Ngôn ngữ lập trình

Bài 6:
Nạp Chồng Toán Tử
và Kế Thừa

Giảng viên: Lê Nguyễn Tuấn Thành

Email: thanhlnt@tlu.edu.vn

Bộ Môn Công Nghệ Phần Mềm – Khoa CNTT

Trường Đại Học Thủy Lợi

Nội dung

- Nạp chồng toán tử (Operator Overloading) và Hàm bạn (Friend Functions)
- Kế thừa (Inheritance)

Bài giảng có sử dụng hình vẽ trong cuốn sách "*Absolute C++*. W. Savitch, Addison Wesley, 2002"

1. Nạp chồng toán tử và Hàm bạn

Operator Overloading and Friend Functions

Mục tiêu

- Nạp chồng toán tử cơ bản
  - Toán tử hai ngôi (binary operators)
  - Toán tử một ngôi (unary operators)
  - Nạp chồng bằng hàm thành viên
- Hàm bạn và Lớp bạn

Lớp Money

```cpp
1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  //Class for amounts of money in U.S. currency
6  class Money
7  {
8  public:
9      Money( );
10     Money(double amount);
11     Money(int theDollars, int theCents);
12     Money(int theDollars);
13     double getAmount( ) const;
14     int getDollars( ) const;
15     int getCents( ) const;
16     friend const Money operator +(const Money& amount1, const Money& amount2)
17     friend const Money operator -(const Money& amount1, const Money& amount2)
18     friend bool operator ==(const Money& amount1, const Money& amount2);
19     friend const Money operator -(const Money& amount);
20     friend ostream& operator <<(ostream& outputStream, const Money& amount);
21     friend istream& operator >>(istream& inputStream, Money& amount);
22 private:
23     int dollars; //A negative amount is represented as negative dollars and
24     int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
25     int dollarsPart(double amount) const;
26     int centsPart(double amount) const;
27     int round(double number) const;
28  };
```

Giới thiệu nạp chồng toán tử

- Những toán tử như **+,-, %, ==** etc. thực ra là những hàm!
- Các hàm đặc biệt này được gọi với cú pháp khác so với cách gọi hàm thông thường
  - Gọi hàm thông thường:
    Tên_Hàm (Danh_Sách_Đối_Số)
  - Với toán tử: ví dụ, **x + 7**, "+" là một toán tử 2 ngôi (binary operator) với x, 7 là 2 toán hạng (operands)
- Thử viết theo cách gọi hàm thông thường: ***+(x,7)***
  - "+" là tên hàm
  - x, 7 là tham số của hàm
  - Hàm "+" trả lại giá trị là tổng của 2 đối số

Tại sao dùng nạp chồng toán tử?

- Những toán tử được xây dựng sẵn
  - Ví dụ, **+, -, = , %, ==, /, ***
  - Đã thao tác được với các kiểu dựng sẵn của C++
- Nhưng liệu chúng ta có thể thực hiện phép **+** với 2 đối tượng của lớp *Money?*, giống như:

  money1 + money2;
- **Để làm được điều này, chúng ta phải nạp chồng những toán tử này cho lớp *Money!***

Cơ bản về nạp chồng

- Nạp chồng toán tử
  - Tương tự như với nạp chồng hàm
  - Toán tử bản thân nó là tên của hàm
- Ví dụ khai báo

const Money operator + *(const Money& amount1,*
const Money& amount2);

  - Nạp chồng toán tử + với toán hạng là đối tượng kiểu Money
  - Giá trị trả lại là một kiểu *Money*
  - Mục đích: cho phép thực hiện phép + trên hai đối tượng của lớp Money

Nạp chồng toán tử "+"

```
const Money operator + (const Money& amount1,
                        const Money& amount2);
```

- Chú ý: hàm nạp chồng toán tử "+" này **không phải** hàm thành viên của lớp Money
- Định nghĩa, cài đặt của hàm này phức tạp hơn so với phép cộng thông thường (phải tính đến biến thành viên, kiểm tra giá trị âm/dương, …)

Định nghĩa nạp chồng toán tử "+" cho lớp **Money**

```
52  const Money operator +(const Money& amount1, const Money& amount2)
53  {
54      int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
55      int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
56      int sumAllCents = allCents1 + allCents2;
57      int absAllCents = abs(sumAllCents); //Money can be negative.
58      int finalDollars = absAllCents/100;
59      int finalCents = absAllCents%100;

60      if (sumAllCents < 0)
61      {
62          finalDollars = -finalDollars;
63          finalCents = -finalCents;
64      }

65      return Money(finalDollars, finalCents);
66  }
```

If the **return** *statements puzzle you, see the tip entitled* ***A Constructor Can Return an Object.***

Nạp chồng toán tử "=="

- Toán tử so sánh bằng "=="
  - Cho phép so sánh các đối tượng của lớp *Money*
  - Khai báo:

    bool operator ==(const Money& amount1,
    const Money& amount2);
  - Hàm này cũng không phải là hàm thành viên của lớp *Money*

```
83  bool operator ==(const Money& amount1, const Money& amount2)
84  {
85      return ((amount1.getDollars( ) == amount2.getDollars( ))
86             && (amount1.getCents( ) == amount2.getCents( )));
87  }
```

Nạp chồng toán tử một ngôi (Unary operators)

- Toán tử một ngôi: chỉ có một toán hạng
  - Toán tử phủ định (negation) “-”

    X = -Y // đặt X bằng giá trị phủ định của Y
  - Toán tử tăng ++
  - Toán tử giảm --

Nạp chồng toán tử "-" cho lớp **Money**

- Khai báo hàm nạp chồng toán tử "-" cho lớp Money

  const Money operator –(const Money& amount);

  - Không phải hàm thành viên của lớp
  - Chú ý: chỉ có một đối số, do toán tử này chỉ có một toán hạng
- Định nghĩa hàm nạp chồng toán tử một ngôi "-"

```
const Money operator –(const Money& amount)
{
        return Money(-amount.getDollars(), -amount.getCents());
}
```

- Trả lại một đối tượng vô danh (anonymous object)
- Lưu ý: nạp chồng toán tử "-" có hai trường hợp!
  - Khi nó là toán tử 2 ngôi, với 2 toán hạng/đối số
  - Khi nó là toán tử 1 ngôi, với 1 toán hạng/đối số

Sử dụng nạp chồng toán tử "-"

- Xét ví dụ sau:

Money amount1(10), amount2(6), amount3;
amount3 = amount1 – amount2;

=> Gọi nạp chồng toán tử 2 ngôi "-"

amount3 = -amount1;

=> Gọi hàm nạp chồng toán tử 1 ngôi "-"

Nạp chồng toán tử như hàm thành viên (1/2)

- Những ví dụ ở trước: các hàm đứng độc lập không phải thành viên của lớp
- Có thể nạp chồng như "toán tử thành viên", được xem như hàm thành viên
- Khi toán tử là hàm thành viên
  - Chỉ có **MỘT tham số**, không phải có 2 tham số!
  - Được tượng được gọi (phía sau toán tử) được xem là tham số duy nhất

Nạp chồng toán tử như hàm thành viên (2/2)

- Ví dụ:

  Money cost(1, 50), tax(0, 15), total;

  total = cost + tax;

- Nếu toán tử "+" được nạp chồng như toán tử thành viên thì:
  - Biến/ đối tượng **cost** là đối tượng gọi hàm nạp chồng
  - Đối tượng **tax** là tham số duy nhất của hàm nạp chồng
  - Tưởng tượng giống như cách viết sau

    total = cost.+(tax);

- Khai báo của toán tử "+" trong định nghĩa lớp

  const Money operator +(const Money& amount);

- Chú ý CHỈ CÓ MỘT đối số

Nạp chồng một số toán tử khác

- Toán tử gọi hàm: **()**
- Toán tử **&**, **||**, *dấu phẩy*
- Toán tử gán **=** (assignment operator), phải được nạp chồng như hàm thành viên!
- Toán tử tăng, giảm: **++, --**
  - Mỗi toán tử có 2 phiên bản:
    - Tiền tố (prefix notation): ++x;
    - Hậu tố (postfix notation): x++;
- Toán tử mảng **[ ]**, nạp chồng như hàm thành viên!
- Toán tử **>>**, **<<**

Nạp chồng toán tử >> và <<

- Cho phép nhập và xuất dữ liệu cho đối tượng
- Tăng tính dễ đọc cho chương trình
- Ví dụ chúng ta sẽ viết:

```
cout << myObject;
cin >> myObject;
```

  - Thay vì phải viết:

```
myObject.output();
myObject.input();
```

Toán tử chèn << (1/2) (Insertion operator)

- Được sử dụng với *cout*, ví dụ: *cout* << "*Hello*";
- Là toán tử hai ngôi:
  - Toán hạng đầu tiên là đối tượng được định nghĩa sẵn *cout*, từ thư viện *iostream*
  - Toán hạng thứ hai là dữ liệu/đối tượng cần in ra màn hình
- Giả sử khai báo: *Money amount(100);*
- Nếu chúng ta đã nạp chồng toán tử << với lớp Money, chúng ta có thể viết:

  cout << "I have " << amount << endl;

  thay vì sử dụng hàm thành viên output() và viết:

  cout << "I have ";
  amount.output()

Toán tử chèn << (2/2) (Insertion operator)

- Nạp chồng << nên trả về giá trị
- Giá trị nào được trả về?
  - Đối tượng *cout* !
  - Trả về kiểu của đối số đầu tiên, ***ostream***
- Hai cách viết sau là tương đương:

```
cout << "I have " << amount;
(cout << "I have ") << amount;
```

Chương trình nạp chồng toán tử << và >> (1/5)

Display 8.5 Overloading << and >>

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <cmath>
4  using namespace std;

5  //Class for amounts of money in U.S. currency
6  class Money
7  {
8  public:
9      Money( );
10     Money(double amount);
11     Money(int theDollars, int theCents);
12     Money(int theDollars);
13     double getAmount( ) const;
14     int getDollars( ) const;
15     int getCents( ) const;
16     friend const Money operator +(const Money& amount1, const Money& amount2)
17     friend const Money operator -(const Money& amount1, const Money& amount2)
18     friend bool operator ==(const Money& amount1, const Money& amount2);
19     friend const Money operator -(const Money& amount);
20     friend ostream& operator <<(ostream& outputStream, const Money& amount);
21     friend istream& operator >>(istream& inputStream, Money& amount);
22 private:
23     int dollars; //A negative amount is represented as negative dollars and
24     int cents; //negative cents. Negative $4.50 is represented as -4 and -50.
```

Chương trình nạp chồng toán tử << và >> (2/5)

```
25      int dollarsPart(double amount) const;
26      int centsPart(double amount) const;
27      int round(double number) const;
28  };

29  int main( )
30  {
31      Money yourAmount, myAmount(10, 9);
32      cout << "Enter an amount of money: ";
33      cin >> yourAmount;
34      cout << "Your amount is " << yourAmount << endl;
35      cout << "My amount is " << myAmount << endl;
36
37      if (yourAmount == myAmount)
38          cout << "We have the same amounts.\n";
39      else
40          cout << "One of us is richer.\n";

41      Money ourAmount = yourAmount + myAmount;
```

Chương trình nạp chồng toán tử << và >> (3/5)

Display 8.5 Overloading << and >>

```
42      cout << yourAmount << " + " << myAmount
43           << " equals " << ourAmount << endl;

44      Money diffAmount = yourAmount - myAmount;
45      cout << yourAmount << " - " << myAmount
46           << " equals " << diffAmount << endl;

47      return 0;
48  }
```

Since << returns a reference, you can chain << like this. You can chain >> in a similar way.

<Definitions of other member functions are as in Display 8.1. Definitions of other overloaded operators are as in Display 8.3.>

```
49  ostream& operator <<(ostream& outputStream, const Money& amount)
50  {
51      int absDollars = abs(amount.dollars);
52      int absCents = abs(amount.cents);
53      if (amount.dollars < 0 || amount.cents < 0)
54          //accounts for dollars == 0 or cents == 0
55          outputStream << "$-";
56      else
57          outputStream << '$';
58      outputStream << absDollars;
```

In the **main** *function,* **cout** *is plugged in for* **outputStream**.

For an alternate input algorithm, see Self-Test Exercise 3 in Chapter 7.

Chương trình nạp chồng toán tử << và >> (4/5)

```
59      if (absCents >= 10)
60          outputStream << '.' << absCents;
61      else
62          outputStream << '.' << '0' << absCents;

63      return outputStream;
64  }
65
66  //Uses iostream and cstdlib:
67  istream& operator >>(istream& inputStream, Money& amount)
68  {
69      char dollarSign;
70      inputStream >> dollarSign; //hopefully
71      if (dollarSign != '$')
72      {
73          cout << "No dollar sign in Money input.\n";
74          exit(1);
75      }

76      double amountAsDouble;
77      inputStream >> amountAsDouble;
78      amount.dollars = amount.dollarsPart(amountAsDouble);
```

Returns a reference

In the **main** *function,* **cin** *is plugged in for* **inputStream**.

Since this is not a member operator, you need to specify a calling object for member functions of **Money**.

(continued)

Chương trình nạp chồng toán tử << và >> (5/5)

Display 8.5 **Overloading << and >>**

```
79        amount.cents = amount.centsPart(amountAsDouble);

80        return inputStream;
81    }
```

Returns a reference

SAMPLE DIALOGUE

Enter an amount of money: **$123.45**
Your amount is $123.45
My amount is $10.09.
One of us is richer.
$123.45 + $10.09 equals $133.54
$123.45 - $10.09 equals $113.36

Hàm bạn
(Friend functions)

- Nhớ lại:
  - Nạp chồng toán tử có thể không phải hàm thành viên
  - Khi đó, truy xuất dữ liệu phải thông qua các hàm *accessor* và *mutator*
  - Cách làm này không hiệu quả (tăng phụ phí khi gọi các hàm này!)
- Hàm bạn *Không phải* hàm thành viên của lớp nhưng có thể truy xuất trực tiếp đến các dữ liệu trong khu vực *private* của lớp
  - Không có phụ phí khi gọi hàm => hiệu quả hơn
- Vì vậy: cách tốt nhất là cài đặt nạp chồng toán tử là khai báo chúng như các hàm bạn
- Sử dụng từ khóa **friend** ở trước khai báo hàm

Lớp bạn
(Friend classes)

- Toàn bộ một lớp có thể là bạn của một lớp khác
  - Tương tự như một hàm là bạn trong một lớp
- Nếu lớp F là bạn của lớp C => tất cả hàm thành viên của lớp F đều là bạn của lớp C
  - Điều ngược lại không đúng
- Cú pháp: *friend class F*

Tóm tắt nạp chồng toán tử và hàm bạn

- Những toán tử dựng sẵn (built-in) trong C++ có thể được nạp chồng để thao tác với đối tượng của lớp mà bạn định nghĩa
- Toán tử thực ra là những hàm!
- Toán tử có thể được nạp chồng như hàm ngoài (không phải thành viên) hoặc hàm thành viên của lớp
  - Toán hạng đầu tiên là đối tượng gọi
- Hàm bạn truy xuất trực tiếp được các thành viên trong khu vực *private*

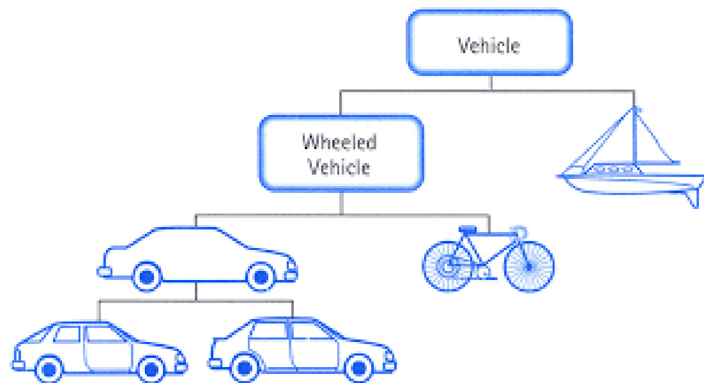
2. Kế thừa

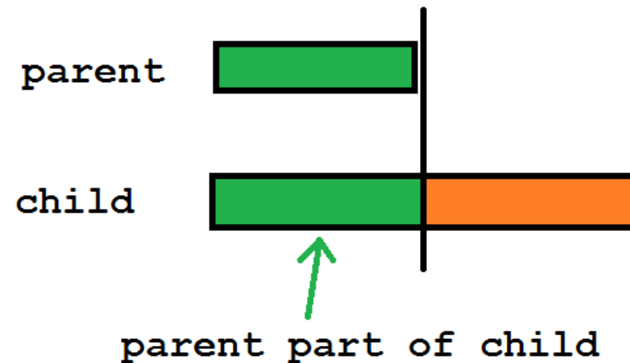
Inheritance

Mục tiêu

- Cơ bản về kế thừa (inheritance)
  - Lớp thừa kế (derived classes), với hàm tạo
  - Khu vực *Protected*
  - Định nghĩa lại hàm thành viên
  - Hàm không kế thừa
- Chương trình với kế thừa
  - Toán tử gán và hàm tạo
  - Đa kế thừa (multiple inheritance)

Giới thiệu về kế thừa





- Thế nào là kế thừa? Định nghĩa?
- Một kỹ thuật lập trình mạnh, khái niệm trừu tượng
- Cấu trúc tổng quát về một khái niệm được định nghĩa trong một lớp (lớp cha / lớp cơ sở)
  - Những phiên bản chuyên biệt (lớp con) sau đó kế thừa thuộc tính của lớp tổng quát đó
  - Lớp con có thể mở rộng hay thay đổi chức năng cho phù hợp

Cơ bản về kế thừa

- Một lớp mới được kế thừa từ một lớp khác
- Lớp cơ sở (lớp cha)
  - Lớp tổng quát mà từ đó các lớp khác sẽ kế thừa
- Lớp thừa kế (lớp con)
  - Một lớp mới
  - Tự động có những hàm/biến thành viên của lớp cơ sở
  - Sau đó có thể thêm những hàm/biến thành viên mới
- Thuật ngữ (terminology) về kế thừa giống như quan hệ gia đình
  - Lớp cha (Parent class) ~ Lớp cơ sở (Base class)
  - Lớp con (Child class) ~ Lớp thừa kế (Derived class)
  - Lớp tổ tiên (Ancestor class)
  - Lớp con cháu (Descendant class)

Lớp thừa kế (Derived classes)

- Xét ví dụ về lớp *Nhân_Viên (Employee)*
- Có thể bao gồm nhiều loại nhỏ:
  - Nhân viên được trả lương (*Salaried employees*)
  - Nhân viên bán thời gian, theo giờ (*Hourly employees*)
  - …
- Khái niệm tổng quát về *Nhân_Viên* là hữu ích! Được định nghĩa trước như một khung chung:
  - Tất cả nhân viên đều có những thông tin chung như: *Tên, Tuổi, Giới Tính, Quốc Tịch, CMTND*
  - Các hàm thành viên liên quan đến những dữ liệu này là giống nhau (cơ sở) cho tất cả nhân viên
    - Ví dụ: các hàm *accessor, mutator*

Định nghĩa lại hàm thành viên

- Xét hàm *printCheck()* của lớp cơ sở Nhân_Viên
  - Được định nghĩa lại trong các lớp thừa kế
  - Do các loại nhân viên khác nhau có thể có các kiểm tra khác nhau

Giao diện cho lớp thừa kế **HourlyEmployee** (1/2)

Display 14.3 Interface for the Derived Class HourlyEmployee

```
1
2   //This is the header file hourlyemployee.h.
3   //This is the interface for the class HourlyEmployee.
4   #ifndef HOURLYEMPLOYEE_H
5   #define HOURLYEMPLOYEE_H

6   #include <string>
7   #include "employee.h"

8   using std::string;

9   namespace SavitchEmployees
10  {
```

Giao diện cho lớp thừa kế **HourlyEmployee** (2/2)

```
11      class HourlyEmployee : public Employee
12      {
13      public:
14          HourlyEmployee( );
15          HourlyEmployee(string theName, string theSsn,
16                                   double theWageRate, double theHours);
17          void setRate(double newWageRate);
18          double getRate( ) const;
19          void setHours(double hoursWorked);
20          double getHours( ) const;
21          void printCheck( ) ;
22      private:
23          double wageRate;
24          double hours;
25      };

26  }//SavitchEmployees

27  #endif //HOURLYEMPLOYEE_H
```

You only list the declaration of an inherited member function if you want to change the definition of the function.

Giao diện lớp **HourlyEmployee**

- Lưu ý phần đầu chương trình
  - Cấu trúc #ifndef
  - Khai báo bao gồm (include) các thư viện liên quan
  - Khai báo bao gồm lớp cơ sở ***employee.h***!
- Khai báo cấu trúc kế thừa

  class HourlyEmployee : public Employee
- Giao diện (interface) của lớp thừa kế chỉ liệt kê những thành viên mới hoặc sẽ được định nghĩa lại
  - Bởi vì tất cả những thành viên khác kế thừa từ lớp cơ sở đã được định nghĩa trước đó!
- Lớp HourlyEmployee thêm các thành viên sau:
  - Hàm khởi tạo
  - Biến thành viên: *wageRate, hours*
  - Hàm thành viên: *setRate(), getRate(), setHours(), getHours()*

Định nghĩa lại hàm thành viên trong lớp **HourlyEmployee**

- Lớp HourlyEmployee định nghĩa lại:
  - Hàm thành viên *printCheck*() của lớp cơ sở
  - Phiên bản mới của hàm *printCheck*() sẽ "ghi đè" (overrides) phiên bản cũ đã được cài đặt trong lớp cơ sở Employee
- Cài đặt của hàm thành viên này phải được thực hiện trong lớp *HourlyEmployee*
- **Định nghĩa lại hàm khác nạp chồng hàm thế nào?**
  - Rất khác nhau
  - Định nghĩa lại hàm trong lớp thừa kế
    - CÙNG danh sách tham số
    - Thực chất là viết lại cùng một hàm
  - Nạp chồng hàm
    - Danh sách tham số khác nhau
    - Định nghĩa một hàm mới với tham số khác

Truy xuất hàm định nghĩa lại

- Khi được định nghĩa lại một hàm trong lớp con, định nghĩa của hàm này trong lớp cơ sở không bị mất đi!

Employee JaneE;
HourlyEmployee SallyH;
JaneE.printCheck();//gọi hàm printCheck của lớp
Employee
SallyH.printCheck();//gọi hàm printCheck của lớp
HourlyEmployee
SallyH.Employee::printCheck();//gọi hàm printCheck của
lớp Employee!

Hàm tạo trong lớp thừa kế (1/2)

- Hàm tạo của lớp cơ sở không được kế thừa tự động trong lớp con !
  - Nhưng chúng có thể được gọi bên trong hàm tạo của của lớp con!
- Hàm tạo của lớp cơ sở nên khởi tạo tất cả các biến thành viên
- Xét ví dụ hàm tạo của lớp *HourlyEmployee*

```
HourlyEmployee::HourlyEmployee(string theName, string
theNumber, double theWageRate, double theHours)
        : Employee(theName, theNumber),
        wageRate(theWageRate), hours(theHours)
{}
```

Hàm tạo trong lớp thừa kế (2/2)

- Nếu lớp con không gọi hàm tạo nào của lớp cơ sở:
  - Hàm tạo mặc định của lớp cơ sở tự động được gọi
- Ví dụ:

```
HourlyEmployee::HourlyEmployee()
                    : wageRate(0), hours(0)
{ }
```

Lưu ý: dữ liệu private của lớp cơ sở

- Lớp con kế thừa biến thành viên trong khu vực *private*
  - Nhưng vẫn không thể truy xuất trực tiếp "theo tên" (by-name) đến những biến thành viên này
  - Ngay cả truy xuất biến *private* thông qua các hàm thành viên của lớp con!
- Biến thành viên *private* có thể CHỈ được truy xuất "theo tên" trong các hàm thành viên của lớp cơ sở mà chúng được định nghĩa!

Lưu ý: hàm thành viên private của lớp cơ sở

- Không thể được truy xuất bên ngoài giao diện và cài đặt của lớp cơ sở
- Ngay cả trong định nghĩa hàm thành viên của lớp con

Khu vực protected

- Một khu vực mới cho thành viên của lớp
- Cho phép truy xuất "theo tên" thành viên trong lớp thừa kế
  - Nhưng không cho phép truy xuất trong các lớp không kế thừa!
- Trong lớp mà những thành viên protected này được định nghĩa, hoạt động giống như các thành viên private

Đa kế thừa
(Multiple inheritance)

- Lớp con có thể kế thừa nhiều hơn một lớp cơ sở!
  - Cú pháp: các lớp cơ sở được phân tách bằng dấu phẩy
  - Ví dụ:

class derivedMulti : public base1, base2 {…}

Bài tập

- Định nghĩa lớp *Nhân viên (Employee)*
  - Private: *Tên, Tuổi, Giới Tính, Quốc Tịch*
  - Public: void *printCheck*()
  - Protected: *CMTND*
- Định nghĩa hai lớp con kế thừa từ lớp *Nhân_Viên*
  - Nhân viên được trả lương (*SalariedEmployee*)
  - Nhân viên bán thời gian, theo giờ (*HourlyEmployee*)
  - Định nghĩa lại hàm *printCheck*() riêng của hai lớp con

Tóm tắt về kế thừa

- Kế thừa cho phép sử dụng lại code
  - Cho phép một lớp kế thừa từ lớp khác và thêm các chức năng mới
- Lớp con kế thừa những thành viên của lớp cơ sở và có thể thêm thành viên mới
- Biến thành viên private trong lớp cơ sở không thể được truy xuất "theo tên" trong lớp con
- Hàm thành viên private không được kế thừa, chỉ được sử dụng riêng ở lớp cơ sở
- Có thể định nghĩa lại hàm thành viên của lớp cơ sở trong lớp con
  - Các lớp con khác nhau có thể có những định nghĩa khác nhau
- Thành viên trong khu vực protected của lớp cơ sở có thể được truy xuất "theo tên" trong lớp con

Giáo trình Tham khảo

- **Giáo trình chính: W. Savitch, *Absolute* C++, Addison Wesley, 2002**
- Tham khảo:
  - A. Ford and T. Teorey, *Practical Debugging in C++*, Prentice Hall, 2002
  - Nguyễn Thanh Thủy, *Kĩ thuật lập trình C++*, NXB Khoa học và Kĩ Thuật, 2006