Inheritance

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.jku.at

Class Hierarchies

Classes represent collections of uniform objects.
- In reality, objects come in variants.
- Often the variants can be hierarchically classified.

A bear is a predator, is a mammal, is an animal.

Parent Classes and Child Classes

Two objects may share some features and differ in others.
- A wolf and a mouse are both mammals.
  - Both wolves and mice breastfeed their offspring.
  - A wolf is a predator while a mouse is a rodent.
    - A wolf eats animals.
    - A mouse eats corn.
- “Mammal” is the parent of children “predator” and “rodent”.
  - Predators and rodents are both mammals, but of a different kind.

Object-oriented languages like C++ offer a similar organization of classes; their objects satisfy corresponding properties.

1. Deriving Classes from Base Classes
2. Generic Methods and Types
3. Virtual Functions and Overriding
4. Abstract Classes, Interfaces, Frameworks
Example: An Internet Shop

The shop offers as articles both books and CDs.

- Books:
  - Article number, title, price.

- CDs:
  - Article number, title, price.
  - Interpreter, list of songs.

A shopping cart shall list the number, title, and price of the selected articles; by clicking on an article the full information is displayed.

Base Class

The common article functionality may be extracted to a base class.

```c++
class Article {
private:
  char* number;
  char* title;
  int price;
public:
  Article(...): ...
  { }
  char* getNumber() const
  { return number; }
  char* getTitle() const
  { return title; }
  int getPrice() const
  { return price; }
};
```

Books and CDs are special cases of articles.

Derived Class

Special functionality may be added to the common functionality.

```c++
class Book : public Article { // Book is derived from Article
private:
  char* author;
  char* publisher;
  char* ISBN;
public:
  Book(...): ...
  { }
  char* getAuthor() const
  { return author; }
  char* getPublisher() const
  { return publisher; }
  char* getISBN() const
  { return ISBN; }
};
```

Class Book inherits all features of Article.

Inheritance

Derived classes inherit from their base classes.

```c++
class Derived : public Base, ...
{ ...
}
```

- Class Derived is derived from Base.
  - Base is the (direct) base class of Derived.
  - Derived classes are also called "subclasses" or "child" classes.
  - Base classes are also called "superclasses" or "parent" classes.

- Class Derived inherits from Base.
  - All data members and object functions of Base.
  - Can access them like its own except those declared private.

Inheritance is transitive.

- Derived inherits also from its indirect base classes, i.e. from the base class of Base, from the base class of the base class, and so on.

A derived class inherits from all its ancestor classes.
Access Specifiers

Base classes may be provided with an access specifier.

class Derived : public Base, ... {
...
}
class Derived : protected Base, ... {
...
}
class Derived : private Base, ... {
...
}
class Derived : Base, ... {
...
}

- Restricts access to members of Base for the children of Derived:
  - public: all access specifiers in Base preserve their meaning.
  - protected: public members of Base become protected.
  - private: all members of Base become private.
- Default is private for class.
- public for struct.

Typically, simply public inheritance is applied.

Derived Classes

A class may have multiple children.

class CD : public Article {
  // CD is derived from Article
  private:
    char* interpreter;
    char*[] songs;
  public:
    CD(...): ... {
      }
    char* getInterpreter() const { return interpreter; }
    char*[] getSongs() const { return songs; }
  };

Also CD inherits all features of Article.

Multiple Inheritance

In C++, a class may also have multiple parents.

class Derived: public Base1, public Base2, ... {
  ...
};

- Derived inherits from Base1, and Base2, and ....
- Object contains separate “subobjects” for each base class.
- Name clashes have to be resolved by qualification with base class.
  - Assume both Base1 and Base2 declare a data member x.
  - Derived can refer to Base1::x and Base2::x but not just to x.
- Thus a directed acyclic inheritance graph can be constructed.
  - If both Base1 and Base2 have a common ancestor class A, two
    separate subobjects of type A are created.
  - Specifier virtual lets corresponding subobjects be shared.

Multiple inheritance may lead to complex class designs; use with care.
Constructors

class Article {
    private:
        char* number;
        char* title;
        int price;
    public:
        Article(char* n, char* t, int p):
            number(n), title(t), price(p)
        {
        }
};

The constructors of a base class are not inherited.

Constructors in Derived Classes

class Book : public Article {
    private:
        char* author;
        char* publisher;
        char* ISBN;
    public:
        Book(char *n, char* t, int p, char *a, char *u, char *i):
            Article(n, t, p), author(a), publisher(u), ISBN(i)
        {
        }
};

A derived class must define its own constructor.
- May call (in its initialization list) first a constructor of the base class.
- Otherwise, default constructor of base class is called first.

Derived class is responsible for initializing data members of base class.

Copy Assignment Operators

class Article {
    ...;
    // this definition is automatically generated
    Article& operator=(const Article& a) {
        number = a.number; title = a.title; price = a.price;
        return *this;
    }
};

class Book : public Article {
    ...;
    // this definition is automatically generated
    Book& operator=(const Book& b) {
        Article::operator=(b);
        author = b.author; publisher = b.publisher; ISBN = b.ISBN;
        return *this;
    }
};

Also the copy assignment operator of a base class is not inherited.

Inheritance for Code Sharing

Inheritance reduces the amount of code to be written.
- Imperative programming:
    Whenever there are two or more functions that share common functionality, this functionality should be put in a separate function; this function is then called by the other functions.
- Object-oriented programming:
    Whenever there are two or more classes that share common functionality, this functionality should be put in a separate base class; from this base class, the other classes are then derived.

Avoid code duplication among classes by inheritance.
1. Deriving Classes from Base Classes

2. Generic Methods and Types

3. Virtual Functions and Overriding

4. Abstract Classes, Interfaces, Frameworks

Is-Relationship

Inheritance constructs a subset relationship.

- A class denotes the set of objects belonging to the class.
- A derived class denotes a subset of the base class.
- An object of a derived class is also an object of the base class (and therefore of any ancestor class).

An object of type Book is also of type Article (but not vice versa).

Type Compatibility

- A derived class is compatible with the base class.
- Has all data members and member functions of the base class.

General rule:
- Wherever an object of a class \( C \) is expected, also an object of a class may be used that is (directly or indirectly) derived from \( C \).

Example: internet shop.
- Implement shopping cart that works with object of type Article.
- Later derive classes Book, CD, . . . from Article.
- Shopping cart can hold objects of type Book, CD, . . .

Inheritance may be used to implement programs that are “generic” i.e. operate on multiple data types.

Object Assignment

Objects may be assigned to object variables.

```
void printTitle(Article a) { cout << a.getTitle(); }
```

Book book("1234", "My Title", 2490,
    "My Author", "My Publisher", "12345678");

Article a = book; // copy constructor
a = book; // copy assignment
printTitle(book); // copy constructor
```

- An object of a derived class may be assigned to a variable of a base (in general: ancestor) class.
- By the assignment, the object is sliced.
  - The additional members of the derived class are removed.

By object slicing, all additional information is lost; while this is technically legal, it is costly and often denotes a programming error.
Object pointers may be assigned to pointer variables.

```cpp
void printTitle(Article* a) { cout << a->getTitle(); }

Book* book = new Book("1234", "My Title", 2490, "My Author", "My Publisher", "12345678");

Article* a = book; // pointer assignment
a = book; // pointer assignment
printTitle(book); // pointer assignment
```

A pointer to an object of a derived class may be assigned to a variable whose type is a pointer to the base (ancestor) class.

By the assignment, only the static (compile-time) type information is lost; the object itself preserves in memory its original identity.

This is the preferred way of writing generic code; objects are not sliced because only pointers are copied.

Dynamic casts must be explicitly applied for assigning pointers of base classes to pointer variables of derived classes.

```cpp
#define NULL (0)

Article *a = ...;
Book *book = dynamic_cast<Book*>(a);
if (book != NULL) { cout << book->getAuthor(); }
CD *cd = dynamic_cast<CD*>(a);
if (cd != NULL) { cout << cd->getInterpreter(); }
```

Dynamic casts check whether the pointer points to an object of a class (or a subclass of a class). If so, it returns a pointer of type to that object. If not, a NULL pointer is returned.

Static versus Dynamic Types

An object (or object pointer) variable has two different types.

- **Static type**: the type appearing in the declaration.
  ```cpp
  Article* ap = ...;
  ```
  Determines which members can be accessed.

- **Dynamic type**: the type of the object stored at runtime.
  ```cpp
  Book* bp = new Book(...);
  Article* ap = bp;
  ```
  Determines which virtual member functions are called (see later).

While the static type is fixed at compile time, the dynamic type can change at runtime.
Generic Methods

void printInfo(Article *a) {
    cout << "Article" << a->getTitle();
    cout << " (" << a->getNumber() << ") : ";
    int price = a->getPrice();
    cout << (price/100) << "." << (price%100) << "Euro\n";
}

Book* book = new Book(...);
CD* cd = new CD(...);
printInfo(book);
printInfo(cd);

Generic methods can operate on arguments of multiple dynamic types.

Generic Types

class ShoppingCart {
...
    void add(Article* a);
    Article* getArticle(int index);
};

ShoppingCart cart(...);
Book* book = new Book(...);
CD* cd = new CD(...);
cart.add(book);
cart.add(cd);

Article* a = cart.getArticle(0); // may be book or CD

Generic containers can contain elements of multiple dynamic types.

Generic Pointers

The type void* can refer to an object of any class.

class Stack {
    Book *book = new Book(...);
    int number;
    int size;
    void** stack;
    void resize();
public:
    Stack();
    int length();
    void push(void *e);
    void *pop();
    void *top();
};

Stack::Stack(): number(0), size(10), stack(new void*[size]) { }
int Stack::length() { return number; }
void* Stack::pop() { number = number-1; return stack[number]; }
void* Stack::top() { return stack[number-1]; }

void Stack::push(void *e) {
    if (number == size) resize();
    stack[number] = e;
    number = number+1;
}

void Stack::resize() {
    int size0 = 2*size;
    void **stack0 = new void*[size0];
    for (int i=0; i<size; i++) stack0[i] = stack[i];
    delete[] stack;
    size = size0; stack = stack0;
}

Generic containers can also hold arbitrary objects.
1. Deriving Classes from Base Classes

2. Generic Methods and Types

3. Virtual Functions and Overriding

4. Abstract Classes, Interfaces, Frameworks

---

### Declaring Methods in Base Classes

```cpp
class Article {
private:
    char* number;
    char* title;
    int price;
public:
    ...
    void printInfo();
};

void Article::printInfo()
{
    cout << "Article" << getTitle();
    cout << " (" << getNumber() << ": ");
    int price = getPrice();
    cout << (price/100) << "." << (price%100) << "Euro\n";
}
```

Method `printInfo` is inherited by all classes derived from `Article`.

---

### Inheriting Methods from Base Classes

Classes `Book` and `CD` may use `printInfo`.

```cpp
class Book: public Article { ... };
class CD: public Article { ... };
```

```cpp
Book* book = new Book(...); book->printInfo();
CD* cd = new CD(...); cd->printInfo();
```

**Problem:** `printInfo()` is too general.

- Only prints generic information on articles.
- Does not print information specific to books or CDs.

How to customize `printInfo` for derived classes?

---

### Virtual Functions

```cpp
class Base {
    virtual T func(...); // overrides Base::func()
};

T Base::func(...) { ... } T Derived::func(...) { ... }
```

```cpp
Base *object = new Base(...); // dynamic type is Base
... object->func(...) ... // calls Base::func()
```

```cpp
Base *object = new Derived(...); // dynamic type is Derived
... object->func(...) ... // calls Derived::func()
```

- A function declared as virtual can be overridden.
  - In a derived class, a function is declared with same name and same types for parameters and return value.
- When a virtual function is called on an object, the function definition for the dynamic type of the object is executed.
  - Form of genericity called **type polymorphism**.
- Base function may be still called (e.g. by the overriding function).
Example

```cpp
class Article {
    ...
    virtual void printInfo();
};
void Article::printInfo() { ... }

class Book: public Article {
    char *author;
    char *interpreter;
    virtual void printInfo();
};
void Book::printInfo() {
    Article::printInfo();
    cout << author << 
    cout << interpreter << 
}

CD* cd = new CD(...); cd->printInfo(); // CD::printInfo()
Article* a1 = book; a1->printInfo(); // Book::printInfo()
Article* a2 = cd; a2->printInfo(); // CD::printInfo()
```

Overriding functions may use functionality of base function.

Wolfgang Schreiner
http://www.risc.jku.at
33/51

Generic Types/Methods

```cpp
class ShoppingCart {
    int number;
    Article* articles[];
    ...
    void add(Article* a) { ...; articles[number] = a; ... }
    void printArticles() {
        for (int i=0; i<number; i++) {
            articles[i]->printInfo(); // Book::printInfo() or CD::printInfo()
        }
    }
};

ShoppingCart cart(...);
Book* book = new Book(...); cart.add(book);
CD* cd = new CD(...); cart.add(cd);
cart.printArticles();
```

Core of object-oriented programming: generic types/methods call the methods associated to the dynamic types of their elements/arguments.

Wolfgang Schreiner
http://www.risc.jku.at
34/51

Covariant Return Types

The return type of an overriding function may be actually more special than the return type of the base function.

```cpp
class Number {
    ...
    virtual Number* add(Number* n);
};

class Fraction : public Number {
    ...
    virtual Fraction* add(Number* n);
};
```

- Pointer/reference to some base type may be replaced by a pointer/reference to some derived type.
- Need not be the type of the class itself.
- Only for the return type, not for the argument types!

The signature of the overriding function may be a bit more specific.

Wolfgang Schreiner
http://www.risc.jku.at
35/51

Constructors/Destructors

Inside a constructor/destructor, also for virtual functions the definitions of the current class are applied.

```cpp
class Base {
    virtual void func();
};

class Derived: public Base {
    virtual void func();
};

Base::func() { ... }
Derived::func() { ... }
```

Derived object:

- When object is constructed, first constructor of base class is called:
  - Executes Base::func()
- Afterwards, constructor of derived class is called:
  - Executes Derived::func()

Prevents access to still uninitialized part of the object.

Wolfgang Schreiner
http://www.risc.jku.at
36/51
Virtual Destructors

- By default, the destructor of a class is not virtual.
  - If an object is deleted, the destructor of its static type is called.
    ```cpp
    class Base {}
    Base* object = new Derived(...);
    delete object;
    ```
  - In most situations, this is not what is wanted/expected.
    The compiler may produce a corresponding warning.
- A destructor can be declared as virtual in the base class.
  - Then the destructor of the dynamic type is called.
    ```cpp
    class Base {
        virtual ~Base() {}
    }
    Base* object = new Derived(...);
    delete object;
    ```
  - A class that is intended for derivation should have a virtual destructor.

Abstract Classes

- A virtual function need not have a definition.
  ```cpp
  // abstract class
  class Base {
      virtual T func(...) = 0;
  }
  ```
- A pure virtual function is declared with the pure specifier `"=0"`.
  ```cpp
  class Derived: public Base {
      virtual T func(...) = 0;
  }
  ```
- A concrete class has no pure virtual functions.
  ```cpp
  class Derived: public Base {
      T Derived::func(...) { ... }
  }
  ```

Interfaces

- Abstract classes can represent interfaces.
  ```cpp
  // abstract class
  class Base {
      virtual T func(...) = 0;
  }
  ```
  ```cpp
  class Derived: public Base {
      virtual T func(...) = 0;
  }
  ```
  An interface only defines the signature of a data type.
  - Names and types of the operations on the type.
  - E.g. an interface IntStack with the usual operations for a stack of integer values.
  - A (concrete) class represents an implementation of the data type.
    - Defines its concrete representation and the concrete realization of the operations on the type.
    - E.g. a class IntArrayStack representing a stack by an array or a class IntListStack representing a stack by a linked list.
  - By an interface, we thus get an abstract datatype.
    - IntStack serves as the static type for all stack objects.
    - IntArrayStack or IntListStack are only used when new stack objects are created.
  By the use of interfaces, the concrete representation of an abstract datatype can be easily replaced without modifying the program.
Interfaces

Interfaces represent “shields” for object representations.

Only the functions of the interface are accessible to users of the object.

Wolfgang Schreiner http://www.risc.jku.at

An Interface

An interface is an abstract class with only pure virtual functions.

```
// IntStack.h
class IntStack {
    public:
        // a virtual dummy destructor
        virtual ~IntStack { };
        // the operations to be defined by any implementation
        virtual bool isEmpty() = 0;
        virtual void push(int value) = 0;
        virtual int pop() = 0;
        virtual int top() = 0;
};
```

The signature of an abstract datatype “stack of integers”.

Wolfgang Schreiner http://www.risc.jku.at

An Implementation of the Interface

An interface is implemented by deriving from the abstract class a concrete class.

```
// IntArrayStack.h
class IntArrayStack: public IntStack {
    private:
        // representation of the stack
        int number;
        int size;
        int* stack;
    public:
        // the concrete constructor
        IntArrayStack();
        virtual ~IntArrayStack();
        int length();
        bool isEmpty();
        void resize();
        int pop();
        int top();
};
```

```
// IntArrayStack.cpp
IntArrayStack::IntArrayStack(): number(0), size(10), stack(new int[size]) { }
IntArrayStack::~IntArrayStack(): number(0), size(10), stack(new int[size]) { }
```

Wolfgang Schreiner http://www.risc.jku.at

An Implementation of the Interface

// IntArrayStack.cpp
IntArrayStack::IntArrayStack(): number(0), size(10), stack(new int[size]) { }
IntArrayStack::~IntArrayStack() { delete[] stack; }
int IntArrayStack::length() { return number; }
bool IntArrayStack::isEmpty() { return length() == 0; }
int IntArrayStack::pop() { number = number-1; return stack[number]; }
int IntArrayStack::top() { return stack[number-1]; }
void IntArrayStack::push(int e) {
    if (number == size) resize();
    stack[number] = e;
    number = number+1;
}
void IntArrayStack::resize() {
    int size0 = 2*size;
    int* stack0 = new int[size0];
    for (int i=0; i<size; i++) stack0[i] = stack[i];
    delete[] stack;
    size = size0; stack = stack0;
}

Wolfgang Schreiner http://www.risc.jku.at
Another Implementation of the Interface

An interface can be implemented by multiple classes.

```cpp
class IntListStack: public IntStack { // IntListStack.h
private:
    IntNode *head; // stack represented by a
    class IntNode; // sequence of linked nodes
public:
    IntListStack();
    virtual ~IntListStack();
    virtual bool isEmpty();
    virtual void push(int e);
    virtual int pop();
    virtual int top();
};
```

```cpp
class IntListStack::IntNode { // IntListStack.cpp
public:
    int value;
    IntNode* next;
    IntNode(int v, IntNode *n): value(v), next(n) {
    }
};
```

```cpp
IntListStack::IntListStack() { head = NULL; }
IntListStack::~IntListStack() { while (head != NULL) pop(); }
bool IntListStack::isEmpty() { return head == NULL; }
void IntListStack::push(int e) { head = new IntNode(e, head); }
int IntListStack::top() { return head->value; }
int IntListStack::pop() {
    int result = head->value;
    IntNode *next = head->next;
    delete head;
    head = next;
    return result;
}
```

The Use of the Interface

// a generic function on stacks
public void push(IntStack* s, int n, int v) {
    for (int i=0; i<n; i++) s->push(v);
}

```cpp
int main() { // original program
    IntStack* stack = new IntArrayStack();
    push(stack, 10, 5); cout << stack.pop();
    // cout << stack.length(); // illegal, length() not in interface
    delete stack;
}

int main() { // program with new data representation
    IntStack* stack = new IntListStack();
    push(stack, 10, 5); cout << stack.pop();
    delete stack;
}
```

Use interfaces to make programs independent of data representations.

Application Frameworks

An abstract class need not be just an interface without own functionality.

- **(Application) framework**: an abstract class A that also has some concrete functions.
- The concrete functions provide actual application functionality.
- The abstract functions are “hooks” for customizing this functionality.
- Some concrete functions of A call the abstract functions.
- Functionality depends on how abstract functions are overridden.
- **Application**: a concrete class C that is derived from A.
- Has to override the abstract functions of A by concrete functions.
- Inherits the functionality of A with appropriate customization.

Application frameworks allow the development of “generic applications”.

Wolfgang Schreiner
http://www.risc.jku.at
Application Frameworks

Framework provides “hooks” for customization of application.

Example Framework

class Printer { // an application framework
    public:
        Printer() { } ;
        virtual ~Printer() { };
        void print(int n); // functionality of framework
        virtual char* getText() = 0; // hook for customization
    }

    // print n lines containing the denoted text
    void Printer::print(int n)
    {
        for(int i=0; i<n; i++)
        {
            cout << getText() << "\n";
        }
    }

A framework for printing text in a formatted manner.

Example Application

class IntPrinter: public Printer { // an application
    int i;
    char text[20];
    public:
        IntPrinter(int i) { this->i = i; }
        virtual char* getText(); // customization of framework
    };

    char* IntPrinter::getText() {
        sprintf(text, "%d", i);
        return text;
    }

    int main() {
        IntPrinter p(7);
        p.print(3); // 7 7 7
        p.print(5); // 7 7 7 7 7
    }

An application for printing integers in a formatted manner.