ECE 449 – OOP and Computer Simulation
Lecture 15 Review

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

December 3, 2015
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Final Exam

- Time: Wed. Dec. 9, 2:00 PM – 4:00 PM
- Closed book/notes, cheat sheet allowed
- Main campus students and Internet students staying at main campus should take exams in this room.
- Check http://www.iit.edu/registrar/important_dates/final_exam_schedule.shtml for other issues.
- Makeup exams will NOT be given.
Deadlines

- Please complete the course evaluation survey through the myIIT portal today.
- Project 4: final release 12/04, report 12/07
- Bonus Project: 12/04
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Challenges

▶ Software systems are complicated, and require a lot of people to collaborate over a long period of time.
▶ Trade-offs are always necessary.
  ▶ Time-to-market and non-recurring engineering (NRE) cost
  ▶ Functionality: more features or less bugs?
  ▶ Performance: optimize for now or future?
Elements of Software Development

- Programming paradigms
  - Define abstraction levels for developers to reason about complex computing systems.
  - Various trade-offs lead to different language designs.

- Software engineering
  - Define methodologies that developers can follow to reduce risk.
  - Effective methodologies vary for different software projects and as workforce changes.
A conventional process of software development.

Stage 1. Requirements Analysis and Definition
Stage 2. System and Software Design
Stage 3. Programming and Unit Testing
Stage 4. Integration and System Testing
Stage 5. Operation and Maintenance

- Waterfall: never go back and revise previous stages

- Advantages
  - Detailed planning for time/personnel/budget within the constraints
  - Goals are well-defined within each stage
Clients usually learn how software can actually help them during the development process.

There may not be an expert to make a feasible plan if you are building something new.

Communication is required to resolve issues arising during integration and system testing but expensive, especially when people leave or enter the team.

Demo is too late and very risky if delay happens.

Complex OS and third-party libraries make operation and maintenance difficult.

Overall not flexible for the rapidly changing world nowadays.
Agile Software Development

- A set of software development methods that teams may choose to satisfy their needs for a specific project.
- Our choices:
  - Iterative and incremental development (IID)
  - Test-driven development (TDD)
  - Code refactoring
  - Continuous integration (CI)
Iterative and Incremental Development (IID)

- An incremental/iterative cycle: build a small portion or make a small revision.
  - The whole system can be assembled and improved across multiple cycles.
  - Progress can be demonstrated at the end of each cycle.
  - Utilize new understandings learned from previous cycles in the next cycle.
- Within each incremental/iterative cycle, the waterfall model can be applied.
  - Changes are limited – much less risky than applying it to the whole system.
Test-Driven Development (TDD)

- Testings at different stages of development
  - Unit testing: for a small unit, e.g. a class
  - Integration testing: for the whole system
  - Acceptance testing: determine whether requirements are met.

- Test-driven development
  - Tests are created before writing the code.
  - The whole system is decomposed into testable pieces.

- Testing help to reproduce bugs easily, and you need to create smaller test cases reproducing bugs to simplify debugging.
Code Refactoring

► Modify code to accommodate more functionalities without breaking existing ones.
► Two steps of code refactoring
  ► Modify code without adding more functionalities and validate your changes using old tests.
  ► Add more functionalities and validate with new tests.
Continuous Integration (CI)

- Manual unit testing is tedious.
  - Need to run many tests and wait.
- Manual integration testing is difficult.
  - Especially if the team is big or the production environment differs from the development environment.
- Continuous integration: enable automated and frequent testing in production environment.
  - As frequent as anyone like while requiring little to no effort from developers – improve quality.
  - The only constraint is the available computational resource for testing.
  - A CI system usually depends on a revision control system, which provides additional benefits of storing all versions of your source code.
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Types and Operators

- Every object has a type.
  - Built-in types: numbers (integer/real), pointers, arrays
  - Class types: defined by `class` or `struct`
- To manipulate an object, we apply operators associated with its type.
  - Either an operator is applied directly in an expression,
  - Or the object is passed to a function where an operator is applied eventually.
- Operator overloading allows us to define operators for class types.
  - Implemented as functions with special names, where operands become parameters
  - Make class types more intuitive for use
Types and Operators

- Every object has a type.
  - Built-in types: numbers (integer/real), pointers, arrays
  - Class types: defined by `class` or `struct`

- To manipulate an object, we apply operators associated with its type.
  - Either an operator is applied directly in an expression,
  - Or the object is passed to a function where an operator is applied eventually.

- Operator overloading allows us to define operators for class types.
  - Implemented as functions with special names, where operands become parameters
  - Make class types more intuitive for use
Every object has a type.
  - Built-in types: numbers (integer/real), pointers, arrays
  - Class types: defined by `class` or `struct`

To manipulate an object, we apply operators associated with its type.
  - Either an operator is applied directly in an expression,
  - Or the object is passed to a function where an operator is applied eventually.

Operator overloading allows us to define operators for class types.
  - Implemented as functions with special names, where operands become parameters
  - Make class types more intuitive for use
More about Operator Overloading

- Most operators in C++ can be overloaded.
  - Except :: (scope) and . (member)
  - It is wise NOT to overload key operators like &(address).

- Limitations
  - One operand must be of a class type.
  - You can only overload operators that are already in C++.
  - Operator precedences and associativities cannot be changed.
  - The placement and the number of the operands cannot be changed.
    - Except (), which can take any number of operands.
More about Operator Overloading

- Most operators in C++ can be overloaded.
  - Except :: (scope) and . (member)
  - It is wise NOT to overload key operators like & (address).

- Limitations
  - One operand must be of an class type.
  - You can only overload operators that are already in C++.
  - Operator precedences and associativities cannot be changed.
  - The placement and the number of the operands cannot be changed.
    - Except (), which can take any number of operands
const std::string hello = "Hello";
const std::string message = hello +", world" +"!";

- + is left-associative: (hello +", world") +"!"
- hello +", world" is interpreted as hello.operator+("", world") or operator+(hello, ",", world).
  - The compiler will call the available one.
  - If both are available, the compiler will complain as it don’t know which one to call.
- The return value +"!" will be handled similarly.
Example 1

```cpp
const std::string hello = "Hello";
const std::string message = hello+", world"+"!";

- + is left-associative: (hello+", world")+"!
- hello+", world" is interpreted as hello.operator+(", world") or operator+(hello, ", world").
  - The compiler will call the available one.
  - If both are available, the compiler will complain as it don’t know which one to call.
- The return value +"!" will be handled similarly.
```
const std::string hello = "Hello";
const std::string message = hello+", world"+"!";

▶ + is left-associative: (hello+", world")+"!"
▶ hello+", world" is interpreted as
   hello.operator+(", world") or
   operator+(hello, ", world").
   ▶ The compiler will call the available one.
   ▶ If both are available, the compiler will complain as it don’t
     know which one to call.
▶ The return value +"!" will be handled similarly.
Example II

```cpp
const std::string exclam = "!";
const std::string message = "Hello"+, world"+exclam;
```

- + is left-associative: ("Hello"+, world")+exclam
- Both "Hello" and ", world" are string literals, which is a built-in type.
  - + is not defined and cannot be redefined.
Example II

```cpp
const std::string exclam = "!";
const std::string message = "Hello"+, world"+exclam;

▶ + is left-associative: ("Hello"+, world")+exclam
▶ Both "Hello" and ", world" are string literals, which is a built-in type.
    ▶ + is not defined and cannot be redefined.
```
const std::string world = "world";
const std::string message = "Hello, " + world + "!";

- + is left-associative: ("Hello, " + world) + "!
- "Hello, " + world is interpreted as
  operator+("Hello, ", world).
  - Note that since "Hello, " is of a built-in type, the compiler
    won’t look for "Hello, ".operator+(world).
- The return value +"!" will be handled as in Example I.
// Example III

const std::string world = "world";
const std::string message = "Hello, "+world+"!";

- + is left-associative: ("Hello, "+world)+"!
- "Hello, "+world is interpreted as operator+("Hello, ", world).
  - Note that since "Hello, " is of a built-in type, the compiler won’t look for "Hello, ".operator+(world).
- The return value +"!" will be handled as in Example I.
const std::string world = "world";
const std::string message = "Hello, " + world + "!";

- + is left-associative: ("Hello, " + world) + "!
- "Hello, " + world is interpreted as operator+("Hello, ", world).
  - Note that since "Hello, " is of a built-in type, the compiler won’t look for "Hello, ".operator+(world).
- The return value +"!" will be handled as in Example I.
double d = f()[n];

- What’s the type of \texttt{f} for the above code to be valid?
  - \texttt{f} is not necessary a function.
  - \texttt{(\quad)} must be defined/overloaded for the type of \texttt{f}.
    - A function taking no argument
  - \texttt{[\quad]} must be defined/overloaded for the return type of \texttt{(\quad)}.
    - It should take \texttt{n} as an argument – note that \texttt{n} is not necessary an integer.
    - The return type should convert to \texttt{double} implicitly.
Example IV

double d = f()[n];

» What’s the type of \texttt{f} for the above code to be valid?
» \texttt{f} is not necessary a function.
» (\texttt{f}) must be defined/overloaded for the type of \texttt{f}.
  » A function taking no argument
» [\texttt{f}] must be defined/overloaded for the return type of (\texttt{f}).
  » It should take \texttt{n} as an argument – note that \texttt{n} is not necessary an integer.
» The return type should convert to \texttt{double} implicitly.
double d = f()[n];

- What’s the type of \( f \) for the above code to be valid?
- \( f \) is not necessary a function.
- \( () \) must be defined/overloaded for the type of \( f \).
  - A function taking no argument
- \([n] \) must be defined/overloaded for the return type of \( () \).
  - It should take \( n \) as an argument – note that \( n \) is not necessary an integer.
  - The return type should convert to \texttt{double} implicitly.
double d = f()[n];

- What’s the type of \( f \) for the above code to be valid?
- \( f \) is not necessary a function.
- \( () \) must be defined/overloaded for the type of \( f \).
  - A function taking no argument
- \( [] \) must be defined/overloaded for the return type of \( () \).
  - It should take \( n \) as an argument – note that \( n \) is not necessary an integer.
  - The return type should convert to \texttt{double} implicitly.
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Class Design

- Class types can be defined with `struct` or `class`.
  - The only difference is the default protection: `struct` is `public` and `class` is `private`.

- Each class type has a class invariant.
  - Constructors establish the class invariant.
  - Public member functions maintain the class invariant.

- An object is needed to access members.
  - Use `->` if the object is referred through a pointer, or `.` otherwise.
  - The object becomes an implicit argument to the member functions, though you can refer to it by `this`.

- The compiler verifies constness and protection labels at compile-time.
Class Design

- Class types can be defined with `struct` or `class`.
  - The only difference is the default protection: `struct` is public and `class` is private.
- Each class type has a class invariant.
  - Constructors establish the class invariant.
  - Public member functions maintain the class invariant.
- An object is needed to access members.
  - Use `->` if the object is referred through a pointer, or `.` otherwise.
  - The object becomes an implicit argument to the member functions, though you can refer to it by `this`.
- The compiler verifies constness and protection labels at compile-time.
Class Design

- Class types can be defined with `struct` or `class`.
  - The only difference is the default protection: `struct` is `public` and `class` is `private`.
- Each class type has a class invariant.
  - Constructors establish the class invariant.
  - Public member functions maintain the class invariant.
- An object is needed to access members.
  - Use `->` if the object is referred through a pointer, or `.` otherwise.
  - The object becomes an implicit argument to the member functions, though you can refer to it by `this`.
- The compiler verifies constness and protection labels at compile-time.
Class Design

- Class types can be defined with `struct` or `class`.
  - The only difference is the default protection: `struct` is `public` and `class` is `private`.
- Each class type has a class invariant.
  - Constructors establish the class invariant.
  - Public member functions maintain the class invariant.
- An object is needed to access members.
  - Use `->` if the object is referred through a pointer, or `.` otherwise.
  - The object becomes an implicit argument to the member functions, though you can refer to it by `this`.
- The compiler verifies constness and protection labels at compile-time.
class netlist {
    typedef std::list<gate *> gate_list;
    gate_list gates_;  
    ...  
    public:
    void display(std::ostream &os) const;
}; // class netlist

▷ How to call **gate::display** through iterators?

```cpp
void netlist::display(std::ostream &os) const {
    for (gate_list::const_iterator it = gates_.begin();
         it != gates_.end(); ++it) {
        it->display(os);  // Wrong! it-> returns gate **
    }
}
```

▷ The correct one

```cpp
void netlist::display(std::ostream &os) const {
    for (gate_list::const_iterator it = gates_.begin();
         it != gates_.end(); ++it) {
        (*it)->display(os);  // Correct. *it returns gate *
    }
}
```
Example I

class netlist {
    typedef std::list<gate *> gate_list;
    gate_list gates_
    ...

public:
    void display(std::ostream &os) const;
}; // class netlist

▶ How to call \texttt{gate::display} through iterators?

```cpp
void netlist::display(std::ostream &os) const {
    for (gate_list::const_iterator it = gates_.begin();
        it != gates_.end(); ++it) {
    it->display(os); // Wrong! it-> returns gate **
    }
}

▶ The correct one

```cpp
void netlist::display(std::ostream &os) const {
    for (gate_list::const_iterator it = gates_.begin();
        it != gates_.end(); ++it) {
        (*it)->display(os); // Correct. *it returns gate *
    }
}
Object Composition

- Compose objects from others by
  - Include child objects as data members of parent objects
  - Keep pointers to objects as data members
  - Inherit from a base type and access the base object as an anonymous data member

- Ctors and dtors ensure all the objects are constructed and destroyed properly.
- The compiler will synthesize default ctor, copy ctor, dtor, and `operator=`
  - The semantics are member-wise.
- We can implement them if the ones synthesized by the compiler are not correct, as well as other ctors.
  - In such case, the compiler will still generate some code to make our life easier.
Object Composition

- Compose objects from others by
  - Include child objects as data members of parent objects
  - Keep pointers to objects as data members
  - Inherit from a base type and access the base object as an anonymous data member

- Ctors and dtors ensure all the objects are constructed and destroyed properly.
  - The compiler will synthesize default ctor, copy ctor, dtor, and operator=.
    - The semantics are member-wise.
  - We can implement them if the ones synthesized by the compiler are not correct, as well as other ctors.
    - In such case, the compiler will still generate some code to make our life easier.
Object Composition

- Compose objects from others by
  - Include child objects as data members of parent objects
  - Keep pointers to objects as data members
  - Inherit from a base type and access the base object as an anonymous data member

- Ctors and dtors ensure all the objects are constructed and destroyed properly.
- The compiler will synthesize default ctor, copy ctor, dtor, and \texttt{operator\textbar{}}.
  - The semantics are member-wise.
- We can implement them if the ones synthesized by the compiler are not correct, as well as other ctors.
  - In such case, the compiler will still generate some code to make our life easier.
Object Composition

- Compose objects from others by
  - Include child objects as data members of parent objects
  - Keep pointers to objects as data members
  - Inherit from a base type and access the base object as an anonymous data member

- Ctors and dtors ensure all the objects are constructed and destroyed properly.

- The compiler will synthesize default ctor, copy ctor, dtor, and \texttt{operator=}.
  - The semantics are member-wise.

- We can implement them if the ones synthesized by the compiler are not correct, as well as other ctors.
  - In such case, the compiler will still generate some code to make our life easier.
Data Member Construction and Destruction

- Members are constructed before the body of a ctor.
  - They are constructed in the order they appear in the class definition.
    - Base objects are constructed before all explicit members.
  - We can specify how the members are constructed using the initializer list.
  - The compiler will default initialize all the members whose constructions are not specified.

- Members are destroyed after the body of the dtor.
  - In the reverse order as they are constructed
  - The compiler will destroy them automatically.
Data Member Construction and Destruction

- Members are constructed before the body of a ctor.
  - They are constructed in the order they appear in the class definition.
    - Base objects are constructed before all explicit members.
  - We can specify how the members are constructed using the initializer list.
  - The compiler will default initialize all the members whose constructions are not specified.

- Members are destroyed after the body of the dtor.
  - In the reverse order as they are constructed.
  - The compiler will destroy them automatically.
Example II

class B {
    B(const B &);
};
class A {
    B member;
};
void some_function() {
    A a; // call synthesized default ctor
    A b = a; // cannot synthesize copy ctor for A
}

▶ You can make it work by defining a copy ctor for A.
    class A {
        B member;
    public:
        A(const A &) {}
    }

Example II

class B {
    B(const B &);
};
class A {
    B member;
};

void some_function() {
    A a; // call synthesized default ctor

    A b = a; // cannot synthesize copy ctor for A
}

▶ You can make it work by defining a copy ctor for A.

class A {
    B member;
    public:
        A(const A &) {}
};
Example III

class B {
    ~B();
};
class A {
    B member;
};
void some_function() {
    A a;

    // cannot synthesize dtor for A
}

▶ There is no way to make the above code working by changing A (without removing member).
class B {
    ~B();
};
class A {
    B member;
};
void some_function() {
    A a;
    // cannot synthesize dtor for A
}

- There is no way to make the above code working by changing A (without removing member).
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Exception and Exception Safety

- Exception allows us to indicate errors during construction.
- Exception safety: ensure proper program behavior when exception happens
  - There are many levels of exception safety.
- We are concerned of the minimum level – no leakage
  - All the objects that are constructed successfully should be destroyed eventually.
- The compiler will take care of most cases by
  - Stack unwinding: local objects are guaranteed to be destroyed
  - All or None construction: if a member construction fails, all the members that are already constructed will be destroyed, and the construction of the parent object fails
  - Assumption: dtors NEVER throw
Exception and Exception Safety

- Exception allows us to indicate errors during construction.
- Exception safety: ensure proper program behavior when exception happens
  - There are many levels of exception safety.
- We are concerned of the minimum level – no leakage
  - All the objects that are constructed successfully should be destroyed eventually.
- The compiler will take care of most cases by
  - Stack unwinding: local objects are guaranteed to be destroyed
  - All or None construction: if a member construction fails, all the members that are already constructed will be destroyed, and the construction of the parent object fails
  - Assumption: dtors NEVER throw
Exception and Exception Safety

- Exception allows us to indicate errors during construction.
- Exception safety: ensure proper program behavior when exception happens
  - There are many levels of exception safety.
- We are concerned of the minimum level – no leakage
  - All the objects that are constructed successfully should be destroyed eventually.
- The compiler will take care of most cases by
  - Stack unwinding: local objects are guaranteed to be destroyed
  - All or None construction: if a member construction fails, all the members that are already constructed will be destroyed, and the construction of the parent object fails
  - Assumption: dtors NEVER throw
Exception and Exception Safety

- Exception allows us to indicate errors during construction.
- Exception safety: ensure proper program behavior when exception happens
  - There are many levels of exception safety.
- We are concerned of the minimum level – no leakage
  - All the objects that are constructed successfully should be destroyed eventually.
- The compiler will take care of most cases by
  - Stack unwinding: local objects are guaranteed to be destroyed
  - All or None construction: if a member construction fails, all the members that are already constructed will be destroyed, and the construction of the parent object fails
  - Assumption: dtors NEVER throw
template <class T>
class unique_ptr {
    unique_ptr(const unique_ptr<T> &); // no copy
    unique_ptr &operator=(const unique_ptr<T> &); // no assignment
    T *p_;
public:
    explicit unique_ptr(T *p) : p_(p) {}
    ~unique_ptr() {delete p_;}
    T *get() const {return p_;}
}; // class unique_ptr<T>

void may_throw(int *p);
void test() {
    unique_ptr<int> p(new int(10));
    may_throw(p.get());
}

- Stack unwinding ensures \( p \) to be destroyed when \texttt{may\_throw} throws an exception.
  - So the \texttt{int} object created on the heap will be deleted.
Example II: the Classes

```cpp
struct will_not_throw {
    will_not_throw() {std::cout << "ctor of will_not_throw" << std::endl;}
    ~will_not_throw() {std::cout << "dtor of will_not_throw" << std::endl;}
}; // struct will_not_throw

class ctor_throw {
    will_not_throw wnt_;  //ctor of will not throw

public:
    ctor_throw() {
        std::cout << "ctor of ctor_throw" << std::endl;
        throw std::runtime_error("from ctor of ctor_throw");
    }
    ~ctor_throw() {std::cout << "dtor of ctor_throw" << std::endl;}
}; // class ctor_throw
```
Example II: the Output

class ctor_throw {
    will_not_throw wnt_;  
    ...
}; // class ctor_throw

void test() {
   try {
      ctor_throw ct;
   }
   catch (std::exception &e) {
      std::cout << e.what() << std::endl;
   }
}

- The output
  - ctor of will_not_throw
  - ctor of ctor_throw
  - dtor of will_not_throw
  - from ctor of ctor_throw

- Dtor of ctor_throw is not called.
  - That's a desired behavior since the object ct is not constructed successfully.
Example II: the Output

class ctor_throw {
    will_not_throw wnt_;
    ...
}; // class ctor_throw
void test() {
    try {
        ctor_throw ct;
    }
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}

-The output
ctor of will_not_throw
ctor of ctor_throw
dtor of will_not_throw
from ctor of ctor_throw

-Dtor of ctor_throw is not called.
    ▶ That's a desired behavior since the object ct is not constructed successfully.
Example III: the Classes

```cpp
struct will_throw {
    will_throw() {
        std::cout << "ctor of will_throw" << std::endl;
        throw std::runtime_error("from ctor of will_throw");
    }
    ~will_throw() {std::cout << "dtor of will_throw" << std::endl;}
}; // struct will_throw

class member_throw {
    will_not_throw wnt_;  
    will_throw wt_; 

public:
    member_throw() {std::cout << "ctor of member_throw" << std::endl;}
    ~member_throw() {std::cout << "dtor of member_throw" << std::endl;}
}; // class member_throw
```
Example III: the Output

class member_throw {
    will_not_throw wnt_;
    will_throw wt_;  
    ...
}; // class member_throw
void test() {
    try {
        member_throw mt;
    } 
    catch (std::exception &e) { 
        std::cout << e.what() << std::endl;
    }
}

▶ The output
ctor of will_not_throw
ctor of will_throw
dtor of will_not_throw
from ctor of will_throw

▶ Body of ctor of member_throw is not called.
▶ Neither dtor of will_throw nor dtor of member_throw is called.
class member_throw {
    will_not_throw wnt_;  
    will_throw wt_;  
    ...
}; // class member_throw

void test() {
    try {
        member_throw mt;
    }
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}

- The output
  ctor of will_not_throw
  ctor of will_throw
  dtor of will_not_throw
  from ctor of will_throw

- Body of ctor of member_throw is not called.

- Neither dtor of will_throw nor dtor of member_throw is called.
Example III: the Output

class member_throw {
    will_not_throw wnt_;  
    will_throw wt_; 
    ...
}; // class member_throw

void test() {
    try {
        member_throw mt;
    } 
    catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}

- The output
  ctor of will_not_throw
  ctor of will_throw
  dtor of will_not_throw
  from ctor of will_throw

- Body of ctor of member_throw is not called.
- Neither dtor of will_throw nor dtor of member_throw is called.
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Polymorphism

- A single piece of code behaves differently under different circumstances.
  - Enable code reusing without modification
  - Object interfaces: constraints objects should satisfy when used for the code
  - Programming against interface: the code depends on the object interfaces but not their exact types

- Runtime polymorphism
  - Interfaces are defined as virtual functions in base classes.
  - The code should leverage dynamic binding for polymorphism.
  - Good for decoupling classes as exact types are hidden
A single piece of code behaves differently under different circumstances.

- Enable code reusing without modification
- Object interfaces: constraints objects should satisfy when used for the code
- Programming against interface: the code depends on the object interfaces but not their exact types

Runtime polymorphism

- Interfaces are defined as `virtual` functions in base classes.
- The code should leverage dynamic binding for polymorphism.
- Good for decoupling classes as exact types are hidden
Dynamic Binding

- Calling a **virtual** function through a pointer or a reference
  - Not through a fully-qualified name
  - Not in ctors or dtor
  - The **virtual** function could also be called through the implicit **this** pointer, i.e. in member functions.
Dynamic Binding

- Calling a \textit{virtual} function through a pointer or a reference
- Not through a fully-qualified name
- Not in ctors or dtor
- The \textit{virtual} function could also be called through the implicit \texttt{this} pointer, i.e. in member functions.
Dynamic Binding

- Calling a \texttt{virtual} function through a pointer or a reference
- Not through a fully-qualified name
- Not in ctors or dtor
- The \texttt{virtual} function could also be called through the implicit \texttt{this} pointer, i.e. in member functions.
Dynamic Binding

- Calling a `virtual` function through a pointer or a reference
- Not through a fully-qualified name
- Not in ctors or dtor
- The `virtual` function could also be called through the implicit `this` pointer, i.e. in member functions.
struct base {
    virtual void vfun(std::string info) {
        std::cout << "calling base::vfun from " << info << std::endl;
    }
    base() {vfun("ctor of base");}
    virtual ~base() {vfun("dtor of base");}
    void usual_member() {vfun("base::usual_member");}
}; // struct base

struct derived : public base {
    void vfun(std::string info) {
        std::cout << "calling derived::vfun from " << info << std::endl;
    }
    derived() {vfun("ctor of derived");}
    ~derived() {vfun("dtor of derived");}
}; // struct derived
Example I

1: calling base::vfun from ctor of base
2: calling derived::vfun from ctor of derived
3: calling derived::vfun from base::usual_member
4: calling derived::vfun from test_obj
5: calling base::vfun from test_obj
6: calling derived::vfun from dtor of derived
7: calling base::vfun from dtor of base

void test_obj() {
  derived d;
  // body of ctor of d’s base  line 1, static binding in ctor of base
  // body of ctor of d  line 2, static binding in ctor of derived

  d.usual_member();  // line 3, dynamic binding in member function
  d.vfun("test_obj");  // line 4, static binding
  d.base::vfun("test_obj");  // line 5, static binding via fully-qualified
    // name

  // body of dtor of d  line 6, static binding in dtor of derived
  // body of dtor of d’s base  line 7, static binding in dtor of base
}
Example II

1: calling base::vfun from ctor of base
2: calling derived::vfun from ctor of derived
3: calling derived::vfun from base::usual_member
4: calling derived::vfun from test_ptr
5: calling base::vfun from test_ptr
6: calling derived::vfun from dtor of derived
7: calling base::vfun from dtor of base

void test_ptr() {
    derived *temp = (derived *)::operator new(sizeof(derived));
    new (temp) derived; // *temp is constructed
    // body of ctor of *temp’s base line 1, static binding in ctor of base
    // body of ctor of *temp line 2, static binding in ctor of derived
    base *p = temp;
    p->usual_member(); // line 3, dynamic binding in member function
    p->vfun("test_ptr"); // line 4, dynamic binding via base pointer
    p->base::vfun("test_ptr"); // line 5, static binding via fully-qualified
    // name
    p->~base(); // dynamic binding, dtor of derived is called
    // body of dtor of *temp line 6, static binding in dtor of derived
    // body of dtor of *temp’s base line 7, static binding in dtor of base
    ::operator delete(p);
}
Outline

Software Development

Types and Operators

Class Design and Object Composition

Exception and Exception Safety

Polymorphism

Smart Pointers
Garbage Collection

- There are cases where it is not possible or not convenient for one to decide the exact place objects allocated from heap should be deleted – garbage collection (GC) is a must.
  - Programmers decide when objects should be created on the heap.
  - The compiler/runtime library decide when they should be deleted.

- Typical garbage collection algorithms
  - Reachability analysis: reclaim memory when necessary
  - Reference counting: delete an object immediately when it’s no longer in use

- Both have their advantages and disadvantages.
Garbage Collection

- There are cases where it is not possible or not convenient for one to decide the exact place objects allocated from heap should be deleted – garbage collection (GC) is a must.
  - Programmers decide when objects should be created on the heap.
  - The compiler/runtime library decide when they should be deleted.
- Typical garbage collection algorithms
  - Reachability analysis: reclaim memory when necessary
  - Reference counting: delete an object immediately when it’s no longer in use
- Both have their advantages and disadvantages.
There are cases where it is not possible or not convenient for one to decide the exact place objects allocated from heap should be deleted – garbage collection (GC) is a must.

- Programmers decide when objects should be created on the heap.
- The compiler/runtime library decide when they should be deleted.

Typical garbage collection algorithms

- Reachability analysis: reclaim memory when necessary
- Reference counting: delete an object immediately when it’s no longer in use

Both have their advantages and disadvantages.
Smart Pointers

- Smart pointers: class types that can be used as pointers but are smarter than built-in pointers.
  - They will delete an object when it is no longer in use.
  - Provide GC to C++ programs

- `std::unique_ptr`: keep a non-copyable pointer to an object
  - It is straightforward to determine when the object should be deleted – in its dtor.

- `std::shared_ptr`: allow to share a pointer to an object
  - Reference counting is used to determine when the object should be deleted.
  - Limitation: at runtime, if the smart pointers form a cycle, resources won’t be released properly.
Smart Pointers

- Smart pointers: class types that can be used as pointers but are smarter than built-in pointers.
  - They will delete an object when it is no longer in use.
  - Provide GC to C++ programs
- `std::unique_ptr`: keep a non-copyable pointer to an object
  - It is straightforward to determine when the object should be deleted – in its dtor.
- `std::shared_ptr`: allow to share a pointer to an object
  - Reference counting is used to determine when the object should be deleted.
  - Limitation: at runtime, if the smart pointers form a cycle, resources won’t be released properly.
Smart Pointers

- Smart pointers: class types that can be used as pointers but are smarter than built-in pointers.
  - They will delete an object when it is no longer in use.
  - Provide GC to C++ programs

- `std::unique_ptr`: keep a non-copyable pointer to an object
  - It is straight-forward to determine when the object should be deleted – in its dtor.

- `std::shared_ptr`: allow to share a pointer to an object
  - Reference counting is used to determine when the object should be deleted.
  - Limitation: at runtime, if the smart pointers form a cycle, resources won’t be released properly.
```cpp
struct A {
    shared_ptr<B> b_of_A;
    A() {std::cout << "ctor of A" << std::endl;}
    ~A() {std::cout << "dtor of A" << std::endl;}
}; // struct A

struct B {
    shared_ptr<A> a_of_B;
    B() {std::cout << "ctor of B" << std::endl;}
    ~B() {std::cout << "dtor of B" << std::endl;}
}; // struct B

void test1() {
    shared_ptr<A> pa(new A); // (A, 1) output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    pa->b_of_A = pb; // (A, 1), (B, 2)
    pb->a_of_B = pa; // (A, 2), (B, 2)

    // pb is destroyed: (A, 2), (B, 1)
    // pa is destroyed: (A, 1), (B, 1)
}
```
Example II

```cpp
void test2() {
    shared_ptr<A> pa(new A); // (A, 1) output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    pa->b_of_A = pb; // (A, 1), (B, 2)

    // pb is destroyed: (A, 1), (B, 1)
    // pa is destroyed: (A, 0), (B, 1)
    // A is destroyed output: dtor of A
    // b_of_A is destroyed: (B, 0)
    // B is destroyed output: dtor of B
    // a_of_B is destroyed
}
```
void test3() {
    shared_ptr<A> pa(new A); // (A, 1) output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    pb->a_of_B = pa; // (A, 2), (B, 1)

    // pb is destroyed: (A, 2), (B, 0)
    // B is destroyed output: dtor of B
    // a_of_B is destroyed (A, 1)
    // pa is destroyed: (A, 0)
    // A is destroyed output: dtor of A
    // b_of_A is destroyed
}

Example IV

```cpp
void test4() {
    shared_ptr<A> pa(new A); // (A, 1) output: ctor of A
    shared_ptr<B> pb(new B); // (A, 1), (B, 1) output: ctor of B

    // pb is destroyed: (A, 1), (B, 0)
    // B is destroyed output: dtor of B
    // a_of_B is destroyed
    // pa is destroyed: (A, 0)
    // A is destroyed output: dtor of A
    // b_of_A is destroyed
}
```
While the compiler will help the programmers to correct typos, syntactic errors, and mismatched types, the programmers need to understand what the programs they have written mean to the compiler.