~std::cout << l.back();~~        // Not available
```


The Standard Template Library

`std::forward_list` – common methods

```
Person p1 {"Larry", 18};
std::forward_list<Person> l;

l.push_front(p1);         // add p1 to the front
l.pop_front();            // remove p1 from the front


l.emplace_front("Moe", 24);        // add to front
```


The Standard Template Library

`std::forward_list` – methods that use iterators

```
std::forward_list<int> l {1,2,3,4,5};
auto it = std::find(l.begin(), l.end(), 3);

l.insert_after(it, 10);     // 1 2 3 10 4 5
l.emplace_after(100);       // 1 2 3 100 10 4 5

l.erase_after(it);          // erases the 100,  1 2 3 10 4 5

l.resize(2);                // 1 2

l.resize(5);                // 1 2 0 0 0
```

The Standard Template Library

The STL Set containers

- Associative containers
  - Collection of stored objects that allow fast retrieval using a key
  - STL provides Sets and Maps
  - Usually implemented as a balanced binary tree or hashsets
  - Most operations are very efficient
- Sets
  - `std::set`
  - `std::unordered_set`
  - `std::multiset`
  - `std::unordered_multiset`

The Standard Template Library

`std::set`

```
#include <set>
```

- Similar to a mathematical set
- Ordered by key
- No duplicate elements
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

`std::set` – initialization and assignment

```
std::set<int> s {1,2,3,4,5};

std::set<std::string> stooges {
      std::string("Larry"},
      "Moe",
      std::string("Curly"}
};

s = {2,4,6,8,10};
```


The Standard Template Library

`std::set` – common methods

```
std::set<int> s {4,1,1,3,3,2,5};   // 1 2 3 4 5

std::cout << s.size();         // 5
std::cout << s.max_size;       // a very large number
```

- No concept of front and back

```
s.insert(7);                   // 1 2 3 4 5 7
```


The Standard Template Library

`std::set` – common methods

```
Person p1 {"Larry", 18};
Person p2 {"Moe", 25};

std::set<Person> stooges;

stooges.insert(p1);                // adds p1 to the set
auto result = stooges.insert(p2);  // adds p2 to the set
```

- uses `operator<` for ordering!
- returns a `std::pair<iterator, bool>`
  - `first` is an iterator to the inserted element or to the duplicate in the set
  - `second` is a boolean indicating success or failure

The Standard Template Library

`std::set` – common methods

```
std::set<int> s {1,2,3,4,5};


s.erase(3);              // erase the 3 : 1 2 4 5

auto it = s.find(5);
if (it != s.end())
   s.erase(it);          // erase the 5: 1 2 4
```


The Standard Template Library

`std::set` – common methods

```
std::set<int> s {1,2,3,4,5};

int num = s.count(1);    // 0 or 1

s.clear();               // remove all elements

s.empty();               // true or false
```


The Standard Template Library

`std::multi_set`

```
#include <set>
```

- Sorted by key
- Allows duplicate elements
- All iterators are available

The Standard Template Library

`std::unordered_set`

```
#include <unordered_set>
```

- Elements are unordered
- No duplicate elements allowed
- Elements cannot be modified
  - Must be erased and new element inserted
- No reverse iterators are allowed

The Standard Template Library

`std::unordered_multiset`

```
#include <unordered_set>
```

- Elements are unordered
- Allows duplicate elements
- No reverse iterators are allowed

The Standard Template Library

The STL Map containers

- Associative containers
  - Collection of stored objects that allow fast retrieval using a key
  - STL provides Sets and Maps
  - Usually implemented as a balanced binary tree or hashsets
  - Most operations are very efficient
- Maps
  - `std::map`
  - `std::unordered_map`
  - `std::multimap`
  - `std::unordered_multimap`

The Standard Template Library

`std::map`

```
#include <map>
```

- Similar to a dictionary
- Elements are stored as Key, Value pairs (std::pair)
- Ordered by key
- No duplicate elements (keys are unique)
- Direct element access using the key
- All iterators available and invalidate when corresponding element is deleted

The Standard Template Library

`std::map` – initialization and assignment

```
std::map<std::string, int> m1 {
    {"Larry", 18},
    {"Moe", 25}
};

std::map<std::string, std::string> m2 {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};
```


The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

std::cout << m.size();            // 3
std::cout << m.max_size;          // a very large number
```

No concept of front and back

The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

std::pair<std::string, std::string> p1 {"James", "Mechanic"};

m.insert(p1);

m.insert(std::make_pair("Roger", "Ranger"));
```


The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

m["Frank"] = "Teacher";       // insert

m["Frank"] = "Instructor";    // update value
m.at("Frank") = "Professor";  // update value
```


The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

m.erase("Anne");                        // erase Anne

if (m.find("Bob") != m.end())           // find Bob
    std::cout << "Found Bob!";

auto it = m.find("George");
if (it != m.end())
    m.erase(it);                        // erase George
```


The Standard Template Library

`std::map` – common methods

```
std::map<std::string, std::string> m {
    {"Bob", "Butcher"},
    {"Anne", "Baker"},
    {"George", "Candlestick maker"}
};

int num = m.count("Bob");  // 0 or 1

m.clear();                 // remove all elements

m.empty();                 // true or false
```


The Standard Template Library

`std::multi_map`

```
#include <map>
```

- Ordered by key
- Allows duplicate elements
- All iterators are available

The Standard Template Library

`std::unordered_map`

```
#include <unordered_map>
```

- Elements are unordered
- No duplicate elements allowed
- No reverse iterators are allowed

The Standard Template Library

`std::unordered_multimap`

```
#include <unordered_map>
```

- Elements are unordered
- Allows duplicate elements
- No reverse iterators are allowed

The Standard Template Library

`std::stack`

- Last-in First-out (LIFO) data structure
- Implemented as an adapter over other STL container
  Can be implemented as a vector, list, or deque
- All operations occur on one end of the stack (top)
- No iterators are supported

The Standard Template Library

`std::stack`

```
#include <stack>
```

- `push` – insert an element at the top of the stack
- `pop` – remove an element from the top of the stack
- `top` – access the top element of the stack
- `empty` – is the stack empty?
- `size` – number of elements in the stack

The Standard Template Library

`std::stack` – initialization

```
std::stack<int> s;                          // deque

std::stack<int, std::vector<int>> s1;       // vector

std::stack<int, std::list<int>> s2;         // list

std::stack<int, std::deque<int>> s3;        // deque
```


The Standard Template Library

`std::stack` – common methods

```
std::stack<int> s;

s.push(10);

s.push(20);

s.push(30);
```

The Standard Template Library

`std::stack` – common methods

```
std::cout << s.top();       // 30

s.pop();                    // 30 is removed

s.pop();                    // 20 is removed
std::cout << s.size();      // 1
```

The Standard Template Library

`std::queue`

- First-in First-out (FIFO) data structure
- Implemented as an adapter over other STL container
  Can be implemented as a list or deque
- Elements are pushed at the back and
  popped from the front
- No iterators are supported

The Standard Template Library

`std::queue`

```
#include <queue>
```

- `push` – insert an element at the back of the queue
- `pop` – remove an element from the front of the queue
- `front` – access the element at the front
- `back` – access the element at the back
- `empty` – is the queue empty?
- `size` – number of elements in the queue

The Standard Template Library

`std::queue` – initialization

```
std::queue<int> q;                          // deque

std::queue<int, std::list<int>> q2;         // list

std::queue<int, std::deque<int>> q3;        // deque
```


The Standard Template Library

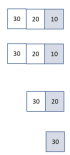
`std::queue` – common methods

```
std::queue<int> q;

q.push(10);

q.push(20);

q.push(30);
```

The Standard Template Library

`std::queue` – common methods

```
std::cout << q.front();     // 10
std::cout << q.back();      // 30

q.pop();                    // remove 10

q.pop();                    // remove 20

std::cout << q.size();      // 1
```



The Standard Template Library

`std::priority_queue`

- Allows insertions and removal of elements in order from the front of the container
- Elements are stored internally as a vector by default
- Elements are inserted in ***priority*** order
  (largest value will always be at the front)
- No iterators are supported

The Standard Template Library

`std::priority_queue`

- Allows insertions and removal of elements in order from the front of the container
- Elements are stored internally as a vector by default
- Elements are inserted in ***priority*** order
  (largest value will always be at the front)
- No iterators are supported

The Standard Template Library

`std::priority_queue`

```
#include <queue>
```

- `push` – insert an element into sorted order
- `pop` – removes the top element (greatest)
- `top` – access the top element (greatest)
- `empty` – is the queue empty?
- `size` – number of elements in the queue

The Standard Template Library

`std::priority_queue` – initialization

```
std::priority_queue<int> pq;          // vector

pq.push(10);
pq.push(20);
pq.push(3);
pq.push(4);

std::cout << pq.top();                // 20 (largest)
pq.pop();                             // remove 20
pq.top();                             /  10 (largest)
```