C++ Programming
~ Object-oriented Programming ~

Prepared for Ingenix, Inc.

FRESH SOURCES

Object-oriented Programming

- Quantifies “is-a” relationships between classes
- Increases code quality through code reuse
- Enhances power of expression and code maintainability through dynamic function binding (polymorphism)

Agenda

- Inheritance
- Protected members
- Virtual functions
- Virtual destructors
- Abstract classes
- Interfaces
- Exceptions
- Volume 1: 14, 15; Volume 2: 1, 9

An Employee Class

```
class Employee
{
    string name;
    double rate;
    double timeWorked;

public:
    Employee(const string& ename, double erate)
        : name(ename)
    {
        rate = erate;
    }

    string getName() const        {return name;}
    double getRate() const        {return rate;}
    double getTimeWorked() const  {return timeWorked;}

    void recordTime(double etime) {timeWorked = etime;}
    double computePay() const;
};
```

```
double Employee::computePay() const
{
    const double& hours = timeWorked;
    if (hours > 40)
        return 40*rate + (hours - 40)*rate*1.5;
    else
        return rate * hours;
}


int main()
{
    using namespace std;
    Employee e("John Hourly",16.50);
    e.recordTime(52.0);
    cout << e.getName() << " gets "
         << e.computePay() << endl;
}
```

John Hourly gets 957.00

Salaried Employees

- Suppose we now want to process *salaried* employees
- Only the `computePay` method changes
- We don’t want to have to repeat the other code
- How can we reuse/extend Employee?

The SalariedEmployee Class

```
class SalariedEmployee : public Employee
{
public:
    SalariedEmployee(const string&, double);
    double computePay() const;
};

SalariedEmployee::SalariedEmployee(const string& ename,
                                   double erate)
                 : Employee(ename, erate)
{}

double SalariedEmployee::computePay() const
{
    return getRate() * getTimeWorked();
}
```

```
int main()
{
    using namespace std;
    SalariedEmployee e("Jane Salaried",1125.00);
    e.recordTime(1.0);
    cout << e.getName() << " gets "
         << e.computePay() << endl;
}
```

Jane Salaried gets 1125

Inheritance

- (Public) inheritance implements an "is-a" relationship
  - you rarely use more restrictive inheritance
  - access of inherited members doesn't change
  - private members of a base class are *not* accessible in derived class methods
  - public inheritance is the default for **struct**
- A SalariedEmployee object inherits all data and methods from Employee
  - because it "is-a" Employee
  - but it overrides `computePay`

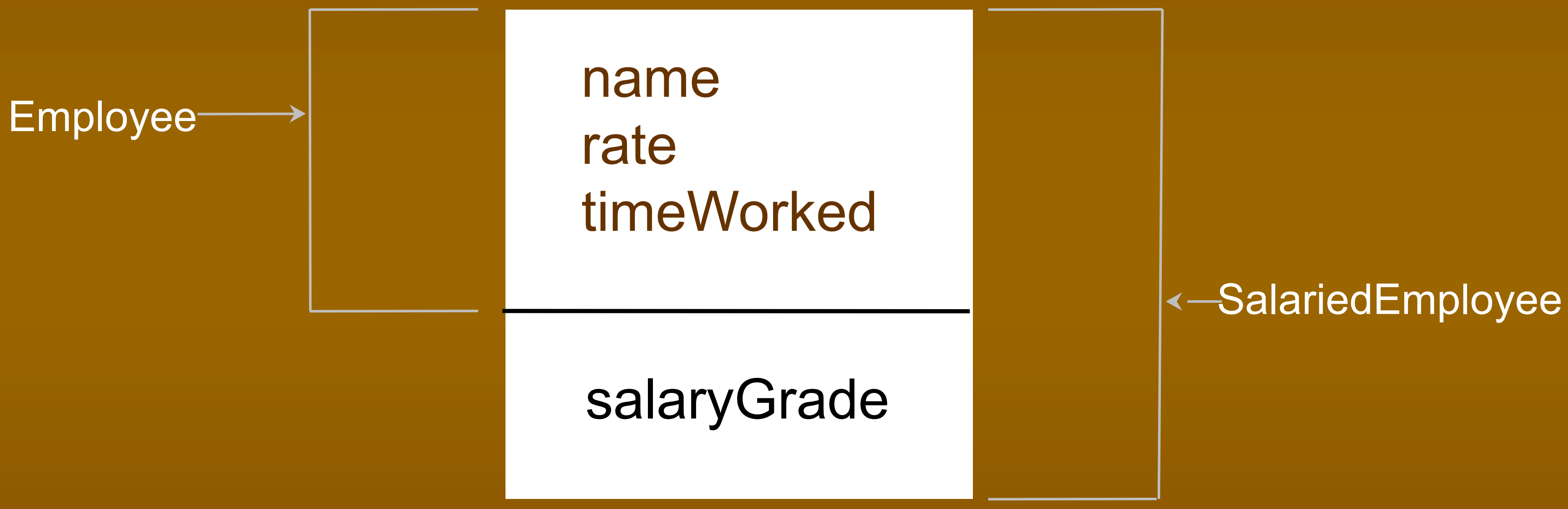
Terminology

- A *base class* is a class you extend by inheritance
  - also called a *superclass*
- A *derived class* is a class that inherits from another
  - also called a *subclass*
  - you can add new data members
  - you can add/replace member functions
- Member functions are also called *methods*

Adding New Members

```
class SalariedEmployee : public Employee
{
    int salaryGrade;              // a new member

public:
    // ...
    void setSalaryGrade(int g) {salaryGrade = g;}
    int getSalaryGrade()      {return salaryGrade;}
};
```

Base Class Subobjects



Object Initialization
The Real Story

- The base class constructor(s) run(s) first
  - in declaration order, if multiple inheritance
  - you pass arguments to base class constructors *only* through the member initializer list
  - if the base class has a default constructor, no explicit initializer is necessary
- Then any member objects are initialized
  - in declaration order
- Then the derived class constructor runs
- Destruction is the reverse of this process

```cpp
#include <iostream>
using namespace std;

struct A {
    A()  {cout << "A::A()\n";}
    ~A() {cout << "A::~A()\n";}
};

struct B {
    B()  {cout << "B::B()\n";}
    ~B() {cout << "B::~B()\n";}
};

struct C : A {
    C()  {cout << "C::C()\n";}
    ~C() {cout << "C::~C()\n";}
    B b;
};

int main() {
    C c;
}
```

```
A::A()
B::B()
C::C()
C::~C()
B::~B()
A::~A()
```

```
// Using Initializers
#include <iostream>
using namespace std;

struct A
{
    A(int i)  {cout << "A::A(" << i << ")\n";}
    ~A() {cout << "A::~A()\n";}
};

struct B
{
    B(int j)  {cout << "B::B(" << j << ")\n";}
    ~B() {cout << "B::~B()\n";}
};

struct C : A
{
    C(int i, int j) : A(i), b(j)
    {
        cout << "C::C(" << i << ',' << j << ")\n";
    }
    ~C() {cout << "C::~C()\n";}
    B b;
};

int main()
{
    C c(1,2);
}
```

```
A::A(1)
B::B(2)
C::C(1,2)
C::~C()
B::~B()
A::~A()
```

Exercise

- Create two classes called **Traveler** and **Pager** without default constructors, but with constructors that take an argument of type **string**, which they simply copy to an internal **string** variable. Now derive a class named **BusinessTraveler** from **Traveler** and give it a member object of type **Pager**. Write the a default constructor, a constructor that takes two **string** arguments, and a stream inserter.

Protected Members

- There is a third access specifier - `protected`
  - protected members are available to derived classes but not to other clients

Protected Employee Members

```
class Employee
{
protected:
    string name;
    double rate;
    double timeWorked;

public:
    Employee(const string& ename, double erate)
        : name(ename)
    {
        rate = erate;
    }

    // (Access functions no longer necessary)
    void recordTime(double etime) {timeWorked = etime;}
    double computePay() const;
};
```

Protected Employee Members

- `SalariedEmployee::computePay` is now simpler

```
double SalariedEmployee::computePay() const
{
    return rate * timeWorked;   // access protected
                                // members
}
```

Name Hiding "Gotcha"

- Beware when "overriding" functions in derived classes
- Example: Hide.cpp

Name Lookup Rules

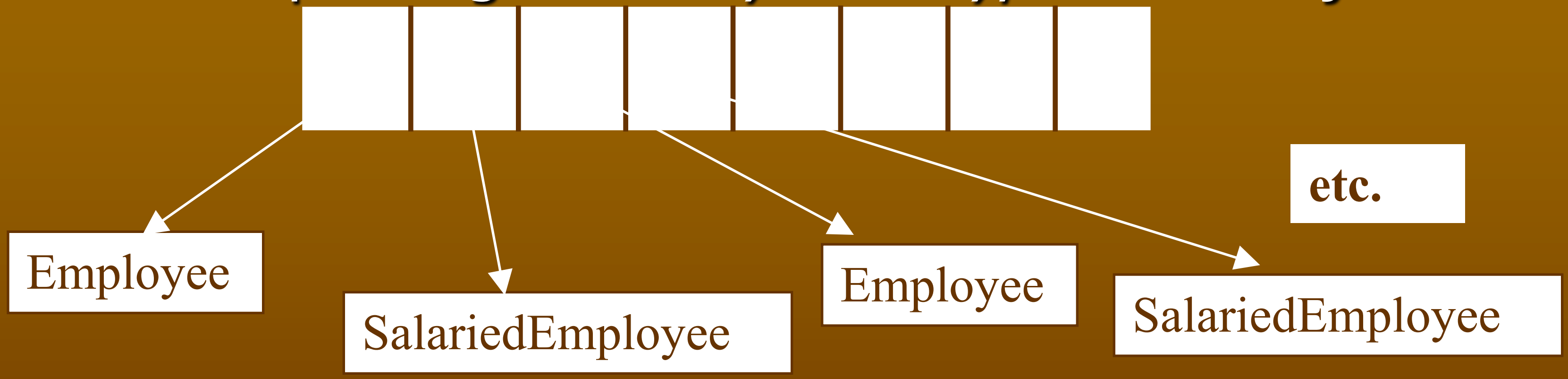
- 1. Find a scope for the name
  - A class constitutes a scope
  - A derived class scope is "nested" in the base class's scope
- 2. Perform overload resolution in that scope
  - Pick unambiguous "best fit"
- 3. Check access permission
- Examples: Lookup1-3.cpp

Upcasting

- An `Employee*` can hold a `SalariedEmployee*`
- Because of the "is-a" relationship
- A SalariedEmployee can take the place of an Employee object
  - unless the code assumes hourly employee behavior
- Hence, an array of `Employee*` can hold pointers to a mixture of the two types

The Goal

- To treat all objects as base objects
  - via a pointer-to-base
- But to have their behavior vary automatically
  - depending on the *dynamic type* of the object



Heterogeneous Collections

```
int main()
{
    using namespace std;
    Employee e("John Hourly",16.50);
    e.recordTime(52.0);
    SalariedEmployee e2("Jane Salaried",1125.00);
    e2.recordTime(1.0);
    Employee* elist[] = {&e, &e2};
    int nemp = sizeof elist / sizeof elist[0];

    for (int i = 0; i < nemp; ++i)
        cout << elist[i]->getName() << " gets "
             << elist[i]->computePay() << endl;
}

John Hourly gets 957
Jane Salaried gets 1125    // beware a subtle bug!
```

Which `computePay`?

```
// After inserting trace statements in
// Employee::computePay and
// SalariedEmployee::computePay

Employee::computePay
John Hourly gets 957
Employee::computePay      // Oops!
Jane Salaried gets 1125
```

Traditional Solution

- The traditional (non-OO) approach to polymorphism is a `switch` statement

```
void computePay(Employee* emp)
{
    switch(emp->type())     // type tag required
    {
    case EMPLOYEE:
        computeHourlyPay(emp);
        break;
    case SALARIED:
        computeSalariedPay(emp);
      break;
    // ...
    }
}
```

Problems with the Traditional Method

- The programmer is completely responsible for the function selection process.
- Each time a new type of customer is added, each switch statement in the system will need to be updated
- The code quickly grows and becomes complicated.

Function Binding

- Determines the code that executes for a functions call
- *Static binding* occurs at compile time
  - what we're used to
- *Dynamic binding* occurs at run time
  - what Java and SmallTalk folks are used to
  - what we want here
  - determined by the dynamic type of object pointed to

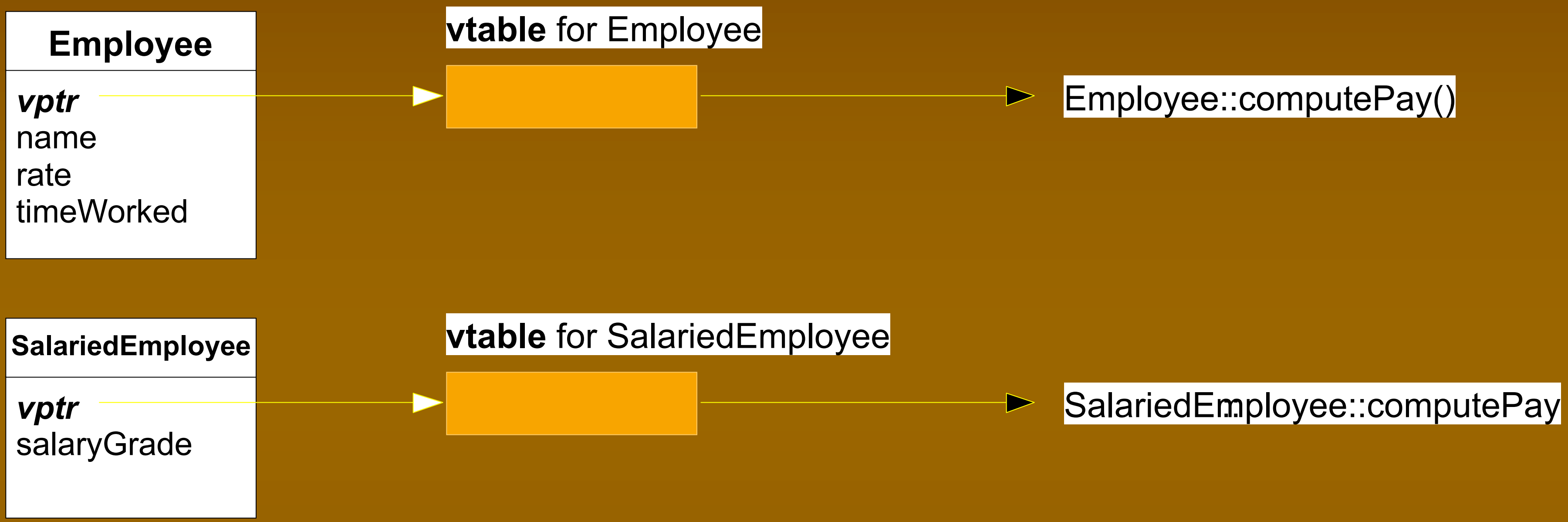
Polymorphism

- G[r]eek for "many forms"
- "One interface, many implementations"
- Overloading is a form of static polymorphism
- We want run time polymorphism
  - i.e., dynamic binding
  - what is usually meant by "polymorphism"
- Achieved in C++ via virtual functions

A virtual `computePay`

```
class Employee
{
    // ...
public:
    // ...
    virtual double computePay() const;
};

Employee::computePay
John Hourly gets 957
SalariedEmployee::computePay    // Right!
Jane Salaried gets 1125
```

How Virtual Functions Work



- Each class has a *vtable* (pointers to its virtual functions)
- Each object has a *vptr* (points to its class's vtable)

Advantages of Dynamic Binding

- Client code can just deal with the base type (e.g., `Employee*`)
- Behavior varies *transparently* according to an object’s dynamic type
- Client code remains unchanged when new derived types are created!
- No “ripple effect” for maintainers

Derived Destructors

- Recall that base class destructors are called automatically when a derived object dies:

```
struct B
{
    ~B() {std::cout << "~B\n";}
};

struct D : B      // public by default
{
    ~D() {std::cout << "~D\n";}
};

int main()
{
    D d;
}

~D
~B
```

Deleting via a Pointer-to-Base

```
int main()
{
    B* pb = new D;
    delete pb;
}

~B             // Oops! Derived part not destoyed!
```

Virtual Destructors

- Needed when deleting via a pointer-to-base

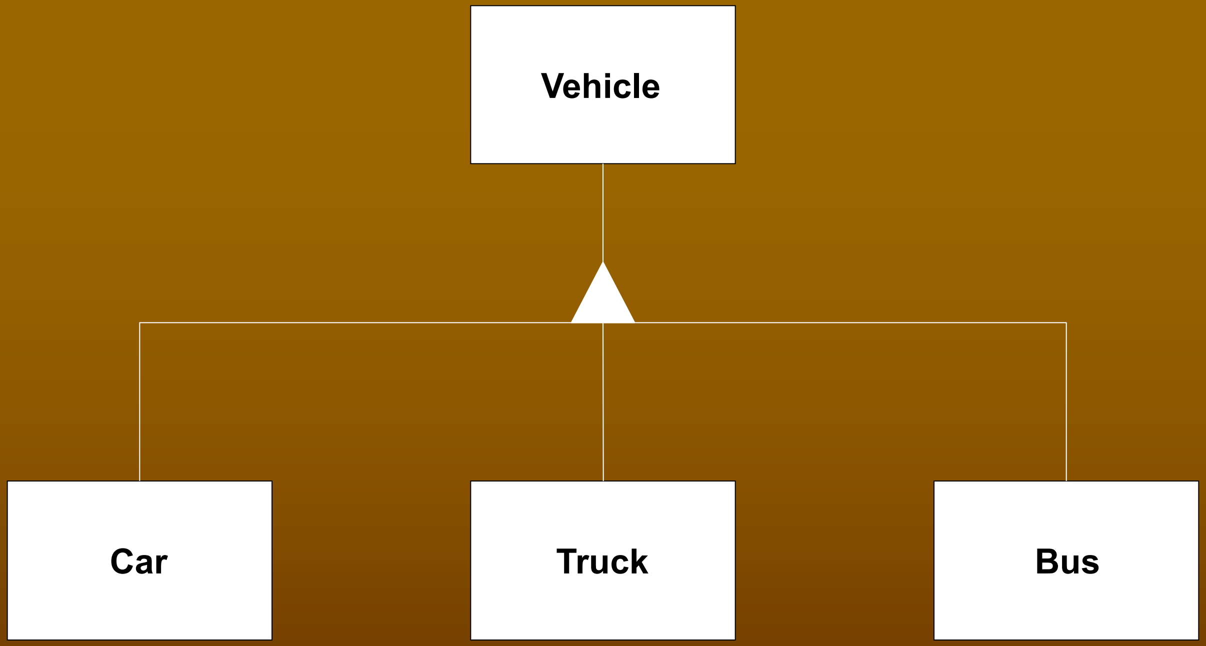
```
struct B
{
    virtual ~B() {std::cout << "~B\n";}
};

int main()
{
    B* pb = new D;
    delete pb;
}


~D                    // Fixed!
~B
```

Virtual Destructors

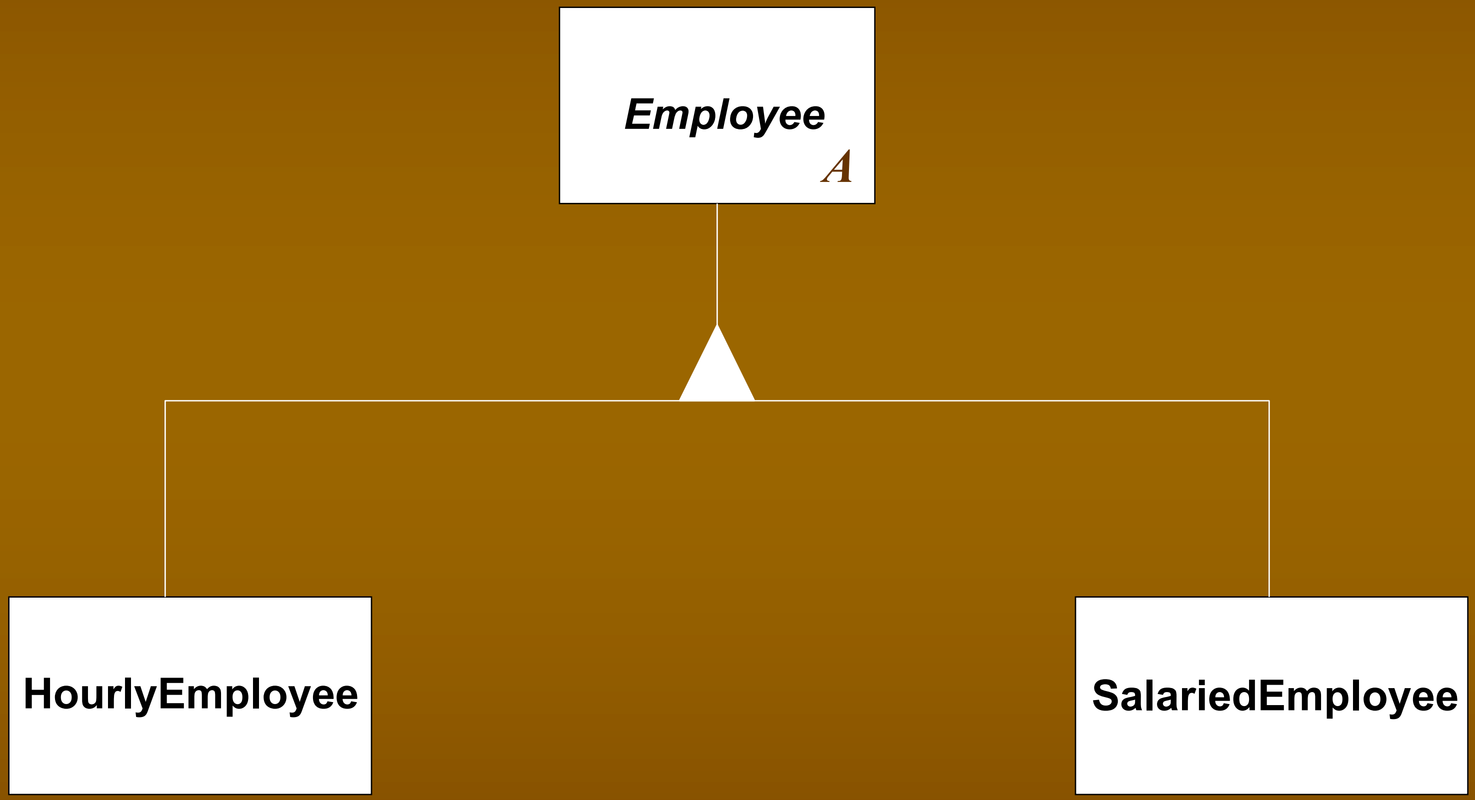
- Destructors can be declared virtual
  - necessary when a base class pointer or reference refers to a derived class object
  - if the destructor is not declared virtual, only the base class destructor is called
  - this may cause a memory leak
- Rule: A class that contains a virtual function should also declare a virtual destructor

Abstract Classes

- Sometimes a base class is just a conceptual entity
  - a category, or umbrella for related classes
  - you won't instantiate any objects of that type



A Better Employee Hierarchy



Pure Virtual Functions

- Abstract classes usually have abstract methods:
  - "Place holder" member functions to be overridden in derived classes
  - Don't need an implementation in the base class
- The presence of such a *pure virtual* function makes its class abstract
- Append "= 0" to the function's declaration

An Abstract Employee Class

```
class Employee
{
public:
    Employee(const string& ename, double erate)
        : name(ename)
    {
        rate = erate;
    }

    string getName() const {return name;}
    double getRate() const {return rate;}

    void recordTime(double etime) {timeWorked = etime;}
    virtual double computePay() const = 0; // pure virt.
    virtual ~Employee() {}      // !!!

protected:
    string name;
    double rate;
    double timeWorked;
};
```

The HourlyEmployee Class

```
class HourlyEmployee : public Employee
{
public:
    HourlyEmployee(const string& ename, double erate)
        : Employee(ename, erate)
    {}
    double computePay() const;
};

double HourlyEmployee::computePay() const
{
    cout << "HourlyEmployee::computePay()\n";
    const double& hours = timeWorked;
    if (hours > 40)
        return 40*rate + (hours - 40)*rate*1.5;
    else
        return rate * hours;
}
```

Interfaces

- An interface is a set of function specifications
- Simply defines a group of functions
  - Non-static functions (meant to be applied to objects of types that *implement* the interface)
  - An interface is also called a (*formal) type*
- In C++, an interface is a class containing only pure virtual functions (no bodies)
- Example: interfaces.cpp

OO Programming - Summary

- Inheritance supports:
  - an "is-a" relationship between classes
  - consolidation of code (just define what changes)
  - class specialization
- Upcasting treats a derived object as a base object
- Virtual Functions implement dynamic binding
- A class with a virtual function should have a virtual destructor also
- Abstract classes represent a concept, and cannot be instantiated

Exercise

- Define an abstract class named Vehicle with an id number as a data member, and two pure virtual functions, `stop()` and `go()`. Derive two classes from Vehicle, Car and Truck. Override `stop()` and `go()` in the derived classes to print a statement that identifies what they're doing (e.g., `Truck::stop()` might say "Stopping Truck #2"). Define a destructor in the derived classes that just announces itself. Create an array of pointers to Vehicle in `main()` that holds a Car and a Truck on the heap, then iterate through the array calling stop() and go() to verify that dynamic binding is taking place. Now define a virtual destructor in Vehicle and test again.

Exceptions

- The Philosophy of Exceptions
- The Mechanics of Exceptions
- Exceptions and Resource Management
- Exception Specifications
- Exception Safety

Pop Quiz!

- What does printf( ) return?

Leading Question

- When was the last time you checked the return value from printf( )?

Error Handling via Return Codes

- You don’t always check them
  - (Did I make my point? :-)
- If you do, the extra code clutter obscures the readability of your program logic
- Even if no errors occur, you’re always wasting cycles checking for them
  - applies to other error-code schemes as well
    - e.g., errno

The Philosophy of Exceptions

- You can’t ignore them
  - Handle them or die!
- Error handling code is localized
  - Code is more readable
- Your code runs faster!
  - If no errors occur
- Yes, there is a space penalty
  - But it’s minimal and worth it!

```
// Illustrates handling "deep errors"
#include <iostream>
using namespace std;

void h()
{
    throw "h() has a problem";
}

void g()
{
    h();
    cout << "doing g..." << endl;
}

void f()
{
    g();
    cout << "doing f..." << endl;
}
```

```
int main()
{
    try
    {
        f();
    }
    catch(const char* msg)
    {
        cerr << "Error: " << msg << endl;
    }

    cout << "back in main" << endl;
}

/* Output:
Error: h() has a problem
back in main
*/
```

Preliminary Details

- The purpose of a try-block is to place exception handlers (“catch-clauses”) into the execution stream
- The `throw` expression transfers control to an upstream handler
  - the nearest-enclosing “matching” handler
    - according to the type of exception thrown
  - so it can recover from the error

Pretty Good Idea #1

- Use exceptions to indicate *errors*
- For functions that can’t fulfill their specification
- Not for alternate returns under normal circumstances

Potential Problem

- What if local objects are created?
  - In f( ), g( ), say
- They may need their destructor called
- Not a problem

Stack Unwinding

- As execution backtracks up the call stack, local objects have their destructors called
- Allows for convenient resource deallocation
  - A key to exception safety

```cpp
#include <iostream>
using namespace std;

void h()
{
    Foo f3;
    throw "h() has a problem";
}

void g()
{
    Foo f2;
    h();
    cout << "doing g..." << endl;
}

void f()
{
    Foo f1;
    g();
    cout << "doing f..." << endl;
}
```

```
int main()
{
    try
    {
        f();
    }
    catch(const char* msg)
    {
        cerr << "Error: " << msg << endl;
    }

    cout << "back in main" << endl;
}

/* Output:
Foo
Foo
Foo
~Foo
~Foo
~Foo
Error: h() has a problem
back in main
*/
```

How to Throw Exceptions

- **`throw`** keyword
- Throw objects of user-defined classes
  - Can hold auxiliary information
  - Allows clear categorization of errors
- Use constructor syntax

```cpp
// Exception class
class MyError
{
    string msg;
public:
    MyError(const string& s) : msg(s) {}
    string what() {return msg;}
};

// ...
```

```
void h()
{
    throw MyError("h() has a problem");
}
```

```
int main()
{
    try
    {
        f();
    }
    catch(MyError& x)
    {
        cerr << "MyError: " << x.what() << endl;
    }

    // Control goes here ("termination semantics")
    cout << "back in main" << endl;
}
```

Catching Exceptions

- Execution backtracks until it finds a matching handler
- Exact type, or
- An accessible base class type
- Beware built-in types
  - rules are complicated; use classes!
  - string literals are **`const char*`**
    - (not caught via a **`char*`** catch parameter)
- Not all conversions apply!
  - Sufficient info not available at runtime!

If `D` derives from `B`…

- **`catch(B&)`** catches a **`B`** or a **`D`**
  - so order of handlers in code matters!
  - B must be an unambiguous, public base for D
- **`catch(B*)`** catches a **`B*`** or **`D*`**
- **`catch(void*)`** catches all pointer types

Order Matters!

- Handlers are tried in order of their appearance in the code
- Most specific handlers should appear first
- Derived class handlers should precede base class handlers
- catch(…), if present, should be last

Uncaught Exceptions

- If no handler is found, the library function `terminate()` is called
  - Which just calls `abort()`
- If you want to prevent termination:
  - Make sure all exceptions are caught!
- You can install your own *terminate handler*
  - With `set_terminate()`

What should `terminate` do?

- Log the error
- Tidy-up as needed (release global resources, if any)
- exit the program
- **`terminate`** *cannot:*
  - return
  - throw exceptions

set_terminate

```
#include <iostream>
#include <exception>   // for set_terminate()
#include <cstdlib>     // for exit()
using namespace std;

void handler()
{
    cout << "Renegade exception!\n";
    exit(1);
}

int main()
{
    void f();
    set_terminate(handler);

    try
    {
        f();
    }
```

```
    catch(long)
    {
        cerr << "caught a long" << endl;
    }
}

void f()
{
    throw "oops";    // Doesn't match a long
}

// Output:
Renegade exception!
```

`terminate()` is called when…

- A matching handler is not found, including when:
  - a constructor for a static object throws
  - An *exit handler* (from `atexit`) throws
- A destructor throws during stack unwinding
  - Only one exception at a time, thank you!
  - Destructors shouldn’t emit exceptions

How does all this really work?

- `throw` is *conceptually* like a function call
  - Takes the exception object as a "parameter"
- This special "function" backtracks up the program stack (the dynamic call chain)
  - Reading information placed there by *each function invocation*
    - Information placed in each "Stack Frame"
    - About each function's local objects and try blocks
- If no matching handler is found in a function, local objects' are destroyed and the search continues
  - Until a matching handler is found
  - Or terminate( ) is ultimately called

Space Overhead

```
struct C
{
   ~C(){}
};

void g();   // for all we know, g may throw

void f()
{
   C c;     // Destructor must be called
   g();
}
```

Compiler Exception Support

- Microsoft Visual C++ .NET (-GX)
  - 1,420 bytes vs. 2,069 bytes
- Borland C++ Builder 6.0 (-x-)
  - 813 bytes vs. 2,150 bytes

Runtime Overhead

- Two Types
  - Adding exception-related info to each stack frame
  - The work done during stack unwinding
    - This is *good overhead*, since you want things cleaned up
    - Following return-code paths the old-fashioned way has a cost too, you know!

The Zero-cost Model

- Adorning each stack frame with exception-related info can have a *runtime* cost
- Can be avoided
  - Offsets for objects with destructors can be computed once at compile time and stored outside the runtime stack
- GNU and Metrowerks compilers currently support this

Another Leading Question

Since exception objects originate in a different scope from where they're caught, how are they accessible in a handler?

Answer

- Exception objects are *temporaries*
  - A *copy* is thrown
    - Const-ness is stripped away (except for string literals)
  - Exceptions must be *copyable* and *destructible*
    - accessible in the context of the throw expression
- Catching by *value* creates an *additional copy*
  - And derived objects caught as a base are sliced
- Catch-by-pointer, is problematic (how to know whether you have to `delete` it)?

Pretty Good Idea #2

- Catch exceptions by reference.
- What about *const* reference?
  - A local stylistic concern
  - Const and volatile are ignored in finding a matching handler
  - You can modify the exception object as it moves up the stack
    - because the same object is re-thrown

Standard Exceptions

- Thrown by the Standard Library
- Hierarchy of *Logic* vs. *Runtime* Errors
- `exception` base class

Standard Exceptions

- **exception**
  - **logic_error** (client program error)
    - domain_error, invalid_argument, length_error, out_of_range
  - **runtime_error** (external error)
    - range_error, overflow_error, underflow_error
  - bad_alloc (memory failure)
  - bad_cast (bad dynamic_cast w/ref)
  - bad_exception (unexpected)
  - bad_typeid (typeid w/null)

```
try
{
    string s;
    cout << s.at(100) << endl;  // invalid arg
}
catch (logic_error& x)
{
    cout << "logic_error: " << x.what()
         << endl;
}
catch (runtime_error& x)
{
    cout << "runtime_error: " << x.what()
         << endl;
}
catch (exception& x)
{
    cout << "exception: " << x.what()
         << endl;
}
```

```
// Output:
logic_error: position beyond end of string
```

Using Standard Exceptions

```
#include <iostream>
#include <stdexcept>
#include <string>
using namespace std;

// Exception class (polymorphic because
// std::exception is)
struct MyError : runtime_error
{
   MyError(const string& msg)
   : runtime_error(msg){}
};
```

```
int main()
{
    try
    {
        f();
    }
    catch (MyError& x)
    {
        cerr << x.what() << endl;
    }
    catch (exception& x)
    {
        cerr << x.what() << endl;
    }
    catch (...) // catch-all
    {
        cerr << "Unknown error\n";
    }

    cout << "back in main" << endl;
}
```

```
// Using RTTI (a sometimes-useful trick):
int main()
{
    try
    {
        f();
    }
    catch(exception& x)
    {
        cerr << typeid(x).name() << ':'
             << x.what() << endl;
    }
    catch (...)      // catch-all
    {
        cerr << "Unknown error\n";
    }

    cout << "back in main" << endl;
}

MyError:h()has a problem
Back in main
```

Pretty Good Idea #3

- Throw objects of classes derived (ultimately, not necessarily directly) from **`std::exception`**
- (**`std::exception`** does not take a std::string parameter in its ctor)

What Should a Handler Do?

- Fully recover, then resume somehow, or
- Partially recover and *re-throw* the exception

  (by using `throw;`)

Pretty Good Idea #4

- If you can’t do anything about an exception, don’t catch it!
- Unless you need to release resources
  - then re-throw the exception

Pretty Good Idea #5

- **`catch(...)`** should usually re-throw

Resource Management

- Dangling Resource Problem
  - a function that allocates a resource might throw before deallocating the resource
- Solutions:
  - Handle the situation locally
  - use an Object Wrapper (RAII)
- **`auto_ptr`**, the standard wrapper for memory
  - a *smart pointer*

A Dangling Resource

```
void f(const char* fname)
{
    FILE* fp = fopen(fname,"r");
    if (fp)
    {
        g(fp);       // Suppose g() throws?
        fclose(fp); // Then this won't happen!
    }
}

// continued...
```

Local Handlers

```
void f(const char* fname)
{
    FILE* fp = fopen(fname,"r");
    if (fp)
    {
        try
        {
            g(fp);
        }
        catch(...)
        {
            fclose(fp);
            puts("File closed");
            throw;  // Re-throw for
                    // other handlers
        }
        fclose(fp); // The normal close
    }
}
```

RAII

- "Resource Allocation is Initialization"
- Use objects on the stack to control resources
- The constructor allocates
- The destructor deallocates

Object Wrappers

(To leverage stack unwinding)

```
class File
{
    FILE* f;

public:
    File(const char* fname, const char* mode)
    {
        f = fopen(fname, mode);  // allocate
    }
    ~File()
    {
        fclose(f);               // deallocate
        puts("File closed");
    }
};
```

```
void f(const char* fname)
{
    File x(fname,"r");
    g(x.getFP());
}
```

Pretty Good Idea #6

- Use object wrappers to manage resources

Memory Leaks

```
void f()
{
    T* p = new T;
    g(p);       // Suppose g() throws?
    delete p;   // Then this won't happen!
}
```

auto_ptr

```
void f()
{
    auto_ptr<T> p(new T);
    g(p);
}

// delete p is implicit
```

Another `auto_ptr` Example

```
Employee* Employee::read(istream& in)
{
  // Create object from file data
  auto_ptr<Employee> p(new Employee);
  in >> *p;
  if (in.fail())
    throw EmployeeError("File input error");
  return p.release();
}
```

Pretty Good Idea #7

- Wrap local & member heap allocations in an `auto_ptr` object
  - scalars only – no arrays!
- Don't do much else with it
  - Herb Sutter, "Using `auto_ptr` Effectively", CUJ, October 1999, pp. 63-67.

Dynamic Memory Mgt.

- **`new`** operator throws **`bad_alloc`** when memory is exhausted
- You can request traditional null-return behavior with **`nothrow_t`** version
- Or call **`set_new_handler`** to install your own new handler

`new` and Exceptions

```
#include <new>
#include <iostream>

int main()
{
    try
    {
        int* p = new int;
        cout << "memory allocated\n";
    }
    catch (bad_alloc& x)
    {
        cout << "memory failure: " << x.what()
             << endl;
    }
}
```

`new` - Traditional Behavior

```
#include <new>
#include <iostream>
using namespace std;

int main()
{
     int* p = new (nothrow) int;
     if (p)
          cout << "memory allocated\n";
     else
          cout << "memory failure\n";
}
```

What’s Wrong Here?

```
void StackOfInt::grow()
{
   // Enlarge stack's data store
   capacity += INCREMENT;
   int* newData = new int[capacity];
   for (size_t i = 0; i < count; ++i)
      newData[i] = data[i];
   delete [] data;
   data = newData;
}
```

An Improvement

```
void StackOfInt::grow()
{
   // Enlarge stack's data store
   size_t newCapacity = capacity + INCREMENT;
   int* newData = new int[newCapacity];
   for (size_t i = 0; i < count; ++i)
      newData[i] = data[i];

   // Update state only when "safe" to do so
   delete [] data;
   data = newData;
   capacity = newCapacity; // moved
}
```

Fundamental Principle of Exception Safety

- Separate operations that may throw from those that change state
  - only change state when exceptions can no longer occur
- Corollary:
  - Do one thing at a time (cohesion)
  - why std::stack<T>::pop() returns void
    - The returned copy might throw
      - and the state has changed!

Rules of Exception Safety

- If you can't handle an exception, let it propagate up ("Exception neutral")
- Leave your data in a *consistent* state
  - Use RAII to allocate resources
  - Only change your state with non-throwing ops
  - An object should only own one resource
- Functions should perform only one logical operation
- Destructors should never throw
- Good references:
  - Sutter, *Exceptional C++* and *More Exceptional C++*
  - Abrahams, www.boost.org/more/generic_exeption_safety.html

Really Good Idea #8

- Don't let an exception escape from a destructor.
- If you see no alternative, however, make sure an exception isn't pending with the **`uncaught_exception()`** library function, then proceed.
  - I've never seen it done

```
#include <exception>
#include <iostream>
using namespace std;

class C
{
public:
    ~C()
    {
        if (uncaught_exception())
            cout << "unwinding..\n";
        else
            throw 1;
    }
};
```

```
int main()
{
    try
    {
        C c;
    }
    catch (int&)
    {
        cout << "caught an int\n";
    }
```

caught an int

```
    try
    {
        C c;
        throw "";
    }
    catch (char*)
    {
        cout << "caught a char*\n";
    }
}
```

unwinding..
*caught a char**

Destructors that Throw

- Are Evil
- Unfit for use in containers
- So use uncaught_exception( ) only under controlled (non-container) conditions