BAB VII
PEWARISAN

Untuk mendapatkan subtipe dari suatu tipe, dapat digunakan mekanisme inheritansi atau pewarisan (*inheritance*). Sebagai contoh , kita dapat membuat IntArrayRC sebagai suatu subtipe dari IntArray yang memberikan pemeriksaan selang (*range*) dari nilai indeksnya sehingga tidak akan muncul kesalahan program pada saat mendefinisikan variabel array dengan indeks melebihi batas maksimumnya.

```
# include “IntArray.h”
class IntArrayRC : public IntArray {
public :
     IntArrayRC (int = ArraySize) ;
     IntArrayRC (const int *, int) ;
     int & operator [ ] (int);
} ;
```

Kelas IntArrayRC adalah turunan dari kelas IntArray. IntArrayRC hanya perlu mendefinisikan yang berbeda atau sebagai tambahan terhadap IntArray, yaitu misalnya: (a) operator subkrip atau indeks yang mengikutsertakan pengecekan range, (b) opreasi masuk pengecekan range, dan (c) konstruktor-konstruktor karena elemen ini tidak ikut diturunkan. Implementasi untuk operator indeks dan konstruksi-konstruksinya adalah:

```
# include < assert .h>
int& IntArrayRC :: operator [ ] (int indeks) {
     assert ( indeks  > = 0   && indeks < size ) ;
     return ia [ indeks ];
}

IntArrayRC :: IntArrayRC (int sz)
          : IntArray (sz) {  }
IntArrayRC : : IntArrayRC  ( const int *iar , int sz )
          :IntArray  (iar , sz ) {  }
```

Dengan demikian, kesalahan indeks dalam program berikut akan terdeteksi:

```
# include “ “IntArray.h”
main ( )
{
     IntArrayRC  ia ;
     for ( int = 1 ; i < ia.getSize ( ) ; ++ i )
          ia [i] ;
     return 0;
}

Output :
Assertion failed  : indeks > = 0  && indeks < size
```

FUNGSI VIRTUAL

Implementasi fungsi virtual dapat bergantung pada tipe kelas dari obyek yang memanggilnya. Jika suatu kelas BASE mempunyai suatu fungsi virtual VF dari kelas DERIVED yang diturunkan dari BASE juga mempunyai suatu fungsi VF yang sama tipnya, maka pemanggilan fungsi VF untuk objek dari kelas DERIVED akan menjalankan DERIVED :: VF, walaupun melalui suatu pointer atau referensi ke BASE. Perhatikan contoh berikut:

```
#include < iostream.h>

class BASE {
public :
     virtual void vf1 ( )      { cout << " BASE : : vf1  ( ) \n" ;}
     virtual void vf2 ( )      { cout << "BASE : :  vf2 ( ) \n " ;}
     void f ( )  { cout  << "BASE : : f ( ) \n " ; }
} ;

class DERIVED : public BASE  {
public :
     void vf1  ( )    { cout << "DERIVED : : vf1 ( ) \n" ;}
     void vf2 (int)  { cout << "DERIVED : : vf2 ( ) \n" ; }
     void  f ( )  {  cout << "DERIVED : : f ( ) \n " ; }
} ;

void main ( )   {
     DERIVED d ;
     BASE *  bp = &d ;
     bp → vf1 ( ) ;                    // panggil DERIVED : : vf1
     bp → vf2 ( ) ;                    // panggil BASE : : vf2
     bp → f ( ) ;                      // panggil BASE : : f
}
```

Pemeriksaan runtime pada kelas IntArrayRC dapat diimbangi dengan membuat operator [] sebagai fungsi interval sehingga suatu aplikasi dapat menggabungkan tipe kelas IntArrayRC dengan tipe IntArrayRC dalam suatu program seperti pada contoh berikut:

```
#include < iostream.h>
const int ArraySize = 12 ;

class IntArray  {
public :
     IntArray ( int sz = ArraySize) { size = sz , ia = new int [ size ] ;}
     int getSize ( )   { return size ; }
     virtual int& operator [ ] ( int index )  { return ia [index] ; }
};

void swap (IntArray &array , int I, int j)   {
     int tmp = arra [ i ] ;
     array [ i ] =array [ j ];
     array [ j ]= tmp;
}
```

```
void main ( )  {
     IntArray ia ;
     IntArrayRC iaRC;              // ada kesalahan keluar dari range
     cout << “swap  ( ) dengan IntArray ia \n “;
     int size = ia.getSize ( ) ;
     swap  (ia, 1,size ) ;         // tidak ada range  checking
     cout << “ swap ( ) dengan IntArrayRC iaRC \n “ ;
     size = iaRC.getSIze ( );
     swap (iaRC,1,size) ;          // ada range checking
}
```

TEMPLATE

Dengan kata kunci Template memugkinkan tipe sebagai parameter dalam suatu fungsi seperti terlihat pada contoh berikut:

```
#include < iostream .h>
#include <assert.h>
const intArraySize = 12 ;

template  < class Tipe>
class Array  {
public :
     Array ( int sz = ArraySize)  { size = sz ; ia = new Tipe [size] ; }
     int  getSize ( )  { return size  ; }
     virtual Tipe& operator [ ] (int index)  { return ia [ index ] ; }
protected ;
     int size ;
     Tipe *ia;
};

classArrayRC  : public Array  < Tipe >    {
public :
     ArrayRC (int size = ArraySize ) : Array , <Tipe> ( sz ) { } ;
     Tipe& operator [ ] ( int index )  {
          assert   ( index >= 0 && index < size) ;
          return ia [index]; }
};

void main ( )  {
     Array < int > ia ( 5 ) ;
     Array < double > da ( 5 );
     Array <char >  ca ( 5 );
     int indeks;
     for ( index = 0 ;indeks<ia.getSize ( ) ; ++index)  {
          ia [indeks] = indeks;
          da [indeks] = indeks *1.25;
          ca [indeks ] = ia [indeks] + 97 ;
     }
```

```
    for (indeks = 0 ;indeks < da.getSize ( ); ++indeks )
        cout << " [ << indeks >>"] ia : "<< [indeks]
          << "      ca : " << [ indeks ] <<"    da : " << da [ indeks ] << end1;
}
```

Keluarannya :

```
[ 0 ] ia : 0    ca : a      da : 0
[ 1 ] ia : 1    ca : b      da : 1.25
[ 2 ] ia : 2    ca : c      da : 2.5
[ 3 ] ia : 3    ca : d      da : 3.75
[ 4 ] ia : 4    ca : e      da : 5
```

KONVERSI TIPE DAN PENURUNAN

Ada empat konversi baku dari suatu kelas turunan ke kelas basis publiknya, yaitu:

1. Obyek kelas turunan akan dikonversi secara implisit ke obyek kelas basis publik-nya.
2. Reference kelas tutunan akan dikonversi secara implisit ke reference kelas basis publik-nya.
3. Pointer kelas turunan akan dikonversi secara implisit ke pointer kelas basis publik-nya.
4. Pointer ke suatu member dari suatu kelas basis akan dikonversi secara implisit menjadi pointer ke suatu member dari suatu publicly-derived-class.

Konversi-konversi dengan arah sebaliknya memerlukan *cast* secara eksplisit, walaupun perlu diingat bahwa hal ini tidak aman dalam pemrograman. Pelajari contoh berikut ini:

```
#include<iostream.h>

class buah {
public:
    buah (int init_i, float init_f): i(init_i), f(init_f)
        { cout << "buah(int,float)." << endl; }
    void print();
    int i;
    float f;
};

void buah::print() {
    cout << "i = " << i << endl << "f = " << f << endl;
}

class pepaya: public buah {
public:
    pepaya(int init_i, float init_f, int init_j, float init_g)
        : buah(init_i, init_f), j(init_j), g(init_g)
        { cout << "pepaya(int,float,int,float).\n"; }
    void print();
    int j;
    float g;
};
```

```
void pepaya::print()  {
      buah::print();
      cout << “j = “ << j << endl << “g = “ << g << endl;
}

int main()  {
      buah b(2,3.1);
      pepaya p(4,5.2,6,7.3);

      buah* bp = &p;
      cout << “bp->print():\n”;
      bp->print();
      cout << endl;

      pepaya *pp = (pepaya *)&b;
      cout << “pp->printf():\n;
      pp->print();
      cout << endl;

      typedef int buah::* buah_int_ptr;
      typedef int pepaya::*pepaya_int_ptr;

      pepaya_int_ptr pip = &buah::i;
      cout << “p.*pip = “ << p.*pip << endl << endl;

      buah_int_ptr bip = (buah_int_ptr) &pepaya::j;
      cout << “b.*bip = “  << b.*bip << endl << endl;

      typedef void (buah::*buah_func_ptr)();
      typedef void (pepaya::*pepaya_func_ptr)();

      pepaya_func_ptr pfp = buah::print;
      cout << “(p.*pfp)():\n;
      (p.*pfp)();
      cout << endl;

      buah_func_ptr bfp = (buah_func_ptr)pepaya::print;
      cout << “(b.*bfp)():\n”;
      (b.*bfp)();

      return 0;
}
```

Output dari program tersebut adalah:

```
buah(int,float).
buah(int,float).
pepaya(int,float,int,float).
bp->print():
i = 4
f = 5.2
pp->print():
i = 2
f = 3.1
j = 0
g = 3.741467e-43
```

```
p.*pip = 4
b.*pip = 0

(p.*pfp)():
i = 4
f = 5.2

(b.*bfp)():
i = 2
f = 3.1
j = 0
g = 3.741467e-43
```

Selain pewarisan(inheritansi), OOP dicirikan juga oleh *dynamic binding* (pengikatan dinamis) yang diperoleh melalui penggunaan fungsi virtual. Fungsi virtual mendefinisikan operasi yang begantung pada tipe dalam hierarki inheritansi. Fungsi virtual adalah suatu fungsi anggota khusus yang diinvokasi melalui referensi atau pointer kelas basis publik. Fungsi virtual diikat secara dinamis pada *run-time*. Instance yang diinvokasi ditentukan berdasarkan tipe kelas dari obyek sesungguhnya yang ditunjuk oleh pointer atau refernsi tersebut.

Suatu fungsi virtual didefinisikan dengan mendahului deklarasi fungsi dengan kata kunci **virtual**. Hanya fungsi-fungsi anggota suatu kelas dapat dideklarasikan virtual, dan kata virtual hanya boleh muncul dalam class-body. Perhatikan contoh berikut:

```
#include<iostream.h>
class B {
public:
    virtual void print() { cout << “Dalam D.\n”;}
};

class D : public B {
public:
    void print() { cout << “Dalam D.\n”;}
};

main() {
    B *pB, bB;
    D dD;
    pB = &bB;
    pB->printf();          //B::print()
    pB = dD;
    pB->printf();          //D::print()
}
```

Suatu fungsi virtual dapat dideklarasikan (bukan didefinikan) sebagai fungsi virtual murni (*pure virtual function*) dengan menginilisasikan deklarasi itu dengan ∅. Contoh:

```
class Bclass {
public:
     virtual void print(void) =∅ ;
          :
          :
};
```

Kalau dengan satu atau lebih fungsi virtual murni hanya dapat dipakai sebagai base class untuk derivasi selanjutnya dan tidak dibolehkan untuk membuat obyek dari kelas yang mempunyai fungsi virtual murni. Kalau yang pertama kali mendeklarasikan suatu fungsi bersifat virtual maka harus memberikan suatu definisi untuk fungsi virtual tersebut atau mendeklarasikan fungsi itu sebagai suatu fungsi virtual murni.

Jika suatu definisi diberikan, fungsi itu berlaku sebagai default untuk kelas-kelas turunan yang tidak menyediakan definisi untuk fungsi virtual itu. Dan jika fungsi itu dideklarasikan sebagai fungsi virtual murni, maka kelas-kelas turunan harus memberikan definisinya atau mendeklarasikan lagi fungsi itu sebagai suatu fungsi virtual murni.

Pendefinisian kembali suatu fungsi virtual dalam kelas turunan harus mengikuti persis nama, signature, dan return-type yang terdapat dalam kelas basisnya. Kata kunci virtual boleh tidak ditulis.

Tingkat akses suatu fungsi virtual ditentukan oleh tipe kelas dari reference atau pointer tempat fungsi virtual itu diinvokasi. Walaupun destinator mempunyai nama sesuai dengan nama kelas, destinator dapat dideklarasikan virtual. Pelajari contoh program berikut ini :

```
#include<iostream.h>
class buah {
public:
     buah(int init_i, float init_f): i(init_i), f(init_f)
     {
          cout << “buah(int,float).\n”;
     }
     virtual~buah() { cout << “~buah().\n”; }
     virtual void print();
private  :
     int i;
     float f;
};

void buah::print(){
     cout<<” i= “<< i << endl;
     cout<<”f= “<< f << endl;
}
```

```
class pepaya : public buah  {
public :
      pepaya ( int init_i,float init_f,int init_j,float init_g)
            : buah (init_i,init_f), j(init_j), g(init_g)
      {
            cout<< “pepaya (int, float,int,float).\n”;
      }
      ~pepaya(){ cout <<”~pepaya().\n”; }

      void print();
private:
      int j;
      float g;
};

void pepaya :: print() {
      buah :: print ();
      cout << “j=” << j << endl;
      cout << “g=” <<g << endl;
}

int main() {
      buah  f(2, 3.1);
      cout << “f.print():\n”;
      f.print();
      cout << endl;
      pepaya p (4, 5.2, 6, 7.3);
      cout << “p.print()\n”;
      p.print();
      cout << endl;
      f = p;
      cout << “f.print():\n”;
      f.print();
      cout << “ \n “ ;
      return 0;
}

main() {
      buah * bp = new buah (2, 3.1);
      cout << ”bp → print(): \n;
      bp → print(); cout << endl;
      delete bp; cout << endl:
      bp = new pepaya (4, 5.2, 6, 7.3);
      cout << “bp → print() : \n”;
      bp → print(); cout << endl;
      delete bp;
      return 0;
}
```

Output dari program tersebut adalah:

```
buah(int, float).
bp → print():
i=2
f=3.1
~buah().

buah (int,float).
pepaya(int,float,int,float).
bp → print():
i=4
f=5.2
j=6
g=7.3
~pepaya().
~buah().
```

KELAS BASIS VIRTUAL
(*Virtual Base Class*)

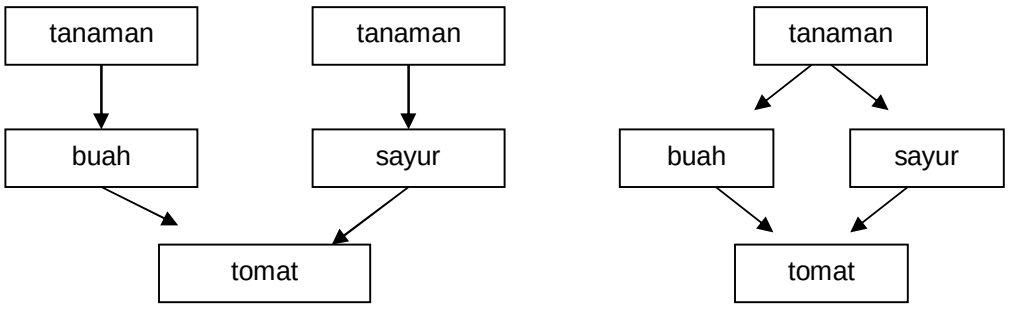
Penggunaan kelas basis virtual adalah untuk mengatasi mekanisme pewarisan default sehingga perancang kelas dapat menentukan suatu shared-base-class. Tak peduli berapa kali suatu kelas basis virtual muncul dalam hirarkhi derivasi, hanya suatu instance dibuat dan suatu kelas basis virtual dispesifikasikan dengan menggunakan kata kunci virtual dalam deklarasinya. Misalnya:

```
class D : public virtual B {.....}
```

menunjukkan bahwa B adalah suatu kelas basis virtual untuk kelas D.

Jika suatu kelas basis virtual mendefinisikan konstruktor, kelas basis virtual itu harus mendefinisikan suatu konstrutkor yang tak memerlukan argumen. Biasanya suatu kelas turunan dapat meng-init secara eksplisit hanya kelas-kelas basis yang langsung di atasnya. Kelas-kelas basis virtual merupakan perkecualian.

Kelas basis virtual di-init oleh kelas turunan terbawah/terakhir. Pelajari contoh berikut ini:



NON – VIRTUAL VIRTUAL

```
#include<iostream.h>

class tanaman {
public :
      tanaman() : i(0), f(0) {cout << “tanaman().\n”; }
      tanaman (int init_i,float init-f):i(init-i),f(init-f)
            {cout << “tanaman(int,float).\n”; }
      ~tanaman() {cout << “~tanaman().\n”; }
      void print();
private :
      int i;
      float f;
};

void tanaman :: print() {
      cout << “i= “ << i << endl;
      cout << “f= “ << f << endl;
}

class buah : virtual public tanaman {
public :
      buah (int init_j, float init_g) : j(init_j), g(init_g)
            {cout << “buah (int,float).\n”; }
      ~buah() { cout << “~buah().\n”; }
      void print();
private :
      int j;
      float g;
};

void buah :: print() {
      tanaman :: print();
      cout << “j= “<< j<< endl;
      cout << “g=”<< g<< endl;
}

class sayur : public virtual tanaman {
public :
      sayur (int init_k,float init_h) : k(init_k), h(init-h)
            {cout << “sayur(int,float).\n”; }
      ~sayur() {cout << “~sayur().\n”; }
      void print();
private :
      int k;
      float h;
};

void sayur :: print() {
      tanaman :: print() ;
      cout << “k= “ << k << endl;
      cout << “h=” << h << endl;
}
```

```
class tomat : public buah, public sayur {
public :
    tomat (int init_l,  float init_m) : buah(init_l, init_m), sayur(int init_l, init_m),
    l(init_l+2), m(init_m +2.1)
        {cout << "tomat (int,float).\n"; }
    ~tomat() { cout << "~tomat().\n"; }
    void print();
private:
    int l;
    float m;
};

void tomat :: print() {
    tanaman :: print();
    buah :: print();
    sayur :: print();
    cout << "l= " << l << endl;
    cout <<"m= "<< m << endl;
}

main() {
    tanaman *pt = new tanaman (2, 3.1);
    cout << "pt → print(): \n";
    pt→ print();
    delete pt;
    cout << endl;

    buah *pb = new buah (2, 3.1);
    cout << "pb→ print() :\n";
    pb→print();
    delete pb;
    cout << "\n";

    sayur *ps = new sayur (2, 3.1);
    cout << "ps →print() :\n";
    ps → print();
    delete ps;
    cout << endl;

    tomat *pto = new tomat (2, 3.1);
    cout << "pto →print(): \n";
    pto → print();
    delete pto;
    return 0;
}
```

Output dari program tersebut adalah:

```
tanaman (int,float).
pt→print():
i=2
f=3.1
~tanaman().

tanaman().
```

```
buah(int,float).
pb→print():
i=0
f=0
j=2
g=3.1
~buah().
~tanaman().
tanaman().
sayur(int,float).
ps→print():
i=0
f=0
k=2
h=3.1
~sayur().
~tanaman().
tanaman().
buah(int,float).
sayur(int,float).
tomat(int,float).
pto→print():
i=0
f=0
i=0
f=0
j=2
g=3.1
i=0
f=0
k=2
h=3.1
l=4
m=5.2
~tomat().
~sayur().
~buah().
~tanaman().
```

Konstruktor kelas basis virtual selalu dijalankan sebelum konstruktor kelas basis non-virtual, tak peduli posisinya dalam derivation-list. Sedangkan urutan destruktor sebaliknya, dan jika suatu kelas turunan melibatkan sekaligus instance public dan instance private dari suatu kelas basis virtual,maka yang menang adalah instance public. Contoh:

```
class tanaman {
public :
    void habitat();
protected:
    short tinggi;
};

class buah : public virtual tanaman {...};
calss sayur : private virtual tanaman {...};
class nangka : public buah, public sayur {...};
```