OOP dan C++ Part 2

Yohanes Nugroho
Departemen Teknik Informatika
Institut Teknologi Bandung

Beberapa hal yang sudah dipelajari

- Objek dan kelas
- Constructor dan Copy Constructor
- Destructor
- Function Overloading

"The time has come," the Walrus said, "To talk of many things: Of shoes--and ships--and sealing-wax- Of cabbages--and kings- And why the sea is boiling hot- And whether pigs have wings."

THROUGH THE LOOKING GLASS

Statik

Static Data Member
And
Static Function Member

Static member

- Static data member adalah data member yang di-share oleh semua objek instan dari suatu kelas
- Static function member adalah fungsi yang bisa dipanggil tanpa perlu menciptakan instan dari kelas tersebut
  - Static function hanya bisa mengakses static member
- Kata kunci untuk keduanya adalah "**static**"

Contoh Data Member

```
class Point { public:
  static int count;
  void incCount() { count++;}
}
  Point p1, p2;
  p1.incCount();
  p2.incCount();
  /*Nilai count di P1 dan P2 sama
    (akhirnya sama-sama menjadi 2)*/
```

Contoh Static Function

```
class Point { public:
   static int count;
   void incCount() { count++;}
   static printCount() {cout << count; }
}
```

- Cara mengaktifkan:

```
Point::printCount();
```

- Tanpa membuat instans kelas Point

Operator Overloading

Operator Overloading

- C++ mengijinkan kita mempersingkat penulisan method tertentu dengan **operator overloading**
- Perhatikan contoh berikut:

```
class Point {
   bool isEqual(Point &otherPoint){  /*bla
    bla */ }
}
Point p1, p2;
if (p1.isEqual(p2)) { /* */ }
```

Bisa dipersingkat dan dipercantik ...

- Sintaks ini:

```
bool isEqual(Point &p){
   return (p.x==x || p.y==y);
}
```

- Bisa Diubah Menjadi:

```
bool operator==(Point &p){
   return (p.x==x || p.y==y);
}
```

Sehingga ...

- Kita bisa memanggilnya seperti ini:

```
if (p1 == p2) { /* bla */}
```

  - atau:

```
if (p1.operator==(p2)) { /* bla */}
```

- Tentu saja "cantik" itu relatif (dan subjektif),
  - bagi penulis kelas, mungkin merepotkan
  - bagi sebagian pengguna kelas akan terasa mudah dan natural

Operator yang mengembalikan nilai

```
class complex { /*complex number*/
    complex operator+(complex right) {
    /**/ }
}
```

- Jika operator tidak mengubah diri (this), tapi mengembalikan nilai lain, tipe kembalian adalah kelas itu sendiri
- Jika operator mengubah diri (this), maka kembaliannya adalah reference ke this (operator yang mengubah diri misalnya: +=, -=, dll)

Contoh

```
class complex { /*complex number*/
    complex operator+(complex right) {
      complex result;
      result.riil = riil + right.riil;
      result.im = im + right.im;
      return result;
   }
```

- Contoh penggunaan:
  - `complex d = d1 + d2;`

Jika operator mengubah diri (this) ...

```
class complex {
   complex& operator+=(complex other){
      r.rill += other.riil; r.im += other.im;
      return (*this);
   }
}
```

- Penggunaan:

```
complex a;
a += complex(1,1);
```

- Perhatikan Jika Operator mengubah diri maka maka kembaliannya **pasti** dirinya sendiri

```
return (*this);
```

Operator di luar kelas

- Kita bisa membuat operator di luar definisi kelas, misalnya:

```
complex operator+(complex a, complex
  b);
```

- Operator + di atas seperti fungsi biasa, namanya saja yang berbeda, sama saja dengan:

```
complex jumlahkompleks(complex a,
  complex b);
```

Operator yang bisa dibuat

- Ada banyak operator yang bisa dibuat:

```
-+, -, *, /, %, ^, &, |, ~, !, =, <,
>, +=, -=, *=, /=, %=, ^=, &=, |=,
<<, >>, >>=, <<=, ==, !=, <=, >=,
&&, ||, ++, --, ->, *, ",", ->, [],
(), new, new[], delete, delete[]
```

- Umumnya hanya operator ***perbandingan***, ***operator =*** dan ***operator aritmatik*** saja yang sering diimplementasikan

Operator = (assignment)

- Ini adalah operator yang memiliki makna khusus
- Untuk mempersingkat penjelasan, operator ini selalu dideklarasikan seperti ini:

```
class X {
   X& operator=(const X&right);
}
```

- Sifatnya mirip copy constructor

Beda isi copy constructor dengan assignment operator

- Dalam copy constructor, isi objek pasti belum ada, karena baru saja dideklarasi
  - `Point p1 = p2; /*p1 belum berisi*/`
- Dalam asignment operator, isi objek pasti sudah ada
  - `Point p1; Point p2; /*p1 dan p2 punya isi*/`
  - `p1 = p2; /* isi p1 harus dihilangkan, isi dengan p2 */`

Kapan operator assignment dipanggil?

- Dalam assignment, tidak disertai inisialisasi, assignment operator dipanggil

```
p1 = p2; /*operator = dipanggil */
```

- Dalam deklarasi+inisialisasi, copy constructor dipanggil

```
Point p = p1; /*copy constructor
  dipanggil*/
```

Kapan perlu copy constructor dan assignment operator?

- Kedua benda tersebut diperlukan jika kelas memegang resource tertentu (misal: memori) yang harus dipakai eksklusif antar objek
- Sebagai pegangan yang sederhana: jika ada memori yang dialokasikan di kelas, maka perlu copy constructor dan assignment operator

Beda Implementasi

- Dalam copy constructor, tidak perlu dealokasi resource lama (ingat, objek masih "kosong")
- Dalam assignment operator, perlu dealokasi resource lama (karena sudah ada "isinya"), dan perlu mengembalikan objek saat ini
  - secara praktis: ada delete untuk setiap resource yang dialokasi dan ada `return (*this)`

Contoh

```
int *nilai;
Mahasiswa(Mahasiswa &mhs){
   nilai = new nilai[mhs.n]; /*salin nilai,
     bla bla ...*/
}
Mahasiswa &operator=(Mahasiswa mhs){
   delete[] nilai;
   nilai = new nilai[mhs.n]; /*salin nilai */
   return (*this);
}
```

Fungsi pre increment/decrement

- Perhatikan contoh operator ++

```
Date& operator++() {
   day += 1; /*hitung implikasi terhadap
    bulan dan tahun, dst */
   return (*this)
}
```

- Ini hanya dipanggil pada kasus:

```
Date d;
++d; /*pre increment operator*/
d++; /*tidak dipanggil */
```

Fungsi post increment/decrement

- Untuk membuat agar bekerja pada post incerement, tambahkan parameter int:

```
Date& operator++(int) {
    day += 1; /*hitung implikasi terhadap bulan dan
      tahun, dst… */
    return (*this)
}
```

- Komite pembuat C++ mempunyai persoalan untuk menentukan operator post increment/decrement, sehingga ditambahkanlah parameter **int** *dummy* yang tidak terpakai
- Operator tersebut akan dipanggil untuk d++;

Initializer

Initializer (Constructor initialization list)

- C++ mengijinkan konstruksi ini:

```
class Point {
   public: int x, y;
   Point(int nx, int ny): nx(x), ny(y) { }
}
```

- Dan lebih disarankan dibanding:

```
Point(int nx, int ny) {
   nx = x; ny = y;
}
```

Mengapa?

- instruksi tersebut dengan jelas terlihat menginisialisasi member variabel dengan nilai yang jelas (mengurangi error)
- instruksi tersebut diperlukan untuk const member
- instruksi tersebut lebih efisien (lihat contoh di slide beikutnya)
- instruksi tersebut diperlukan untuk penurunan

Const member

- Kelas boleh memiliki member data const
- Member ini hanya boleh diinisialisasi ketika objek diciptakan

```
class Test {
   const int r;
   Test(int x) : r(x) { }
}
```

Lihat contoh

- Jika konstruktor menerima Objek, maka:

```
Garis(Point np1, Point np2): p1(np1), p2(np2) { }
```

  Tidak akan membuat objek baru

- Sedangkan:

```
Garis(Point np1, Point np2){
 p1 = np1;
 p2 = np2;
}
```

- Akan membuat aliran eksekusi yang sangat kompleks ... *[cobalah]*

Penjelasan ...

- Perhatikan instruksi

  `p1 = np1;`

  - `np1` diciptakan (parameter)
  - pada (=), asignment operator `p1` dipanggil dengan parameter `np1`
  - pada (}), destruktor `np1` dipanggil

  – Hal yang sama terjadi pada `p2 = np2;`

Catatan

- Ada satu catatan tambahan: member initialization tidak bisa dilakukan terhadap data member yang sifatnya statik

Sekarang...

Semua yang berhubungan dengan sebuah kelas sudah cukup kalian kuasai.

Tapi kelas bisa punya turunan ...

Inheritance

Penurunan/Inheritance

- Konsep ini agak rumit dibanding yang lain, jadi perhatikan baik-baik
- Sebuah kelas bisa diturunkan dari kelas lain
  - Menambah method baru
  - Menambah property baru
  - Mengubah sifat method yang sudah ada
  - Memungkinkan polimorfisme

Jika sudah bisa menjawab ini semua, Anda sudah mengerti inheritance

- Apa gunanya inheritance?
- Bagaimana caranya menurunkan?
- Apa visibilitas member parent?
- Apa jadinya jika ada method yang namanya sama?
- Apa yang harus dilakukan di konstruktor dan destruktor turunan?
- Apa itu virtual function? kelas abstrak? polimorfisme?

Penurunan sederhana: penambahan

- Jika kita punya kelas Lingkaran:

```
class Lingkaran() {
  public: int r;
  int getLuas()  { return pi*r*r; }
  int  getKeliling() { return 2 * pi * r}
}
```

- Dan kita ingin membuat kelas lingkaran yang lain yang berwarna
  - ada property int warna
  - ada method `void setWarna(int w)` dan `int getWarna()`

Diturunkan menjadi

```
class LingkaranBerwarna: public Lingkaran {
   int warna;
   void setWarna(int w) { warna = w;}
   int getWarna() { return warna;}
   }
```

- Semua method kelas warna masih bisa dipanggil:

```
LingkaranBerwarna lb = new
  LingkaranBerwarna();
cout << lb.getLuas();
```

Hubungan kelas

- Sebuah kelas bisa diturunkan dari kelas lain, kelas orang tua disebut sebagai superclass dan kelas anak disebut **derived class** atau **subclass**
- Kelas LingkaranBerwarna is a kind of Lingkaran, *subclass is a kind of superclass*
- Kelas (misal Mobil) bisa punya properti berupa objek (misal Roda), hubungan semacam ini disebut *client supplier*, atau has a (Mobil has a Roda)

Access modifier

- Jika kelas Lingkaran memiliki jari-jari (int r) yang sifatnya **private**, maka kelas LingkaranBerwarna tidak bisa mengakses r
- Jika kelas lingkaran memiliki sesuatu (method/property) yang sifatnya protected maka LingkaranBerwarna bisa mengaksesnya
- Semua yang publik di Lingkaran boleh diakses siapa saja (termasuk kelas turunannya)

Visibilitas penurunan

- Perhatikan:
  ```
  class LingkaranBerwarna: public
  Lingkaran{ }
  ```
- Public artinya: modifier akses di kelas anak, sama dengan kelas bapak (private ya private, public ya tetap public), **ini yang paling umum dipakai**
- Kita bisa memakai kata kunci **`private`** dan **`protected`**, yang artinya adalah ....

Private dan Protected

- `class LingkaranBerwarna: private Lingkaran{ }`
  - Semua yang protected dan public dilingkaran jadi private
- `class LingkaranBerwarna: protected Lingkaran{ }`
  - Semua yang protected tetap protected
  - Semua yang public menjadi protected

Konstruktor turunan

- Jika A turunan B, maka:
  - Jika B punya konstruktor default, dan kita tidak memanggil konstruktor B, maka konstruktor default B dipanggil otomatis
  - Jika B tidak punya konstruktor default, maka kita harus memanggil secara eksplisit konstruktor b
- Jika A turunan B maka A harus memanggil konstruktor B baik secara langsung (otomatis) maupun tak langsung (manual)

Contoh

- Point punya konstruktor default, turunan otomatis memanggil Point() secara implisit

```
class Point {
   Point() { } /*default (0,0) */
   Point(int x, int y) { }
}
```

- Maka boleh:

```
class Point3D() {
      Point 3D() { } /*otomatis memanggil
        Point() */
}
```

Contoh lain

- Garis tidak punya konstruktor default, akan ada error

```
class Garis {
   Garis(Point p1, Point p2) { }
}
```

- Maka tidak boleh:

```
class GarisBerwana() {
     GarisBerwarna(Point p1, Point p2) { }
      /*tidak otomatis memanggil Point(p1,
      p2) */
}
```

Cara yang benar

```
class GarisBerwana() {
      GarisBerwarna(Point p1, Point p2):
       Garis(p1,p2) { }
}
```

- Perhatikan bagian yang ditebalkan, konstruktor Garis dipanggil secara eksplisit
- Agar tidak bingung dan aman:
  - **Selalu panggil konstruktor parent yang sesuai, secara eksplisit**
  - `class Point3D() { Point 3D() : Point(){ } } }`

Meng-override method

- Jika sifat suatu method menjadi berubah pada turunannya (misalnya method Draw di kelas Lingkaran berubah di Lingkaran berwarna), maka gunakan kata kunci **virtual**:

```
virtual void Draw();
```

- Kita bisa mendefinisikan ulang method tanpa keyword virtual, tapi efeknya mungkin tidak seperti yang diinginkan (legal, tapi tidak moral). Lihat contoh berikut

Contoh (dengan virtual)

```
class Rakyat {
  public:  virtual void print() {
      cout << "Rakyat jelata";  }
}
class Raja : public Rakyat {
  void print() { cout << "Paduka YM";}
}
Raja *r = new Raja();
Rakyat *rakyat = &r; /*boleh, raja adalah
  rakyat*/
rakyat->printNama(); /* tampil "Paduka YM" */
```

Tanpa Virtual

```
class Rakyat {
  public:   void print()
  { cout << "Rakyat jelata"; }
}
class Raja : public Rakyat {void print() {
  cout << "Paduka yang mulia"; }
}
Raja *r = new Raja();
Rakyat *rakyat = &r;//boleh, raja is a rakyat
rakyat->printNama();//tampil "Rakyat Jelata"
```

Jadi apa itu overriding?

- Overriding (redefine) adalah pendefinisian ulang suatu method di kelas turunan
  - secara harfiah overidding artinya "penimpaan"
- Method yang dioverride harus ada di kelas parent dan sudah didefinisikan, kecuali untuk method yang *virtual murni (pure virtual).* Method virtual murni tidak punya body di kelas parent, sehingga bukan disebut sebagai "override", melainkan "define"

Method Virtual Murni

```
class Bangun {
   virtual float getLuas() = 0;
}
```

- Method ini **tidak didefinisikan** isinya dalam kelas Bangun (= 0 menyatakan method virtual murni)
- Kelas yang punya method virtual murni di sebut dengan kelas **abstrak**
- **Kelas abstrak tidak bisa diinstansiasi**

Kelas Abstrak

- Kelas untuk mendefinisikan kontrak dengan kelas turunan
  - Misal, semua bangun harus punya `float getLuas();`
  - Kelas yang semua methodnya virtual murni disebut dengan `interface`
- Interface berguna untuk membuat algoritma yang bekerja pada aneka objek yang mengimplementasikan interface tertentu

Kelas Bangun dan turunannya

```
class Lingkaran : public Bangun {
   float getLuas(){return pi * r * r;}
}
class BujurSangkar: public Bangun {
   float getLuas(){return  sisi * sisi;
    }
}
Lingkaran * l = new Lingkaran();
cout << l->getLuas(); /*biasa*/
```

Polymorphism

Polymorphic attachment

- Hal ini bisa dilakukan:

```
Lingkaran * l = new Lingkaran();
/*boleh juga seperti ini, lingkaran
  adalah bangun*/
Bangun *b = new Lingkaran();
```

- atau :

```
BujurSangkar bs;
Bangun &b = bs;
```

- Ini yang namanya polimorfik attachment
- hanya bisa dengan pointer dan reference

Polymorphic attachment

- Sebuah reference/pointer bisa menunjuk ke berbagai objek dan sifatnya menjadi berubah mengikuti apa yang ditunjuk
- kita bisa membuat:

```
void printBangunPtr(Bangun *b) { cout <<
 bangun->getLuas(); }
```

  - atau

```
void printBangunRef(Bangun b){ cout <<
 bangun.getLuas(); }
```

Contoh Penggunaan

```
Lingkaran ling(5);
BujurSangkar bs(4);
printBangunPtr(&ling);//luas lingkaran
printBangunPtr(&bs);//luas bujursangkar
```

atau:

```
printBangunRef(ling); //luas lingkaran
printBangunRef(bs);// luasbujursangkar
```

Perhatikan

- Ketika mengoverride sesuatu, perhatikan apakah perlu memanggil method parent, jika ya, panggil dengan:
  `NamaKelasParent::NamaMethod();`
- misal, kelas Lingkaran punya method draw
  - Kelas LingkaranBerwarna:

```
void draw() {
   Lingkaran::draw();
   /*lalu warnai lingkaran*/
}
```

Mengoverride destruktor

- Jika punya fungsi virtual, **kemungkinan** Anda harus menggunakan virtual destruktor
  - Tambahkan keyword virtual di depan nama destruktor
- Untuk mudahnya, jika Anda punya virtual function, selalu nyatakan destruktor sebagai virtual
- Penjelasannya adalah ...

Penjelasannya agak rumit ...

- Technically speaking, you need a base class's destructor to be virtual if and only if you intend to allow someone to invoke an object's destructor via a base class pointer (this is normally done implicitly via delete), and the object being destructed is of a derived class that has a non-trivial destructor. A class has a non-trivial destructor if it either has an explicitly defined destructor, or if it has a member object or a base class that has a non-trivial destructor (note that this is a recursive definition (e.g., a class has a non-trivial destructor if it has a member object (which has a base class (which has a member object (which has a base class (which has an explicitly defined destructor)))))).
- Sumber: The C++ Language FAQ

Lingkaran bukan elips....

- Terkadang ada sesuatu yang tidak sesuai intuisi di OO, misalnya kita memiliki kelas elips dengan method
  - `void setXYDiameter(int dx, int dy){ xd = dx; yd = dy}`
  - Dan method lain-lain
- Maka kita tidak bisa membuat kelas Lingkaran yang sifatnya tidak menyalahi kontrak (kelas yang konsisten)

Mengapa?

- Lingkaran hanya punya 1 diameter
  - Method tidak bisa dihilangkan dari turunan
  - method `setXYDiameter(x, y)` tidak bisa dibuat konsisten
- Jadi dalam di sini kita tidak bisa membuat kelas Lingkaran dengan menurunkan Elips
  - *Circle is not a kind of ellipse*

Tapi ...

- *Seorang Phd di bidang matematika, mengatakan bahwa lingkaran itu adalah sebuah elips [FAQ C++]*
- Ya di matematika, tidak di OOP
  - lebih tepatnya lagi, hubungan *is a kind of* di OOP tidak seperti di dunia nyata
- Jadi perhatikan jika membuat kelas, intuisi terkadang menipu
  - Intinya orang tua (superclass) tidak boleh menjanjikan terlalu banyak (kasihan anaknya)

Multiple inheritance

- C++ Mendukung multiple inheritance

```
class A: public C, public D {
   /**/
}
```

- Jika mungkin, **jangan gunakan fitur ini**
  - **ini nasihat untuk programmer pemula**
- Dalam kasus sederhana, jika tidak ada konflik (nama, kontrak) antara kedua orang tua, maka hidup akan baik-baik saja
  - Jika ada, maka hidup menjadi rumit

Resolusi konflik

- class A turunan dari B dan C, sementara itu B dan C punya method draw
  - `A *a = new A;`
  - `a->B::draw(); /*draw milik B dipanggil */`
  - `a->C::draw();`
- supaya tidak sulit, biasanya di kelas A dibuat method draw yang memanggil A dan B, sehingga `a->draw()` akan memanggil draw milik A

Friend

Friend

- *C++ Also supports the notion of friends, cooperative classes that are permited to see each others private parts* [Grady Booch]
- Friend memungkinkan suatu entitas lain (fungsi atau kelas) di luar kelas untuk mengakses bagian private milik kelas tersebut
- Biasanya friend jarang dipakai karena menandakan design kelas yang kurang baik

Contoh Friend

```
class B {
   friend class A;
   friend void f(B a, int);
}
```

- Kelas A bisa mengakses bagian private B
- Fungsi f bisa mengakses bagian private B

Template Fungsi dan Kelas

Template Fungsi

- Daripada memiliki banyak fungsi seperti ini:

```
void myswap(int &a, int &b) {  }
void myswap(Orang &a, Orang &b) { }
```

- Kita bisa membuat makro (seperti di C), atau lebih baik lagi: sebuah template fungsi

```
template <class Tipe>
void swap(Tipe &a, Tipe &b) {
   Tipe tmp; tmp = a; a = b; b = a;
}
```

Sifat template fungsi

- Waktu dibuat, fungsi tidak didefinisikan, baru ketika dipakai maka fungsi didefinisikan
- kata kunci **class** tidak menyatakan suatu kelas (dalam pengertian OO), tapi tipe sembarang
- Boleh ada lebih dari satu tipe yang dijadikan template

```
template <class TipeA, class TipeB>
void test(TipeA &a, TipeB &b) { /**/}
```

Sifat template ...

- Template boleh digabung dengan non template

```
template <class Tipe>
void find_min(Tipe t[], int size) { }
```

- Semua signature fungsi harus tipe yang sudah dikenal, atau tercantum dalam class di keyword template

```
template <class T2, class T3>
T1 fungsi(T2 &t2, T3 &t3) { }
/*tidak boleh jika T1 tidak terdefinisi*/
```

Template kelas

- Template fungsi bisa diperluas menjadi template kelas

```
template <class Tipe> class Stack{
  void push(Tipe);
}
Stack<Tipe>::push(Tipe){
  /*lakukan sesuatu*/
}
```

Secara umum

- Ganti tipe yang tidak generik menjadi nama tipe yang generik, misalnya jika sudah punya stack integer, gunakan Tipe yang dideklarasikan di template
  - Tanpa template:

```
class Stack{
   void push(int a);
}
```

  - Dengan template

```
template <class Tipe> {
   void push(Tipe a);
}
```

Deklarasi Objek

- Jika membuat objek temporer dengan tipe yang tidak generik, ubah menjadi versi generik
  - misal:
    - `Stack tmp;`
  - Menjadi
    - `Stack<Tipe> tmp;`

Memakai template kelas

```
Stack<int> s; /*sebuah stack
  integer*/
  -s.push(10);
Stack<char> c; /*sebuah stack
  char*/
  -c.push('a');
```

Penutup

- Masih ada banyak hal yang belum dibahas mengenai C++
  - Fitur-fitur yang jarang dipakai tidak dibahas
- C++ juga masih mengalami perkembangan
  - Draft fitur C++ yang baru sudah ditulis oleh Stroustrup
  - Menurutnya nanti C++ yang baru akan lebih mudah dipelajari