DIKTAT KULIAH
Pemrograman Berorientasi Objek

Buku Latihan : memahami mekanisme inheritance C++

Oleh :

Inggriani Liem

Departemen Teknik Informatika

Institut Teknologi Bandung

2003

Kata Pengantar

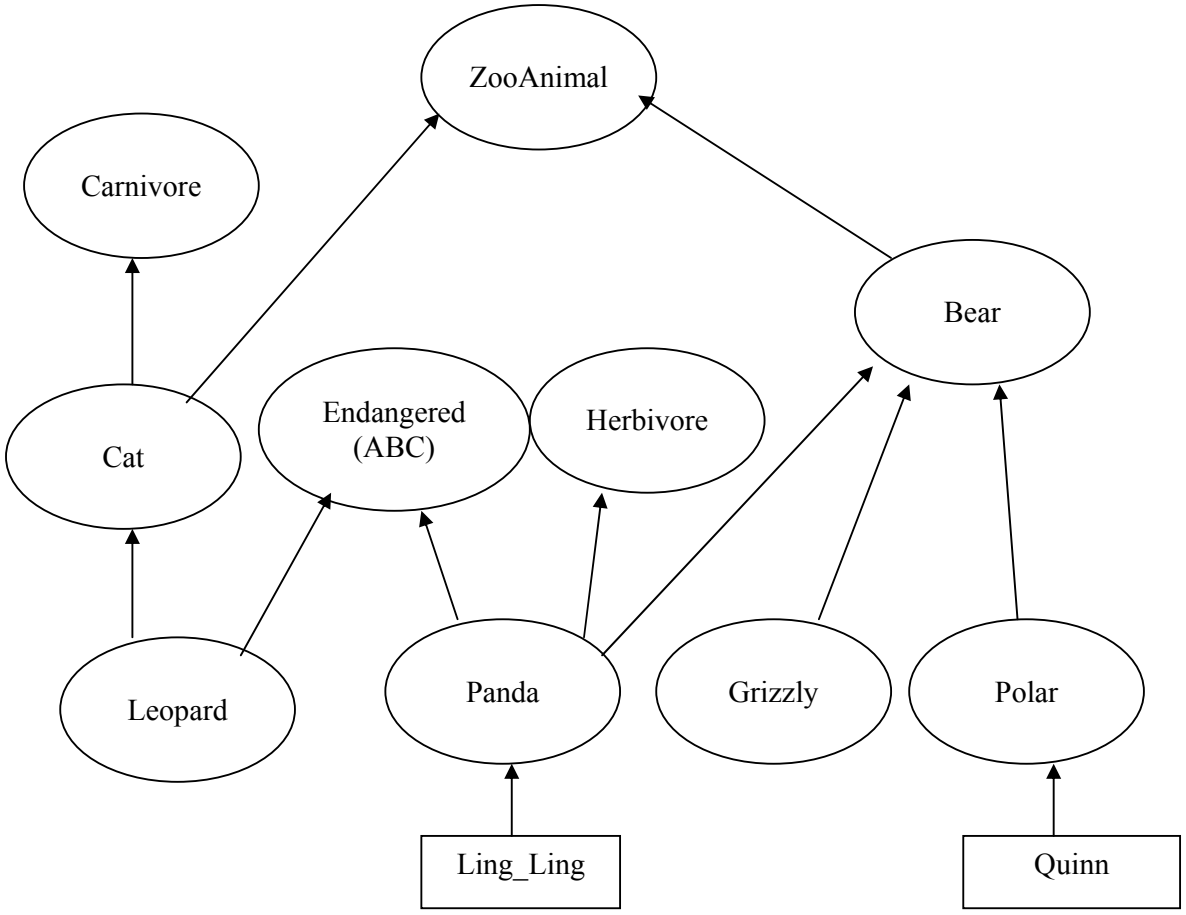
Buku pelengkap diktat ini merupakan buku untuk memahami inheritance program dalam C++ dan segala permasalahannya .

Diktat ini merupakan rangkuman dari yang ditulis dalam buku [Lipmann : "C++ primer, 2nd edition", Addison Wesley, 1991] serta dipakai untuk penjelasan di kelas. Oleh karena itu hanya mengandung sedikit (serta sepotong-sepotong) penjelasan. Mahasiswa seharusnya membaca buku Lipmann secara lengkap

Kenapa harus inheritance ?

```
// Diberikan definisi sebagai berikut
Class Window {


private:
      Screen base;

};



Window W;
W.Clear();
W.base.Clear(); // tidak natural untuk pengguna
W.home();
W.base.home(); // redefine
// supaya tersembunyi:
void Window::Clear() { base.clear();}

// Lebih natural
class Window: public Screen { . . .};
class Menu: puclic Window{ }; //Menu adalah turunan Window dan Screen

Menu m;
m.display();
Window &w=m; //ok
Screen *ps=&w; //ok

extern dumbImage (Screen *s);
Screen s;
Window w;
Menu m;;
Bitvector bv;
dumbImage(&w); //ok, Windows is a kind of Screen
dumbImage(&s); //ok, exact match
dumbImage(bv); //error, bitVector has no relation with screen
// tapi dumbImage harus dideklarasi virtual
class Screen {
public:
      virtual void dumbImage();
};
```

Contoh kasus mengenai inheritance (diambil dari Buku Lipman)



Zoo Animal Inheritance Graph

```
class ZooAnimal{ };
class Endangered{ };
class Carnivore{ };
class Herbivore{ };

// Bear : single inheritance
class Bear : public ZooAnimal{ };

// Cat, Panda : multiple inheritance.
// Cat : inherite dengan mode "private" dari Carnivore
// Tanpa keyword, by default adalah "private"
class Cat : public ZooAnimal, Carnivore { };
class Panda : private Endangered, public Bear, private Herbivore { };
Panda Ling_Ling; // derived class object
```

```
class ZooAnimal{
public:
      ZooAnimal (char*, char*, short);
      virtual ~ZooAnimal;

      virtual void draw();
      void locate();
      void inform();
protected:
      char* name;
      char* infoFile;
      short location;
      short count;
};
```

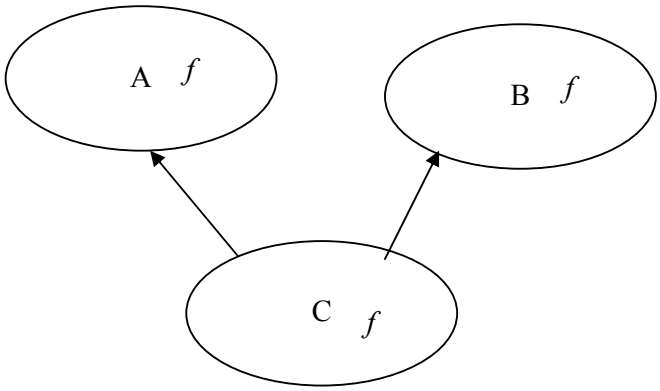
```
#include "ZooAnimal.h"

class Bear: public ZooAnimal{
public:
      Bear (char*, char*, short, char, char);
      virtual ~Bear;

      void locate(int);
      void isOnDisplay();
protected:
      char BeingFed;
      char IsDanger;
      short onDisplay;
      short epoch;
};
```

Inherited Member access

```
Void objectExample (Bear& ursus) {
      if (ursus.onDisplay() {
            ursus.locate(low_intensity);
            if (ursus.BeingFed)
                  cout << ursus.name << "is now being fed\n";
            ursus.inform()
      }
}

// Bagian if … sama, menulis scope adalah redundant
// if (ursus.BeingFed)
//  cout << ursus.ZooAnimal::name << "is now being fed\n";

// untuk memanggil
ursus.ZooAnimal::locate();
```

Scope wajib ditulis :

1. Kalau inherited member's name is reused in the derived class
2. When two or more base classes define an inherited member with the same name
   (A::*f* berbeda dengan B::*f* berbeda dengan C::*f*)




```
class Endangered { public : virtual void highlight (short);};
class Herbivore { public : virtual void highlight (short);};

class Panda: public Bear, public Endangered, public Herbivore{
public:
      void locate ();
//
};

Panda::locate(){
      if (isonDisplay() {
            Bear::locate(low_intensity);
            Highlight (location); // error
            Endangered::highlight(location); //OK
      }
}


// Panda::highlight sbb sekaligus menyediakan fungsionalitas yg ada
// di kedua parent
void Panda::highlight (){
      Endangered ::highlight (location);
      Herbivore ::highlight (location);
}
```

Base Class Initialization

Untuk single base class:

```
Bear::Bear (char* nm, char* fil, short loc, char danger, char age)
// member initialization list must be used to initialized
// a base class requiring arguments to it constructor
: ZooAnimal(nm,fil,loc)
{ epoch=age;
isDanger=danger;
}
// CHINE, BAMBOO, PANDA, MIOCENE:
// enumeration constants defined elsewhere

Panda::Panda (char* nm,  short loc, char sex)
// member initialization list must be used to initialized
// each base class requiring arguments to it constructor
// requiring arguments is palced within member initialization list—
// placement order is not significant
      : Endangered (CHINA),
        Herbivore (BAMBOO),
   Bear ("Auluropoda melaoleuca","Panda", CHINE, PANDA, MIOCENE)
{ name = new char [strlen(nm)+1];
  strcpy (name, nm);
  cell= loc;
  gender= sex;
}
```

Kasus :

1. Base class yang tidak mendefinisikan ctor atau punya cctor tanpa argumen, tidak perlu secara eksplisit diinisialisasi melaluai constructor initialization list
2. Sebuah Base class dapat dinisialisasi dengan argument list yang diharapkan oleh konstruktor. Dapat juga oleh class lain Class object can be the same base class type

```
Bear::Bear(ZooAnimal & z):
       ZooAnimal(z) { . . . };
```

Contoh dimana objek tidak punya relationship dengan base class, tetapi punya definisi convertion operator

```
Class GiantSloth: public Extinct {
public:
   GiantSloth();
   Operator Bear() { return b;} //GiantSloth==> Bear
Protected:
   Bear b;
}; // end GiantSloth

// invokes GiantSloth::operator Bear()
Panda::Panda(GiantSloth& gs):
   Bear(gs) { . . . };
```

Base class constructor diaktifkan sebelum konstruktor derived class diaktifkan

```
// illegal: explicit initialization of the
// inherited base class member ZooAnimal::name
```

```
Bear(char* nm): name(nm)
```

Bab 8.5. Public, protected, private Base Class

```
Class ZooAnimal{
public:
   void locate();
   void inform();
   //
};

class Bear: public ZooAnimal{};
class ToyAnimal : private ZooAnimal{
protected:
   // ok: ZooAnimal::locate() within ToyAnimal
   void where_is_it() { locate();}
};

void main() {
Bear yogi;
ToyAnimal pooh;
   yogi.inform(); //OK
   pooh.inform(); // error, private member
}
```

```
extern void draw (ZooAnimal*pz) { pz->draw();}

void main() {
  Bear fozzie;
  ToyAnimal eeyore;

  draw (&fozzie); //ok
  draw (&eeyore); //error: no standards conversion
  draw ( (ZooAnimal*) &eeyore) ; //ok

  ZooAnimal *pz= &eeyore; // error
  pz= (ZooAnimal*) &eeyore; //OK
}
```

When is a base class private ?

```
Class Base{
public:
   int set_val(int);
   //
};

Class Derived: private Base {
public:
   void mem1(Base&);
   void mem2();
   //
};
```

```
main() {
Base *bp=new Base; //ok
int ival = bp->set_val(1024); //ok
```

```
// . .
}
```

```
void main() {
      Derived d; //ok
      int ival = d.set_val(1024); //error: private
// . .

}
```

Public members of a Base

```
void main() {

Base *bp= new Derived;// error
      Derived d;
      d.mem1(d); //error
// . .
}
```

```
void Derived::mem2() {
      Base *bp = new Derived; //ok
      Derived d;
      d.mem1(d); //ok
// . .

}
```

Protected Base Class

The public interface of the hierarchy has to remain available to each class derived from Rodent; however, it must not be available to the general program through instances of either Rodent or those classes derived from Rodent.

```
Class Rodent : protected ZooAnimal {
// Mulai  Rodent, semua public di ZooAnimal menjadi protected
Public:
   virtual void assuringMessage();
protected:
      virtual void reportSightings(ZooLoc);
      unsigned long howMany();
};
```

```
Class Rat : public Rodent {
Public:
   // . .
```

```
protected:
      void log( ) { inform(); //ok ZooAnimal::inform() }
};
```

```
void main() {
      Bear anyBear;
      Rat anyRat;

      anyBear.inform(); //ok
      anyRat.inform(); // error, non public.
                       // Padahal di ZooAnimal adalah public

      anyRat.assuringMessage(); // ok

}
```

There is no standard conversion between a derived class and its nonpublic base class except within the scope of the derived class. For example

```
extern void draw (ZooAnimal*)

Rat::loc() {
 Rat ratso;
  draw (&ratso); // ok sebab scope ada di ratso::
}

void main() {
   Rat ratso;
   draw (&ratso); // error, no standard conversion. Scope: main
}
```

Private Base Class

Kelas stack yang merupakan turunan dari Array., karena ingin memanfaatkan kelas Array yang sudah ada. Dalam kasus ini, sebenarnya, **stack is not an array**. Stack harus menjamin LIFO, array tidak. Dengan memberikan mode secara private, maka Semua public interface dari base class becomes private in derived class

8.6. Standard Conversions under derivation

Ada empat konversi standard antara sebuah derived class dengan public base classes-nya:

1. Sebuah Derived class object akan dikonversi secara implisit ke public base class object
2. Sebuah Derived class reference akan dikonversi secara implisit ke public base class reference
3. Sebuah Derived class pointer akan dikonversi secara implisit ke public base class pointer

4. Sebuah Pointer ke class member of a base class akan dikonversi secara implisit ke pointer to a class member of a publicicly derived class

Selain itu, sebuah pointer ke any class object akan secara implisit dikonversi ke pointer bertype `void*`. Sebuah pointer bertype `void*` membutuhkan explisit cast supaya dapat dipakai

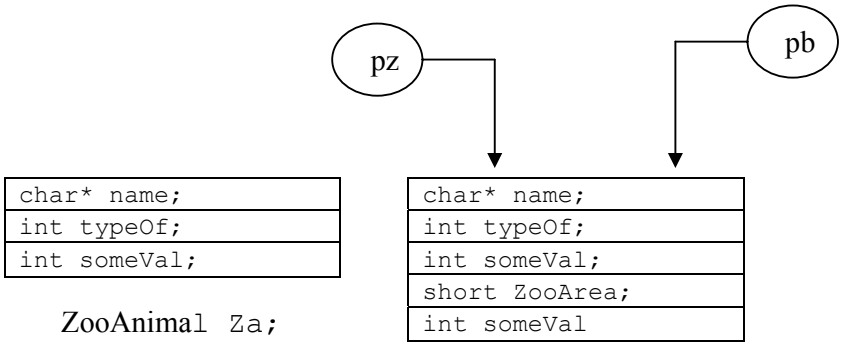
```
class Endangered { } endang;
class ZooAnimal { } zooanim;
class Bear: public ZooAnimal { } bear;
class Panda: public Bear { } panda;
class ForSale : private ZooAnimal { } forsale;

// all class publically derived from ZooAnimal
// can be passed to f() without an explicit cast
extern void f (ZooAnimal&);

f(zooanim);// ok exact match
f(bear);   // ok Bear ->ZooAnimal
f(panda);  // ok Panda -> ZooAnimal
```

Berikut ini ilegal :

```
// f(forsale);  error: ZooAnimal private
// f(endang); error : ZooAnimal not a base class
```

Model Object, memory map

```
class ZooAnimal{
public:
      int isA();
protected:
      char* name;
      int typeOf;
      int someVal;
};
#include "ZooAnimal.h"
class Bear: public ZooAnimal{
public:
      void locate( );

protected:
      short ZooArea;
      int SomeVal;
};
Bear b, *pb=&b;
ZooAnimal *pz=pb;
```



Perhatikan attachment sebagai berikut

```
//pb and pz address the Bear Object "b"
//ok : Bear::locate()
pb->locate();

//error: locate() is not a ZooAnimal member
pz->locate();

pb->someVal; // ok Bear::someVal;
pz->someVal; // ok: ZooAnimal::someVal
```

Except when virtual functions are being invoked, an explicit cast is necessary in order to access a derived class member through a base class object, reference, or pointer. For example :

```
// non object oriented implementation
void locate (ZooAnimal* pz) {
  switch pz->isA() {
     case Bear :
          ((Bear *)pz)->locate;
     case Panda :
          ((Panda *)pz)->locate;
     // . . .
   }
}
```

Perhatikan kode ini

```
//invoke the ZooAnimal member locate()
// cast its return value to Bear*
(Bear *) pz->locate()
```

The assignmnet or initialization of a derived class to one of its base classes always require an explisit cast by the programmer

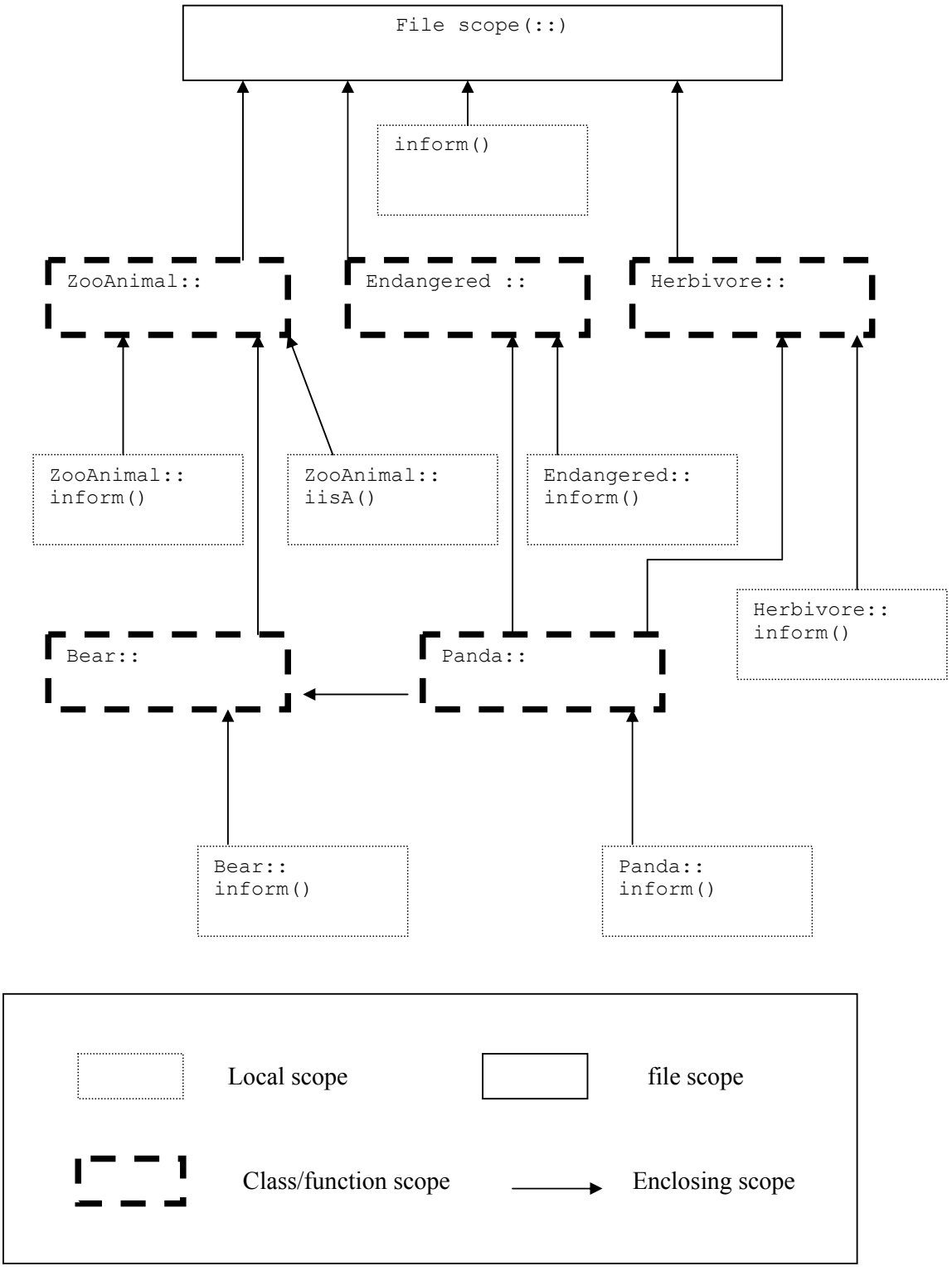
```
Bear *pb= new Panda; // ok, implisit conversion
Panda *pp=new Bear; //error : no implisit conversion
Panda *pp= (Panda *) new Bear; //ok


Bear b;
Bear*pb= new Panda; // ok

Panda *pp= (Panda *) pb; //ok, public base to derived,
                         // but it really is a Panda
pp->cell=24; //ok, programmer knew best in this case
pb=&b; //ok, pb now addresses just a Bear Object
*pp = (Panda *) pb; //Oops: pp address a Bear object this time to
Panda
pp->cell=24; // disaster, no Panda::cell
            // run-time area has been corrupted
            // this is a difficult bug to track down
```

```
int (Panda::*pm)()=0
Panda yinYang;
Pm_Panda = Bear::member; //ok, implicit conversion
(yinYang.*pm_Panda) (); // ok, save execution

int (Bear::*pm_Bear) ()=0;
Bear buddy;
Pm_Bear=Panda::member; //error, requires explicit cast


// explicit cast is necessary when
// initializing or assignning a derived class member to
// a pointer to class member of base class.
// Pm_Zoo, for example,
//is a ZooAnimal pointer to class member initialized to isA():
int (ZooAnimal::*pm_Zoo) ()= ZooAnimal::isA();
it is invoked by binding it to a class object of its type

ZooAnimal z;
(z.*pm_Zoo) ();

int (Bear::*pm_Bear)()=ZooAnimal::isA();
Bear b;
(b.*pm_Bear(); // ok
```

Let's assign pm_Zoo the address of the Bear member function locate ()
The assignment, because it is unsafe, requires an explicit cast

```
Pm_Zoo = (int (ZooAnimal::*) ()) Bear::locate();

ZooAnimal *pz=&pb; // pz address a Bear
(pz->*pm_Zoo)(); //ok
pz=&z;
(pz->*pm_Zoo)(); // disaster
```

8.7. Class Scope under derivation



```
extern inform();
class ZooAnimal {
public:
  int inform();
  int isA();
}
class Endangered {
public:
  int inform(ZooAnimal*);
}
class Herbivore {
public:
  int inform(ZooAnimal*);
}
class Bear : public ZooAnimal {
public:
  int inform(short);
}
class Panda: public Bear, public Endangered, public Herbivore {
public:
  int inform(char*);
}

int Panda::inform (char* str) {
//. . .

int kval =isA();  // resolved to ZooAnimal::isA()

}
```

Derivation is not overloading

```
// inherited ZooAnimal::locate() member has a signature distinct
// from that of the renamed Bear::locate(int)
Bear ursus;
ursus.locate(1); //ok Bear::locate(int);
ursus.locate(); // error, ZooAnimal::locate() is hidden

// to overload locate(), a second Bear instance must be define
inline void Bear::locate() {ZooAnimal.locate();}
```

8.8. Initialization and assignment under derivation

Tiga kasus yang dijelaskan di diktat
Base class constructor is always invoked before execution of constructor for the derived class. Jika base class constructor mempunyai argument list, harus ada member inisialization list.

Kasus 1 : derived class tidak punya cctor, base class punya

Jika derived class tidak punya cctor, default memberwise initialization ketika sebuah derived object diinisialisasi. Base class diinisialisasi dulu sesuai urutan pada deklarasi

```
class Carnivore { public: Carnivore(); };
class ZooAnimal {
public:
  ZooAnimal();
  ZooAnimal(const ZooAnimal&); //cctor };

class Cat : public ZooAnimal, public Carnivore{
    public:
      Cat(); //ctor
};

// Inisialisasi Ftritz  dengan Felix akan mengaktifkan
// ZooAnimal (const ZooAnimal &)
Cat Felix;
Cat Fritz=Felix;
```

Kasus 2 : derived class dan base class punya cctor

Penanganan inisialisasi dari base class menjadi tanggung jawab derived class

```
// ZooAnimal punya cctor

class Bear : public ZooAnimal{
    public:
       Bear(); //ctor
      Bear(const Bear&); //cctor
};

Bear Yogi;
Bear Smokey=Yogi;
// mengakibatkan pengaktifan konstruktor sbb
// Zooanimal();
// Bear (const Bear&);
```

Karena ZooAnimal mempunyai cctor, maka lebih disukai bahwa cctor yang diaktifkan, sbb:

```
Bear ::Bear (const Bear& b)
//ZooAnimal (const ZooAnimal &&) invoked
:ZooAnimal(b)
{     }

// Maka inisialisasi Smokey dengan Yogi mengakibatkan pengaktifan
// Zooanimal(const ZooAnimal& );
// Bear (const Bear&);
```

Kasus 3 : derived class punya cctor dan base class tidak

Default memberwise initialization is recursively applied to the base class
Contoh : kelas Point (Parent) mempunyai anak kelas PointLabel (Child) karena punya label yang bertype string maka punya cctor

Memberwise assignment : penanganannya sama dengan cctor

```
// ZooAnimal dan Bear mempunyai operator=
Bear ursus;
ZooAnimal on_loan;
OnLoan =  ursus; // operan kiri menenteukan type operan.
// Pada kasus di atas, ZooAnimal Part of ursus is assigned to onLoan
// Maka: ZooAnimal object operator is invoked

Bear Fozzie;
ZooAnimal on_loan=Fozzie; //invoke cctor of ZooAnimal
```

8.9. Initialization Order Under Derivation

Given simplified definition of Panda

```
class Panda:public Bear, public endangered, public herbivore {
      public:
            Panda();
            // . . . rest of public member
      protected:
            BitVector status;
            String name;
            // . . rest of Panda member
};
Panda yinYang;
```

Urutan pengaktifan konstruktor:

```
//1. Each base class constructor
ZooAnimal(); //base class of Bear
Bear();
Endangered();
Herbivore();
//2. Each member class declaration
BitVector();
String();
//3. derived class constructor
Panda();
```

Urutan pengaktifan destruktor: kebalikannya

Cobalah!!

8.10. Overloaded function with Class arguments

```
//A class object matches exactly
// only a formal argument of its own class type
// a class object and a reference to class object match
extern void ff (const Bear&);
extern void ff (const Panda&);
Panda yinYang;
ff(yinYang); //ok ff(const Panda&);

//A pointer to a class object matches exactly
//only a formal argumen of a pointer to its own class type
// an array of objects and a pointer to class object match
// berikut ini contoh yang salah kompilasi, karena
```

```
// overload mechanism tidak dapat membedakannya

extern void ff (Panda);
extern void ff (const Panda&);
```

Standard Conversion:

```
//a derived class object, ref or pointer is implicitly converted
// into a corresponding base class type
extern void ff (const ZooAnimal&);
extern void ff (const Screen&);
ff(yinYang); //ok: ff (const ZooAnimal&);

// a pointer to any class type is implicitly converted
// into a pointer of type void*
extern void ff (const Screen*);
extern void ff (const void *);
ff(&yinYang); //ok: ff (const void*);

//a conversion of a base class object, ref or pointer into its
corresponding derived class type is not applied
// the following call fails to achieve a match
extern void ff (const Bear &);
extern void ff (const Panda &);
ZooAnimal za;
ff(za); //error : no match;


// ambiguous
extern void ff (const Bear &);
extern void ff (const Endangered &);
// ff(yinYang); // error
// to resolve
ff(Bear(yinYang)); //ok

// a derived class is considered more nearly the type of its
immediate base class than of a base class further removed
// the following call is not ambiguous
// Panda is treated as being more nearly to Bear than
// a kind of ZooAnimal
extern void ff (const ZooAnimal &);
extern void ff (const Bear &);
ff(yinYang);

// given
extern void ff (void*);
extern void ff (ZooAnimal *);
ff(yinYang*); // match ZooAnimal*
```

User defined conversion

```
// constructor that take one argument, or
// an explisit conversion operator

class ZooAnimal{
public:
      // conversion long==>ZooAnimal
      ZooAnimal(long);
      // conversion ZooAnimal ==> char*
      operator char*();
```

```
}
// given
extern void ff (const ZooAnimal &);
extern void ff (const Screen &);
long lval;
ff(lval); //ok
// ff(1024); //ERROR: 1024 is int

// Conversion operator is only invoked if there is no match
// of standard conversion
extern void ff (const ZooAnimal &);
extern void ff (char);
long lval;
ff(ZooAnimal (lval)); // ok ff (const ZooAnimal &);


// ZooAnimal char* conversion operator is applied
// since there is no standard conversion from a bse class
// to a derived class object
extern void ff (const Panda*);
extern void ff (const void*);
ZooAnimal za;
ff(za)); // za==>char*=>void*; ok ff(const void*)
```

```
// Conversion operator (but not constructor) are inheried in the
// same way as of class member
// Bear, Panda inherit char* conversion from ZooAnimal
extern void ff (const char*);
extern void ff (const Bear*);
Bear yog;
Bear *pBear= &yogi;
ff(yogi); //yogi==>char*; ff(const char*)
ff(pBear); //ff(const Bear*)
```

```
// ambiguous examples
class Endangered {
public:
      //conversion:Endangered ==> int
      operator int();
};
class Extinct {
public:
      //conversion:Endangered ==> Extinct
      Extinct(Endangered&);
};
extern void ff (const Extinct &);
extern void ff (int);
Endangered e;
ff(e);// error, ambiguous
```

8.11. Inheriting operator function

Derived class inherits all the member functions of each base classes except the ctor, dtor and assignment operators of each of its base class

```
class Base {
public:
      int operator ==(const char*); // compare Base to char*
};

class Derived: public Base {
public:
      int operator ==(int); // compare Derived to char*
};

// program with compilation error
void main(){
Derived d1;
  if (d1==1024 && ok
       d1==”anabelle” ) // error
}
```

Solusi

```
class Derived: public Base {
public:
      int operator ==(int); // compare Derived to char*
      int operator ==(const char* ch) {
      // call the base instance explicitly
         return Base::operator==(ch);  }
};
```

Second situation

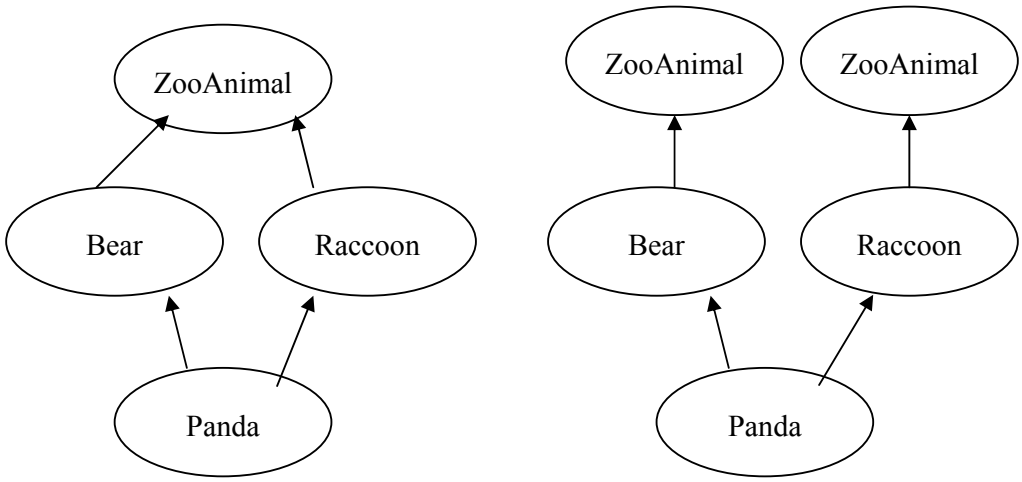
```
class Base {
public:
      const Base &operator =(int);
};

class Derived: public Base {
public:
      Derived();
};
// fail..
void main(){
 Derived d1;
 d1=1; // assignment is not inheritated
}
```

```
// inheritance of operator new and delete
// a class intended as a candidate for derivation, should always
// if it provides a delete operator,
// supply the optional second argument of type size_t
// This second argument (if present) is initialized with the size
// in bytes of the object that has been deleted.
// if this argument is not present and the derived class
// defined memory additional to that base class,
// there is no way for the delete operator
// to act upon the additional memory
```

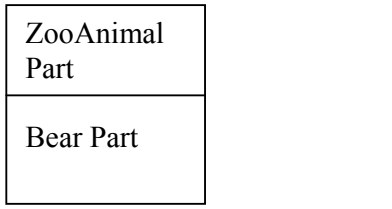
```
class Screen {
public:
      // opreator new, size_t is predefined type in stdef.h
      // size_t : size of the object for which the operator
      //           has been invoked
      void *operator new (size_t);
      // new char [ScreenChunk*size]
      // size adalah argument size_t

      //operator delete
      void operator delete(void* p, size_t);
};

class ScreenBuf: public Screen {
public:
      //
protected:
   long some_val();
};

//Each call of
new Screen;
// object is passed for a call
new ScreenBuf; // has unsufficient memory to contain
                  //ScreenBuf::some_val
// read : ambiguous pattern of Screen
// write: corrupts that memory
// HATI-HATI dengan SIZE ketika reuse new dan delete
```

9.1. VIRTUAL FUNCTIONS

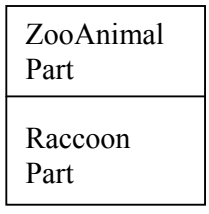
A virtual base class serves as a asingle, shared instance



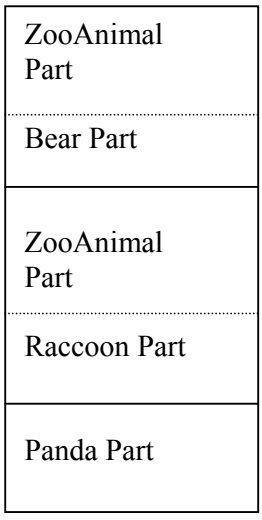

```
// tanpa virtual
class Panda: public Bear, public Racoon {.. }
// invocation
ZooAnimal();
Bear();
ZooAnimal();
Raccoon();
Panda();
```




```
class Bear:public ZooAnimal
```

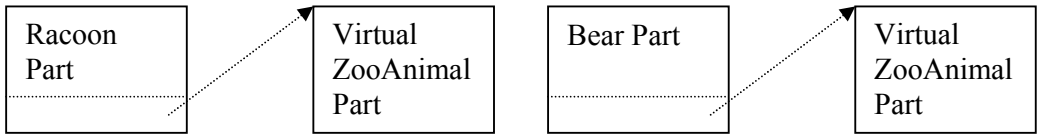



```
class Raccoon:public ZooAnimal
```

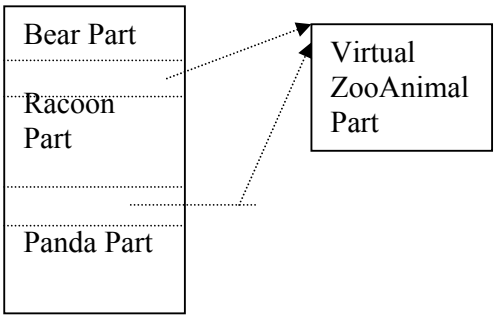



```
class Panda: public Bear,
             public Raccoon
```

Dengan Virtual




```
class Racoon: public virtual ZooAnimal  class Bear: public virtual ZooAnimal
```




```
class Panda: public  Bear: public virtual ZooAnimal
```

Inisialiasi sebuah virtual base class dilakukan oleh turunan paling akhir. Contoh di atas : oleh Panda. Contoh kosntruktor Panda

```
Panda:: Panda(char*nm):ZooAnimal(nm,PANDA), Bear(nm), Racoon(nm){. .}
```

Default ZooAnimal Ctor akan diaktifkan jika konstruktor Panda adalah sbb :

```
Panda:: Panda(char*nm): Bear(nm), Racoon(nm){. .}
```

Dominance

```
// Bear defines its own instance of the ZooAnimal locate() and
ZooArea

class Bear : public virtual ZooAnimal {
public:
      void locate();
protected:
      short ZooArea;
};

Bear pooch;
pooch.locate();// Bear::locate()

Raccoon rocky;
rocky.locate(); // Zooanimal::locate();

// Contoh berikut, tanpa adanya virtual, akan ambiguous
class Panda:public Raccoon, public Bear{ };
Panda tui_li;
tui_li.locate(); //eror, ambiguous
```

Urutan penciptaan objek

Virtual base class akan diciptakan lebih dulu dari non-virtual base class tanpa memperhatikan urutan dalam list ataupun hirarki turunan

```
// example
class Character { . . };
class BookCharacter:public Character { };
class TeddyBear: public BookCharacter,
                  public Bear;
                  public virtual ToyAnimal { };
TeddyBear pooh;
// urutan penciptaan:
ToyAnimal(); //immediate virtual base
ZooAnimal();// Bear's virtual base
Character(); // BookCharacter's nonvirtualbase
BookCharacter(); // immediate nonvirtual base
Bear();// immediate nonvirtual base
TeddyBear(); // derived class constructor
```