Chapter 8 - Polymorphism

Polimorfisme

Secara teknis polimorfisme merupakan suatu konsep untuk merelasikan diatara kelas-kelas C++ melalui overriding metode-metode virtual, sehingga dengan demikian satu tipe kelas dapat konversikan menjadi kelas lain. Aspek penting pertama dalam Pewarisan (*Inheritance*) adalah pengoganisasian kelas yang mengijinkan kelas lain berbagi program dan data (*code reuse*) sehingga progam tidak harus dibangun ulang dari awal. Aspek penting kedua dari Pewarisan (*Inheritance*) adalah pengelompokan fitur-fitur kelas yang serupa kedalam sebuah kelas dasar (*base class*) dan kemudian membuat kelas lain dengan cara menurunkan kelas tersebut sehingga bentuk utama/ pokok dari kelas-kelas turunan menjadi serupa dengan kelas dasar sedemikian rupa sehingga pointer atau referensi bertipe kelas dasar dapat menerima nilai berbagai macam bentuk objek bertipe kelas turunannya (berubah tipe kelas) dan dapat mengeksekusi fitur-fitur yang serupa tesebut. Inilah yang disebut dengan polimorfisme.

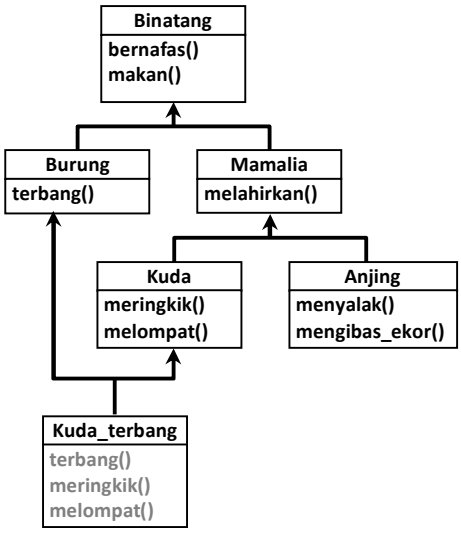
Berbeda dengan tipe data standard (int, float, double, bool, dsb.) yang jenisnya terbatas dan sudah pasti dikenal dalam pemrograman C++, kelas adalah tipe data terstruktur yang bisa dibuat sendiri oleh programer, sehingga tidak terbatas ada berapa macam kelas dalam C++. Namun di sisi lain, dalam pemrograman kita pasti perlu untuk dapat berhubungan antara kelas satu dengan yang lain, sehingga diperlukan suatu kerangka yang dapat memberikan arahan mengenai bentuk dan fitur dari suatu kelas sehingga dengan demikian ada suatu pointer atau referensi yang dapat menerima berbagai macam bentuk yang dinamakan polimorfisme.

Mekanisme untuk dapat terjadi polimorfisme adalah pewarisan (*inheritance*), oleh karena itu polimorfisme hanya bisa terjadi diantara kelas-kelas yang mempunyai hubungan kekerabatan, tepatnya pointer atau referensi bertipe kelas dasar hanya dapat menerima berbagai macam bentuk objek yang bertipe kelas-kelas turunannya, sehingga pointer atau referensi tersebut dapat mengeksekusi fitur-fitur yang serupa dengannya. Contoh sederhana dari polimorfisme adalah seperti sudah dibahas pada bab 7 yaitu mengenai metode vitual pada percobaan **Lab.10**.

Berikut ini akan dibahas secara lebih mendalam mengenai berbagai aspek polimorfisme dari problem pada *single inheritance*, *multiple inheritance* hingga *abstract data type*.

Problema Pewarisan Tunggal (Single Inheritance)

Supaya terlihat sederhana, untuk menjelaskan masalah ini marilah kita menggunakan ilustrasi mengenai kelas-kelas dengan perumpamaan kelas binatang dan turunan-turunanya. Misalnya kelas **Binatang** mempunai hirarki keturuan **Mamalia** dan **Burung**, kelas **Burung** mempunyai *member function* **terbang()**, sedangkan kelas **Mamalia** sudah diturunkan menjadi beberapa kelas diantaranya **Kuda** dan **Anjing**. Kelas **Kuda** mempunyai *member fucntion* **meringkik()** dan **melompat()**. Suatu saat terpikir untuk membuat kelas **Kuda_terbang** yang merupakan kombinasi antara kelas **Burung** dan kelas **Kuda**, karena **Kuda_terbang** bisa **terbang()**, **meringkik()** dan **melompat()**.



Dengan *single inheritance* hal ini tidak bisa dilakukan dengan mudah, karena hanya bisa menurunkan dari salah satu kelas yang sudah ada, kita bisa membuat **Kuda_terbang** adalah turunan **Burung** yang bisa **tebang()** tetapi akibatnya tidak bisa **meringkik()** dan **melompat()**, sebaliknya jika dibuat sebagai turunan dari **Kuda** akan bisa **meringkik()** dan **melompat()** tatapi tidak bisa **terbang()**.

Untuk memaksakan hal ini bisa dilakukan dengan membuat metode **terbang()** di dalam kelas **Kuda_terbang** dan kelas ini diturunkan dari kelas **Kuda**. Ini akan dapat dilakukan, akan tetapi harga yang harus dibayar sekarang adalah kita mempunyai metode **terbang()** dalam dua kelas yang berbeda (**Burung** dan **Kuda_terbang**), jika ada perubahan di salah satu metode **terbang()**, maka harus selalu diingat untuk merubah yang lainnya, ini sangat berisiko akan adanya ketidakkonsistenan program.

Belum lagi nanti akan timbul masalah polimorfisme ketika akan dibuat daftar objek **Kuda** atau daftar objek **Burung**, kalau **Kuda_terbang** dapat dimasukkan ke dalam dafatar **Kuda** maka ia tidak akan bisa dimasukkan kedalam daftar **Burung**. Akibatnya akan timbul ide untuk mengubah metode **melompat()** pada **Kuda** menjadi **berpindah()** kemudian melakukan override dalam kelas **Kuda_terbang** yang melakukan pekerjaan **terbang()** dan pada turunan **Kuda** lainnya melakukan override **berpindah()** yang melakukan pekerjaan **melompat()**. Kemudian karena seharusnya **Kuda_terbang** tetap dapat melompat kita membatasi jika jarak dekat melompat jika jarak jauh terbang seperti berikut:

```
Kuda_terbang::berpindah(long jarak)
{
if (jarak > sangat_jauh)
      terbang(jarak);
else
      melompat(jarak);
}
```

Tapi ini menjadikan terbatas, bagaimana jika nanti ternyata **Kuda_terbang** bisa **melompat()** lebih jauh atau **terbang()** lebih dekat? Ini akan menjadi masalah.

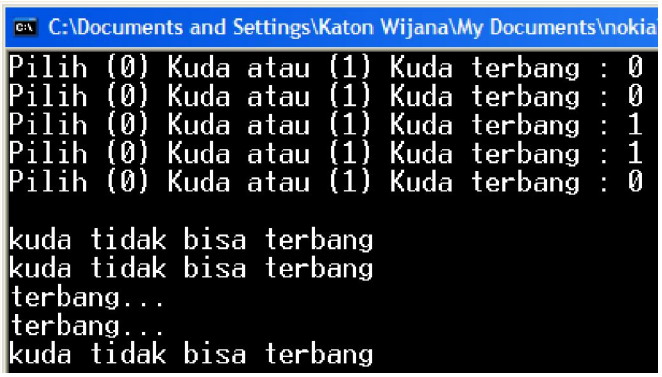
Solusi lainnya untuk membahas keterbatasan *single inheritance* ini adalah membuat metode **terbang()** pada kelas **Kuda** dan pada kelas ini tidak melakukan **terbang()**, baru nanti kalau berupa objek **Kuda_terbang**, barulah **terbang()** yang sesungguhnya dikerjakan. Marilah kita melakukan percobaan berikut ini.

Lab.1 Meletakkan metode kelas turunan di kelas dasar.

1. Buka Qt Creator dan buat project Qt Console Application baru dengan nama **Lab1**, kemudian tulis kode berikut.

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;
class Kuda{
public:
    void melompat(){ cout << "Lompat!" << endl;}
    virtual void terbang(){cout <<"kuda tidak bisa terbang" << endl;}
};

class Kuda_terbang : public Kuda{
public:
    void terbang() {cout << "terbang..." << endl;}
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Kuda* kandang[5];
    Kuda* kudanya;
    int pilih;
    for(int nomor=0; nomor<5; nomor++){
       cout << "Pilih (0) Kuda atau (1) Kuda terbang : ";
       cin >> pilih;
       if(pilih==0)
           kudanya = new Kuda();
       else
           kudanya = new Kuda_terbang();
       kandang[nomor] = kudanya;
    }
    cout << endl;
    for(int nomor=0; nomor<5; nomor++){
        kandang[nomor]->terbang();
        delete kandang[nomor];
    }
    return a.exec();
}
```

2. Kemudian jalankan kode diatas dengan menekan tombol Ctrl+R, outputnya adalah sebagai berikut.



Analisa Program :

- Pada program diatas ada dua kelas, yaitu **Kuda** sebagai kelas dasar dan **Kuda_terbang** sebagai kelas turunan, **Kuda** mempunyai metode **terbang()** yang sebenarnya diada-adakan supaya bisa terjadi proses polimorfisme dengan baik, akibatnya metode **terbang()** ini tidak melakukan terbang yang sesungguhnya. Kelas **Kuda_terbang** melakukan override terhadap metode virutal **terbang()**, pada metode ini melakukan terbang yang sesungguhnya.
- Pada program utama disediakan program untuk menentukan 5 jenis kuda, yang ditampung dalam pointer **kudanya**. Pointer ini bisa berpolimorfisme kaena memenuhi syarat bahwa dia bertipe kelas dasar (**Kuda**) dan memanggil metode virutual **terbang()**.
- Pada percobaan eksekusi di atas dipilih **0**, **0**, **1**, **1**, **0** yang berarti array kandang berisi :

  Kandang[0] <-- berisi objek **Kuda**
  Kandang[1] <-- berisi objek **Kuda**
  Kandang[2] <-- berisi objek **Kuda_terbang**
  Kandang[3] <-- berisi objek **Kuda_terbang**
  Kandang[4] <-- berisi objek **Kuda**

  Sehingga hasil keluaran berikutnya:

  kuda tidak bisa terbang
  kuda tidak bisa terbang
  terbang...
  terbang...
  kuda tidak bisa terbang
- Dari percobaan ini tampak polimorfisme bisa berhasil dengan baik. Namun seperti sudah dijelaskan di atas, harga yang harus dibayar adalah sekarang ada metode **terbang()** pada kelas **Kuda_terbang** dan **Burung**, sehingga jika ada perubahan di salah satu metode **terbang()**, maka harus selalu diingat untuk merubah yang lainnya, ini sangat berisiko akan adanya ketidakkonsistenan program.

Catatan:
Pada percobaan ini kelas telah disederhanakan untuk fokus pada pokok masalah yang hendak dijelaskan. Konstruktor, destruktor dan sebagainya dihilangkan supaya program tampak sederhana untuk

membahas masalah *single inheritance*. Tidak direkomendasikan untuk menulis program yang seperti ini.

Peletakan ke atas (*Pecolating Upward*)

Meletakkan fungsi yang diperlukan pada kelas yang berada pada hierarki di atasnya, seperti percobaan di atas, adalah solusi yang biasa digunakan untuk keperluan polimorfisme dan juga dalam hal ini mengatasi *single inheritance* dan berakibat menghasilkan banyak fungsi-fungsi "yang diletakkan di atas" ("*Percolating up*") ke dalam kelas dasar. Kemudian kelas dasar tersebut akan menjadi sangat berbahaya karena menjadi *global namespace* untuk semua fungsi yang mungkin diperlukan kelas turunan yang berpotensi merusak tatanan tipe kelas C++ dan menciptakan kelas dasar yang berukuran besar dan tidak efisien.

Sebenarnya kejadian ini terjadi karena keinginan untuk menaruh fungsionalitas bersama ke hirarki di atasnya tanpa mengubah antarmuka (interface) dari tiap kelas. Artinya jika ada dua kelas yang memakai kelas dasar yang sama (misalnya **Burung** dan **Kuda** sama-sama bersal dari kelas dasar **Binatang**) dan keduanya sama-sama mempunyai sebuah fungsi yang sama (misalnya **Burung** dan **Kuda** sama-sama bisa **makan**), maka akan timbul ide untuk memindahkan fungsi tersebut ke hirarki di atasnya dan membuat **metode virtual** pada kelas dasar.

Konversi ke bawah (Casting Down)

Alternatif lain untuk mengatasi *single inheritance* adalah tetap membuat metode **terbang()** dalam kelas **Kuda_terbang** dan metode ini hanya dipanggil jika pointer menunjuk objek bertipe **Kuda_terbang**. Untuk keperluan ini diperlukan untuk dapat mendeteksi objek tipe apa yang sedang ditunjuk oleh pointer yang dikenal sebagai Runtime Type Identification (RTTI).

Catatan:
Hati-hati dalam menggunakan RTTI dalam program. Kebutuhan untuk memakai RTTI bisa menjadi pertanda adanya desain hirarki pewarisan yang kurang baik. Untuk itu sebaiknya gunakan metode virtual, template atau *multiple inheritance* daripada memakai RTTI.

Pada percobaan Lab.1 di atas, kita menunjuk baik **Kuda** maupun **Kuda_terbang** dengan array **Kuda** (kandang). Semuanya dimasukkan sebagai **Kuda**. Dengan RTTI, kita akan memeriksa tiap-tiap elemen array apakah objek yang ditunjuk berupa **Kuda** atau sebenarnya **Kuda_terbang**.

Pada percobaan ini kita tidak melakukan "*percolating* upward", yaitu menuliskan metode **terbang()** ke dalam kelas **Kuda** melainkan melakukan "*down casting*" untuk memanggil metode **terbang()**.Untuk memanggil metode **terbang()** tersebut harus dipastikan bahwa pointer sedang menunjuk objek bertipe **Kuda_terbang**, bukan **Kuda**. C++ mendukung "*down casting*" (RTTI) memakai operator **`dynamic_cast`**.

Cara kerja **`dynamic_cast`** adalah demikian, jika kita memiliki pointer bertipe kelas dasar seperti **Kuda** dan digunakan untuk menunjuk objek bertipe kelas turunan misalnya **Kuda_Terbang**, maka pointer tersebut bisa langsung dipergunakan secara polimorfisme. Kemudian jika kita ingin mengambil objek **Kuda_terbang** yang sudah ditunjuk oleh pointer bertipe **Kuda** itu dapat dilakukan dengan cara membuat

pointer bertipe **Kuda_terbang** kemudian gunakan operator **dynamic_cast** untuk mengkonversikannya.

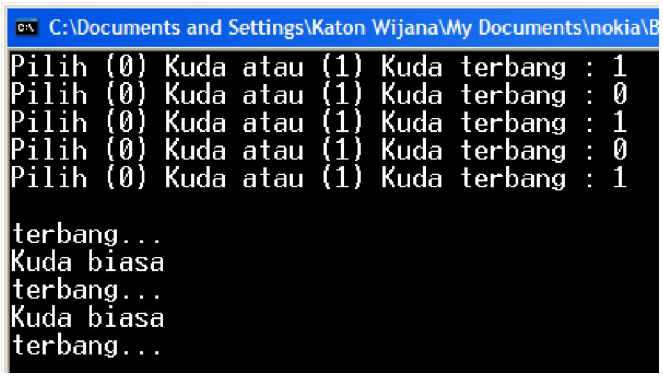
Pada saat eksekusi (runtime), pointer dasar akan diperiksa, jika sesuai maka pointer Kuda_terbang bekerja dengan baik, jika tidak sesuai maka sebenarnya bukan objek Kuda_terbang yang ditunjuk meliankan pointer kosong (null). Lakukan percobaan berikut ini.

Lab.2 Melakukan Down Casting.

1. Buka Qt Creator dan buat project Qt Console Application baru dengan nama **Lab2**, kemudian tulis kode berikut.

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;
class Kuda{
public:
    virtual void melompat(){ cout << "Lompat!" << endl;}
};

class Kuda_terbang : public Kuda{
public:
    virtual void terbang() {cout << "terbang..." << endl;}
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Kuda* kandang[5];
    Kuda* kudanya;
    int pilih;
    for(int nomor=0; nomor<5; nomor++){
       cout << "Pilih (0) Kuda atau (1) Kuda terbang : ";
       cin >> pilih;
       if(pilih==0)
           kudanya = new Kuda();
       else
           kudanya = new Kuda_terbang();
       kandang[nomor] = kudanya;
    }
    cout << endl;
    for(int nomor=0; nomor<5; nomor++){
        Kuda_terbang *pKterb = dynamic_cast<Kuda_terbang*> (kandang[nomor]);
        if (pKterb != NULL)
            pKterb->terbang();
        else
            cout << "Kuda biasa " << endl;
        delete kandang[nomor];
    }
    return a.exec();
}
```

2. Kemudian jalankan kode diatas dengan menekan tombol Ctrl+R, outputnya adalah sebagai berikut.



Analisa Program :

- Jalan keluar *single inheritance* ini berjalan dengan baik, namun tidak direkomendasikan untuk memakai cara ini. Hasilnya hanya berupa tambal-sulam saja, sebenarnya metode **terbang()** tidak ada di kelas **Kuda** dan tidak dipanggil dari sana. Ketika dipanggil dengan poiter **Kuda_terbang** dilakukan casting secara eksplisit, maka harus dipastikan bahwa pointer **pKterb** benar-bena berisi objek bertipe **Kuda_terbang** sebelum memanggil metode **terbang()**.
- Pemakain down casting semacam ini merupakan pertanda bahwa desain yang dibuat kurang baik, program semacam ini merusak fungsi virtual polimorfisme karena ini dilakukan casting objek menjadi tipe yang sesungguhnya (bukan polimorfisme).

Menambahkan ke Dua Daftar

Masalah lain yang dihadapi oleh karena memakai solusi di atas adalah bahwa kita telah mendeklarasikan **Kuda_terbang** bertipe **Kuda**, sehingga kita tidak bisa menambahkan objek **Kuda_terbang** ke dalam daftar objek **Burung**. Dalam hal ini bisa saja kita melakukan pemindahan metode **terbang()** ke hirarki atasnya yaitu kedalam kelas **Kuda** atau melakukan ***down casting*** pada pointer, tetapi tetap saja kita tidak mendapatkan fungsionalitasnya secara penuh.

Satu solusi terakhir untuk mengatasi masalah *single inheritance* ini adalah memindahkean semua fungsi terbang(), meringkik() dan melompat() ke kelas dasar dari Burang dan Kuda, yaitu kelas Binatang. Dengan demikian kita bisa mendaftar objek-objek **Burung, Kuda** maupun objek-objek **Kuda_terbang** dalam satu kesatuan daftar **Binatang**. Namun akibatnya kelas dasar mempunyai semua karakteristik kelas turunannya sehingga ukuran kelas dasar menjadi sangat besar dan ini tidak diinginkan.

Pewarisan Ganda (Multiple Inheritance)

Dengan C++ dimungkinkan untuk menurunkan kelas baru yang berasal dari lebih dari satu kelas dasar, yang dinamakan pewaisan ganda (*multiple inheritance*). Penurunan lebih dari satu kelas dasar dilakukan dengan cara menuliskan kelas dasar berikutnya dipisahkan dengan tanda koma (,) seperti berikut:

```
class KelasTurunan : public KelasDasar1, public KelasDasar2{}
```

Percobaan berikut ini memberikan ilustrasi cara deklarasi kelas **Kuda_terbang** yang mewarisi kelas dasar **Kuda** dan **Burung**, kemudian program menambahkan objek **Kuda_terbang** ini di kedua jenis daftar objek tersebut.

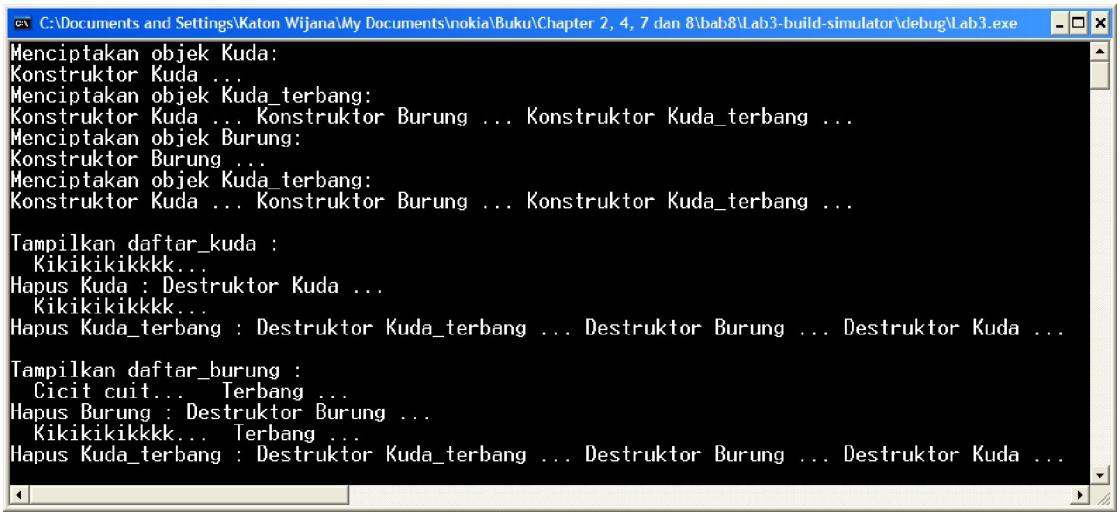
Lab.3 Multiple Inheritance.

1. Buka Qt Creator dan buat project Qt Console Application baru dengan nama **Lab3**, kemudian tulis kode berikut.

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;
class Kuda{
public:
    Kuda() {cout << "Konstruktor Kuda ... ";}
    virtual ~Kuda() {cout << "Destruktor Kuda ... \n";}
    virtual void meringkik() const {cout << "  Kikikikikkkk...";}
};

class Burung{
public:
    Burung() { cout << "Konstruktor Burung ... "; }
    virtual ~Burung() { cout << "Destruktor Burung ... "; }
    virtual void berkicau() const { cout << "  Cicit cuit... "; }
    virtual void terbang() const { cout << "  Terbang ... "; }
};

class Kuda_terbang : public Kuda, public Burung{
public:
    void berkicau() const { meringkik(); }
    Kuda_terbang() { cout << "Konstruktor Kuda_terbang ... "; }
    ~Kuda_terbang() { cout << "Destruktor Kuda_terbang ... "; }
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Kuda* daftar_kuda[2];     //<-- kumpulan Kuda
    Burung* daftar_burung[2]; //<-- kumpulan Burung
    cout << "Menciptakan objek Kuda:" << endl;
    daftar_kuda[0] = new Kuda();          //<-- objek Kuda
    cout << "\nMenciptakan objek Kuda_terbang:" << endl;
    daftar_kuda[1] = new Kuda_terbang(); //<-- objek Kuda_terbang
    cout << "\nMenciptakan objek Burung:" << endl;
    daftar_burung[0] = new Burung();      //<-- objek Burung
    cout << "\nMenciptakan objek Kuda_terbang:" << endl;
    daftar_burung[1] = new Kuda_terbang();//<-- objek Kuda_terbang
    cout << "\n\nTampilkan daftar_kuda :" << endl;
    daftar_kuda[0]->meringkik(); //<-- berisi objek Kuda
    cout << "\nHapus Kuda : ";
    delete daftar_kuda[0];
    daftar_kuda[1]->meringkik(); //<-- berisi objek Kuda_terbang
    cout << "\nHapus Kuda_terbang : ";
    delete daftar_kuda[1];
```

```
    cout << "\nTampilkan daftar_burung :" << endl;
    daftar_burung[0]->berkicau(); //<-- berisi objek Burung
    daftar_burung[0]->terbang();  //<-- berisi objek Burung
    cout << "\nHapus Burung : " ;
    delete daftar_burung[0];
    cout << endl;
    daftar_burung[1]->berkicau(); //<-- berisi objek Kuda_terbang
    daftar_burung[1]->terbang(); //<-- berisi objek Kuda_terbang
    cout << "\nHapus Kuda_terbang : ";
    delete daftar_burung[1];
    return a.exec();
}
```

2. Kemudian jalankan kode diatas dengan menekan tombol Ctrl+R, outputnya adalah sebagai berikut.



Analisa Program :

- Pada kelas **Kuda_terbang** tampak penggunaan *multiple inheritance*, yaitu pada deklarasi kelas **Kuda_terbang** yang merupakan keturunan dari **Kuda** dan **Burung** :

  ```
  class Kuda_terbang : public Kuda, public Burung
  ```

  Kemudian kelas ini melakukan *override* terhadap metode **berkicau()**, metode **berkicau()** pada **Kuda_terbang** ini mengerjakan pemanggilan metode **meringkik()**, yaitu metode warisan dari kelas **Kuda**:

  ```
  void berkicau() const { meringkik(); }
  ```

- Tampak pada percobaan **Lab.3** ini ketika diciptakan objek bertipe **Kuda** maka yang bekerja adalah **konstruktor Kuda**, demikian juga ketika diciptakan objek **Burung** yang bekerja adalah **konstruktor Burung**, namun ketika diciptakan **Kuda_terbang** yang bekerja tiga konstruktor sekaligus, yaitu : **konstruktor Kuda**, **konstruktor Burung** dan **konstruktor Kuda_terbang**. Ini mempelihatkan bahwa kelas **Kuda_terbang** merupakan turunan dari kelas **Kuda** sekaligus turunan kelas **Burung**. Dengan kata lain objek **Kuda_terbang** adalah objek yang di dalamnya

terkandung bagian objek **Kuda** dan bagian objek **Burung**.

- Kebalikannya saat dihapus, destruktor yang dijalankan : **destruktor Kuda_terbang**, **destruktor Burung** baru kemudian **destruktor Kuda**. Ini adalah akibat adanya destruktor yang selalu virtual.
- Pada saat ditampilkan **daftar_kuda**, tampak pada program sebenarnya yang pertama berisi objek **Kuda** sedangkan yang kedua berisi objek **Kuda_terbang** :

```
daftar_kuda[0] = new Kuda();          //<-- objek Kuda
daftar_kuda[1] = new Kuda_terbang(); //<-- objek Kuda_terbang
```

Demikian juga pada **daftar_burung**, yang pertama berisi objek **Burung** sedangkan yang kedua berisi **Kuda_terbang** :

```
daftar_burung[0] = new Burung();       //<-- objek Burung
daftar_burung[1] = new Kuda_terbang();//<-- objek Kuda_terbang
```

Ini menunjukkan bahwa baik **daftar_kuda[]** maupun **daftar_burung[]** dapat berpolimorfisme dengan sempurna berkat adanya *multiple inheritance*.

- Pada waktu daftar_burung berisi objek Kuda_terbang, ketika dipanggil metode berkicau berikut :

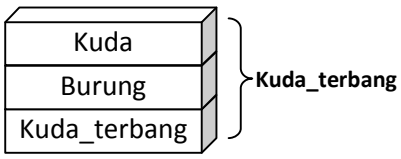
```
daftar_burung[1]->berkicau(); //<-- berisi objek Kuda_terbang
```

Maka yang menanggapi adalah metode hasil override di dalam kelas Kuda_terbang yaitu pemanggilan metode meringkik(), sehingga keluarannya adalah :

```
"  Kikikikikkkk..."
```

Komponen Objek Multi Inheritance

Ketika objek **Kuda_terbang** diciptakan di memori, kedua kelas dasarnya juga tercipta sebagai bagian pembentuk objek tersebut. Gambar di bawah ini menggambarkan sebuah objek bertipe **Kuda_terbang** termasuk fitur-fitur baru yang ditambahkan pada kelas **Kuda_terbang** maupun fitur-fitur warisan kelas-kelas dasarnya.



Ada beberapa hal yang perlu diperhatikan pada objek yang merupakan turunan dari beberapa kelas dasar. Sebagai contoh misalnya, apa yang terjadi jika kedua kelas dasar tersebut memiliki metode virtual atau data yang sama? Bagaimana cara menginisialisasi konstruktor kelas dasarnya? Bagaimana jika kedua kelas dasar yang diturunkan merupakan keturunan dari suatu kelas dasar yang sama? Berikut ini akan kita bicarakan mengenai isu-isu tersebut.

Konstruktor Kelas Banyak Turunan (*Multiple Inheritance*)

Seperti pada umumnya, kelas turunan pasti memanggil konstruktor kelas dasarnya, demikian juga konstruktor kelas banyak turunan (*multiple inheritance*) juga harus memanggil konstruktor-kontruktor kelas dasarnya. Jika bentuk konstruktor default (konstruktor yang diciptakan secara otomatis oleh kompiler jika kelas turunan tidak menuliskan konstruktor secara eksplisit) pada *single inheritance* yaitu:

```
KelasTurunan():KelasDasar(){} //<-- Konstruktor default kelas Turunan tunggal
```

Maka konstruktor default kelas banyak turunan (*mutiple inheritance*) adalah :

```
KelasTurunan():KelasDasar1(),KelasDasar2(){}
```

Jadi pada waktu membuat kelas banyak turunan, khususnya jika kelas dasarnya tidak mempunyai konstruktor tanpa parameter, maka konstruktor kelas turunan tersebut harus memanggil salah satu dari konstruktor kelas-kelas dasarnya, karena kalau tidak kompiler akan memanggil konstruktor default kelas dasar yang sebenarnya tidak ada sehingga menimbulkan kesalahan kompilasi.

Supaya bisa fokus pada pokok permasalahan, yaitu konstruktor, percobaan berikut ini menghilangkan berbagai macam anggota yang lain supaya terlihat sederhana dan mudah dipahami.

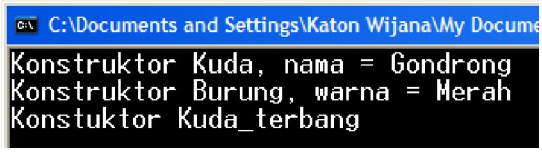
Lab.4 Konstruktor Kelas Multiple Inheritance.

1. Buka Qt Creator dan buat project Qt Console Application baru dengan nama **Lab4**, kemudian tulis kode berikut.

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;

class Kuda{
public:
    Kuda(string nama){
        cout << "Konstruktor Kuda, nama = " << nama << endl;
    }
};

class Burung{
public:
    Burung(string warna){
        cout << "Konstruktor Burung, warna = " << warna << endl;
    }
};

class Kuda_terbang : public Kuda, public Burung{
public:
    Kuda_terbang():Kuda("Gondrong"),Burung("Merah"){
        cout << "Konstuktor Kuda_terbang";
    }
};

int main(int argc, char *argv[])
{
```

```
    QCoreApplication a(argc, argv);
    Kuda_terbang* gondrong = new Kuda_terbang();
    return a.exec();
}
```

2. Kemudian jalankan kode diatas dengan menekan tombol Ctrl+R, outputnya adalah sebagai berikut.



Analisa Program :

- Kelas **Kuda** hanya mempunyai **sebuah** konstruktor dengan sebuah parameter bertipe **string**, demikian juga kelas **Burung** hanya mempunyai **sebuah** konstruktor dengan **sebuah** parameter bertipe **string**.
- Pada kelas **Kuda_terbang** tampak penggunaan *multiple inheritance*, yaitu pada deklarasi kelas **Kuda_terbang** yang merupakan keturunan dari **Kuda** dan **Burung** :

  ```
  class Kuda_terbang : public Kuda, public Burung
  ```

  Oleh karena itu konstruktor kelas Kuda_terbang ini harus memanggil secara eksplisit konstruktor-konstruktor kelas dasarnya seperti berikut:

  ```
  Kuda_terbang():Kuda("Gondrong"),Burung("Merah")
  ```

- Tampak pada hasil output, instansiasi kelas **Kuda_terbang** menjalankan konstruktor **Kuda** dengan satu parameter dan konstruktor **Burung** dengan satu parameter, baru kemudian menjalankan konstruktornya sendiri.

Problem Ambiguitas

Pada kelas *multi inheitance* yang beberapa kelas dasarnya mempunyai metode virtual yang sama akan menimbulkan masalah ketika objek kelas turunan akan memanggil metode tersebut, sebab kelas tersebut mendapatkan warisan dari keduanya, sehingga ketika metode tersebut dipanggil akan menimbulkan masalah ambiguitas bagi kompiler.

Sebagai contoh misalnya kelas Kuda dan kelas Burung sama-sama mempunyai metode virutal makan() dam jika kita memanggil metode dari objek Kuda_terbang yang merupakan multi inheritance dari kedua kelas tersebut di atas.

```
pKterbang->makan();
```

Maka kompile akan menampilkan pesan kesalahan :

```
Member is ambiguous: 'Kuda::makan' and 'Burung::makan'
```

Untuk memecahkan masalah ini dengan pemanggilan secara eksplisit bisa dilakukan dengan cara menyebutkan nama kelas dasar pemilik metode virtual yang akan dieksekusi, misalnya:

```
pKterbang->Kuda::makan();
```

Kapan saja kita ingin mengeksekusi fungsi anggota atau variabel anggota kelas mana yang akan diakses, kita dapat melakukan dengan cara tersebut di atas. Jika kelas **Kuda_terbang** akan melakukan overriding fungsi tersebut, maka pemanggilan tersebut bisa dilakukan di dalam fungsi anggota **Kuda_terbang**:

```
virtual void makan() const { Kuda::makan() }
```

Dengan cara ini kita bisa menyembunyikan masalah ambiguitas ini dari pemakai kelas **Kuda_terbang** dengan cara mengenkapsulasi di dalam kelas **Kuda_terbang** mengenai kelas dasar mana yang diambil warisan fungsi **makan()**-nya. Namun pemakai tetap saja bebas untuk dapat memanggil waisan lainnya dengan menulis :

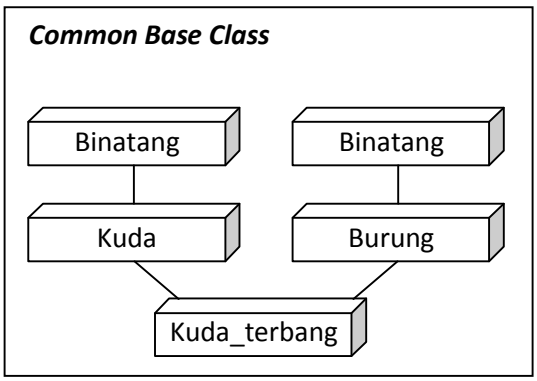
```
pKterbang->Burung::makan();
```

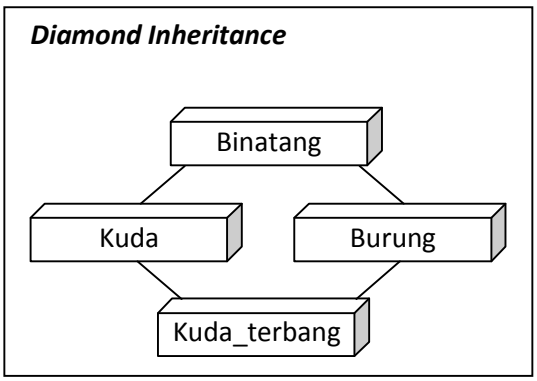
Penurunan dari Kelas Dasar Bersama

Jika dua kelas dasar yang dijadikan *multiple inheritance* merupakan turunan dari kelas dasar yang sama, maka ketika diciptakan objek dari kelas turunan akan bisa terbentuk objek dengan 2 kemungkinan, yaitu:

1. Tercipta 2 objek kelas dasar (***Common Base Class***)
2. Tercipta 1 Objek kelas dasar (***Diamond Inheritance***)

Sebagai ilustrasi seperti contoh sebelumnya, Kuda dan Burung merupakan turunan dari sebuah kelas dasar yang sama yaitu kelas Binatang, sedangkan Kuda_terbang merupakan hasil multiple inheritance dari kelas Kuda dan Burung. Pada saat instansiasi kelas Kuda_terbang terdapat 2 kemungkinan bentuk kelas yang terjadi:





Common base class adalah penurunan seperti biasa yang terjadi, yaitu ketika kelas paling bawah (**Kuda_terbang**) melakukan instansiasi, secara otomatis kelas-kelas dasarnya (**Kuda** dan **Burung**) juga terbentuk, dan ketika kedua kelas dasar tersebut terbentuk (**Kuda** dan **Burung**), masing-masing memanggil secara terpisah kostruktor kelas dasar paling atas (**Binatang**), sehingga terbentuklah objek

kelas dasar paling atas untuk masing-masing (sendiri-sendiri). Sedangkan *Diamon inheritance* terjadi ketika dilakukan penurunan secara **virtual**, yaitu konstruktor kelas turunan paling bawah (**Kuda_terbang**) memanggil secara langsung konstruktor kelas dasar paling atas (**Binatang**), sehingga pemanggilan oleh kedua kelas dasarnya (Kuda dan Burung) terhadap konstruktor kelas dasar paling atas (**Binatang**) diabaikan. Hal ini bisa terjadi jika penurunan dilakukan secara **virtual** yang akan dibahas nanti.

Untuk penurunan pada umumnya (*Common base class*) ada kemungkinan kelas **Kuda** maupun **Burung** masing-masing sudah melakukan overriding terhadap metode milik kelas dasarnya (**Binatang**). Sebagai contoh misalnya kelas **Binatang** mempunyai member variabel **umur** dan member function **getUmur()**. Jika kelas turunan paling bawah (**Kuda_terbang**) memanggil metode **getUmur()** tersebut, makan akan timbul ambiguitas lagi, yaitu metode yang dipanggil merupakan **getUmur()** Kuda atau **getUmur()** Burung? Pemecahan problem ini sama dengan pada kelas *multi inheitance* yang yang kedua kelas dasarnya mempunyai metode virtual yang sama, yaitu bisa dilakukan dengan cara melakukan override terhadap metode **getUmur()** kemudian di dalamnya memilih metode yang akan merespon, misalnya dipilih **getUmur()** dari **Kuda** seperti berikut:

```
virtual int getUmur() const { return Kuda::getUmur() }
```

Supaya lebih jelas, lakukan pecobaan berikut ini.

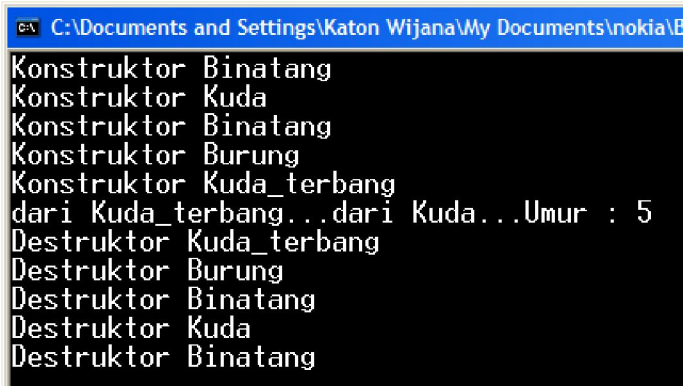
Lab.5 Penurunan pada umumnya (*Common Base Class*).

1. Buka Qt Creator dan buat project Qt Console Application baru dengan nama **Lab5**, kemudian tulis kode berikut.

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;
class Binatang{
public:
    Binatang(int umur=5):umur(5){cout << "Konstruktor Binatang\n";}
    ~Binatang(){cout << "Destruktor Binatang\n";}
protected:
    int umur;
public:
    virtual int const getUmur(){
        cout << "dari Binatang...";
        return umur;
    }
};

class Kuda : public Binatang{
public:
    Kuda(){cout << "Konstruktor Kuda\n";}
    ~Kuda(){cout << "Destruktor Kuda\n";}
    virtual int const getUmur(){
        cout << "dari Kuda...";
        return umur;
    }
```

```
};

class Burung : public Binatang{
public:
    Burung(){cout << "Konstruktor Burung\n";}
    ~Burung(){cout << "Destruktor Burung\n";}
    virtual int const getUmur(){
        cout << "dari Burung...";
        return umur;
    }
};

class Kuda_terbang : public Kuda, public Burung{
public:
    Kuda_terbang(){cout << "Konstruktor Kuda_terbang\n";}
    ~Kuda_terbang(){cout << "Destruktor Kuda_terbang\n";}
    virtual int const getUmur(){
        cout << "dari Kuda_terbang...";
        return Kuda::getUmur();
    }
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Kuda_terbang* gondrong = new Kuda_terbang();
    cout << "Umur : " << gondrong->getUmur() << endl;
    delete gondrong;
    return a.exec();
}
```

2. Kemudian jalankan kode diatas dengan menekan tombol Ctrl+R, outputnya adalah sebagai berikut.



Analisa Program :

- Pada program Utama hanya menciptakan objek bertipe **Kuda_terbang**. Tampak pada hasil keluaran, dilihat dari konstruktornya maka bisa disimpulkan bahwa terbentuk objek **Binatang**, objek **Kuda**, objek **Binatang**, objek **Burung** baru kemudian objek **Kuda_terbang**. Ini menunjukkan bahwa ada 2 objek **Binatang** yang masing-masing merupakan bagian **Kuda**

dan **Burung**.

- Oleh karena **Kuda_terbang** adalah turunan dari **Kuda** dan **Burung**, maka jika ia memanggil metode **getUmur()** yang merupakan warisan dari mereka, akan terjadi ambiguitas, oleh karena itu pada kelas **Kuda_terbang** dilakukan overriding seperti berikut :

```
virtual int const getUmur(){
      cout << "dari Kuda_terbang...";
      return Kuda::getUmur();
}
```

- Tampak pada hasil pemanggilan metode tersebut pada kelas **Kuda_terbang**:

```
dari Kuda_terbang...dari Kuda...Umur : 5
```

  Ini menunjukkan bahwa metode yang dipanggil adalah metode hasil override pada kelas Kuda_tebang yang di dalamnya sudah mengatasi problem ambiguitas dengan memastikan yang dipanggil adalah getUmur() dari kelas Kuda (yaitu : `Kuda::getUmur()`).

- Sekali lagi pada saat objek dihapus (delete), destruktor yang dijalankan adalah untuk objek **Kuda_terbang**, **Burung**, **Binatang**, **Kuda** dan **Binatang** ini menunjukkan bahwa ada 2 objek **Binatang** tadi.

Penurunan Virtual (*Virtual Inheritance*)

Berbeda dengan penurunan biasa (*Common Base Class*), penurunan virtual pada kedua kelas dasar (Kuda dan Burung) menyebabkan kelas turunan *multi inheritance* Kuda_terbang dapat langsung memanggil konstruktor kelas dasar paling atas (Binatang), sehingga terbentuklah objek dengan bentuk *Diamond Inheritance* seperti gambar tadi. Supaya lebih jelas, lakukan pecobaan berikut ini.

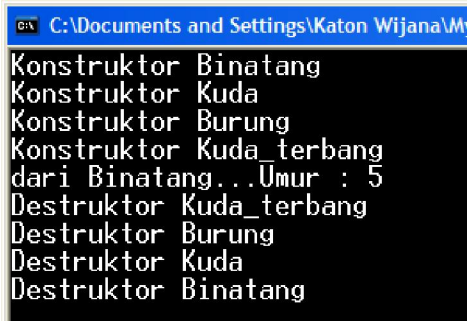
Lab.6 Penurunan pada umumnya (*Common Base Class*).

1. Buka Qt Creator dan buat project Qt Console Application baru dengan nama **Lab6**, kemudian tulis kode berikut.

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;
class Binatang{
public:
    Binatang(int umur=5):umur(5){cout << "Konstruktor Binatang\n";}
    ~Binatang(){cout << "Destruktor Binatang\n";}
protected:
    int umur;
public:
    virtual int const getUmur(){
        cout << "dari Binatang...";
        return umur;
    }
};
```

```
class Kuda : virtual public Binatang{
public:
    Kuda(){cout << "Konstruktor Kuda\n";}
    ~Kuda(){cout << "Destruktor Kuda\n";}
};

class Burung : virtual public Binatang{
public:
    Burung(){cout << "Konstruktor Burung\n";}
    ~Burung(){cout << "Destruktor Burung\n";}
};

class Kuda_terbang : public Kuda, public Burung{
public:
    Kuda_terbang():Kuda(),Burung(),Binatang(){
        cout << "Konstruktor Kuda_terbang\n";}
    ~Kuda_terbang(){cout << "Destruktor Kuda_terbang\n";}
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Kuda_terbang* gondrong = new Kuda_terbang();
    cout << "Umur : " << gondrong->getUmur() << endl;
    delete gondrong;
    return a.exec();
}
```

2. Kemudian jalankan kode diatas dengan menekan tombol Ctrl+R, outputnya adalah sebagai berikut.



Analisa Program :

- Pada program Utama, sama seperti percobaan **Lab5**, hanya menciptakan objek bertipe **Kuda_terbang**. Tampak pada hasil keluaran, dilihat dari konstruktornya maka bisa disimpulkan bahwa terbentuk objek **Binatang**, objek **Kuda**, ~~objek **Binatang**~~, objek **Burung** baru kemudian objek **Kuda_terbang**. Ini menunjukkan bahwa ~~ada 2~~ hanya 1 objek **Binatang** yang ~~masing masing~~ merupakan bagian **Kuda,** ~~dan~~ **Burung** dan **Kuda_terbang**.
- Walaupun **Kuda_terbang** adalah turunan dari **Kuda** dan **Burung**, namun saat instansiasi kelas **Kuda_terbang** memanggil langsung konstruktor **Binatang** seperti berikut:

```
Kuda_terbang():Kuda(),Burung(),Binatang()
```

Memang kelas **Kuda_terbang** bukan keturunan langsung kelas **Binatang**, namun hal ini bisa dimungkinkan karena kelas dasar dari **Kuda_terbang**, yaitu **Kuda** dan **Binatang** melakukan penurunan secara virutal :

```
class Kuda : virtual public Binatang{}
class Burung : virtual public Binatang{}
```

sehingga dimungkinkan konstruktor **Kuda_terbang()** memanggil langsung konstruktor **Binatang()**. Konsekuensinya maka jika ia memanggil metode **getUmur()** yang merupakan warisan dari **Binatang**, metode-metode warisan yang ada pada **Kuda** dan **Burung** akan diabaikan (kecuali mereka melakukan override) dan di-"By pass" langsung ke kelas Binatang. Oleh karena itu pada kelas **Kuda_terbang** tidak lagi terjadi ambigu seperti tadi.

- Tampak pada hasil pemanggilan metode ***getUmur()*** tersebut pada kelas **Kuda_terbang**:

```
dari Binatang...Umur : 5
```

Ini menunjukkan bahwa metode yang dipanggil adalah metode warisan dari kelas **Binatang** secara langsung.

- Sekali lagi pada saat objek dihapus (delete), destruktor yang dijalankan adalah untuk objek **Kuda_terbang**, **Burung**, ~~**Binatang**~~, **Kuda** dan **Binatang** ini menunjukkan bahwa ~~ada 2~~ hanya ada 1 objek **Binatang**.
- Jika dibayangkan, maka "silsilah kekerabatan" penurunan virtual ini adalah seperti gambar ***Daimond Inheritance*** di atas.

Catatan:

- Untuk memastikan kelas turunan hanya mempunyai sebuah kelas dasar bersama, deklarasikan kelas-kelas turunan secara virtual dari kelas dasar. Contoh:

```
class Binatang //<-- common base class (kelas dasar bersama)
class Kuda : virtual public Binatang   //<-- penurunan secara virtual
class Burung : virtual public Binatang //<-- penurunan secara virtual
class Kuda_terbang : public Kuda, public Burung
```

- Jika penurunan dilakukan secara virtual seperti di atas, secara otomatis kelas **Kuda_terbang** akan memanggil konstruktor **Binatang()** walaupun tidak ditulis secara eksplisit, contoh konstruktor **Kuda_terbang()** tanpa memanggil konstruktor **Binatang()** berikut ini akan menghasilkan hasil yang sama:

```
Kuda_terbang(){//<-- pemanggilan Kuda(), Burung() dan Binatang() implisit
cout << "Konstruktor Kuda_terbang\n";}
```

Masalah Pada Multiple Inheritance

Meskipun *multiple inheritance* menawarkan banyak keuntungan dibanding penurunan tunggal, banyak

pemrogram C++ menghindari penggunaan *multiple inheritance*. Mereka mengatakan bahwa *multiple inheritance* membuat pelacakan kesalahan menjadi lebih sulit, dengan mengembangkan kelas *multiple inheritance* hirarki semakin rumit dan semakin berisiko dibandingkan penurunan tunggal dan bahwa hampir semua yang bisa dilakukan dengan *multiple inheritance* juga dapat dilakukan dengan *single inheritance*. Bahasa pemrograman lain seperti **Java** dan **C#** tidak mendukung *multiple inheritance* dengan alasan yang sama. Oleh karena itu, jika bisa dilakukan dengan cara *single inheritance* jangan memakai *multiple inheritance*.

Catatan

- Pakailah *multiple inheritance* jika suatu kelas baru memerlukan fitur-fitur yang ada di dua kelas yang berbeda.
- Pakailah penurunan secara virtual jika ada lebih dari satu kelas turunan namun hanya diperlukan sebuah instan dari kelas dasar.
- Pasti terjadi pemanggilan konstruktor kelas dasar bersama (*shared base class*) dari kelas turunan paling bawah (*multiple inheritance*) ketika memakai kelas dasar virtual.