Classes

Class adalah metode logical untuk organisasi data dan fungsi dalam struktur yang sama. Class dideklarasikan menggunakan keyword **class**, yang secara fungsional sama dengan keyword **struct**, tetapi dengan kemungkinan penyertaan fungsi sebagai anggota, formatnya sbb :

```
class class_name {
  permission_label_1:
    member1;
  permission_label_2:
    member2;
  ...
  } object_name;
```

Dimana ***class_name*** adalah nama class(user defined *type*) dan field optional ***object_name*** adalah satu atau beberapa identifier objek yang valid. *Body* dari deklarasi berisikan ***members***, yang dapat berupa data ataupun deklarasi fungsi, dan ***permission labels*** (optional), dapat berupa satu dari tiga keyword berikut : **private:**, **public:** atau **protected:**. Digunakan untuk menentukan batasan akses terhadap *members* yang ada :

- **private** , anggota class dapat diakses dari danggota lain pada kelas yang sama atau dari class "*friend*".
- **protected** , anggota dapat diakses dari anggota class yang sama atau class *friend* , dan juga dari anggota class turunannya(*derived*).
- **public** , anggota dapat diakses dari class manapun.

Default permission label : **private**

Contoh :

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Deklarasi class **CRectangle** dan object bernama **rect**. Class ini berisi empat anggota: dua variable bertipe **int** (**x** and **y**) pada bagian **private** (karena private adalah default permission) dan dua fungsi pada bagian **public** : **set_values()** dan **area()**, dimana hanya dideklarasikan propotype_nya.

Perhatikan perbedaan antara nama classdan nama object. Pada contoh sebelumnya **CRectangle** adalah nama class(contoh, user-defined type), dan **rect** adalah object dari tipe **CRectangle**. Sama halnya dengan deklarasi berikut :

```
int a;
```

int adalah nama *class* (type) dan **a** adalah nama *object*(variable).

Contoh :

```
// classes example
#include <iostream.h>

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
}
```

Output :

area: 12

Hal baru dari contoh diatas adalah operator :: dari lingkup yang disertakan dalam pendefinisian **set_values()**. Digunakan untuk mendeklarasikan anggota dari class diluar class tersebut.

Scope operator (::) menspesifikasikan class dimana anggota yang dideklarasikan berada, memberikan scope properties yang sama seperti jika dideklarasikan secara langsung dalam class.

Contoh :

```
// class example
#include <iostream.h>

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
```

```
int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;

}
```

Output **:**

```
rect area: 12
rectb area: 30
```

Perhatikan pemanggilan **rect.area()** tidak memberikan hasil yang sama dengan pemanggilan **rectb.area()**. Ini disebabkan karena objek dari class **CRectangle** mempunyai variable sendiri, **x** dan **y**, dan fungsi **set_value()** dan **area()**.

Constructor dan destructor

Objek biasanya memerlkan inisialisasi variable atau menentukan memori dinamik selama proses untuk mencapai hasil akhie yang diharapkan dan menghindari pengembalian nilai yang tidak diharapkan.

Untuk mengatasinya dapat digunakan fungsi spesial : *constructor*, yang dapat dideklarasikan dengan pemberian nama fungsi dengan nama yang sama untuk class. Fungsi dari constructor ini akan dipanggil secara otomatis ketika instance baru dari sebuah class dibuat. Contoh :

```
// classes example
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

Output :

```
rect area: 12
rectb area: 30
```

Hasil dari contoh diatas sama seperti contoh sebelumnya. Dalam hal ini, hanya menggantikan fungsi **set_values**, yang sudah tidak ada dengan class *constructor*. Perhatikan cara parameter diberikan ke constructor pada saat instance class dibuat :

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Destructor berfungsi kebalikannya. Secara otomatis akan dipanggil jika objek di keluarkan dari memory, ataupun karena keberadaannya telah selesai (contoh : jika didefinisikan sebuah objek local dalam function dan function tersebut selesai) atau karena merupakan objek yang secara dinamis ditetapkan dan dikeluarkan dengan menggunakan operator **delete**.

Destuctor harus mempunyai nama yang sama dengan class, diberi awalan tile (~) dan tidak mengembalikan nilai. Contoh :

```
// example on constructors and destructors
#include <iostream.h>

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
}

int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Output :

```
rect area: 12
rectb area: 30
```

Overloading Constructors

Sama halnya seperti fungsi, constructor juga dapat mempunyai nama yang sama tetapi mempunyai jumlah dan tipe yang berbeda pada parameternya. Pada saat pemanggilan kompiler akan meng-eksekusi yang sesuai pada saat objek class di deklarasikan.

Pada kenyataannya, ketika dideklarasikan sebuah class dan tidak disebutkan constructornya, maka kompiler secara otomatis akan mengasumsikan dua constructor overloaded ("*default constructor*" dan "*copy constructor*"). Contoh :

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

Jika tanpa constructor, Kompiler secara otomatis mengasumsikan anggota-anggota fungsi constructor berikut :

- **Empty constructor**

  Merupakan constructor tanpa parameters didefinisikan sebagai ***nop*** (blok instruksi kosong). Tidak melakukan apapun.

  ```
  CExample::CExample () { };
  ```

- **Copy constructor**

  Merupakan constructor dengan satu parameter dengan tipe yang sama yang ditetapkan untuk setiap anggota variable class nonstatik objek yang disalin dari objek sebelumnya.

  ```
  CExample::CExample (const CExample& rv) {
    a=rv.a;  b=rv.b;  c=rv.c;
    }
  ```

Penting untuk mengetahui, bahwa kedua constructor default : *empty construction* dan *copy constructor* ada jika tidak ada constructor lain yang dideklarasikan. Jika terdapat constructor dengan sejumlah parameter dideklarasikan, maka tidak satupun dari constructors default ini ada.

Contoh :

```
// overloading class constructors
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

Output :

```
rect area: 12
rectb area: 25
```

Contoh diatas **rectb** dideklarasikan tanpa parameter, sehingga diinisialisasikan dengan *constructor* tanpa parameters, yang mendeklarasikan **width** dan **height** dengan nilai **5**.
Perhatikan jika dideklarasikan objek baru dan tidak diberikan parameter maka tidak diperlukan tanda kurung **()**:

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

Relationships between classes

Friend functions (friend keyword)
Terdapat tiga akses berbeda untuk anggota class : **public**, **protected** dan **private**. Untuk anggota *protected* dan *private*, tidak dapat diakses dari luar fungsi dimana mereka dideklarasikan. Namun, aturan ini dapat di lewati dengan menggunakan keyword *friend* dalam class, sehingga fungsi eksternal dapat mengakses anggota **protected** dan **private** suatu class.

Untuk itu, harus dideklarasikan prototype dari fungsi eksternal yang akan mengakses, Contoh :

```
// friend functions
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}
```

```
int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
}
```

Output :
24

Dari fungsi **duplicate**, yaitu *friend* dari **CRectangle**, dapat mengakses anggota **width** dan **height** untuk objek yang berbeda dari tipe **CRectangle**. Perhatikan bahwa dalam deklarasi **duplicate()** maupun **main()** tidak dipertimbangkan apakah **duplicate** merupakan anggota dari class **CRectangle**. Secara umum keguanaan fungsi *friend* diluar methodology pemrograman, jadi jika memungkinkan sebaiknya menggunakan anggota dari clas yang sama.

Friend classes (friend)
Selain dapat mendefinisikan fungsi friend, dapat juga didefinisikan class sebagai *friend* dari class lainnya, sehingga memungkinkan class kedua dapat mengakses anggota **protected** dan **private** class pertama. Contoh :

```
// friend class
#include <iostream.h>

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area (void)
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
```

```
  cout << rect.area();
  return 0;
}
```

Output :

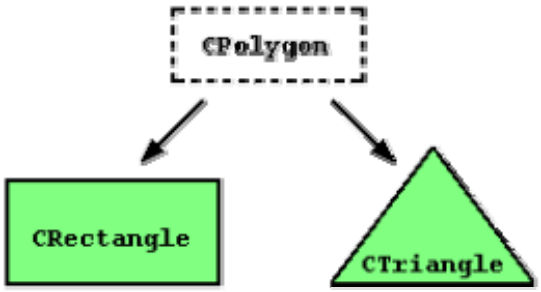
16

Pada contoh diatas, dideklarasikan **CRectangle** sebagai *friend* dari **CSquare**, sehingga **CRectangle** dapat mengakses anggota **protected** dan **private** dari **CSquare**, lebih jelasnya **CSquare::side**, Mendefinisikan lebar dari kotak.

Juga terdapat *empty prototype* dari class **CSquare** diawal program. Merupakan hal yang penting karena dalam deklarasi **CRectangle** mengacu kepada **CSquare** (sebagai parameter dalam **convert()**). Pendefinisian **CSquare** disertakan nanti, jika tidak disertakan pada deklarasi sebelumnya untuk **CSquare**, maka class ini tidak akan terlihat dalam pendefinisian **CRectangle**.

Inheritance between classes

Inheritance memungkinkan kita untuk membuat objek dari objek sebelumnya, sehingga memungkinkan untuk menyertakan beberapa anggota objek sebelumnya ditambah dengan anggota objeknya sendiri. Contoh, membuat class untuk mengetahui apakah segi empat (**CRectangle**), atau (**CTriangle**). Masing-masing mempunyai hal yang sama yaitu, dasar dan tinggi. Dapat direpresentasikan dengan class **CPolygon** kemudian diturunkan menjadi **CRectangle** dan **CTriangle**.



Class **CPolygon** dapat berisi anggota yang dipakai untuk setiap *polygon,* dalam hal ini **width** dan **height**, dan **CRectangle** dan **CTriangle** adalah class turunannya.

Class turunan akan menurunkan seluruh anggota yang dimiliki oleh class dasar(parent)nya. Jadi jika class paret mempunyai anggota **A** dan diturunkan pada class lain dengan anggota **B**,maka clas turunan ini akan memiliki **A** dan **B**.

Untuk menurunkan class, menggunakan operator **:** (colon) pada saat deklarasi, syntax :

```
class derived_class_name: public base_class_name;
```

Dimana *derived_class_name* adalah nama dari *derived* class dan *base_class_name* adalah nama dari class asal. **public** dapat digantikan dengan tipe akses lainnya : **protected** atau **private**, Contoh:

```
// derived classes
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Output :

20
10

Class **CRectangle** dan **CTriangle** masing-masing mengandung anggota dari **CPolygon**, yaitu : **width**, **height** dan **set_values()**.

Rangkuman tipe akses dan anggota yang bisa mengaksesnya :

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not-members	yes	no	no

Dimana "*not-members*" merepresentasikan referensi dari luar class, seperti dari **main()**, dari class lain atau dari fungsi lain baik global ataupun local.

Pada contoh diatas, anggota yang diturunkan kepada **CRectangle** dan **CTriangle** diikuti dengan hak akses yang sama dengan class asalnya, **CPolygon**:

```
CPolygon::width          // protected access
CRectangle::width        // protected access
```

```
CPolygon::set_values()    // public access
CRectangle::set_values()  // public access
```

Ini diakibatkan karena menurunkan class sebagai **public** :

```
class CRectangle: public CPolygon;
```

What is inherited from the base class?

Pada dasarnya setiap anggota dari class dasar diturunkan kepada class turunannya, kecuali :

- **Constructor and destructor**
- **operator=() member**
- **friends**

Jika class dasar tidak mempunyai default constructor atau akan melakukan pemanggilan terhadap constructor overloaded ketika objek turunan yang baru dibuat maka dapat dituliskan dalam setiap definisi constructor dari class turunan:

```
derived_class_name (parameters) : base_class_name
(parameters) {}
```

Contoh :

```
// constructors and derivated classes
#include <iostream.h>

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (1);
  son daniel(1);

  return 0;
}
```

Output :

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Terlihat perbedaan mana yang merupakan constructor dari mother yang dipanggil ketika objek **daughter** dibuat dan ketika objek **son dibuat**. Perbedaannya disebabkan dari deklarasi untuk **daughter** dan **son**:

```
daughter (int a)          // nothing specified: call
default constructor
son (int a) : mother (a)  // constructor specified: call
this one
```

Multiple inheritance

Dalam C++ memungkinkan untuk menurunkan field atau method dari satu atau lebih class dengan menggunakan operator koma dalam deklarasi class turunan. Contoh, akan dibuat class untuk menampilkan dilayar (**COutput**) dan akan diturunkan ke class **CRectangle** and **CTriangle** maka dapat dituliskan :

```
class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {
```

Contoh :

```
// multiple inheritance
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class COutput {
  public:
    void output (int i);  };

void COutput::output (int i) {
  cout << i << endl;  }

class CRectangle: public CPolygon, public COutput {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon, public COutput {
  public:
    int area (void)
      { return (width * height / 2); }
```

```
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

Output :

20
10