Advanced Programming
Topic 5: Inheritance and Polymorphism

Nguyen, Thien Binh
Mechatronics and Sensor Systems Technology
Vietnamese-German University

Intake 2024

Outline

Outline

An Introductory Example: Matrix Types

- Depending the sizes and entry patterns, matrices can be classified into different types.
- A general matrix $A \in \mathbb{R}^{m \times n}$ is any matrix with m and n as the number of rows and columns, respectively.
- A square matrix $S \in \mathbb{R}^{m \times m}$ has the same number of rows and columns.
- Matrix $B \in \mathbb{R}^{m \times m}$ is called symmetric if $B_{ij} = B_{ji}$.
- A lower triangular matrix $L \in \mathbb{R}^{m \times m}$ has zero entries in the upper triangular part; whereas a upper triangular matrix $S \in \mathbb{U}^{m \times m}$ has zero entries in the lower triangular part.
- A diagonal matrix $D \in \mathbb{R}^{m \times m}$ has non-zero entries only in the diagonals. A common diagonal matrix is the so-called tri-diagonal matrix T.

An Introductory Example: Matrix Types

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{bmatrix}, S = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 6 & 7 & 8 \\ 3 & 7 & 11 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 5 & 6 & 0 & 0 \\ 9 & 10 & 11 & 0 \\ 13 & 14 & 15 & 16 \end{bmatrix}, U = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 11 & 12 \\ 0 & 0 & 0 & 16 \end{bmatrix}, T = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix},$$

An Introductory Example: Matrix Types

- Question: What makes a matrix have its specific type?
  - Consider one needs to approximate the solution u of the following 1D Poisson equation

$$\begin{cases} \Delta u(x) = b(x), \quad x \in \Omega = (0,1), \\ u(x=0) = u_0, \\ u(x=1) = u_1, \end{cases} \tag{1}$$

Dividing the domain Ω into $N+1$ equidistant points $\{0 = x_0, x_1, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_{N-1}, x_N = 1\}$ where $\Delta x = x_{j+1} - x_j$ is the grid size. Using a 2nd-order finite difference method, Eq. (1) is discretized as follows,

$$\frac{u_{j+1} - 2u_j + u_{j-1}}{\Delta x^2} = b_j, \tag{2}$$

An Introductory Example: Matrix Types

- Question: What makes a matrix have its specific type?
  - Writing Eq. (2) at each grid point, we obtain that

$$\begin{cases} u_2 - 2u_1 = \Delta x^2 b_1 - u_0, \quad j = 1 \\ u_{j+1} - 2u_j + u_{j-1} = \Delta x^2 b_j, \quad 1 < j < N-1, \\ u_{N-2} - 2u_{N-1} = \Delta x^2 b_{N-1} - u_1, \quad j = N-1 \end{cases} \tag{3}$$

which can be written in a matrix form as

$$\begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 & 0 \\ -1 & 2 & -1 & \dots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & -1 & 2 & -1 \\ 0 & 0 & 0 & \dots & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{bmatrix} = - \begin{bmatrix} \Delta x^2 b_1 - u_0 \\ \Delta x^2 b_2 \\ \vdots \\ \Delta x^2 b_{N-2} \\ \Delta x^2 b_{N-1} - u_1 \end{bmatrix} \tag{4}$$

$\Rightarrow$ *Note that the LHS matrix is exactly our tri-diagonal T.*

An Introductory Example: Matrix Types

- Lower and upper triangular matrices are obtained through the LU decomposition process, and are particularly useful when solving a linear system using a direct method.
- Symmetric positive definite matrices occur frequently in simulations, and are subject to a very powerful iterative linear solver, i.e., the Conjugate Gradient (CG) method

An Introductory Example: Matrix Types

- Question: What are the similarities and differences of the general matrix A and the square matrix S?

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \end{bmatrix}, S = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix},$$

An Introductory Example: Matrix Types

- Question: What are the similarities and differences of the general matrix A and the square matrix S?
  - Similarities: S can be seen as a general matrix, i.e., possesses all the properties (`**entries`, `numRows`, `numCols`), and all possible operators for a general matrix ($S_1 + S_2$, ++S, etc.)
    $\Rightarrow$ *A square matrix* *is a* general matrix.
  - Differences: S has some unique properties in which a general matrix does not:
    - ★ `size = numRows = numCols`
    - ★ S has a different type from A
    - ★ We can calculate the *determinant* of the square matrix S
    - ★ We can check whether S is symmetric, lower triangular, or upper triangular, etc.

An Introductory Example: Matrix Types

- Question: Given class `MatrixDouble` in which each object is a general matrix A, how could we implement class `MatrixDoubleSquare` for objects being square matrices, e.g., S?

Vietnamese - German University

An Introductory Example: Matrix Types

- Question: Given class `MatrixDouble` in which each object is a general matrix A, how could we implement class `MatrixDoubleSquare` for objects being square matrices, e.g., S?
  - One possible solution: *Just copy and paste the reusable parts from class `MatrixDouble` and implement the new properties and methods for class `MatrixDoubleSquare`?*

An Introductory Example: Matrix Types

- Question: Given class `MatrixDouble` in which each object is a general matrix A, how could we implement class `MatrixDoubleSquare` for objects being square matrices, e.g., S?
  - One possible solution: *Just copy and paste the reusable parts from class `MatrixDouble` and implement the new properties and methods for class `MatrixDoubleSquare`?*
  - Many drawbacks:
    - ★ A lot of code lines are repeated ⇒ Code redundancy ⇒ Waste of coding time and energy, prone to bugging
    - ★ What if you want to modify or improve class `MatrixDouble`? Every single change must be updated manually in class `MatrixDoubleSquare`
    - ★ What if there are bugs in class `MatrixDouble`? You have to debug them and trace their locations in class `MatrixDoubleSquare` to make appropriate changes
    - ★ What if someone accidentally modifies your class `MatrixDouble`? Now classes `MatrixDouble` and `MatrixDoubleSquare` are not synchronized properly!

An Introductory Example: Matrix Types

- Question: Given class `MatrixDouble` in which each object is a general matrix A, how could we implement class `MatrixDoubleSquare` for objects being square matrices, e.g., S?
  - A much better solution: *Thanks to OOP, C++ offers the so-called* *`inheritance`* *which allows* *`derived classes`* *(e.g., square matrix S) to inherit most of the properties from their* *`base class`* *(e.g., general matrix A.)*
  - Benefits of inheritance:
    - ★ Inheritance resolves most of the aforementioned disadvantages of the *copy-and-paste* approach
    - ★ Moreover, inheritance enables `polymorphism` which is a mechanism that allows a method of a derived class to *override* its contents defined in the base class.

Outline

Introduction to Inheritance

- Object relationships:
  - *Composition:* constitutes a *has-a* relationship.
    e.g. The tri-diagonal matrix T is called a *sparse* matrix since most of its entries are just zeros. It would be a lot of waste if memory is allocated to save every single entry. Instead, one can use vectors $v_0 = \{2,\ 2,\ 2,\ 2\}$ and $v_1 = \{-1,\ -1,\ -1\}$ to save just the non-zero diagonals. In this case, class `MatrixDoubleTriDiag` uses class `VectorDouble` as its member data.
  - *Inheritance:* constitutes a *is-a* relationship.
    e.g. Classes `MatrixDouble` and `MatrixDoubleSquare` has a *is-a* relationship in which the latter class acquires the data and methods of the former class and then extend or modify them accordingly.

Introduction to Inheritance

- In C++, the class being inherited is called the `base class`, `parent class`, or `superclass`, and the class which inherits is called the `derived class`, `child class`, or `subclass`
- *Inheritance hierarchy:* a derived class itself can be the base class of another derived class.
  e.g. Classes `MatrixDoubleLowTri` (for lower triangular matrices) and `MatrixDoubleUpTri` (for upper triangular matrices) can be both derived classes of `MatrixDoubleSquare`, which itself is a derived class of `MatrixDouble`.
- Methods that are NOT inherited:
  - `private` members and methods
  - Constructors and destructors
  - Assignment operators
  - Non-member methods
  - `friend` methods

Outline

Access Privileges for Derived Classes I

- Class Derived inherited from class Base has the following syntax

```
    class Base
    {
      // declarations go here
    };

    class Derived : access_specifier Base
    {
      // declarations go here
    };
```

- access_specifier determines what is inherited and what is not from the base class. If not specified, access_specifier = private by default.

Vietnamese - German University

Access Privileges for Derived Classes II

1. `access_specifier = public`

$$\begin{cases} \text{private of Base is inaccessible} \\ \text{public of Base} = \texttt{public} \text{ of } \texttt{Derived} \\ \text{protected of Base} = \texttt{protected} \text{ of } \texttt{Derived} \end{cases}$$

2. `access_specifier = protected`

$$\begin{cases} \text{private of Base is inaccessible} \\ \text{public of Base} = \texttt{protected} \text{ of } \texttt{Derived} \\ \text{protected of Base} = \texttt{protected} \text{ of } \texttt{Derived} \end{cases}$$

3. `access_specifier = private`

$$\begin{cases} \text{private of Base is inaccessible} \\ \text{public of Base} = \texttt{private} \text{ of } \texttt{Derived} \\ \text{protected of Base} = \texttt{private} \text{ of } \texttt{Derived} \end{cases}$$

1. access_specifier = public:

```
class Base
{
private:
  // inaccessible to Derived
public:
protected:
};

class Derived : public Base
{
private:
  // private of Derived
protected:
  // protected of Base (inherited)
public:
  // public of Base (inherited)
};
```

Vietnamese - German University

access_specifier = public: II

- Example: What are the outputs?

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

// class Base
class Base
{
private:
  int  b_private;

protected:
  int b_protected;

public:
  int b_public;
  Base() : b_private(1), b_protected(2) {}

};

class Derived : public Base
{
```

access_specifier = public: III

```
public:
  void print()
  {
    cout << "b_private␣=␣" << b_private << endl;
    cout << "b_protected␣=␣" << b_protected << endl;
    cout << "b_public␣=␣" << b_public << endl;
  }
};

int main()
{
  Derived obj;
  obj.b_protected = 2;
  obj.b_public = 3;
  obj.print();
  return 0;
}
```

access_specifier = public:

- Example: What are the outputs?
  - Line 24: not allowed since `b_private` (`private of Base`) is not accessible
  - Line 33: not allowed since `b_protected`, inherited from `protected of Base`, is protected of Derived

access_specifier = protected:

2. access_specifier = protected:

```
class Base
{
private:
  // inaccessible to Derived
public:
protected:
};

class Derived : protected Base
{
private:
  // private of Derived
protected:
  // protected of Base (inherited)
  // public of Base (inherited)
public:
};
```

Vietnamese - German University

access_specifier = protected: II

- Example: What are the outputs?

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

class Base
{
private:
  int  b_private;

protected:
  int b_protected;

public:
  int b_public;
  Base() : b_private(1), b_protected(2) {}

};

class Derived : protected Base
{
public:
```

access_specifier = protected: III

```
void print()
  {
    cout << "b_private = " << b_private << endl;
    cout << "b_protected = " << b_protected << endl;
    cout << "b_public = " << b_public << endl;
  }
};

int main()
{
  Derived obj;
  obj.b_protected = 2;
  obj.b_public = 3;
  obj.print();
  return 0;
}
```

access_specifier = protected:

- Example: What are the outputs?
  - Line 24: not allowed since `b_private` (`private of Base`) is not accessible
  - Line 33: not allowed since `b_protected`, inherited from `protected of Base`, is protected of Derived
  - Line 34: not allowed since `b_public`, inherited from `public of Base`, is protected of Derived

access_specifier = private:

2. access_specifier = private:

```
class Base
{
private:
  // inaccessible to Derived
public:
protected:
};

class Derived : private Base
{
private:
  // private of Derived
  // protected of Base (inherited)
  // public of Base (inherited)
protected:
public:
};
```

access_specifier = private: II

- Example: What are the outputs?

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

class Base
{
private:
  int  b_private;

protected:
  int b_protected;

public:
  int b_public;
  Base() : b_private(1), b_protected(2) {}

};

class Derived : private Base
{
public:
```

Vietnamese - German University

access_specifier = private: III

```
  void print()
  {
    cout << "b_private␣=␣" << b_private << endl;
    cout << "b_protected␣=␣" << b_protected << endl;
    cout << "b_public␣=␣" << b_public << endl;
  }
};

int main()
{
  Derived obj;
  obj.b_protected = 2;
  obj.b_public = 3;
  obj.print();
  return 0;
}
```

access_specifier = private:

- Example: What are the outputs?
  - Line 24: not allowed since b_private (private of Base) is not accessible
  - Line 33: not allowed since b_protected, inherited from protected of Base, is private of Derived
  - Line 34: not allowed since b_public, inherited from public of Base, is private of Derived

Access Privileges for Derived Classes I

- Exercise: Class hierarchy: What are the outputs?

```
#include <iostream>
using namespace std;
class Base
{
private:
  int  b_private;
protected:
  int b_protected;
public:
  int b_public;
  Base() : b_private(1), b_protected(2), b_public(3) {}
};
class Derived: protected Base
{
protected:
  int d_protected;
public:
  int d_public;
  Derived() : d_protected(4), d_public(5) {}
};
```

Access Privileges for Derived Classes II

```
class DerivedofDerived : public Derived
{
public:
void print()
  {
    cout << "b_protected␣=␣" << b_protected << endl;
    cout << "b_public␣=␣" << b_public << endl;
    cout << "d_protected␣=␣" << d_protected << endl;
    cout << "d_public␣=␣" << d_public << endl;
  }
};
int main()
{
  DerivedofDerived obj;
  obj.b_public = 20;
  obj.d_public = 30;
  obj.print();
  return 0;
}
```

Vietnamese - German University

Scopes of Variables and Methods

- Question: Which method `void print()` is called and which variable `int var` is printed to the screen in the following code?

```cpp
#include <iostream>
using namespace std;
class Base
{
public:
   int var;
   void print() {cout << "In␣Base.␣var␣=␣" << var << endl;}
};
class Derived : public Base
{
public:
   int var;
   void print() {cout << "In␣Derived.␣var␣=␣" << var << endl;}
};
int main()
{
   Derived obj;
   obj.var = 10;
   obj.print();
   return 0;
}
```

Scopes of Variables and Methods

- When a member is called with a derived class object, the compiler prioritizes the member defined in the derived than in the base class.
- In order to specify the use of a member in the base class, its class scope must be included with a double colon `::`

```
int main()
{
  Derived obj;
  obj.var = 10;
  obj.Base::var = 20;
  obj.print();
  obj.Base::print();
  return 0;
}
```

Outline

Constructors

- Constructors of a base class are always called before those of a derived class. This is comprehensible since objects of the derived class are constructed based on those of the base class.
- Example:

```
#include <iostream>
using namespace std;
class Base
{
public:
  Base() { cout << "Base constructor" << endl; }
};
class Derived : public Base
{
public:
  Derived() { cout << "Derived constructor" << endl; }
};
int main()
{
  Derived obj;
  return 0;
}
```

Constructors I

- Question: Is the following code compilable?

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

class Base
{
protected:
  int b_var;
public:
  Base()
  {
    cout << "Calling␣the␣default␣Base()" << endl;
    b_var = 1;
  }
};

class Derived : public Base
{
protected:
  int d_var;
public:
```

Constructors II

```cpp
  Derived(const int& d_var_)
  {
    cout << "Calling␣Derived(const␣int&)" << endl;
    d_var = d_var_;
  }

  print()
  {
    cout << "b_var␣=␣" << b_var << endl;
    cout << "d_var␣=␣" << d_var << endl;
  }
};

int main()
{
  Derived obj(2);
  obj.print();
  return 0;
}
```

Constructors III

- Question: What if we want to initialize `Base::b_var` when defining an object of class `Derived`, e.g., `Derived obj(1,2)`?

Constructors I

- A constructor of a base class can be placed in the initialization list of a derived class constructor.
- Example: `Base(b_var_)` is called in the initialization list of `Derived(d_var_)` to initialize `Base::b_var`

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

class Base
{
protected:
  int b_var;
public:
  Base(const int& var_) { b_var = var_; }
};

class Derived : public Base
{
protected:
  int d_var;
```

Constructors II

```
public:
  Derived(const int& d_var_, const int& b_var_) : Base(b_var_)
  {
    d_var = d_var_;
  }

  print()
  {
    cout << "b_var␣=␣" << b_var << endl;
    cout << "d_var␣=␣" << d_var << endl;
  }
};

int main()
{
  Derived obj(1, 2);
  obj.print();
  return 0;
}
```

Destructors

- Destructors are called in a reverse order as that of constructors, i.e., the destructor of a derived class is always called before the base class destructor.

```
#include <iostream>
using namespace std;
class Base
{
public:
  ~Base() { cout << "Base destructor" << endl; }
};
class Derived : public Base
{
public:
  ~Derived() { cout << "Derived destructor" << endl; }
};
int main()
{
  Derived obj;
  return 0;
}
```

Outline

Calling Inherited Methods I

- When a method is called by objects of a derived class, the compiler will check for the method's definition backward in the inheritance hierarchy, i.e., starting from the called derived class and go back to its base class, and uses the first one it finds
- Example: Which `void print()` method is called in the following code?

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

class Base
{
public:
  int b_var;
  void print()
  {
    cout << "Base: b_var = " << b_var << endl;
  }
};
```

Calling Inherited Methods II

```cpp

class Derived : public Base
{
public:
  int d_var;
  void print()
  {
    cout << "Derived: d_var = " << d_var << endl;
  }
};

int main()
{
  Derived obj;
  obj.b_var = 1;
  obj.d_var = 2;
  obj.print();
  return 0;
}
```

Vietnamese - German University

Calling Inherited Methods III

- Question: What is the output?

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

class Base
{
public:
   int b_var;
   void print()
   {
      cout << "Base:␣b_var␣=␣" << b_var << endl;
   }
};

class Derived : public Base
{
public:
   int d_var;
};

int main()
```

Calling Inherited Methods IV

```
{
  Derived obj;
  obj.b_var = 1;
  obj.d_var = 2;
  obj.print();
  return 0;
}
```

- Question: Is the following code compilable?

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

class Base
{
public:
  int b_var;
private:
  void print()
  {
    cout << "Base: b_var = " << b_var << endl;
  }
```

Calling Inherited Methods V

```
};

class Derived : public Base
{
public:
  int d_var;
  void print()
  {
    cout << "Derived:␣d_var␣=␣" << d_var << endl;
  }
};

int main()
{
  Derived obj;
  obj.b_var = 1;
  obj.d_var = 2;
  obj.print();
  return 0;
}
```

Calling Inherited Methods VI

⇒ *When a method is redefined in a derived class, the `access_specifier` of inheritance is ignored.*

- A derived method can call its version defined in the base class by adding the base class with a double colon.
- Example: derived method `void print()` calls `void Base::print()` and adds additional information.

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

class Base
{
public:
  int b_var;
  void print()
  {
    cout << "Base:␣b_var␣=␣" << b_var << endl;
  }
};
```

Vietnamese - German University

Calling Inherited Methods VII

```cpp

class Derived : public Base
{
public:
  int d_var;
  void print()
  {
    Base::print();
    cout << "Derived: d_var = " << d_var << endl;
  }
};

int main()
{
  Derived obj;
  obj.b_var = 1;
  obj.d_var = 2;
  obj.print();
  return 0;
}
```

Calling Inherited Methods VIII

- BUT a constructor cannot directly call another constructor of the base class. It must be done through the constructor's initialization list.
- Question: How to correct the following code?

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

class Base
{
public:
  int b_var;
  Base(const int& b_var_) : b_var(b_var_)
  {
    cout << "Base: b_var = " << b_var << endl;
  }
};

class Derived : public Base
{
public:
  int d_var;
```

Calling Inherited Methods IX

```
  Derived(const int& d_var_) : d_var(d_var_)
  {
    Base::Base(1);
    cout << "Derived: d_var = " << d_var << endl;
  }
};

int main()
{
  Derived obj(2);
  return 0;
}
```

Outline

virtual methods

- Consider the following examples. What are the outputs?

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

class Base
{
public:
  void print() { cout << "This␣is␣BASE::print()" << endl; }
};

class Derived : public Base
{
public:
  void print() { cout << "This␣is␣DERIVED::print()" << endl; }
};
```

virtual methods

- Consider the following examples. What are the outputs?

```
int main()
{
  Base b_obj;
  Derived d_obj;
  b_obj.print();
  d_obj.Base::print();
  d_obj.print();

  Base *p;
  // is it possible to use p to point to d_obj?
  p = &d_obj;
  p->print();

  return 0;
}
```

`virtual` methods I

- Although `p` is of type `Base*`, it is possible to point to `d_obj` of type `Derived` since a derived object contains a base part (constructed first) and a derived part (constructed later).
- `p` is of type `Base*`, it can only see the base part of is of type `d_obj`.
- Question: How to use `Base* p` to print out `Derived::print()`?

virtual methods

- Question: How to use Base* p to print out Derived::print()? ⇒ virtual methods

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

class Base
{
public:
  // Base::print() is now virtual
  virtual void print() { cout << "This␣is␣BASE::print()" << endl; }
};

class Derived : public Base
{
public:
  void print() { cout << "This␣is␣DERIVED::print()" << endl; }
};
```

virtual methods

- Question: How to use Base* p to print out Derived::print()? ⇒ virtual methods

```
int main()
{
  Base b_obj;
  Derived d_obj;

  Base *p;
  p = &d_obj;
  p->print();

  return 0;
}
```

`virtual` methods

- If a member method is declared as `virtual`, the *most derived* version of the method will be called.
- A member method can only declared as `virtual` in a *base* class, adding the keyword in a derived method is possible but takes no effects.
- This capability is called `polymorphism` (poly- = many, -morphism = the process to change forms), and the derived method `overrides` its base virtual one.
- Note that overridden methods have exactly the same argument list as that of the base virtual method.

virtual methods I

- Example: What are the outputs?

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

class Base
{
public:
  virtual void print() {cout << "This␣is␣BASE::print()" << endl;}
};

class Derived_1 : public Base
{
public:
  virtual void print() {cout << "This␣is␣DERIVED-1::print()" << endl;
};

class Derived_2 : public Base
{
public:
  virtual void print() {cout << "This␣is␣DERIVED-2::print()" << endl;
};
```

virtual methods II

```

int main()
{
  Derived_1 d1_obj;
  Derived_2 d2_obj;

  Base *p;
  p = &d1_obj;
  p->print();
  p = &d2_obj;
  p->print();

  return 0;
}
```

virtual methods I

- Example: What are the outputs? Which method is overridden by polymorphism? Which method is overloaded? Which line is incorrect?

```
 1 #include <iostream>
 2 using namespace std;
 3
 4 class Base
 5 {
 6 public:
 7   virtual void print() {cout << "This␣is␣BASE::print()" << endl;}
 8 };
 9
10 class Derived_1 : public Base
11 {
12 public:
13   void print() {cout << "This␣is␣DERIVED-1::print()" << endl;}
14   void print(const int& val) {cout << "DERIVED-1:␣" << val << endl;}
15 };
16
17 class Derived_2 : public Derived_1
```

virtual methods II

```
{
  public:
  void print() {cout << "This is DERIVED-2::print()" << endl;}
  void print(const int& val) {cout << "DERIVED-2: " << val << endl;}
};

int main()
{
  Derived_1 d1_obj;
  Derived_2 d2_obj;

  cout << "... for p1 ..." << endl;
  Base *p1;
  p1 = &d1_obj;
  p1->print();
  p1 = &d2_obj;
  p1->print();
  p1->print(2);

  cout << "... for p2 ..." << endl;
  Derived_1 *p2;
  p2 = &d1_obj;
```

virtual methods III

```
  p2->print();
  p2 = &d2_obj;
  p2->print();
  p2->print(2);

  return 0;
}
```

`virtual` methods

- `Overloading` vs. `Overriding`:
  1. `Overloading`:
     - ★ Both class member and non-member functions can be overloaded.
     - ★ Overloaded functions have exactly the same name, but different input argument lists ⇒ help distinguishes which function to be called.
  2. `Overriding`:
     - ★ Only a `virtual` method of a base class can be overridden by its derived methods.
     - ★ Overridden methods have exact the same name and input argument lists ⇒ The most derived method will be called.

Vietnamese - German University

Abstract Classes

- Question: Why should one use a base-class pointer to point to a derived object? ⇒ *for generic programming!*
- Example: Write a function to sum up the areas of polygons of different types. That is, our function sumArea takes `Polygons *pt[]`, which could be of any specific type, as its input parameter. `computeArea()` has to compute the area of which type correspondingly.

```
double sumArea(const int& size, Polygons* poly[])
{
  double area(0.0);
  for (int i = 0; i < size; ++i)
  area += poly[i]->computeArea();
  return area;
}
```

Abstract Classes

- Example: Write a function to sum up the areas of polygons of different types.

```
int main()
{
  Triangles T(2.0, 1.0);
  Squares S(2.0);
  Rectangles R(2.0, 3.0);

  Polygons *p_obj[3];
  p_obj[0] = &T;
  p_obj[1] = &S;
  p_obj[2] = &R;

  // area of each type of polygons
  for (int i = 0; i < 3; ++i)
    cout << "area = " << p_obj[i]->computeArea() << endl;

  // area sum of all polygons of different types
  cout << "Total area = " << sumArea(3, p_obj) << endl;

  return 0;
}
```

Abstract Classes I

- Example: Write a function to sum up the areas of polygons of different types. *computeArea is declared virtual in the base class so that it can be overridden in the derived classes.*

```
// abstract class: Polygons
class Polygons
{
public:
  double area;
  Polygons() : area(0.0) {}

  // pure virtual member
  virtual double computeArea() = 0;
};


// class Triangles
class Triangles : public Polygons
{
protected:
  double base;
```

Abstract Classes II

```
  double height;
  public:
  Triangles(const double& base_, const double& height_)
  {
    base = base_;
    height = height_;
  }

  virtual double computeArea()
  {
    area = 0.5 * base * height;
    return area;
  }
};


// class Squares
class Squares : public Polygons
{
protected:
  double side;
  public:
```

Vietnamese - German University

Abstract Classes III

```
  Squares(const double& side_)
  {
    side = side_;
  }

  virtual double computeArea()
  {
    area = side * side;
    return area;
  }
};


// class Rectangles
class Rectangles : public Polygons
{
protected:
  double base;
  double height;
  public:
  Rectangles(const double& base_, const double& height_)
  {
```

Abstract Classes IV

```
    base = base_;
    height = height_;
  }

  virtual double computeArea()
  {
    area = base * height;
    return area;
  }
};
```

Vietnamese - German University

Abstract Classes

- Note that in line 9, computeArea does not have its body, instead, is set to be zero. Such a function is called pure virtual method.

```
// pure virtual member
virtual double computeArea() = 0;
```

- A class is called abstract if it has at least one pure virtual method.
- Abstract classes are served as interfaces for generic programming, e.g., class Polygons.

Abstract Classes

- Some rules for abstract classes:
  - It is possible to define an abstract pointer, for pointing to specific objects.

```
Polygons *pt;
```

  - But it is prohibited to instantiate abstract objects

```
Polygons obj; // NOT OK!
```

  - An abstract class cannot be an argument type, but it is OK to have it as an input reference or pointer

```
double sumArea(const int& size, Polygons* poly[])   // OK
double foo(Polygons poly)   // NOT OK!
```

virtual Destructors I

- Question: Which destructor is called when the object is deleted?

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

class Base
{
public:
  ~Base() {cout << "~Base is called" << endl;}
};

class Derived : public Base
{
public:
  int size;
  double *v;
  // constructor
  Derived(const int& size_)
  {
    size = size_;
    //v = new double[size];
  }
```

virtual Destructors II

```
    // destructor
    ~Derived()
    {
      cout << "~Derived␣is␣called" << endl;
      //delete []v;
    }
};

int main()
{
  Base *p_obj = new Derived(5);
  delete p_obj;

  return 0;
}
```

Vietnamese - German University

`virtual` Destructors I

- Question: Which destructor is called when the object is deleted? $\Rightarrow$ *We expect that both of the destructors are called, but only one of them, `Base::~Base()`, was.*
- What if the commented part with dynamic memory allocation is un-commented? $\Rightarrow$ *Since `Derived::~Derived()` is not called, memory will be leaked.*
- How to avoid this situation?
  For inheritance, always use a virtual destructor for the base class.

virtual Destructors I

- How to avoid this situation?
  For inheritance, always use a virtual destructor for the base class.

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

class Base
{
public:
  // now made virtual
  virtual ~Base() {cout << "~Base is called" << endl;}
};

class Derived : public Base
{
public:
  int size;
  double *v;
  Derived(const int& size_)
  {
```

virtual Destructors II

```
    size = size_;
    v = new double[size];
  }
  // this virtual has no effects
  // just explicitly notify that the destructor
  // is virtual
  virtual ~Derived()
  {
    cout << "~Derived␣is␣called" << endl;
    delete []v;
  }
};

int main()
{
  Base *p_obj = new Derived(5);
  delete p_obj;
  return 0;
}
```

Reading

1. Capper, *Introducing C++ for Scientists, Engineers, and Mathematicians*, **Chapter 12**
2. Pitt-Francis, and Whiteley, *Guide to Scientific Computing in C++*, **Chapter 7**
3. LearnCpp, `https://www.learncpp.com/` , **Chapter 11**