TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN ĐIỆN TỬ VIỄN THÔNG

BÀI BÁO CÁO

TÌM HIỂU KHÁI NIỆM VỀ CLASS, KẾ THỪA, ĐA NHIỆM

HỌ VÀ TÊN: HOÀNG MINH SƠN

MSSV: 20193081

LỚP: ĐIỆN TỬ 06- K64

CHÍ LINH, NGÀY THÁNG NĂM 2020

MỤC LỤC

1. KHÁI NIỆM VỀ CLASS, ĐỐI TƯỢNG

1.1 Đối tượng :

1.1.1 Đối tượng là gì:

- Mô hình đại diện của một đối tượng vật lý:
  - ♦ Person, students, employee…
  - ♦ Car, bus, verhicle…
- Đối tượng logic:
  - o Trend, report, button, window…

1.1.2 Thành phần của một đối tương:

- Các thuộc tính
- Trạng thái
- Hành vi
- Căn cước
- Ngữ nghĩa

1.2 Lớp:

1.2.1 Khái niệm về lớp:

- Là sự thực thi của các đối tượng có chung các thuộc tính, hành vi, quan hệ, ngữ nghĩa.
- Lớp là một kiểu dữ liệu mới có cấu trúc, trong đó việc truy nhập các biến thành viên được kiểm soát thông qua các hàm thành viên
- Các dữ liệu của lớp => Biến thành viên
- Các hàm của lớp => hàm thành viên
- Một biến của lớp => một đối tượng

```
class Box{// Ten lop
    double chieudai, chieurong, chieucao;// bien thanh vien
    public: // quyen truy cap
        void setBox( double dai, double rong, double cao)// ham thanh vien
        {
            chieudai = dai;
            chieurong = rong;
            chieucao = cao;
        }
}
```

Ví dụ về đối tượng trong C++:

```
void main()
{
    Box box1;// doi tuong box1 thuoc lop Box.
    Box box2;// doi tuong box2 thuoc lop Box.
}
```

1.2.2 Truy cập thành viên dữ liệu

Các thành viên dữ liệu public của các đối tượng của một lớp có thể được truy cập bởi sử dụng toán tử truy cập thành viên trực tiếp là dấu chấm.

```
void main()
{
    Box box1;// doi tuong box1 thuoc lop Box.
    Box box2;// doi tuong box2 thuoc lop Box.

    double theTich;
    // Khai bao cac thuoc tinh cua box1
    box1.chieudai = 2;// có the truy cap truc tiep bien thanh vien vi de quyen
truy cap la public
    box1.chieurong = 3;
    box1.chieucao = 4;
    //the tich cua box 1
    theTich = box1.chieudai * box1.chieurong * box1.chieucao;
    cout <<"The tich cua box1 la: "<<theTich;
}
```

1.2.3 Hàm thành viên

Một hàm thành viên của một lớp là một hàm mà có định nghĩa hoặc prototype của nó bên trong định nghĩa lớp giống như bất kỳ biến nào khác. Nó hoạt động trên bất kỳ đối tượng nào của lớp mà nó là một thành viên, và có sự truy cập tới tất cả thành viên của một lớp cho đối tượng đó.

Chúng ta truy cập các thành viên của lớp bởi sử dụng một hàm thành viên thay vì trực tiếp truy cập chúng:

Ví dụ: Các hàm thành viên có thể được định nghĩa bên trong định nghĩa lớp hoặc sử dụng **toán tử phân giải phạm vi ::**. Định nghĩa một hàm thành viên bên trong định nghĩa lớp sẽ khai báo hàm **inline**, ngay cả khi bạn không sử dụng inline specifier.

```cpp
class box
{
  public:
  double chieudai, chieurong, chieucao;
  //khai bao ham thanh vien
  double theTich(void);
  void setChieuDai(double dai);
  void setChieuRong(double rong)
  {
    chieurong = rong;//dinh nghia ham thanh vien ngay trong lop
  }
  void setChieuCao(double cao);
};
// dinh nghia cac ham thanh vien ben ngoai lop bang toan tu phan giai pham vi :
:
void box::setChieuDai(double dai)
{
  chieudai = dai;
}
void box::setChieuCao(double cao)
{
  chieucao = cao;
}
double box::theTich(void)
{
  return chieudai*chieurong*chieucao;
}

//ham main cua chuong trinh

int main()
{
  box box1, box2;
  double thetich;
  //thong tin ve box1:
  box1.setChieuDai(2.1);
  box1.setChieuRong(2.5);
  box1.setChieuCao(3.4);
  //the tich box1:
  thetich = box1.theTich();
  cout<<"The tich cua box1 la: "<<thetich;
  return 0;
}
```

1.2.4 Access Modifier (giới hạn truy cập):

Data Hiding là một trong những đặc điểm quan trọng của Lập trình hướng đối tượng mà cho phép ngăn cản hàm của một chương trình truy cập trực tiếp tới biểu diễn nội vi của một kiểu lớp. Giới hạn truy cập tới các thành viên lớp được xác định bởi các khu vực **public, private** và **protected** được gán nhãn bên trong thân lớp. Từ khóa public, private và protected được gọi là Access Specifier.

Một lớp có thể có nhiều khu vực public, private và protected được gán nhãn. Mỗi khu vực vẫn còn giữ hiệu quả tới khi gặp: hoặc nhãn khu vực khác hoặc dấu ngoặc đóng của thân lớp. Access Specifier mặc định cho các thành viên và các lớp là private.

```
class Base {

   public:

  // tai day la cac thanh vien public

   protected:

  // tai day la cac thanh vien protected

   private:

  // tai day la cac thanh vien private

};
```

- **Thành viên public trong C++:**

Thành viên **public** là có thể truy cập từ bất cứ đâu bên ngoài lớp nhưng là bên trong một chương trình. Bạn có thể thiết lập và lấy giá trị của các biến public mà không cần các hàm thành viên, như ví dụ sau:

```
class HCN
{
  public:
  int chieuDai, chieuRong;
  void setChieuDai(int dai);
  int laychieuDai(void);
};
```

```
void HCN::setChieuDai(int dai)
{
  chieuDai = dai;
}
int HCN::laychieuDai(void)
{
  return chieuDai;
}


int main()
{
  HCN hcn1;
  // thiet lap chieu dai cho hcn1:
  hcn1.setChieuDai(3);
  // thiet lap chieu rong:
  hcn1.chieuRong = 2;// dieu nay la ok vi chieuRong la public.
  cout<<"chieu dai hcn1 la: "<<hcn1.laychieuDai();
  return 0;
}
```

- **Thành viên private trong C++:**

Một biến hoặc một hàm thành viên **private** trong C/C++ không thể được truy cập, hoặc được quan sát từ bên ngoài lớp. Chỉ có lớp và các hàm friend có thể truy cập các thành viên private trong C/C++.

Theo mặc định, tất cả thành viên của một lớp sẽ là private, ví dụ: trong lớp sau, *chieurong* là một thành viên private, nghĩa là, tời khi bạn gán nhãn một thành viên, nó sẽ được xem như là một thành viên private.

```
class Box
{
   double chieurong;
   public:
      double chieudai;
      void setChieuRong( double rong );
      double layChieuRong( void );
};
```

Thực tế, chúng ta định nghĩa dữ liệu trong khu vực private và các hàm liên quan trong khu vực public để mà chúng có thể được gọi từ bên ngoài lớp.

- **Thành viên protected trong C++:**

Một biến hoặc một hàm thành viên **protected** là khá tương tự như một thành viên private, nhưng nó cung cấp một lợi ích bổ sung là chúng có thể được truy cập trong các lớp con, mà được gọi là các lớp kế thừa.

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

class Box
{
   protected:
      double chieurong;
};

class SmallBox:Box // SmallBox la mot lop ke thua cua Box.
{
   public:
      void setChieuRongCon( double rong );
      double layChieuRongCon( void );
};

// phan dinh nghia cac ham thanh vien cua lop con
double SmallBox::layChieuRongCon(void)
{
    return chieurong ;
}

void SmallBox::setChieuRongCon( double rong )
{
    chieurong = rong;
}

// ham main cua chuong trinh
int main( )
{
   SmallBox box;

   // thiet lap chieu rong cua Box boi su dung ham thanh vien
   box.setChieuRongCon(15.3);
   cout << "Do rong cua Box la: "<< box.layChieuRongCon() << endl;

   return 0;
}
```

Sau khi chạy chương trình ta sẽ được kết qủa:

```
Do rong cua Box la: 15.3
```

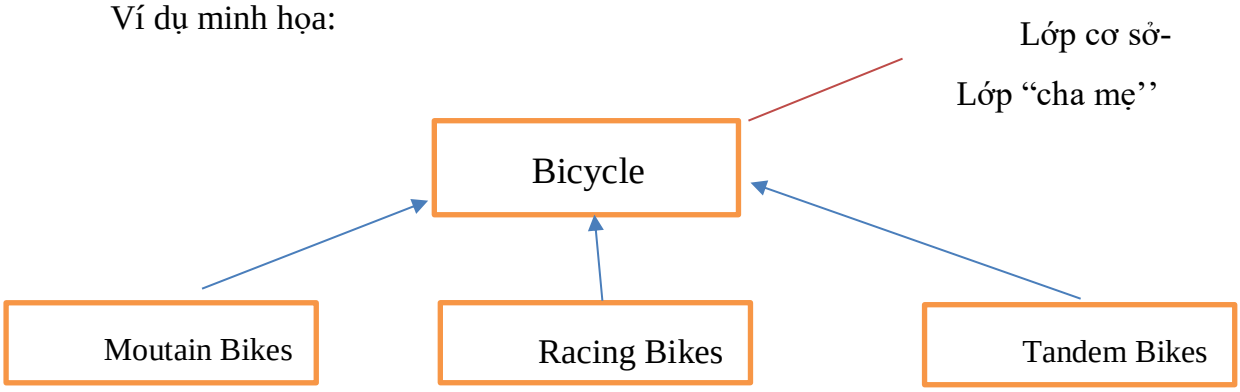
2. TÍNH KẾ THỪA & TÍNH ĐA NHIỆM

2.1 Tính kế thừa:

2.1.1 Khái niệm:

- Cơ chế dẫn xuất / thừa kế là một kỹ thuật lập trình hướng đối tượng cho phép chuyên biệt hóa
- Dân xuất cho phép tạo ra một lớp mới (lớp dẫn xuất) của các đối tượng bằng cách sử dụng các lớp cũ như là lớp cơ sở
    - Lớp dẫn xuất thừa hướng các thuộc tính và hành vi của lớp "cha-mẹ"
    - Lớp dẫn xuất là một phiên bản chuyên biệt hóa của lớp "cha-mẹ"

Ví dụ minh họa:



Lớp dẫn xuất

Một lớp có thể được kế thừa từ hơn một lớp khác, nghĩa là, nó có thể kế thừa dữ liệu và hàm từ nhiều lớp cơ sở. Để định nghĩa một lớp kế thừa (Derived Class), chúng ta sử dụng một danh sách để xác định các lớp cơ sở. Danh sách này liệt kê một hoặc nhiều lớp cơ sở và có form sau:

```
class lop_ke_thua: access_modifier lop_co_so
```

Ở đây, *access_modifier* là **public, protected** hoặc **private**, và *lop_co_so* là tên của lớp đã được định nghĩa trước đó. Nếu access_modifier không được sử dụng, thì mặc định là private.

Ví dụ sau với **Hinh** là lớp cơ sở và **HinhChuNhat** là lớp kế thừa:

```
#include <iostream>
```

```
using namespace std;

// lop co so: Hinh
class Hinh
{
   public:
      void setChieuRong(int rong)
      {
         chieurong = rong;
      }
      void setChieuCao(int cao)
      {
         chieucao = cao;
      }
   protected:
      int chieurong;
      int chieucao;
};

// day la lop ke thua: HinhChuNhat
class HinhChuNhat: public Hinh
{
   public:
      int tinhDienTich()
      {
         return chieurong * chieucao;
      }
};

int main(void)
{
   HinhChuNhat Hcn;

   Hcn.setChieuRong(14);
   Hcn.setChieuCao(30);

   // in dien tich cua doi tuong.
   cout << "Tong dien tich la: " << Hcn.tinhDienTich() << endl;

   return 0;
}
```

Kết quả khi chạy chương trình:

```
Tong dien tich la: 420
```

2.1.2 Điều khiển truy cập và tính thừa kế

Một lớp kế thừa có thể truy cập tất cả thành viên không phải là private của lớp cơ sở của nó. Vì thế, các thành viên lớp cơ sở, mà là hạn chế truy cập tới các hàm thành viên của lớp kế thừa, nên được khai báo là private trong lớp cơ sở.

Chúng ta tổng kết các kiểu truy cập khác nhau, tương ứng với ai đó có thể truy cập chúng như sau:

Truy cập	**Public**	**Protected**	**Private**
Trong cùng lớp	Có	Có	Có
Lớp kế thừa	Có	Có	Không
Bên ngoài lớp	Có	Không	Không

Một lớp kế thừa (Derived Class) sẽ kế thừa tất cả các phương thức của lớp cơ sở, ngoại trừ:

- Constructor, destructor và copy constructor của lớp cơ sở.
- Overloaded operator (toán tử nạp chồng) của lớp cơ sở.
- Hàm friend của lớp cơ sở.

2.1.3 Kiểu kế thừa

Khi kế thừa từ một lớp cơ sở, lớp cơ sở đó có thể được kế thừa thông qua kiểu kế thừa là **public, protected** hoặc **private**. Kiểu kế thừa trong C++ được xác định bởi Access-specifier đã được giải thíc ở trên.

Chúng ta hiếm khi sử dụng kiểu kế thừa **protected** hoặc **private**, nhưng kiểu kế thừa **public** thì được sử dụng phổ biến hơn. Trong khi sử dụng các kiểu kế thừa khác sau, bạn nên ghi nhớ các quy tắc sau:

Kiểu kế thừa Public: Khi kế thừa từ một lớp cơ sở là **public**, thì các thành viên **public** của lớp cơ sở trở thành các thành viên **public** của lớp kế thừa; và các thành viên **protected** của lớp có sở trở thành các thành viên **protected** của lớp kế thừa. Một thành viên là **private** của lớp cơ sở là không bao giờ có thể được truy cập trực tiếp từ một lớp kế thừa, nhưng có thể truy cập thông qua các lời gọi tới các thành viên **public** và **protected** của lớp cơ sở đó.

Kiểu kế thừa protected: Khi kế thừa từ một lớp cơ sở là **protected**, thì các thành viên **public** và **protected** của lớp cơ sở trở thành các thành viên **protected** của lớp kế thừa

Kiểu kế thừa private: Khi kế thừa từ một lớp cơ sở là **private**, thì các thành viên **public** và **protected** của lớp cơ sở trở thành các thành viên **private** của lớp kế thừa.

2.1.4 Đa kế thừa

Một lớp trong C++ có thể kế thừa các thành viên từ nhiều lớp, và đây là cú pháp:

```
class lop_ke_thua: access_modifier lop_co_so_1, access_modifier lop_co_so_2 ...
```

Tại đây, access_modifier là **public, protected** hoặc **private** và sẽ được cung cấp cho mỗi lớp cơ sở, và chúng sẽ được phân biệt với nhau bởi dấu phảy như trên.

Ví dụ:

```
#include <iostream>

using namespace std;

// Lop co so: Hinh
class Hinh
{
   public:
      void setChieuRong(int rong)
      {
         chieurong = rong;
      }
      void setChieuCao(int cao)
      {
         chieucao = cao;
      }
   protected:
      int chieurong;
      int chieucao;
};

// Lop co so: ChiPhiSonMau
class ChiPhiSonMau
{
   public:
      int tinhChiPhi(int dientich)
      {
         return dientich * 300000;
      }
```

```
};

// day la lop ke thua: HinhChuNhat
class HinhChuNhat: public Hinh, public ChiPhiSonMau
{
   public:
      int tinhDienTich()
      {
         return chieurong * chieucao;
      }
};

int main(void)
{
   HinhChuNhat Hcn;
   int dientich;

   Hcn.setChieuRong(14);
   Hcn.setChieuCao(30);

   dientich = Hcn.tinhDienTich();

   // in dien tich cua doi tuong.
   cout << "Tong dien tich la: " << Hcn.tinhDienTich() << " m2." << endl;

   // in tong chi phi de son mau
   cout << "Tong chi phi de son mau la: " << Hcn.tinhChiPhi(dientich) << " VND.
" << endl;

   return 0;
}
```

Kết quả hiện thị:

```
Tong dien tich la: 420 m2.
Tong chi phi de son mau la : 126000000 VND.

------------------------------------------------------------
```

2.2 Tính đa hình:

2.2.1 Khái niệm:

Từ đa hình có nghĩa là nhiều hình dạng. Nói một cách đơn giản, chúng ta có thể định nghĩa đa hình là khả năng của một thông điệp được hiển thị dưới nhiều dạng.

Ví dụ thực tế:

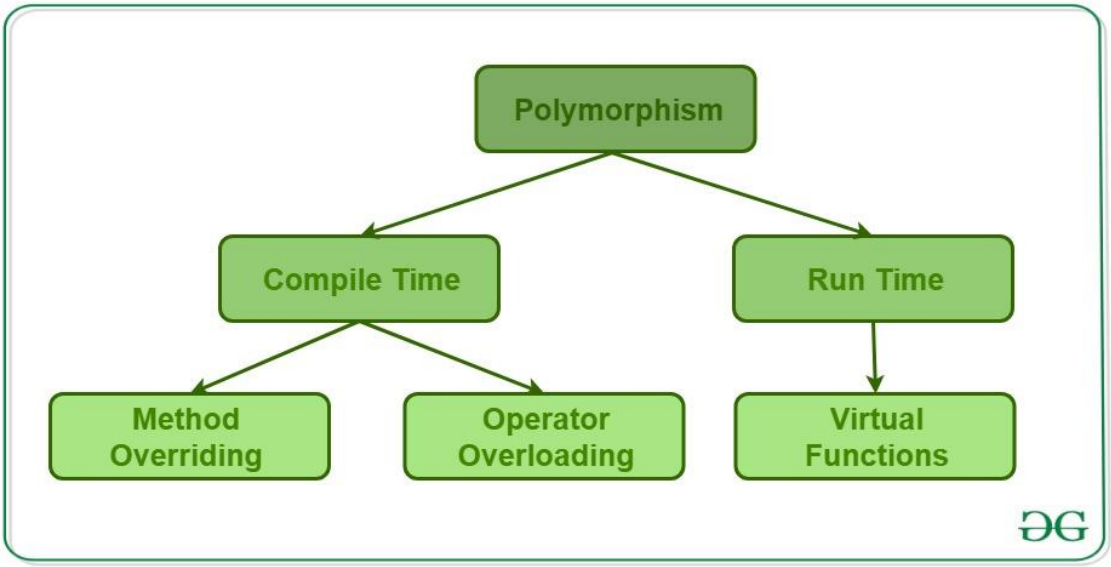
Một người cùng lúc có thể có đặc điểm khác nhau. Giống như một người đàn ông đồng thời là một người cha, một người chồng, một nhân viên. Vì vậy, cùng một người sở hữu những hành vi khác nhau trong các tính huống khác nhau. Điều này được gọi là đa hình.

Đa hình được coi là một trong những tính năng quan trọng của Lập trình hướng đối tượng.

2.2.2 Phân loại đa hình

Trong ngôn ngữ C++, tính đa hình chủ yếu được chia làm 2 loại:

- Compile time Polymorphism
- Runtime Polymorphism



2.2.2.1 Compile time Polymorphism:

Tính đa hình này được sử dụng bằng cách nạp chồng hàm hoặc nạp chồng toán tử.

Ví dụ:

Nạp chồng hàm:

```
class OOP
{
public:
    // Hàm có một tham số
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }
```

```
    // Hàm cùng tên có một tham số nhưng khác kiểu
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // Hàm cùng tên nhưng có 2 tham số
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};

int main()
{

    OOP obj;

    obj.func(7);
    obj.func(9.132);
    obj.func(85, 64);
    return 0;
}
```

Sau khi thực hiện được kết quả:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

Trong ví dụ trên, ta chỉ dùng một hàm duy nhất có tên là `func` nhưng có thể dùng được cho 3 tình huống khác nhau. Đây là một thể hiện của tính đa hình.

Nạp chồng toán tử

```
class SoPhuc
{
private:
    int thuc, ao;

public:
    SoPhuc(int thuc = 0, int ao = 0)
    {
        this->thuc = thuc;
        this->ao = ao;
    }
```

```
    ~SoPhuc()
    {
        this->thuc = 0;
        this->ao = 0;
    }

    SoPhuc operator+(SoPhuc const &obj)
    {
        SoPhuc res;
        res.thuc = thuc + obj.thuc;
        res.ao = ao + obj.ao;
        return res;
    }

    void print() { cout << this->thuc << " + " << this->ao << "i" << endl; }
};

int main()
{
    SoPhuc c1(10, 5), c2(2, 4);
    SoPhuc c3 = c1 + c2;
    c3.print();

    return 0;
}
```

Trong ví dụ trên, ta đã nạp chồng lại toán tử cộng.

Định nghĩa của toán tử cộng chỉ dùng cho số nguyên `int`, nhưng sau khi nạp chồng lại, ta có thể sử dụng chúng cho số phức.

Đây cũng là một thể hiện của tính đa hình.

2.2.2.2 Runtime Polyphism

Tính đa hình được thể hiện ở cách nạp chồng toán tử trong kế thừa.

```
class base
{
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }

    void show()
    {
```

```
        cout << "show base class" << endl;
    }
};

class derived : public base
{
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

    bptr->print();
    bptr->show();

    return 0;
}
```

Chương trình cho kết quả:

```
print derived class
show base class
```

Trong ví dụ trên mình đã thêm từ khóa `virtual` vào hàm `print()` trong lớp cơ sở `base`.

Từ khóa `virtual` này dùng để khai báo một hàm là hàm ảo.

Khi khai báo hàm ảo với từ khóa `virtual` nghĩa là hàm này sẽ được gọi theo loại đối tượng được trỏ (hoặc tham chiếu), chứ không phải theo loại của con trỏ (hoặc tham chiếu). Và điều này dẫn đến kết quả khác nhau:

- Nếu không khai báo hàm ảo `virtual` trình biên dịch sẽ gọi hàm tại lớp cở sở `base`

- Nếu dùng hàm ảo `virtual` trình biên dịch sẽ gọi hàm tại lớp dẫn xuất `derived`

Mục đích của hàm ảo:

Các hàm ảo sẽ cho phép chúng ta tạo một danh sách các con trỏ lớp cơ sở và các phương thức của bất kỳ lớp dẫn xuất nào mà không cần biết loại đối tượng của lớp dẫn xuất.

Mình lấy một ví dụ cụ thể nhé:

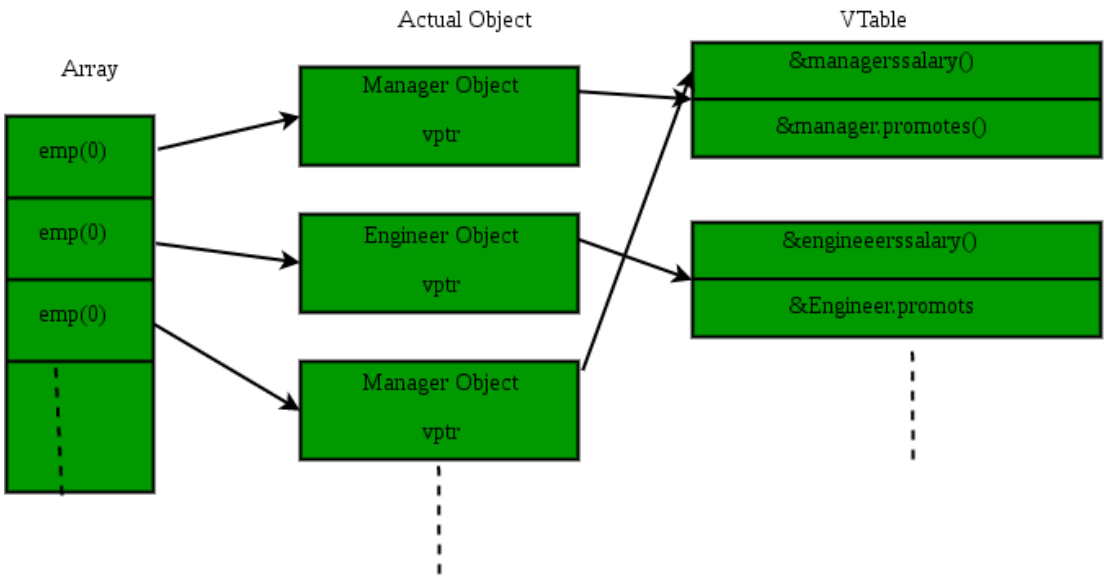
Ta sẽ bắt đầu với một phần mềm quản lý nhân viên.

Đầu tiên, ta sẽ xây dựng một lớp `Nhanvien` sau đó xây dựng các hàm ảo `tangluong()`, `chuyenphong()`, …Từ lớp Nhanvien này ta sẽ kế thừa tới các lớp `Baove`, `NhanvienphongA`, `NhanvienphongB`,.. và tất nhiên các lớp này có thể triển khai riêng biệt các hàm ảo có tại lớp cơ sở `Nhanvien`

Trình biên dịch sẽ thực hiện Runtime Polymorphism như thế nào?

Trình biên dịch sẽ duy trì:

- vtable: Đây là một bảng các con trỏ hàm được duy trì cho mỗi lớp
- vptr: Đây là một con trỏ tới `vtable` và được duy trì cho mỗi một đối tượng.



Trình biên dịch sẽ thêm code bổ sung tại 2 chỗ là:

Code trong mỗi hàm khởi tạo. Nó sẽ khởi tạo `vptr` của đối tượng được tạo và đặt `vptr` trỏ đến `vtable` của lớp.

Code với lệnh gọi hàm ảo. Tại bất cứ chỗ nào tính đa hình được thực hiện, trình biên dịch sẽ chèn code để tìm `vptr` trước bằng cách sử dụng con trỏ hoặc tham chiếu lớp cơ sở. Khi `vptr` được nạp, `vptable` của lớp dẫn xuất có thể được truy cập. Sử dụng `vtable`, địa chỉ của hàm ảo tại lớp dẫn xuất sẽ được truy cập và gọi.

Trên thực tế thì C++ không bắt buộc một `Runtime polymorphism` chạy chính xác như thế này. Nhưng trình biên dịch thường sử dụng các mô hình với biến thể nhỏ, dựa trên mô hình cơ bản bên trên.

Hàm Pure Virtual trong C++

Với `Pure Virtual` nghĩa là bạn chỉ dùng hàm ảo tại lớp cơ sở để khai báo, chứ không có bất kì câu lệnh nào bên trong hàm đó.

```
class base
{
public:
    virtual void print(); // Pure Virtual
    void show()
    {
        cout << "show base class" << endl;
    }
};

class derived : public base
{
public:
    void print()
    {
        cout << "print derived class" << endl;
    }

    void show()
    {
        cout << "show derived class" << endl;
    }
};

int main()
{
    base *bptr;
```

```
    derived d;
    bptr = &d;

    bptr->print();
    bptr->show();

    return 0;
}
```

Trong đoạn code trên, đã sửa hàm `print()` thành một `Pure Virtual` và tất nhiên kết quả vẫn không hề thay đổi.