

Pengenalan Paradigma Objek

IF2210 – Semester II 2022/2023

Tim Pengajar IF2210

Tujuan

- › Mahasiswa memahami dan dapat membedakan metodologi terstruktur (prosedural) dibandingkan dengan berorientasi objek dalam pengembangan perangkat lunak
- › Mahasiswa mengenal Object-Oriented Language dan metodologi “terkait” OO (object-based programming, visual programming, event driven programming)
- › Mahasiswa mendapatkan “sense” mengenai perbedaan aplikasi yang dikembangkan secara OO dan bukan, dengan harapan pada saat kelak mengembangkan aplikasi OO, akan mampu menerapkan konsep OO

Dua Pendekatan Pembangunan Perangkat Lunak

- › Struktural
 - › Sudut pandang pemrogram adalah “*how computers work*” (bagaimana data direpresentasikan dalam memori; **urutan instruksi/proses** terhadap data).
 - › Biasanya analisis dimulai dari input-proses-output.
 - › Inspirasi: proses bisnis yang diterjemahkan ke struktur program (**urutan instruksi + control flow (conditional, loop)**)
- › *Object-Oriented*
 - › Sudut pandang pemrogram adalah objek apa saja yang terlibat dalam domain persoalan.
 - › Analisis dimulai dari identifikasi objek beserta **perilakunya** dan **bagaimana objek-objek saling berinteraksi**.
 - › Inspirasi: sel biologis, “software IC”

Analogi OO

- › Sel biologis – Alan Kay (pencipta OO, GUI, & bahasa Smalltalk)
 - › Alan Kay's favorite metaphor for software objects is a biological system. Like cells, software objects don't know what goes on inside one another, but they communicate and work together to perform complex tasks.
- › Software IC – Brad Cox (pencipta Bahasa Objective-C)
 - › (Dalam buku *Object-oriented programming: an evolutionary approach*, 1986)
 - › “a desirable trait for software objects is the ability to use them as standardized and interchangeable parts to ‘mass-produce’ larger constructs,” seperti IC (integrated circuit) pada elektronika (hardware).

Contoh: structural vs. OO (1)

```
program frequency;
const
    size = 80;
var
    s: string[size];
    i: integer;
    c: character;
    f: array[1..26] of integer;
    k: integer;
begin
    writeln('enter line');
    readln(s);
    for i := 1 to 26 do f[i] := 0;
    for i := 1 to size do
    begin
        c := asLowerCase(s[i]);
        if isLetter(c) then
        begin
            k := ord(c) - ord('a') + 1;
            f[k] := f[k] + 1
        end
    end;
    for i := 1 to 26 do
        write(f[i], ' ')
end.
```

› Struktural: program Pascal untuk menghitung jumlah kemunculan setiap huruf pada sebuah String.

Contoh: structural vs. OO (2)

```
| s c f k |
f := Array new: 26.
s := Prompter prompt: 'enter line' default: ''.
1 to: 26 do: [:i | f at: i put: 0].
1 to: s size do: [
  :I | c := (s at: i) asLowerCase.
  c isLetter ifTrue: [
    k := c asciiValue - $a asciiValue + 1.
    f at: k put: (f at: k) + 1.
  ].
].
^ f
```

- › Bahasa OO (Smalltalk), cara berpikir masih prosedural.

Contoh: structural vs. OO (3)

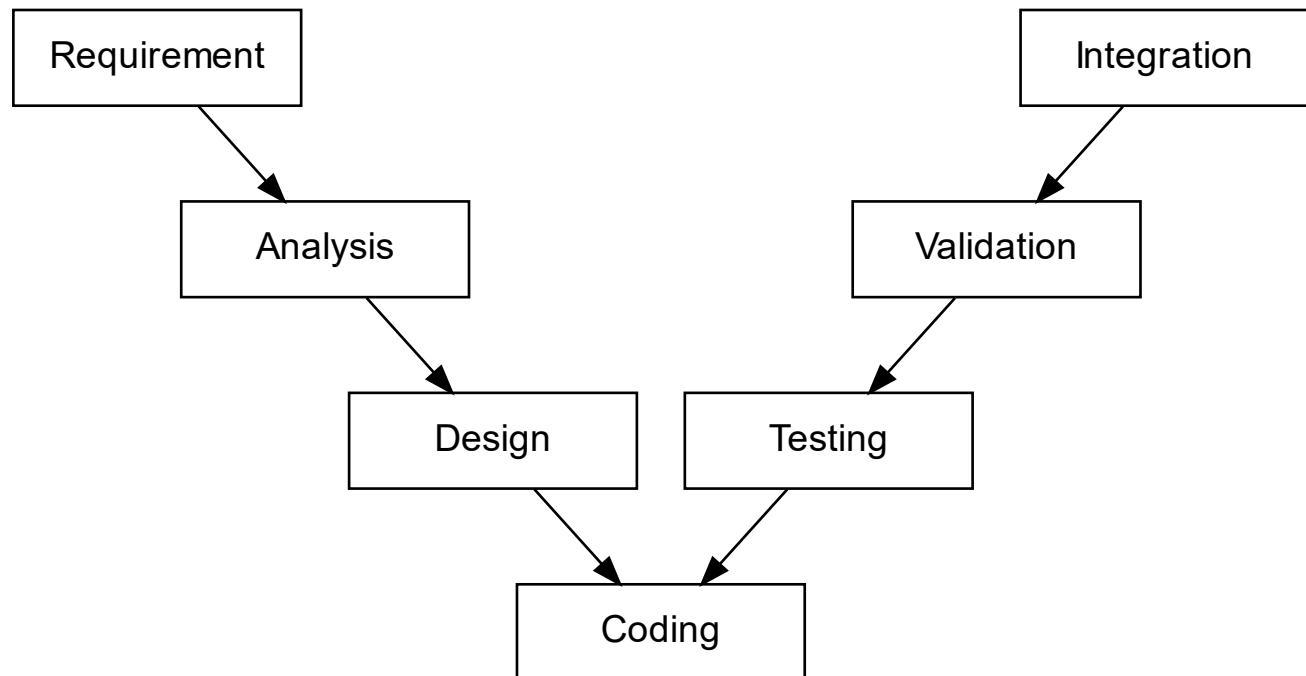
```
| s f |
s := Promter prompt: ' enter line ' default: '' .
f := Bag new.
s do: [ :c | c isLetter ifTrue: [f add: c asLowerCase]] .
^ f.
```

“untuk setiap c bagian dari s, lakukan:
jika c adalah letter, tambahkan c
dalam bentuk huruf kecil ke dalam f”

- › Real OO using Smalltalk.

OO* and Software Lifecycle (Konteks OOP)

- › 00A
- › 00D
- › 00P
- › 00T

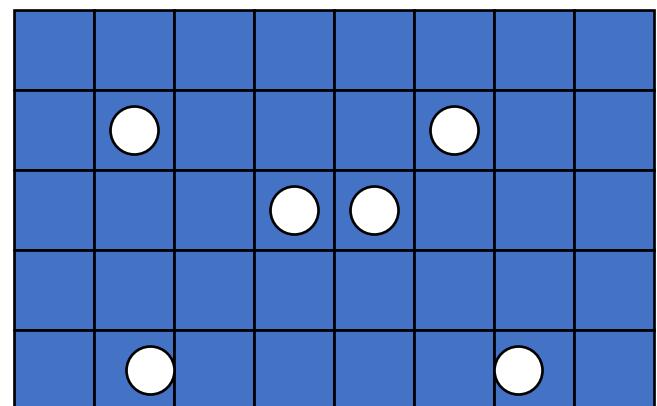
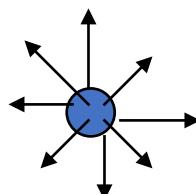


Dalam Software Lifecycle...

- › Ada penentuan *requirement*, untuk apa SW dibuat, data, fungsi, behaviour
- › Ada artefak desain (model PL):
 - › Memodelkan “statis” (*state*, atribut) dan “dinamika” (*behaviour*, perilaku)
 - › bagaimana dekomposisi menjadi modul, *class*, ... Dengan notasi tertentu, misalnya diagram kelas, CRC card, dll.
- › Implementasi desain menjadi source code (*file*) dan artefak lain, sesuai dengan kaidah yang baik. Anda sudah belajar bagaimana membagi-bagi sebuah aplikasi menjadi sejumlah *file* dalam bahasa C.
- › Antara desain dengan implementasi, harus *traceable*. Ibarat gambar rumah dengan rumah yang dibangun.
- › Ada berbagai cara implementasi untuk mewujudkan efek yang sama ke pengguna.

Contoh: bola dalam bidang

- › Sebuah bidang mengandung sekumpulan bola. Setiap bola mempunyai arah
- › Bola bergerak sesuai dengan arahnya
- › Jika “bersitabrak”, maka bola akan mati
- › Jika bola habis sama sekali, sistem mati
- › Jika bola tersisa “sedikit”, akan lahir bola-bola baru hingga jumlah tertentu



Berbagai solusi bola dalam bidang

- › Solusi prosedural sekuensial:
 - › Sebuah main program
 - › ADT list of bola (linked list, matriks), yang dibuat, ditambah, dikurangi bolanya
- › Solusi dengan proses konkuren:
 - › Satu main program menumbuhkan banyak proses
 - › setiap bola adalah proses
 - › Semua bola dipetakan ke sebuah bidang (matriks posisi)
- › Jika benda bergerak bukan hanya bola, namun perilaku sama: Solusi dengan proses konkuren dan ADT generik (list of list, atau list of “things”)
- › Sekilas solusi OO:
 - › Bola-bola adalah objek, *instance* dari sebuah kelas Bola.
 - › Jika benda bergerak bukan hanya bola, maka objek-objek adalah instance dari kelas-kelas yang memiliki hubungan inheritance.

Metodologi OO SW Development

- › Memandang persoalan dari sudut “Objek”
 - › Coadd
 - › Rumbaugh
 - › Jacobson
 - › Booch
 - › Martin Odel
 - › UML

Standar:

- Notasi
- Diagram
- Bahasa Spesifikasi

OO Methodology (1)

Coadd Yourdon

- › Object diagram
- › Class diagram
- › Notion of Subject

Rumbaugh

- › Object modeling - Object diagram
- › Dynamic Modeling - State diagram
- › Functional Modeling - DFD

OO Methodology (2)

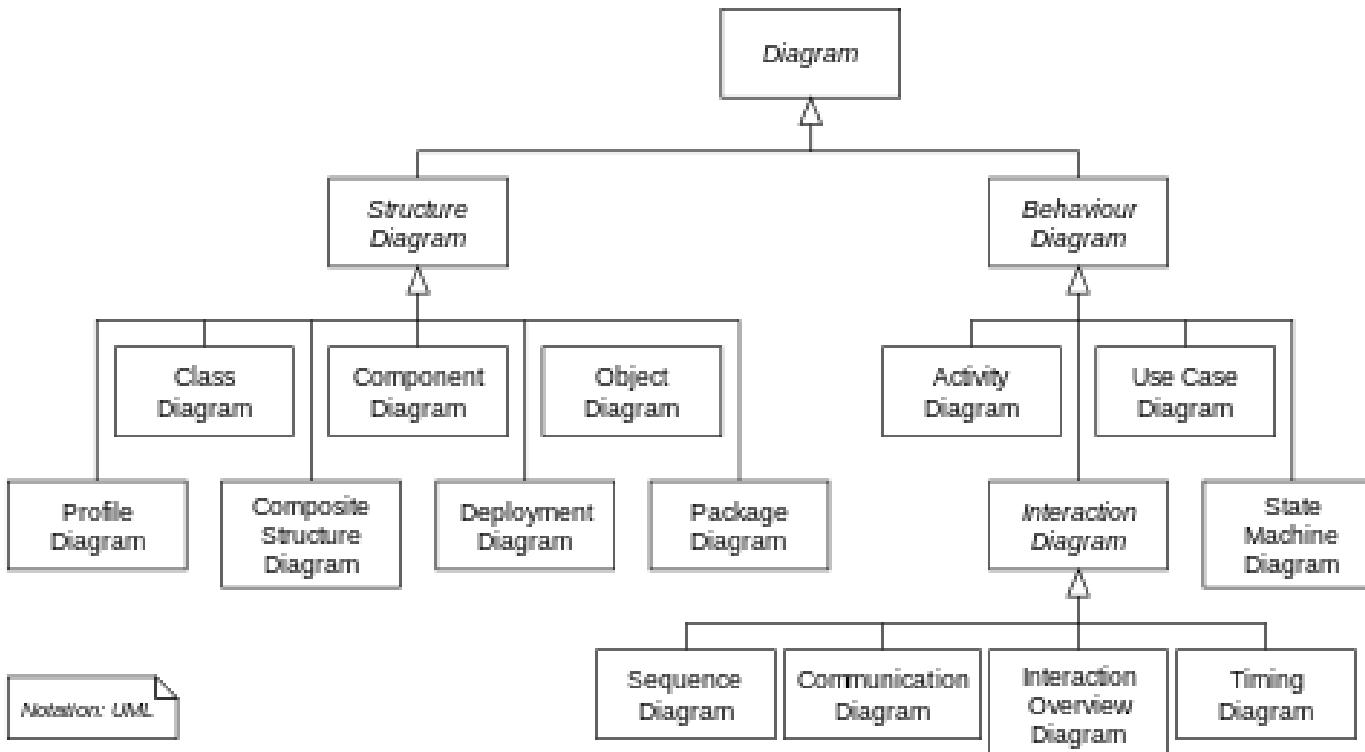
Booch

- › Class Diagram
- › State transition Diagram
- › Object Diagram
- › Timing Diagram
- › Module Diagram
- › Process Diagram

UML Diagram [yang utama]

- › Structural Diagram:
 - › Class, object diagram
 - › Component diagram
 - › Deployment diagram
- › Behavioural Diagram
 - › Use-case
 - › Collaboration diagram
 - › Sequence diagram
 - › State chart diagram
 - › Activity diagram

UML Diagram



- Use Case driven
- Architecture-Centric Process
- Iterrative and Incremental process

OO Language

- › Pure OO: ex. Smalltalk, Eiffel, Java(?)
- › Procedural OO: ex. C++, Ada 95
- › Functional and OO: ex. CLOS, Object LISP, Scala
- › Declarative: prolog extension, ex. Flora-2, Logtalk, Oblog
- › Bacaan :
 - › http://en.wikipedia.org/wiki/Object-oriented_programming
 - › http://en.wikipedia.org/wiki/List_of_object-oriented_programming_languages
 - › http://en.wikipedia.org/wiki/Comparison_of_programming_languages_%28object-oriented_programming%29
 - › http://en.wikipedia.org/wiki/Encapsulation_%28object-oriented_programming%29
 - › http://en.wikipedia.org/wiki/Object-based_language

Apakah ada “ukuran” derajat “OO” sebuah program?

- › Prosedural dan sekuensial, tanpa ada definisi “kelas” misalnya menggunakan bahasa C++ , atau karena dalam bahasa Java kita diharuskan mempunyai kelas, maka hanya berisi sebuah class dengan “void main(...)”
- › Program tidak mengandung definisi “kelas”, tetapi
 - › Menggunakan class library Class (misalnya STL)
 - › It uses available class/“object” (→ object-based programming)
- › Program mengandung definisi “kelas” namun hanya diperlakukan sebagai “ADT”.
- › Program anda dimodelkan dengan OOAD, dan secara konsisten diprogram/dikoding dengan bahasa OO ataupun yang lain (misalnya C yang tidak OO)
- › Domain dimodelkan dengan OOAD, diprogram dengan/tanpa OOL dan direlasikan dengan GUI object (windows, button...) – “event driven programming”

Batasan Kuliah IF2210 OOP

- › Pemodelan perangkat lunak dengan notasi/diagram di atas menjadi bagian dari kuliah Rekayasa Perangkat Lunak
- › Pada kuliah OOP:
 - › Memrogram harus berdasarkan spesifikasi yang jelas, kuliah dimulai dari analisis persoalan secara umum, langsung memodelkan software dalam diagram kelas
 - › Diagram yang banyak dipakai adalah diagram kelas. Diagram-diagram lain yang diperlukan untuk pendukung dapat disinggung/dipakai

OOP dan paradigma terkait

- › Pada contoh bola dan bidang, seringkali gambar visual dapat diplot ke layar dengan menggunakan objek visual yang tersedia:
 - › OOP vs event driven programming
 - › OOP vs Visual programming [VB is not visual programming, but GUI object-based programming]
 - › Object-based programming



Konsep OOP

IF2210 – Semester II 2022/2023

Tim Pengajar IF2210

Tujuan

- › Mahasiswa memahami :
 - › Definisi sebuah perangkat lunak yang dibangun berorientasi Objek
 - › Perbedaan kelas dan objek
 - › Siklus hidup objek: *definition, declaration, penciptaan, manipulation, pemusnahan*
 - › Manipulasi objek:
 - › *Object comparison*
 - › *Assignment, clone dan deep clone*

Definisi OOP

- › [Meyer98]: Sebuah sistem yang dibangun berdasarkan metoda berorientasi objek adalah sebuah sistem yang komponennya di-enkapsulasi menjadi kelompok data dan fungsi, yang dapat mewarisi atribut dan sifat dari komponen lainnya, dan komponen-komponen tersebut saling berinteraksi satu sama lain.

Memrogram Secara OOP

- › Merancang program secara OO “*fundamentally different*” dibandingkan pendekatan structural. [Booch91]
- › Tidak lagi membagi persoalan ke dalam data dan fungsi/prosedur.
- › Melainkan bagaimana membagi ke dalam objek-objek yang memiliki peran & tanggung jawab masing-masing.
- › Berpikir dalam terminologi objek akan berefek terhadap kemudahan mendesain program sebab dunia nyata terdiri atas objek-objek.

Objek & Kelas

Apa itu Objek? (1)

- › [West04]: Object is “*the quanta from which the universe is constructed.*”
- › Objek adalah benda (*thing*) atau sebuah entitas
contoh objek: mobil, buku, *bank account*, gajah, lagu, film, dll.
- › Di dunia komputer, window, mouse, menu, textbox, button
adalah objek.
- › Objek bisa berbentuk objek fisik atau intangible seperti lagu.

Apa itu Objek? (2)

- › Objek terbentuk atas objek-objek lain.
 - › Contoh: objek mobil terdiri atas objek mesin, objek chassis, objek body, dst.
 - › Objek mesin terdiri atas objek blok silinder, objek busi, objek piston, dst.
- › Dalam konsep OO, “semua” adalah objek—termasuk integer, character, dll.
 - › Kebanyakan bahasa pemrograman menganggap integer dst. sebagai tipe data primitif (bukan objek) karena alasan kinerja.

Objek

- › Objek memiliki perilaku tertentu untuk memenuhi suatu tanggung jawab yang disepakati. (“Layanan” yang tersedia.)
 - › Objek harus memiliki akses terhadap informasi yang dibutuhkan untuk menjalankan tanggung jawabnya.
 - › Informasi tersebut bisa dimiliki sendiri, ataupun ditanyakan ke objek lain.
- › Tanggung jawab sebuah objek dapat berupa:
 - › Memberi informasi tertentu bagi objek lain yang meminta.
 - › Melakukan perhitungan (komputasi).
 - › Memberi tahu perubahan state dirinya.
 - › Mengkoordinir objek-objek lain.

Mendefinisikan Objek

- › Pada implementasi di bahasa pemrograman, objek didefinisikan dengan sekelompok **atribut** dan **method**.
 - › **Atribut** adalah informasi yang menjadi bagian dari objek yang dimaksud.
 - › Karena “semua adalah objek” maka atribut pun berupa objek.
 - › Nilai atribut menentukan karakteristik dan state suatu objek.
 - › **Method** adalah aksi atau perilaku suatu objek.
 - › Method dieksekusi untuk memenuhi tanggung jawab suatu objek.

Objek vs. ADT

- › Objek memiliki atribut (\approx data) dan method (\approx prosedur/fungsi). Jadi apa bedanya dari ADT?
 - › Sebuah objek tidak mengekspos isi perutnya ke objek lain.
 - › Objek lain tidak boleh tahu bagaimana sebuah objek mengelola informasi yang dimilikinya secara internal.
 - › Method merefleksikan ekspektasi pada domain persoalan sedangkan fungsi merefleksikan detail implementasi pada program.
 - › Perancangan objek dimulai dari tanggung jawab (method) apa saja yang dimiliki suatu objek, dilanjutkan dengan memutuskan informasi (atribut) apa saja yang diperlukan objek untuk menjalankan tanggung jawab tersebut.

Message

- › **Message** adalah komunikasi formal yang dikirim oleh sebuah objek ke objek lainnya untuk meminta sebuah layanan (meminta objek lain memenuhi tanggung jawabnya).
- › Jenis message:
 - › **Imperative**: menyuruh objek melakukan perubahan state.
 - › **Informational**: hanya “memberitahu”, tidak mengharapkan adanya perubahan state.
 - › **Interrogatory**: meminta informasi.
- › Objek menginvokasi method yang bersesuaian dengan message yang diterima. Misal (C++, Java):
 - › `Stack.push(10); // mengirim message push dengan argumen 10 ke objek Stack.`
 - › Objek Stack menginvokasi method `void push(int item)`

Protocol

- › Protocol (kadang disebut **interface**) adalah spesifikasi message apa saja yang dapat ditangani oleh sebuah objek.
 - › Dalam C++ langsung bersesuaian dengan *header file* (xxx.h).
 - › Pada Java, dapat dilihat pada dokumentasi (Javadocs).

Contoh Objek

- › Sebuah lampu lalu lintas (LLL) adalah sebuah objek.
 - › Tanggung jawab: memberitahukan perubahan state dirinya kepada objek yang “berlangganan”:
 - › Pengemudi mengirim message “subscribe” terhadap layanan LLL ketika mendekati perempatan.
 - › Pengemudi sebagai pengguna layanan dari LLL, tidak perlu tahu bagaimana LLL “menghitung” kapan harus berubah state.
 - › Apakah berdasarkan timer sederhana? → LLL memiliki atribut “Timer”.
 - › LLL cerdas yang menghitung kepadatan dari semua arah supaya bisa menentukan siapa yang harus diberi prioritas? → LLL memiliki atribut sensor atau berkoordinasi dengan objek sensor.
 - › Pengemudi sebagai objek harus patuh terhadap tanggung jawabnya: berhenti jika menerima message “merah”, dst.

Mind Exercise

- › Definisikan apa saja tanggung jawab objek Vending Machine!
- › Tentukan atribut apa saja yang diperlukan objek Vending Machine untuk memenuhi tanggung jawabnya!

Kelas

- › **Kelas** adalah *blueprint* yang mendeskripsikan objek-objek.
 - › Definisi atribut dan method.
- › Misalkan kelas *lampa lalu lintas dengan timer sederhana*, mendefinisikan bahwa setiap objek dari kelas ini memiliki timer di dalamnya.
 - › Setiap *instance* dari kelas ini adalah objek lampu lalu lintas yang memiliki timer, namun konfigurasi timer setiap *instance* dapat berbeda.
- › Karena “semua adalah objek”, secara konseptual Kelas pun adalah sebuah objek yang bertanggung jawab menciptakan objek-objek yang sesuai spesifikasi yang dimilikinya.
 - › Di bahasa pemrograman, umumnya kelas bukan objek.

Analogi

- › *Blueprint* dari sebuah rumah bukanlah rumah.
- › Arsitek membuat *blueprint*. Kontraktor membuat rumah dari *blueprint*.
- › *Programmer* mendefinisikan kelas yang nantinya akan dibentuk objek dari kelas tersebut. Objek adalah instansiasi kelas.
- › *Programmer* dapat bersudut pandang sebagai arsitek atau kontraktor.
 - › Jika sudut pandang bercampur-campur, desain kelas dapat menjadi tidak “bersih”.

Kelas dan Objek

- › Program akan menciptakan objek-objek dari sebuah kelas
 - › Dari sebuah kelas bisa diciptakan banyak objek
 - › Objek dibuat berdasarkan spesifikasi kelas
 - › Setiap objek dimiliki oleh suatu kelas
-
- › Program utama pada OOP “seharusnya” hanya bertugas menciptakan satu objek.
 - › Objek ini akan menciptakan dan mengkoordinir objek-objek lain.

Kelas vs Objek

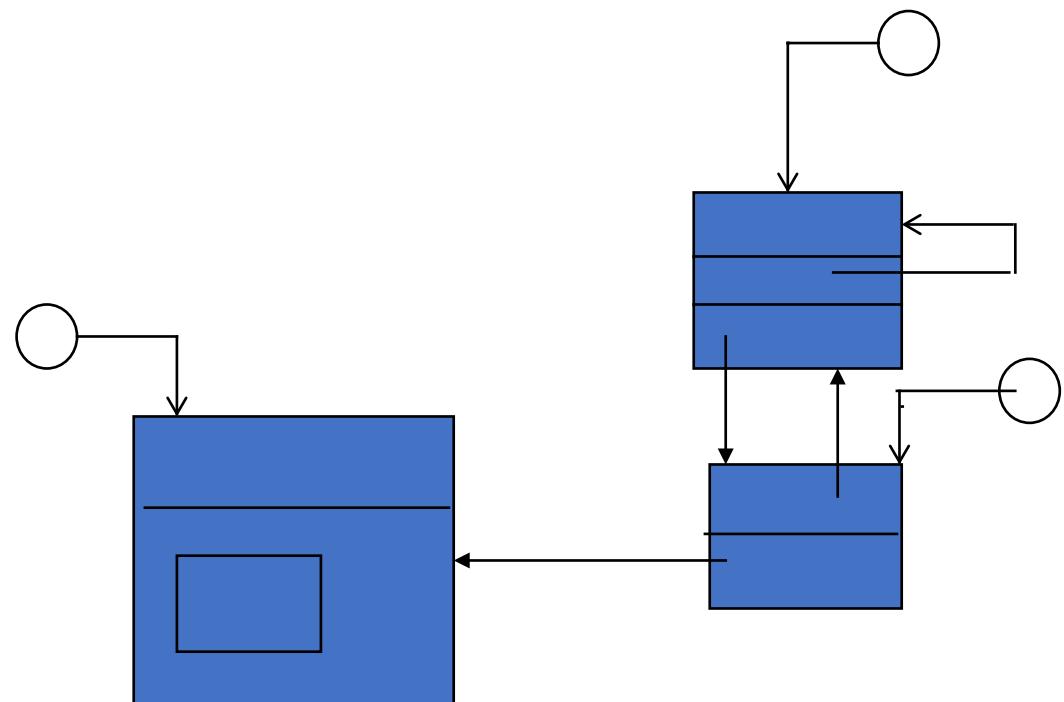
- › Kelas ~ “type”
 - › static definition, defined in source code
- › Objek ~ “variable”
 - › instance of class, exists during run-time (execution)
- › Siklus hidup objek:
 - › creation, manipulation, destruction
- › OOP:
 - › Define class, create objects
 - › Manage objects, their life cycle and states

Penciptaan Objek

- › Objek diciptakan dengan mengirim message kepada kelas untuk menginvokasi suatu method khusus yang disebut **constructor [ctor]**.
 - › Dalam bahasa OO, nama ctor biasanya sama dengan nama kelas.
- › ctor dipakai untuk menciptakan objek dan menginisialisasi atribut-atributnya.
 - › Setiap bahasa memiliki aturan inisialisasi “default” jika pembuat kelas tidak menuliskan ctor.

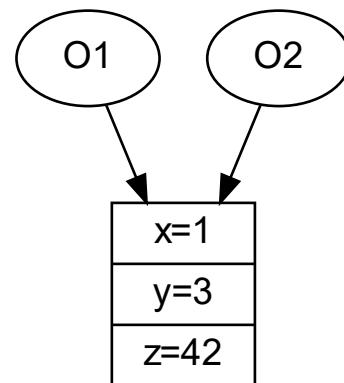
Penugasan Objek

- › Assignment
- › Copy
- › Clone
- › Deep Clone



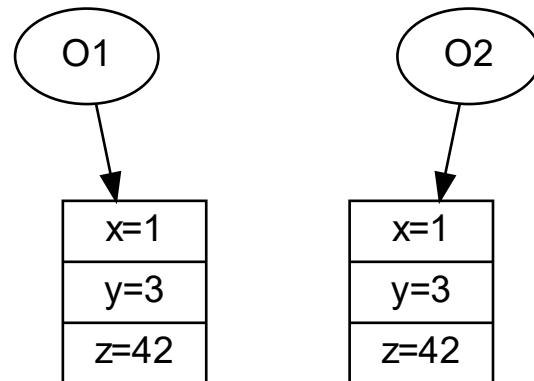
Membandingkan Objek (1)

- › *Reference comparison*: membandingkan apakah O1 dan O2 adalah dua buah reference yang mengacu ke objek yang sama?



Membandingkan Objek (2)

- › *Object comparison*: membandingkan apakah O1 dan O2 adalah dua buah objek yang identik kandungan informasinya?
- › Kasus menjadi rekursif jika atribut objek adalah objek yang mengandung objek lagi (bukan “tipe primitif”).



Penugasan & Pembandingan Objek

- › Pada kebanyakan kasus, kita hanya akan peduli pada *reference comparison*.
 - › Misal: apakah dua *reference* mengacu ke satu orang yang sama.
 - › Umumnya kita tidak peduli apakah dua orang yang berbeda punya nama, tempat, dan tanggal lahir yang sama.
- › Secara semantik, *object comparison* berbicara tentang dua objek yang berbeda tapi mirip → biasanya dapat distrukturkan ulang menjadi hierarki kelas dari pada *object comparison*.
 - › Misal: sesama objek dari kelas X.

Karakteristik OOP

Karakteristik OOP

- › Abstraction
- › Encapsulation
- › Pewarisan (inheritance)
- › Composability
- › →Reuseability
- › Specialization
- › Generalization
- › Communication between objects
- › Polymorphism, dynamic binding

Konsep OOP: Enkapsulasi

- › Menurut kamus: membungkus dengan kapsul
- › Contoh: komputer, membungkus prosesor, memori, *motherboard*, kabel-kabel, dll
 - › Pengguna berinteraksi melalui antarmuka.
- › Contoh lain: TV, DVD player, kamera, mobil, dll
 - › mobil membungkus mesin, chassis, body, dst.

Enkapsulasi

- › Detil teknis disembunyikan dari pengguna → *information hiding*
- › Data dan method, atribut dan fungsionalitas dikombinasikan dalam sebuah unit, sebuah kelas
- › Akses data melalui method atau interface (tidak diakses langsung)
 - › *In fact*, pada sebagian besar kasus, desain yang baik adalah data tidak dapat diakses sama sekali (kembali ke konsep “tanggung jawab” objek)

“Level” of complexity of OOP

- › OOP using ADT
- › OOP with generic class
- › OOP with inheritance
- › OOP with inheritance and polymorphism
- › OOP – concurrent programming
- › OOP – distributed, concurrent and parallel

Konkurensi hanya akan dibahas sebatas fitur bahasa Java. Kuliah fokus sampai dengan inheritance dan polymorphism.



Bahasa C++: Konsep Kelas (bagian I)

IF2210 – Semester II 2022/2023

Sumber: Diktat Bahasa C++ oleh Hans Dulimarta

Latar Belakang C++

- › Diciptakan oleh Bjarne Stroustrup di AT&T Bell Laboratories pada awal 1980an
- › Pada mulanya dikenal sebagai "C with Classes" (nama C++ digunakan sejak 1983, setelah diusulkan oleh Rick Mascitti).
- › 1985: disebarluaskan oleh AT&T, perangkat lunak cfront (C++ translator).
- › Didasarkan pada bahasa C, Simula67, Algol68, Ada.
 - › Simula67: konsep kelas.
 - › Algol68: konsep operator overloading dan kemungkinan penempatan deklarasi di manapun
 - › Ada: konsep template dan exception

Perbandingan C++ dengan C (1)

- › Typecasting dalam C++ dapat dipandang sebagai fungsi.
 - › e.g. `(int) x` vs. `int(x)`
- › *Function name overloading*: fungsi dengan nama yang sama namun dengan *signature* yang berbeda.
- › Nilai default pada parameter formal.
 - › e.g. `int f(int a, int b=2) { ... }` jika dipanggil dengan `f(5)` maka a bernilai 5 dan b bernilai 2.
- › *Template function, operator function, inline function*.
- › *Reference variable*, dan *call by reference* (berbeda dengan *address of*).

Perbandingan C++ dengan C (2)

- › Operator baru seperti *global scope* (unary `:::`), *class scope* (binary `:::`), `new`, `delete`, member pointer selectors (`->*`, `.*`) dan kata kunci baru seperti: `class`, `private`, `operator`, dsb.
- › Nama kelas atau enumerasi (*tag name*) adalah nama tipe (baru)
- › *Anonymous union*

Reference (1)

- › Reference variable: nama alias terhadap variabel tsb.
 - › Jika sudah digunakan untuk mengacu suatu objek/variabel, reference tidak dapat direset untuk mengacu objek/variabel lain
 - › Setiap pendefinisian reference variable harus selalu diinisialisasi dengan variabel lain

```
int x = 5;  
int &xr = x; // xr mengacu pada x  
xr++; // xr merupakan alias dari r
```

Reference (2)

- › Penggunaan *reference* lain: untuk *call-by-reference* dan *return value* dari sebuah fungsi
- › *Reference berbeda* dengan pointer
- › Dalam C++ simbol & digunakan dengan 2 makna: *address-of* dan *reference*

```
int *py;  
int &yr; // error (tidak diinisialisasi)
```

```
int y;  
py = &y; // py akan berisi alamat dari y
```

Kompatibilitas antara C++ dan C

- › Program C yang dikompilasi oleh C++ tidak dapat menggunakan kata kunci dari C++ sebagai nama identifier
- › Setiap fungsi harus dideklarasikan (harus memiliki *prototype*)
- › Fungsi yang bukan bertipe **void**, harus memiliki instruksi **return**
- › Penanganan inisialisasi array karakter:

```
char ch[3] = "C++"; /* C: OK, C++: error */  
char ch[] = "C++"; /* OK untuk C dan C++ */
```

Class

- › Untuk menciptakan tipe data baru.
- › Berisi operasi-operasi dan data yang akan dimiliki objek-objek yang berasal dari kelas tsb.
 - › **Operasi** dalam bentuk **method/function** member biasanya public.
 - › **Data** dalam bentuk **atribut**/data member sebaiknya private. (Secara konsep OOP atribut seharusnya private, namun fitur bahasa C++ mengizinkan atribut public).
- › Peran: perancang kelas dan pengguna kelas.
 - › **Perancang**: menentukan operasi yang disajikan pada pengguna kelas serta representasi internal objek.
 - › **Pengguna**: memanfaatkan operasi tersebut untuk memanipulasi objek.

Class vs Struct (1)

Struct	Class
Memiliki ≥ 1 <i>field</i> , masing-masing berupa data	Memiliki ≥ 1 <i>member</i> , masing-masing berupa data atau fungsi
Setiap field dapat diacu secara bebas dari luar	Pengaksesan member dari luar dapat dikendalikan (kata kunci <i>private</i> , <i>public</i> , dan <i>protected</i>)

Pada class, ada dua jenis member:

- **Function member**, kumpulan operasi (*service/method*) yang dapat diterapkan terhadap objek, seringkali disebut juga sebagai *class interface*
- **Data member**, yang merupakan representasi internal dari kelas (**atribut**)

Class vs Struct (2)

Struct	Class
<pre>typedef struct { int x; int y; } Point; void moveTo (Point&) { // ... }</pre>	<pre>class Point { int x; int y; void moveTo () { // ... } };</pre>

Pada class:

- Pengaturan akses terhadap anggota kelas: `private`, `public`, `protected`
- Perhatikan perubahan parameter aktual pada prosedur `MoveTo`

Class

- › Pengaturan akses terhadap anggota kelas: **private**, **public**, **protected**

Wilayah Deklarasi	Makna
public	Dapat diakses oleh fungsi di luar kelas dengan menggunakan operator selektor (. atau ->) “Fungsi luar”: fungsi yang bukan anggota kelas tersebut
private	hanya dapat diakses oleh fungsi kelas tersebut
protected	seperti private , namun dapat diakses oleh kelas turunan

- › Pendefinisian member function dapat dilakukan dengan dua cara:
 - › Di dalam class body, otomatis menjadi inline function
 - › Di luar class body, nama fungsi harus didahului oleh class scope. Di dalam kelas hanya dituliskan prototipe fungsi

Deklarasi Kelas Stack + definisi method

```
class Stack {  
public:  
    // methods  
    void pop(int& ); // deklarasi (prototype)  
    void push(int); // deklarasi (prototype)  
    /*--- pendefinisian method di dalam class body ---*/  
    int is_empty() {  
        return topStack == 0;  
    }  
private:  
    // attributes  
    int topStack; /* posisi yang akan diisi berikutnya */  
    int *data;  
}; // PERHATIKAN TITIK KOMA !!!
```

Pendefinisian method di luar Class

```
/* pendefinisian Pop dan Push di  
luar class body */  
void Stack::pop(int& item) {  
    if (is_empty()) {  
        // error message  
    } else {  
        topStack--;  
        item = data [topStack];  
    }  
} // TIDAK PERLU TITIK KOMA !!!
```

```
void Stack::push (int item) {  
    if /* penuh */) {  
        // error message  
    } else {  
        data [topStack] = item;  
        topStack++;  
    }  
}
```

Pointer Implisit `this`

- › Setiap *method* memiliki pointer ke objek `this`, yang secara implisit pointer ini dideklarasikan sebagai (untuk kelas X):

`X* this`

- › Pengaksesan anggota kelas (method/atribut) dapat dituliskan dengan menyertakan `this->`

```
void Stack::push (int item) {  
    //...  
    this->data [this->topStack] = item;  
    this->topStack++;  
    //...  
}
```

- › Manfaat: Setiap objek memiliki atribut terpisah, namun method bersama, `this` diperlukan untuk mengakses atribut.

Objek dari Kelas

- › Contoh deklarasi objek:

```
Stack myStack;
Stack oprStack[10];
Stack *pts = new Stack;
Stack ns = myStack; // definition & initialization
```

- › Pengaksesan anggota public:

```
int x;
myStack.push(99);
oprStack[2].pop(x);
pts->push(x);
if (myStack.is_empty()) {
    printf ("Stack masih kosong");
}
```

Penulisan Kode Kelas

- › Kode untuk kelas terdiri dari
 1. Interface / specification: **deklarasi** kelas. Dituliskan ke dalam file **X.h**.
 2. Implementation / body : **definisi** dari method-method dari kelas tersebut. Dituliskan ke dalam file **X.cc**, **X.cpp**, **X.cxx** atau **X.c**.
- › Untuk mencegah penyertaan header lebih dari satu kali, deklarasi kelas dituliskan di antara **#ifdef XXXX_H** dan **#endif**

Contoh Header File

```
// Nama file: Stack.h
// Deskripsi: interface dari kelas Stack
#ifndef STACK_H
#define STACK_H
class Stack {
    //
    // daftar signature method ("protocol") kelas Stack
    //
};

#endif
```

PERHATIAN: Di dalam header file, jangan menuliskan **definisi** objek/variabel karena akan mengakibatkan kesalahan “*multiply defined name*” pada saat linking dilakukan.

Contoh Implementation File

```
// Nama file: Stack.cc
// Deskripsi: implementasi/body dari kelas Stack

#include "Stack.h"

Stack::Stack() {
    // ... ctor
}

Stack::~Stack() {
    // ... dtor
}

// dst...
```

Pemanfaatan Kelas

- › Perancang kelas:
 1. Kompilasi file implementasi (*.cpp) menjadi kode objek (*.o)
 2. Berikan file header (*.h) dan file kode objek (*.o) kepada pemakai kelas (mungkin dalam bentuk *library*)
- › Pemakai kelas:
 1. Sertakan file header di dalam program (main.cc)
 2. Kompilasi program C++ menjadi kode objek (main.o)
 3. Link main.o dengan kode objek yang diberikan perancang kelas

Next Topic

- › 4 sekawan:
 - › ctor, cctor, dtor, operator=

Tugas Baca #2

- › “Konsep tambahan paradigma objek (1)” (3 halaman)
- › Buat summary, 1-2 kalimat untuk setiap *heading*.
- › Kumpulkan di Edunex.



Bahasa C++: Konsep Kelas (bagian II)

IF2210 – Semester II 2022/2023

Sumber: Diktat Bahasa C++ oleh Hans Dulimarta

ctor, dtor, dan cctor

- › *Constructor/destructor* = method khusus yang secara otomatis dipanggil pada saat penciptaan/pemusnahan objek.
 - › Nama konstruktor (ctor) = NamaKelas
 - › Nama destruktur (dtor) = ~NamaKelas
- › Sebuah kelas memiliki ≥ 0 ctor dan ≤ 1 dtor
- › *Copy constructor* (cctor) = konstruktor yang menciptakan objek dengan cara menduplikasi objek lain yang sudah ada
 - › Jika tidak dideklarasikan oleh perancang kelas, cctor akan dilakukan secara *bitwise copy*
- › Untuk menciptakan array dari objek, kelas objek tersebut harus memiliki *default ctor*.

Constructor

- › Tugas utama: menginisialisasi nilai-nilai dari atribut yang dimiliki kelas
- › Dua jenis konstruktor:
 - › ***Default constructor:*** konstruktor yang menginisialisasi objek dengan nilai(-nilai) default yang ditentukan yang ditentukan oleh perancang kelas. ctor ini **tidak** memiliki parameter formal.
 - › ***User-defined constructor:*** konstruktor yang menginisialisasi objek dengan nilai(-nilai) yang diberikan oleh pemakai kelas saat objek diciptakan. ctor ini memiliki satu atau lebih parameter formal.

Penciptaan/Pemusnahan Objek

- › Beberapa jenis objek dalam program C++:
 - › *Automatic object*: diciptakan melalui deklarasi objek di dalam blok eksekusi dan dimusnahkan pada saat blok tersebut selesai eksekusi.
 - › *Static object*: diciptakan satu kali pada saat program dimulai dan dimusnahkan pada saat program selesai.
 - › *Free store object*: diciptakan dengan operator new dan dimusnahkan dengan operator delete. Kedua operator dipanggil secara eksplisit oleh pemakai.
 - › *Member object*: sebagai anggota (atribut) dari kelas.

Contoh Penciptaan/Pemusnahan Objek

```
#include "Stack.h"

Stack s0; /* global (static) */

int reverse() {
    static Stack tStack = ...; /* local static */
    // kode untuk fungsi reverse() di sini
}

main() {
    Stack s1;      // automatic
    Stack s2(20); // automatic
    Stack *ptr;

    ptr = new Stack(50); /* free store object */
    while (...) {
        Stack s3; // automatic

        /* assignment dengan automatic object */
        s3 = Stack(5); // ctor Stack(5) is called
        /* dtor Stack(5) is called */

        // ... instruksi lain ...
    } /* dtor s3 is called, just before next iteration,
       or before iteration stops */

    delete ptr; /* dtor *ptr is called */
}
/* dtor s2 is called */
/* dtor s1 is called */
```

Catatan: kode program di samping hanya digunakan untuk menggambarkan *lifetime* suatu objek. Dalam program OO sebenarnya, tidak ada variabel atau fungsi global—semua di dalam objek.

Copy Constructor

- › Copy constructor (cctor) dipanggil pada saat penciptaan objek secara “duplikasi”, yaitu:
 - › Deklarasi variabel dengan inisialisasi,
 - › e.g. Stack s2 = s1;
 - › Passing parameter aktual ke parameter formal secara “pass by value”
 - › Pemberian *return value* dari fungsi/method yang nilai kembalinya bertipe kelas tersebut (bukan ptr/ref)
- › cctor untuk kelas X dideklarasikan sebagai

X(const X&);

const biar dia ga berubah

biar calling by reference, ga by value

by reference untuk mencegah pengcopyan

```
class Stack {  
public:  
    Stack();           // constructor  
    Stack (int);      // constructor dengan ukuran stack  
    Stack (const Stack&); // copy constructor  
    ~Stack();          // destructor  
    // ...  
};  
  
Stack::Stack (const Stack& s) {  
    size = s.size;  
    topStack = s.topStack;  
    data = new int[size]; // PERHATIKAN: atribut "data"  
                        // harus dialokasi ulang,  
                        // tidak disalin dari "s.data".  
    int i;  
    for (i=0; i<topStack; i++) {  
        data[i] = s.data[i];  
    }  
}
```

```
#include "Stack.h"

void f1(const Stack& _) { /* Instruksi tidak dituliskan */ }

void f2(Stack _) { /* Instruksi tidak dituliskan */ }

Stack f3(int) {
    /* Instruksi tidak dituliskan */
    return ...; // return objek bertipe "Stack"
}

main () {
    Stack s2 (20); // constructor Stack (int)

    /* s3 diciptakan dengan inisialisasi oleh s2 */
    Stack s3 = s2; // BITWISE COPY, jika
                    // tidak ada cctor yang didefinisikan
    f1(s2);      // tidak ada pemanggilan cctor
    f2(s3);      // ada pemanggilan cctor
    s2 = f3(-100); // ada pemanggilan cctor dan assignment
}
```

ctor Initialization List (1)

- › ctor dari atribut akan dipanggil (sesuai urutan deklarasi) sebelum ctor kelas

```
#include "Stack.h"

class Parser {
public:
    Parser(int);
    // ...
private:
    Stack sym_stack, op_stack;
    String s;
};
```

- › Urutan pemanggilan: ctor **Stack** (2x), ctor **String**, lalu ctor **Parser**.
sym_stack dan **op_stack** akan diinisialisasi oleh constructor **Stack::Stack()**

ctor Initialization List (2)

- › Jika ctor yang diinginkan adalah `Stack::Stack(int)`, maka ctor `Parser::Parser()` harus melakukan *constructor initialization list*

```
Parser::Parser(int x): sym_stack(x), op_stack(x) {  
    // ...  
}
```

- › *Initialization list* dapat berisi ≥ 1 inisialisasi, dipisah koma.
- › Setiap inisialisasi mencantumkan nama atribut dengan parameter aktual untuk ctor kelas atribut tsb.

Const Member

- › Keyword `const` dapat diterapkan pada atribut maupun method.
- › Pada atribut: nilai atribut tersebut akan tetap sepanjang waktu hidup objeknya.
 - › Pengisian nilai awal harus dilakukan pada saat objek tersebut diciptakan, yaitu melalui *constructor initialization list*.
- › Pada method: method tersebut tidak bisa mengubah (status) data member yang dimiliki oleh kelasnya.
- › Object juga dapat ditandai sebagai `const`
 - › hanya boleh memanggil const method, untuk memastikan bahwa status object tidak berubah.

Anggota Statik

- › Anggota statik adalah anggota yang “dimiliki” oleh kelas, bukan oleh objek dari kelas tersebut.
- › Dalam konsep OOP, anggota statik kira-kira adalah atribut & method yang dimiliki oleh objek “kelas”.
 - › (Ingat bahwa secara konseptual, kelas pun adalah sebuah objek.)
- › Anggota statik juga dapat berupa atribut maupun method.

atribut Statik

```
class Stack {  
public:  
    // ... method lain  
private:  
    static int n_stack; // static attribute!!  
    // ... atribut & method lain  
};
```

- › Setiap objek dari kelas memiliki sendiri salinan atribut non-statik
- › atribut statik **dipakai bersama** oleh seluruh objek dari kelas tersebut

Inisialisasi Anggota Statik

- › Keberadaan anggota statik (method maupun atribut) **tidak** bergantung pada keberadaan objek dari kelas.
- › Inisialisasi **harus** dilakukan **di luar** deklarasi kelas dan **di luar** method. Apa yang terjadi jika diinisialisasi di dalam ctor?
- › Dilakukan di dalam file implementasi (X.cc), **bukan** di dalam header file.

```
// inisialisasi atribut statik (file Stack.cc)
int Stack::n_stack = 0;

Stack::Stack() {
    // ... dst
}
```

Method Statik

- Method yang hanya mengakses anggota statik dapat dideklarasikan static di dalam file: Stack.h

```
class Stack {  
    // ...  
public:  
    static int numStackObj();  
private:  
    static int n_stack;  
};
```

- Di dalam file: Stack.cc

```
int Stack::numStackObj() {  
    // kode mengakses hanya atribut statik  
    return n_stack;  
}
```

Sifat Anggota Statik

- › method statik dapat dipanggil tanpa melalui objek dari kelas tersebut, misalnya:

```
if (Stack::numStackObj() > 0) {  
    printf("...");  
}
```

- › method statik tidak memiliki pointer implisit `this`
- › Atribut statik diinisialisasi tanpa perlu adanya objek dari kelas tersebut.

Friend (1)

- › Friend = pemberian hak pada fungsi/kelas untuk mengakses anggota *non-public* suatu kelas

```
class B { // kelas "pemberi izin"
    friend class A;
    friend void f (int, char *);
}
```

- › Friend bersifat satu arah
- › Seluruh member kelas A dan fungsi f dapat mengakses anggota *non-public* dari kelas B

Friend (2)

- › Kriteria penggunaan *friend*:
 - › Hindari penggunaan *friend* kecuali untuk fungsi operator
 - › Fungsi friend merupakan fungsi di luar kelas sehingga objek parameter aktual mungkin dilewatkan secara *call-by-value*
 - › Akibatnya operasi yang dilakukan terhadap objek bukanlah objek semula

Nested Class (1)

- › Dalam keadaan tertentu, perancang kelas membutuhkan pendeklarasian kelas di dalam deklarasi suatu kelas tertentu
- › Operasi dalam kelas List didefinisikan di kelas List dan mungkin membutuhkan akses kelas ListElem → kelas List dideklarasikan sebagai friend dari kelas ListElem

```
class List;

class ListElem {
    friend class List;
public:
    // ...
private:
    // ...
};

class List {
public:
    // ...
private:
    // ...
};
```

Nested Class (2)

- › Pemakai kelas `List` tidak perlu mengetahui keberadaan kelas `ListElem`
 - › Yang perlu diketahui: adanya layanan untuk menyimpan nilai (integer) ke dalam list maupun untuk mengambil nilai dari list
- › Kelas `ListElem` dapat dijadikan sebagai *nested class* di dalam kelas `List`

```
class List {  
    //  
    //  
    class ListElem {  
        //  
        //  
    };  
};
```

Nested Class (3)

- › Di manakah nested class didefinisikan? Di bagian public atau non-public
 - › Di bagian public: akan tampak di luar kelas sebagai anggota yang public
 - › Di bagian non-public: akan tersembunyi dari pihak luar kelas, tetapi terlihat oleh anggota kelas → efek yang diharapkan

```
class List {  
public:  
    // bagian public kelas List  
private:  
    class ListElem {  
        public:  
            // semua anggota ListElem berada pada bagian publik  
    }  
  
    // definisi anggota private kelas List  
};
```

Method pada Nested Class

- › Contoh `List` dan `ListElem`
 - › Nama scope yang digunakan untuk kelas `ListElem` adalah `List::ListElem::`, bukan hanya `ListElem::`
 - › Jika merupakan kelas generic dengan parameter generic Type, nama scope menjadi `List<Type>::ListElem`

Tugas Baca #3

- › “Menulis program berorientasi objek” (8 halaman)
- › Buat summary, 1-2 kalimat untuk setiap *heading*.
- › Kumpulkan di Edunex



Bahasa C++: Contoh Program Kecil

IF2210 – Semester II 2022/2023

RSP, IL

Hello World

Bahasa C

```
/* File : helloworld.c */  
/* Author : NIM - Nama */  
#include "stdio.h"  
  
int main(){  
    printf("Hello World\n");  
    return 0;  
}
```

Bahasa C++ [versi prosedural]

```
/* File : helloworld.cpp */  
/* Author : NIM - Nama */  
#include <iostream>  
using namespace std;  
int main(){  
    cout << "Hello World" << endl;  
    return 0;  
}
```

Bahasa C++ [versi 00]

```
/* File : hello.h */
/* Author : NIM – Nama */
class hello {
    public :
        hello();
};
```

```
/* File : hello.cpp */
/* Author : NIM - Nama */
#include "hello.h"
#include <iostream>
using namespace std;
hello::hello(){}
cout << "Hello World"
     << endl;
}
```

```
/* File:hellotest.cpp */
/* Author: NIM -Nama */
#include "hello.h"
#include <iostream>
using namespace std;
```

```
int main(){
    hello h; //ctor dihidupkan
    return 0;
}
```



Passing Parameter

```
// Z.h
#ifndef _Z_H
#define _Z_H
class Z {
public:
    void print();      // print nilai val
    void print(int i); // print nilai i
    void set(int x);  // set val dengan x
    int add(int x, int y); // mengirimkan x+y
    void add(int x);   // menambah val dengan x
private:
    int val;
};
#endif
```

```
// Z.cpp
#include "Z.h"
#include <iostream>
using namespace std;
void Z::print() { cout << "val=" << this->val << endl; }
void Z::print(int i) { cout << "i=" << i << endl; }
void Z::set(int x) { this->val = x; }
int Z::add(int x, int y) { return x+y; }
void Z::add(int x) { this->val = this->val + x; }
```

Perhatikan cara invokasi & passing parameter

```
#include <iostream>
#include "Z.h"
using namespace std;
int main() {
    Z z;
    z.set(2);
    z.print();
    cout << z.add(4,5) << endl;
    z.add(3);
    z.print();
    return 0;
}
```

Perhatikanlah bahwa :

1. Tidak ada ctor, tapi program dapat berjalan dengan baik
2. Prosedur bernama `print` dan fungsi bernama `add` ada dua, namun parameternya berbeda
3. Fungsi `int add(int x, int y)` tidak memakai nilai `val`.

ctor,cctor, dtor

```

// X.h
#ifndef _X_H
#define _X_H
class X {
public:
    X();          // ctor
    X(int);       // ctor dengan parameter
    X(const X&); // cctor
    ~X();         // dtor
    void print(); // prosedur untuk print atribut
private:
    int x;        // atribut kelas
};
#endif

```

```

// X.cpp
#include "X.h"
#include <iostream>
using namespace std;
// Perhatikan bahwa umumnya tidak ada cout pada ctor, cctor, dtor.
// Pada contoh ini hanya dituliskan untuk menunjukkan kapan method dipanggil.
X::X() { x=0; cout << "ctor()" << endl; }
X::X(int a) { x=a; cout << "ctor(int)" << endl; }
X::X(const X& ox) { x=ox.x; cout << "cctor(X)" << endl; }
X::~X() { cout << "dtor()" << endl; } // pada kasus ini dtor does nothing
void X::print() { cout << "Nilai x=" << X::x << endl; }

```

```
// Y.h -- contoh kelas tanpa ctor, cctor, dtor
#ifndef _Y_H
#define _Y_H
class Y {
public:
    void print();
private:
    int y;
};
#endif
```

```
// Y.cpp
#include "Y.h"
#include <iostream>
using namespace std;
void Y::print() { cout << "Nilai y=" << Y::y << endl; }
```

```
// mX.cpp
#include <iostream>
#include "X.h"
#include "Y.h"
using namespace std;
int main() {
    X x;
    X x1=x; // karena ada cctor, maka bukan BITWISE COPY melainkan memanggil
              // cctor
    X* ptrx; // ptrx adalah pointer, harus di-new
    X* ptr1 = new X();
    x.print();
    x1.print();
    ptr1->print();

    Y oy; oy.print();
    Y y1 = oy; y1.print();
    Y* ptry = new Y(); ptry->print();

    return 0;
}
```

const

Tujuan topik ini

1. Mahasiswa memahami tentang sifat “immutability” dari data yang disimpan
2. Secara khusus, mengenai berbagai arti keyword `const` dalam bahasa C++ (melalui program kecil dalam dokumen ini dan dalam bahasa lain (lihat bacaan yang diberikan)

```
const int myconstant = 10;
    // An int which can't change value. similar to #define in C but better.
const int * myconstant;
    // Variable pointer to a constant integer.
int const * myconstant;
    // Same as above.
int * const myconstant;
    // Constant pointer to a variable integer.
int const * const myconstant;
    // Constant pointer to a constant integer.
void mymethod(QString const &myparameter);
/* myparameter will not be altered by the method. & means it can be
   altered but here we just want it to be used because it saves taking
   a copy */
```

How to initialize const member in C++?

// Contoh berikut salah, sebab Val adalah konstanta.

```
class Something {  
private:  
    const int val;  
public:  
    something() { val = 5; }  
}
```

Lantas bagaimana? ctor initialization list

```
class Foo {  
private:  
    const int data;  
public:  
    Foo(int x): data(x) {...} // data diinisialisasi dengan x  
};
```

```
class Bar: Foo {  
public:  
    Bar(int x): Foo(x) {...} // ctor Bar(int) diawali dengan  
                            // memanggil ctor Foo(int)  
};
```

```
class Baz {  
public:  
    Foo* const foo;  
    Baz(Foo* f): foo(f) {...} // foo diinisialisasi dengan f  
};
```



Bahasa C++: Operator Overloading

IF2210 Pemrograman Berorientasi Objek

Sumber: Diktat Bahasa C++ oleh Hans Dulimarta

Operator Overloading

- › *Function overloading* adalah fasilitas yang memungkinkan nama fungsi yang sama dapat dipanggil dengan jenis dan jumlah parameter (*function signature*) yang berbeda-beda

```
int a, b, c;  
float x, y, z;  
c = a + b; /* "fungsi +" di-overload */  
z = x + y;
```

- › Dengan fungsi tambah():

```
c = tambah(a, b);  
z = tambah(x, y);
```

- › Prototipe dua fungsi tambah:

```
int tambah(int, int);  
float tambah(float, float);
```

Fungsi Operator

- › Fungsi operator = fungsi dengan nama “operator@” dengan @ diganti simbol operator yang ada
- › Dapat dimanfaatkan dalam manipulasi objek dengan menggunakan simbol

```
Matrix A, B, C;
```

```
C = A * B; /* perkalian matriks */
```

```
Complex x, y, z;
```

```
x = y / z; /* pembagian bilangan kompleks */
```

```
Process p;
```

```
p << SIGHUP; /* pengiriman sinyal dalam Unix */
```

- › Harus didefinisikan oleh perancang kelas sebagai fungsi public

Pendeklarasian Fungsi Operator

- › Ada 2 cara/bentuk:
 - › Sebagai fungsi non-anggota
 - › Jika mengakses anggota yang non-public maka fungsi tersebut harus dideklarasikan sebagai friend
 - › Jumlah parameter formal = jumlah operan
 - › Sebagai fungsi anggota
 - › Dideklarasikan di wilayah public
 - › Parameter pertama dari operasi harus bertipe kelas tersebut
 - › Jumlah parameter formal = jumlah operan - 1

Contoh Deklarasi Fungsi Operator

```
class Matrix {  
public:  
    // fungsi-fungsi operator  
    friend Matrix operator* (const Matrix&, const Matrix&);  
    // ...  
};  
  
class Complex {  
// ...  
public:  
    Complex operator/ (const Complex&);  
    // ...  
};  
  
class Process {  
// ...  
public:  
    void operator<< (int);  
    // ...  
};
```

Implementasi sebagai Anggota

- › Deklarasi operator<< dan operator>> pada Stack

```
class Stack {  
    // ...  
public:  
    void operator<< (int); // untuk push  
    void operator>> (int&); // untuk pop  
    // ...  
};
```

- › Definisi fungsi operator:

```
void Stack::operator<< (int item) {  
    push(item);  
}  
  
void Stack::operator>> (int& item) {  
    pop(item);  
}
```

Pemanggilan Fungsi Operator Anggota

- › Perbandingan nama “operator@” dengan “nama-biasa”

```
void operator<< (int)    <-> void push(int)
void operator>> (int&)   <-> void pull(int&)
```

- › Sebagai fungsi anggota, fungsi operator dapat dipanggil dengan 2 cara: (1) dengan kata kunci operator; (2) hanya simbol

```
Stack s, t;
s.push(100);
t.operator<<(500);
t << 500;
```

Implementasi sebagai Non-Anggota

```
class Stack {  
    friend void operator<< (Stack&, int);  
    friend void operator>> (Stack&, int&);  
};  
                                bukan const krn dia mengubah stack  
void operator<< (Stack& s, int v) {  
    s.push(v);  
}  
  
void operator>> (Stack& s, int& v) {  
    s.pop(v);  
}
```

Fungsi Operator Non-Anggota

- › Dapat dimanfaatkan hanya dengan simbol << atau >>, tidak sebagai fungsi dengan nama operator<< atau operator>>

```
Stack s;  
s.operator<<(500); // ERROR  
s << 500; // OK
```

- › Error terdeteksi pada saat kompilasi karena kelas Stack tidak memiliki operator<<

Fungsi Anggota atau Non-Anggota?

- › Implementasi sebagai anggota: “ruas kiri” operator harus berupa objek kelas tersebut
- › Jika “ruas kiri” operator bukan bertipe kelas tersebut, operator harus diimplementasikan sebagai non-anggota
- › Operator biner yang ingin dibuat komutatif yang salah satu operannya bukan berasal dari kelas tersebut harus diimplementasikan sebagai dua fungsi dengan nama sama (*overloading*)

```
class Stack {  
    void operator+ (int); // sebagai anggota  
    friend void operator+ (int, Stack&); // sebagai friend  
};  
  
void Stack::operator+ (int m) {  
    push(m);  
}  
  
void operator+ (int m, Stack& s) {  
    s.push(m);  
}
```

Anggota atau Non-Anggota?

- › Operator (biner) yang ruas kirinya bertipe kelas tersebut dapat diimplementasikan sebagai fungsi non-anggota maupun fungsi anggota
- › Operator (biner) yang ruas kirinya bertipe lain **harus** diimplementasikan sebagai fungsi non-anggota
- › Operator *assignment* =, *subscript* [], pemanggilan (), dan selektor - >, harus diimplementasikan sebagai fungsi anggota
- › Operator yang (dalam tipe standar) memerlukan operan lvalue seperti *assignment* dan *arithmetic assignment* (=, +=, ++, *=, dst) sebaiknya diimplementasikan sebagai fungsi anggota
- › Operator yang (dalam tipe standar) tidak memerlukan operan lvalue (+, -, &&, dst.) sebaiknya diimplementasikan sebagai fungsi non-anggota

Manfaat Fungsi Operator

- › Operasi aritmatika terhadap objek-objek matematika lebih terlihat alami dan mudah dipahami oleh pembaca program

```
c = a*b + c/d + e; // lebih mudah dimengerti  
c = tambah(tambah(kali(a,b), bagi(c,d)), e);
```

- › Dapat menciptakan operasi input/output yang seragam dengan memanfaatkan *stream I/O* dari C++
- › Pengalokasian dinamik dapat dikendalikan perancang kelas melalui operator `new` dan `delete`.
- › Batas indeks pengaksesan array dapat dikendalikan lewat operator `[]`.

Perancangan Operator Overloading

- › Assignment = dan address-of & secara otomatis didefinisikan untuk setiap kelas
- › Pilihlah simbol operator yang memiliki makna paling mendekati aslinya
- › Overloading tidak dapat dilakukan pada :: (scope), .* (member pointer selector), . (member selector), ?: (arithmetic if), dan **sizeof()**
- › Urutan presendensi operator tidak dapat diubah
- › Sintaks (*arity*: banyaknya operan) dari operator tidak dapat diubah
- › Operator baru tidak dapat diciptakan
- › ~~Operator ++ dan -- tidak dapat dibedakan antara postfix dan prefix~~
 - › T& T::operator++(); // prefix
 - › T T::operator++(...); // postfix
- › Fungsi operator harus merupakan member atau paling sedikit salah satu argument berasal dari kelas yang dioperasikan

Operator =

- › Assignment \neq copy constructor
 - › Pada assignment $a = b$, objek a dan b sudah tercipta sebelumnya
 - › Pada copy constructor $Stack s = t$, hanya t yang sudah tercipta, objek s sedang dalam proses penciptaan

```
Stack& Stack::operator= (const Stack& s) {  
    /* assign stack "s" ke stack "*this" */  
    int i;  
    delete[] data; // bebaskan memori yang digunakan sebelumnya  
    size = s.size;  
    data = new int[size]; // alokasikan ulang  
    topStack = s.topStack;  
  
    for (i=0; i<topStack; i++)  
        data[i] = s.data[i];  
  
    return *this;  
}
```

Return value operator=()

- › Perintah *assignment* berantai berikut:

```
int a, b, c;  
a = b = c = 4; // aksi eksekusi: a = (b = (c = 4));
```

- › *Assignment* berantai dapat diterapkan pada objek dari kelas jika nilai kembali operator= adalah reference
 - › Fungsi anggota operator= harus mengembalikan nilai kembali berupa objek yang sudah mengalami operasi *assignment* (perintah return *this)

Anggota kelas minimal

- › Perancang kelas “wajib” mendefinisikan empat fungsi berikut:
 - › Constructor
 - › Copy constructor
 - › Operator assignment
 - › Destructor

```
class XC {  
public:  
    XC(...);                      // ctor  
    XC(const XC&);                // cctor  
    XC& operator= (const XC&);    // assignment  
    ~XC();                         // dtor  
    //...member public yang lain  
private:  
    //...  
};
```

Operator []

- › Dapat digunakan untuk melakukan subscripting terhadap objek.
- › Jika digunakan sebagai subscripting, batas index dapat diperiksa.
- › Parameter kedua (index/subscript) dapat berasal dari tipe data apa pun: integer, float, character, string, maupun tipe/kelas yang didefinisikan user.
- › Contoh kasus: membuat Map/Dictionary dengan key berupa string sehingga elemen-elemen Map `m` dapat diakses dengan, e.g., `m["apple"]`.



Bahasa C++: Contoh Operator Overloading

IF2210 – Semester II 2022/2023

Tim Pengajar IF2210

Copy assignment

The assignment operator (`operator=`) has special properties: see [copy assignment](#) and [move assignment](#) for details.

The canonical copy-assignment operator is expected to [perform no action on self-assignment](#), and to return the lhs by reference:

```
// assume the object holds reusable storage,
// such as a heap-allocated buffer mArray
T& operator=(const T& other) { // copy assignment
    supaya tdk dicopy saat dipanggil
    if (this != &other) { // self-assignment check expected
        if (other.size != size) { // storage cannot be reused
            delete[] mArray; // destroy storage in this
            size = 0;
            mArray = nullptr; // preserve invariants in case next line throws
            mArray = new int[other.size]; // create storage in this
            size = other.size;
        }
        std::copy(other.mArray, other.mArray + other.size, mArray);
    }
    return *this;
}
```

Move assignment

The canonical move assignment is expected to leave the moved-from object in valid state (that is, a state with class *invariants intact*), and either do nothing or at least leave the object in a valid state on self-assignment, and return the lhs by reference to non-const, and be noexcept:

```
T& operator=(T&& other) noexcept { // move assignment

    if(this != &other) { // no-op on self-move-assignment
        // (delete[]/size=0 also ok)

        delete[] mArray; // delete this storage
        mArray = std::exchange(other.mArray, nullptr);
        // leave moved-from in valid state
        size = std::exchange(other.size, 0);
    }
    return *this;
}
```

Copy-and-swap assignment

In those situations where copy assignment cannot benefit from resource reuse (it does not manage a heap-allocated array and does not have a (possibly transitive) member that does, such as a member `std::vector` or `std::string`), there is a popular convenient shorthand: the copy-and-swap assignment operator, which takes its parameter by value (thus working as both copy- and move-assignment depending on the value category of the argument), swaps with the parameter, and lets the destructor clean it up.

```
T& T::operator=(T arg) noexcept {
    // copy/move constructor is called to construct arg

    swap(arg); // resources are exchanged
                // between *this and arg
    return *this;
} // destructor of arg is called to release the resources
  // formerly held by *this
```

This form automatically provides strong exception guarantee but prohibits resource reuse.

Contoh-contoh dari TutorialsPoint

https://www.tutorialspoint.com/cplusplus/binary_operators_overloading.htm

Perkalian pecahan

```
#include <iostream>

class Fraction {
    int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
    int n, d;
public:
    Fraction(int n, int d = 1): n(n/gcd(n, d)), d(d/gcd(n, d)) {}
    int num() const { return n; }
    int den() const { return d; }
    Fraction& operator*=(const Fraction& rhs) {
        int new_n = n * rhs.n/gcd(n * rhs.n, d * rhs.d);
        d = d * rhs.d/gcd(n * rhs.n, d * rhs.d);
        n = new_n;
        return *this;
    }
};
```

```

std::ostream& operator<<(std::ostream& out, const Fraction& f) {
    return out << f.num() << '/' << f.den();
}

bool operator==(const Fraction& lhs, const Fraction& rhs) {
    return lhs.num() == rhs.num() && lhs.den() == rhs.den();
}

bool operator!=(const Fraction& lhs, const Fraction& rhs) {
    return !(lhs == rhs);
}

Fraction operator*(Fraction lhs, const Fraction& rhs) {
    return lhs *= rhs;
}

```

```

int main() {
    Fraction f1(3, 8), f2(1, 2), f3(10, 2);
    std::cout << f1 << " * " << f2 << " = " << f1 * f2 << '\n'
                << f2 << " * " << f3 << " = " << f2 * f3 << '\n'
                << 2 << " * " << f1 << " = " << 2 * f1 << '\n';
}

```

Contoh lain

```
#include <iostream>
using namespace std;
class A {
public:
    A();
    A(int nn);
    A(const A& a);
    ~A();
    A& operator=(const A& a);
    A operator+(const A& a);
    friend A operator-(const A& a1, const A& a2);
    friend ostream& operator<<(ostream& os, const A& a);
private:
    int n;
};
```

```
A::A() { // ctor
    cout << "A::ctor 0" << endl;
    n = 0;
}
```

```
A::A(int nn) { //ctor dengan param
    cout << "A::ctor 1" << endl;
    n = nn;
}
```

```
A::A(const A& a) { //cctor
    cout << "A::cctor" << endl;
    n = a.n;
}
```

```
A::~A() { //dtor
    cout << "A::dtor" << endl;
}
```

```
A& A::operator=(const A& a) {  
    cout << "A::opr =" << endl;  
    n = a.n;  
    return *this;  
}
```

```
A A::operator+(const A& a) { //operator+ sebagai anggota kelas  
    cout << "A::opr +" << endl;  
    A t;  
    t.n = n + a.n;  
    return t;  
}
```

```
A operator-(const A& a1, const A& a2) { //operator- bukan anggota kelas  
    cout << "A::opr -" << endl;  
    A t;  
    t.n = a1.n - a2.n;  
    return t;  
}
```

```
ostream& operator<<(ostream& os, const A& a) {  
    os << "n:" << a.n;  
    return os;  
}
```



Binary operator overloading example

```
#include <iostream>
using namespace std;
class Box {
public:
    Box(double len, double bre, double hei): length(len),
                                              breadth(bre),
                                              height(hei) {}
    double volume() { return length * breadth * height; }
    // Overload + operator to add two Box objects.
    Box operator+(const Box& b) {
        Box box(this->length + b.length,
                this->breadth + b.breadth,
                this->height + b.height);
        return box;
    }
private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};
```

```

// Main function for the program
int main()
{
    double volume = 0.0; // Store the volume of a box here

    // box 1 specification
    Box Box1(6.0,7.0,5.0);

    // box 2 specification
    Box Box2(12.0,13.0,10.0);

    // volume of box 1
    volume = Box1.volume();
    cout << "Volume of Box1 : " << volume << endl;

    // volume of box 2
    volume = Box2.volume();
    cout << "Volume of Box2 : " << volume << endl;

    // Add two object as follows:
    Box Box3 = Box1 + Box2;

    // volume of box 3
    volume = Box3.volume();
    cout << "Volume of Box3 : " << volume << endl;

    return 0;
}

```

Output :

Volume of Box1 : 210
 Volume of Box2 : 1560
 Volume of Box3 : 5400

```
#include <iostream>
using namespace std;
class Distance {
private:
    int feet; // 0 to infinite
    int inches; // 0 to 12
public:
    // required constructors
    Distance(int f, int i): feet(f), inches(i) {}
    Distance(): Distance(0,0) {}
    // method to display distance
    void displayDistance() {
        cout << feet << " feet " << inches << " inches" << endl;
    }

    // overloaded minus (-) operator
    Distance operator- () {
        feet = -feet;
        inches = -inches;
        return *this;
    }
};
```

```
int main() {
    Distance D1(11, 10), D2(-5, 11);

    -D1;           // apply negation
    D1.displayDistance(); // display D1

    -D2;           // apply negation
    D2.displayDistance(); // display D2

    return 0;
}
```

Output :

-11 feet -10 inches
5 feet -11 inches



Bahasa C++: Latihan 4 Sekawan

IF2210 – Semester II 2022/2023

Tim Pengajar IF2210

Soal 1.

Sebuah sistem penyewaan kendaraan, terdapat beberapa kendaraan yang diidentifikasi secara unik melalui nomor kendaraan (integer). Setiap kendaraan tergolong dalam suatu kategori tertentu, yang terdiri atas Bus, Minibus, dan Mobil. Selain itu tiap kendaraan memiliki informasi: merk (string), dan tahun keluaran (integer).

1. Buatlah semua operator 4 sekawan untuk kelas Kendaraan
 - › Buat 2 buah ctor: default dan user-defined
 - › Ctor default: nomor = 0; tahun keluaran = 0; merk = “XXX”; kategori = “mobil”

2. Kendaraan dapat menerima *message* berikut:
- printInfo**, mencetak semua informasi yang dimiliki oleh Kendaraan (format pencetakan bebas).
 - biayaSewa(int lamaSewa)**, menghitung dan menghasilkan biaya peminjaman dengan kebijakan yang berbeda tergantung jenis Kendaraan-nya sbb.:

No	Jenis Kendaraan	Biaya Sewa (dalam Rupiah)
1	Bus	1 juta * lamaSewa
2	Minibus	Jika lamaSewa kurang dari atau sama dengan 5 hari, maka biaya sewanya adalah 5 juta, tetapi jika lamaSewa lebih dari 5 hari, maka biaya total adalah $5 \text{ juta} + 500 \text{ ribu} * (\text{lamaSewa} - 5)$
3	Mobil	500 ribu * lamaSewa

lamaSewa adalah durasi/lama suatu Kendaraan disewa (dalam hari)

Soal 2.

Masih berkaitan dengan Soal 1, buatlah sebuah kelas baru bernama **KoleksiKendaraan** yang merepresentasikan semua Kendaraan yang dimiliki oleh sebuah perusahaan rental.

Hint: gunakan array of Kendaraan sebagai memori internal untuk menyimpan kumpulan Kendaraan. Ukuran array tergantung ctor yang dipanggil.

1. Buat semua operator 4 sekawan untuk kelas KoleksiKendaraan
 - › Ctor default: size berukuran 100, Neff = 0.
 - › Ctor user defined: parameter masukan: size. Neff = 0.

2. KoleksiKendaraan dapat menerima *message* berikut:
- a. **printAll**, mencetak data semua kendaraan dalam suatu KoleksiKendaraan; dilakukan dengan mengirimkan *message* PrintInfo kepada setiap Kendaraan yang disimpan dalam KoleksiKendaraan.
 - b. **operator<<(Kendaraan)**, menambahkan sebuah Kendaraan ke dalam array of Kendaraan sebagai elemen yang terakhir. Diasumsikan selalu ada tempat di memory array.
 - c. **operator<<(KoleksiKendaraan)**, menambahkan semua Kendaraan dari KoleksiKendaraan lain ke dalam array Kendaraan. Jika banyaknya Kendaraan yang akan ditambahkan melebihi kapasitas array, maka yang ditampung hanya sejumlah kendaraan yang dapat dimuat.



Konsep Inheritance

IF2210 – Semester II 2021/2022

Inheritance dalam OOP

- › Kemampuan kelas untuk menurunkan atribut dan method dari kelas lain disebut dengan *inheritance*.
- › *Inheritance* adalah salah satu konsep dalam OOP yang membedakan dari memrogram ADT.
- › Istilah:
 - › **Subclass/derived class:** kelas “anak”, yang menurunkan atribut & method dari kelas lain.
 - › **Superclass/base class:** kelas “induk”, yang atribut & method-nya diturunkan ke kelas lain.

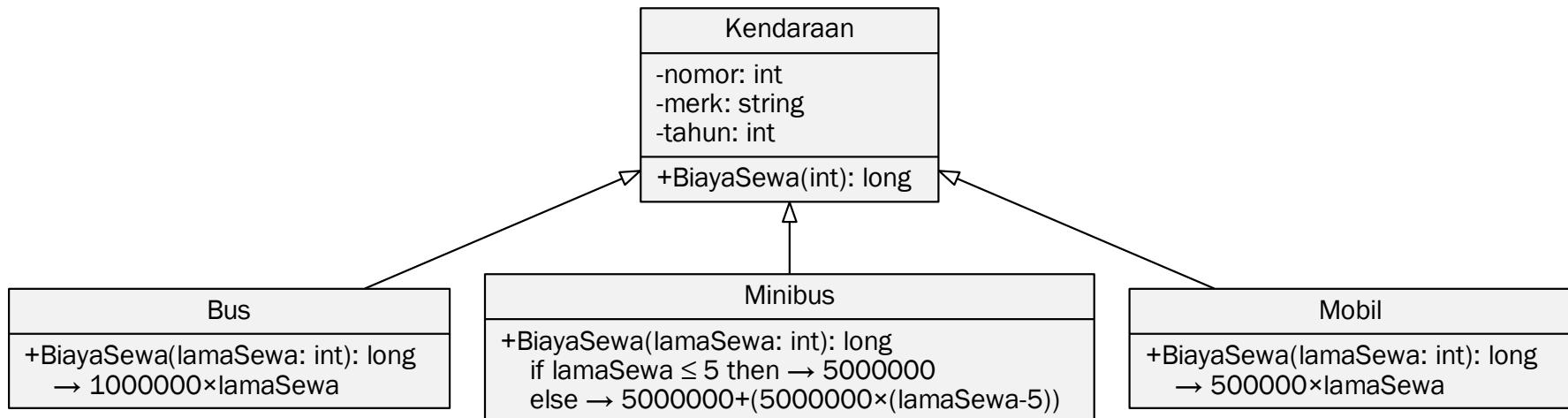


Mengapa inheritance?

- › Tinjau contoh kelas Kendaraan pada latihan soal sebelumnya.
- › Pada implementasi method BiayaSewa(int) terdapat kondisional:
 - › Jika kategori=“bus” maka biaya sewa = $1000000 \times$ lama sewa
 - › Jika kategori=“minibus” maka biaya sewa = ...
 - › Jika kategori=“mobil” maka biaya sewa = ...
- › Ketiga kategori kendaraan memiliki implementasi yang berbeda-beda untuk method BiayaSewa().
- › Kendaraan dengan kategori yang berbeda-beda dapat dirancang sebagai subclass dari kelas Kendaraan.



Ilustrasi



Notasi: diagram kelas (UML)
yang dimodifikasi untuk
memudahkan ilustrasi
(jangan ditiru)

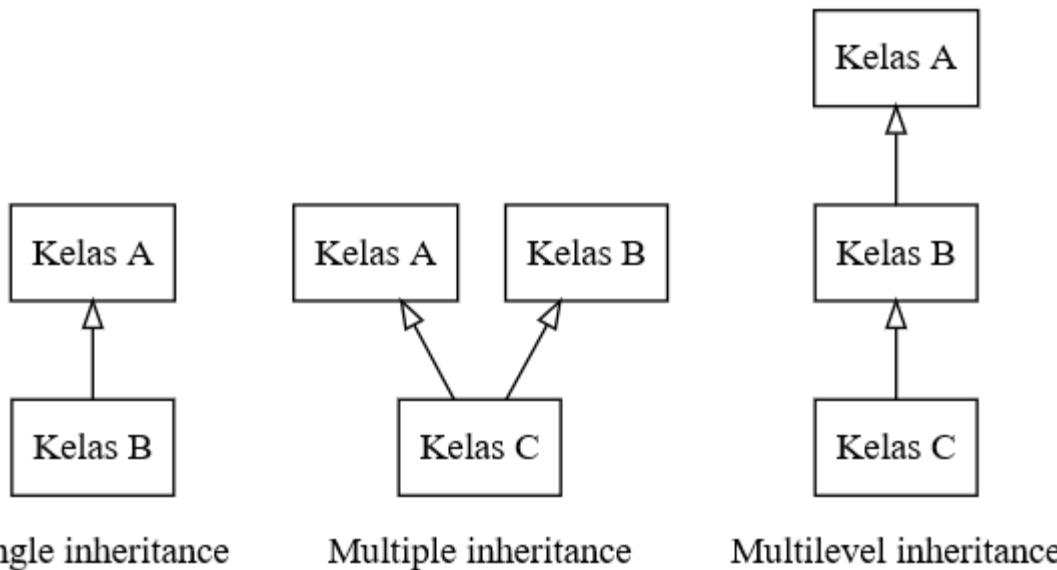
- Tidak ada lagi atribut kategori.
- BiayaSewa() diimplementasikan di *subclass*.
- Efek: jika ada jenis kendaraan baru, dapat membuat *subclass* baru tanpa mengubah kelas Kendaraan.

Kelas abstrak

- › Pada contoh sebelumnya, kelas Kendaraan adalah kelas abstrak.
 - › Ada method yang tidak diimplementasikan (`BiayaSewa(int)`).
- › Kelas abstrak tidak dapat langsung diinstansiasi.
- › Objek yang diciptakan adalah instansiasi dari subclass-nya yang tidak abstrak.
- › Contoh lain: kelas Shape adalah abstrak, tidak bisa digambar (method `Draw()` tidak dapat diimplementasi) jika tidak tahu bentuk “nyata”-nya.
 - › Subclass Rectangle, Circle diketahui bagaimana cara menggambarnya (mengimplementasikan `Draw()` sesuai atribut yang dimiliki: Rectangle memiliki panjang dan lebar, Circle memiliki titik pusat dan radius).



Jenis-jenis *inheritance* (1)

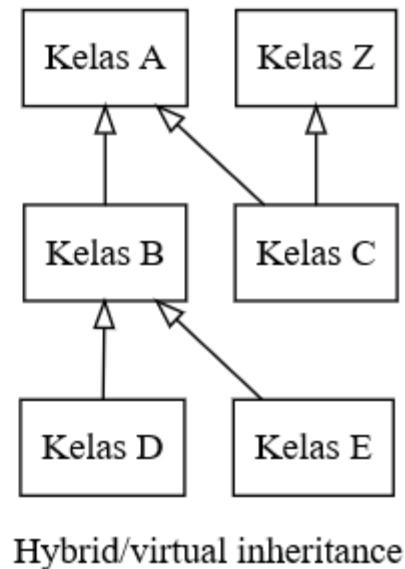
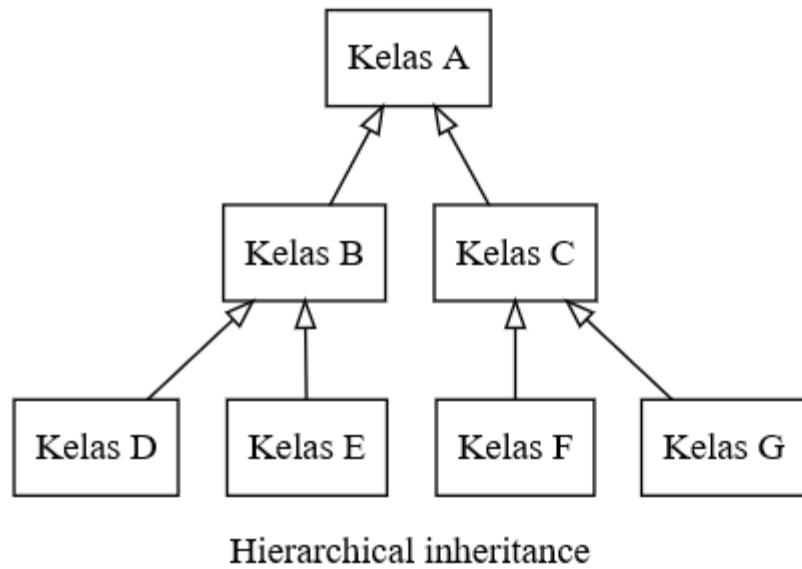


Single inheritance

Multiple inheritance

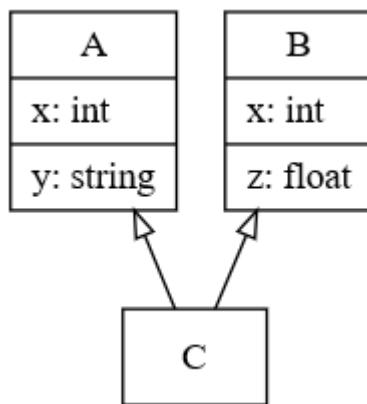
Multilevel inheritance

Jenis-jenis *inheritance* (2)

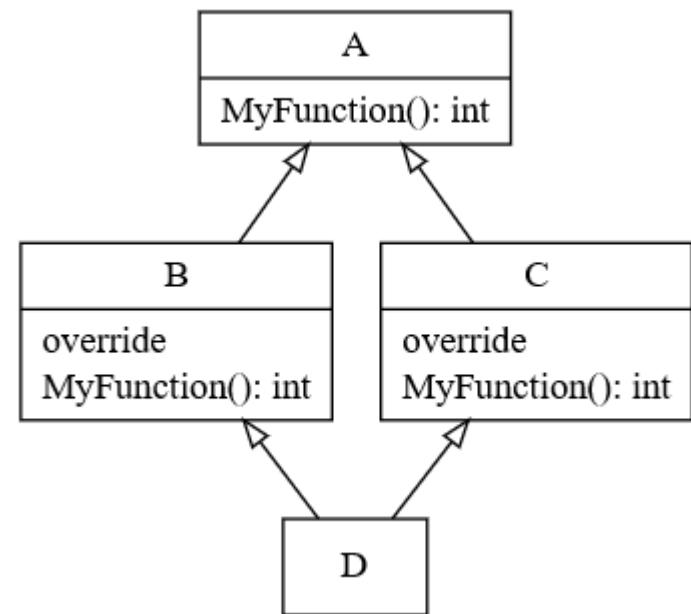


Masalah pada multiple inheritance (1)

- › Beberapa base class memiliki nama anggota yang sama.
 - › Diamond problem/deadly diamond of death (DDD).



- › c.x yang mana?



- › d.MyFunction() yang mana?



Masalah pada multiple inheritance (2)

- › Untuk mengatasi masalah-masalah tersebut, setiap bahasa OO memiliki caranya sendiri, misalnya:
 - › Rename, redefinition
 - › C++: untuk mencegah DDD, kelas B dan C harus dideklarasikan virtual
 - › Java, Ruby, Smalltalk: tidak memperbolehkan *multiple inheritance*
 - › dll.



Root object/class

- › Jika sebuah kelas dapat merupakan turunan dari kelas lain, adakah “*mother of all classes*”?
- › Pada beberapa bahasa OO, semua kelas, baik yang disediakan oleh bahasa/framework maupun yang dibuat sendiri oleh pemrogram, adalah *subclass* dari sebuah *root class* meskipun tidak dituliskan secara eksplisit.
 - › C#: kelas `System.Object`
 - › Java: kelas `java.lang.Object`
 - › Python: kelas `object`



Polymorphism

- › Objek-objek dari kelas turunan memiliki sifat sebagai kelas tersebut dan sekaligus kelas dasarnya.
 - › *polymorphism* (*poly* = banyak, *morph* = bentuk).
- › Sebuah “variabel” dapat dideklarasikan sebagai kelas Kendaraan, namun pada *run time* dapat merupakan instansiasi dari kelas Mobil atau Bus (subclass dari Kendaraan).
- › Sebaliknya, sebuah “variabel” bertipe Mobil dapat diperlakukan sebagai Kendaraan.
 - › i.e. message yang dapat diterima Kendaraan juga dapat diterima oleh Mobil.
- › Hubungan “is-a”: Mobil is a Kendaraan, Bus is a Kendaraan.



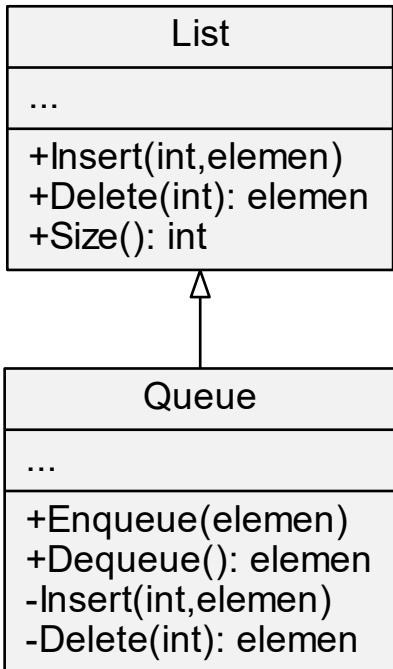
Inheritance vs. Composition

- › Inheritance: hubungan *is-a*, composition: hubungan *has-a*.
 - › Sedan *is a* mobil, SUV *is a* mobil, dst.
 - › Mobil *has a* mesin, mesin *has a* set of busi, dst.
- › Misalkan kita memiliki sebuah kelas List yang bisa ditambah/kurangi elemennya di posisi manapun
 - › Kita dapat memanfaatkan kelas tersebut untuk membuat kelas Queue:
 - › Komposisi? (Queue *has a* List)
 - › Inheritance? (Queue *is a* List)
 - › Queue harus bisa melakukan apa pun yang bisa dilakukan List (seharusnya tidak)
 - › Beberapa bahasa mendukung “penyembunyian” method dari superclass yang public menjadi private di subclass

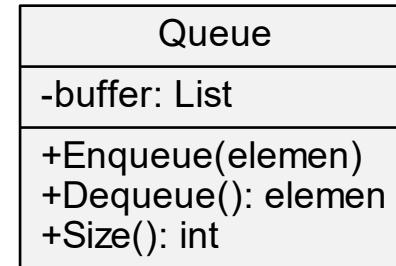


Queue (is a | has a) List

Queue 'is a' List



Queue 'has a' List



- Queue harus menyembunyikan `Insert(int,elemen)` dan `Delete(int)`

- `Enqueue(e)` memanggil `buffer.Insert(last_idx,e)`
- `Dequeue()` mengembalikan `buffer.Delete(0)`
- `Size()` membungkus `buffer.Size()`

Kapan menggunakan *inheritance*?

- › Inheritance should only be used when:
 - › Both classes are in the same logical domain
 - › The subclass is a proper subtype of the superclass
 - › The superclass's implementation is necessary or appropriate for the subclass
 - › The enhancements made by the subclass are primarily additive.

(<https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose>)





Bahasa C++: Inheritance

IF2210 – Semester II 2022/2023

Sumber: Diktat Bahasa C++ oleh Hans Dulimarta

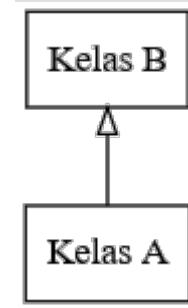
Pewarisan & penurunan kelas

- › Konsep-konsep yang berkaitan erat OOP: objek, kelas, pewarisan (*inheritance*), *polymorphism*, dan *dynamic binding*.
- › Pewarisan merupakan ciri unik dari OOP.
- › Pewarisan: pendefinisian dan pengimplementasian sebuah kelas berdasarkan kelas-kelas yang sudah ada (*reuse*).

Pewarisan & penurunan kelas

- › Kelas A mewarisi kelas B:

- › A = kelas turunan (*derived class/subclass*), dan
- › B = kelas dasar (*base class/superclass*)
- › Seluruh atribut & method B diwariskan ke A, kecuali ctor, dtor, cctor, dan operator=. A memiliki ctor, cctor, dtor, dan operator= sendiri.
- › Kelas A akan memiliki dua bagian:
 1. Bagian yang diturunkan dari B, dan
 2. Bagian yang didefinisikan sendiri (spesifik terhadap A)
- › Fungsi di dalam kelas turunan dapat mengakses semua atribut & method di dalam bagian non-private.



Penurunan kelas dalam C++

```
class kelas-turunan: mode-pewarisan kelas-dasar  
    // ...  
;
```

- › *Mode-pewarisan*: mempengaruhi tingkat pengaksesan setiap anggota (method & atribut) kelas dasar jika diakses melalui fungsi/method **di luar** kelas dasar maupun di luar kelas turunan.
- › Contoh:

```
class Minibus: public Kendaraan {  
    // ...  
};
```

Penurunan kelas dalam C++

- › Perubahan tingkat pengaksesan akibat pewarisan:

Tingkat akses di <i>base class</i>	Mode pewarisan		
	private	protected	public
private	private	private	private
protected	private	protected	protected
public	private	protected	public

- › private: = “sangat tertutup” (hanya method kelas tersebut yang dapat mengakses,
- › public: = “sangat terbuka” (fungsi/method manapun, di dalam atau di luar kelas dapat mengakses anggota dalam bagian ini),
- › protected: “setengah terbuka”/“setengah tertutup” (hanya kelas turunan yang dapat mengakses anggota pada bagian ini).

Contoh pewarisan

- › Growing Stack: Stack yang kapasitasnya dapat bertambah/berkurang secara otomatis
 - › push(): jika Stack penuh perbesar kapasitas
 - › pop(): jika kapasitas tak terpakai cukup besar perkecil kapasitas
- › Kelas GStack dapat diwariskan dari kelas Stack dengan cara:
 1. Mengubah perilaku pop() dan push() yang ada pada kelas Stack.
 2. Menambahkan atribut yang digunakan untuk menyimpan faktor penambahan/pencuitan kapasitas stack

```
// File GStack.h - Deklarasi kelas GStack

#ifndef GSTACK_H
#define GSTACK_H

#include "Stack.h"

class GStack: public Stack {
public:
    // ctor, cctor, dtor, oper= (tidak dituliskan)
    // redefinition of push & pop
    void push (int);
    void pop (int&);

private:
    int gs_unit;
    // method untuk mengubah kapasitas
    void grow();
    void shrink();
};

#endif GSTACK_H
```

```
// File GStack.cc
// Definisi method-method kelas GStack

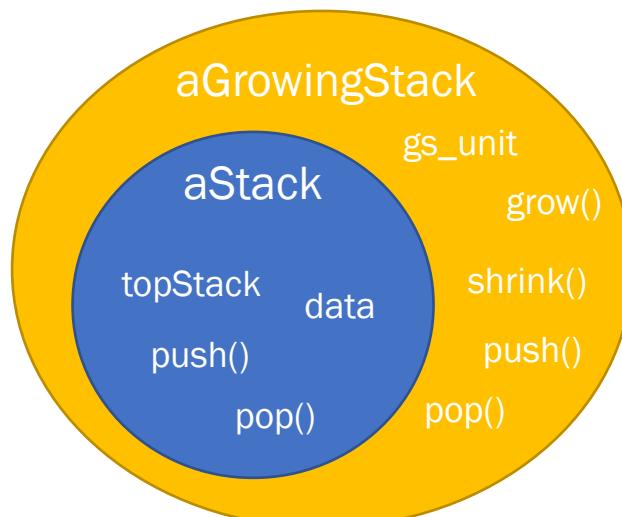
#include <stdio.h>
#include "GStack.h"

void GStack::push (int x) {
    if (isFull())
        grow();
    Stack::push(x);
}

void GStack::pop (int& x) {
    Stack::pop(x);
    if (size - topStack > gs_unit)
        shrink();
}
```

ctor, dtor, cctor, dan operator= kelas dasar

- › Komponen yang berasal dari kelas dasar dapat dianggap sebagai “sub-objek” dari kelas turunan
- › Pada penciptaan objek kelas turunan, konstruktor kelas dasar akan diaktifkan **sebelum** konstruktor kelas turunan.
- › Pada pemusnahan objek kelas turunan, destruktur kelas dasar dipanggil **setelah** destruktur kelas turunan.



Penanganan copy constructor

- › Ada tiga kasus:
 1. Kelas turunan tidak memiliki cctor, kelas dasar memiliki
 2. Kelas turunan memiliki cctor, kelas dasar tidak memiliki
 3. Baik kelas turunan maupun kelas dasar memiliki cctor
- › Kasus (1): cctor kelas dasar akan dipanggil, inisialisasi kelas turunan dilakukan secara *bitwise copy*
- › Pada kasus (2) dan (3), cctor dari kelas dasar **tidak dipanggil**, inisialisasi kelas dasar menjadi tanggung jawab kelas turunan.

Penanganan copy constructor

- › Penginisialisasi kelas dasar oleh kelas turunan melalui ctor atau cctor dilakukan melalui *constructor initialization list*

```
// File GStack.cc
// Definisi method-method kelas GStack

#include <stdio.h>
#include "GStack.h"

GStack::GStack(const GStack& s): Stack(s) {
    gs_unit = s.gs_unit;
}
```

Operator assignment

Assignment ditangani seperti inisialisasi.

- › Jika kelas turunan **tidak** mendefinisikannya, operator= dari kelas dasar akan dipanggil (jika ada).
- › Jika kelas turunan mendefinisikan operator= maka operasi *assignment* dari kelas dasar menjadi tanggung jawab kelas turunan.

```
// File GStack.cc
#include <stdio.h>
#include "GStack.h"

GStack& GStack::operator=(const GStack& s) {
    Stack::operator=(s); // oper= dari Stack
    gs_unit = s.gs_unit;
    return *this;
}
```

Polymorphism

- › Objek-objek dari kelas turunan memiliki sifat sebagai kelas tersebut dan sekaligus kelas dasarnya.
 - › *polymorphism* (*poly* = banyak, *morph* = bentuk).
- › *reference* (“ref”) dan *pointer* (“ptr”) dapat bersifat polimorfik.
- › Dalam C++ ref/ptr dapat digunakan untuk *dynamic binding*. ref/ptr memiliki tipe statik dan tipe dinamik.
 - › Tipe statik = tipe objek pada saat deklarasikan,
 - › Tipe dinamik = tipe objek pada saat eksekusi, dapat berubah bergantung pada objek yang diacu.

Dynamic binding

- › Tipe dinamik digunakan pada saat eksekusi untuk memanggil method yang berasal dari kelas berbeda-beda melalui sebuah ref/ptr dari **kelas dasar**.
- › Method yang akan dipanggil secara dinamik, harus dideklarasikan sebagai *virtual* (dilakukan di **kelas dasar**).
- › Dalam contoh Gstack method push() dan pop() akan dipanggil secara dinamik
 - › Kelas Stack harus mendeklarasikan sebagai method virtual

```
// File: Stack.h
class Stack {
public:
    // ctor, cctor, dtor, & oper=
public:
    // services
    virtual void push(int); // <== penambahan "virtual"
    virtual void pop(int&); // <== penambahan "virtual"
};
```

```
#include <GStack.h>

// tipe memiliki tipe dinamik
void funcVal(Stack s) { s.push (10); }

// memiliki tipe dinamik
void funcPtr(Stack *t) { t->push (10); }

// memiliki tipe dinamik
void funcRef(Stack& u) { u.push (10); }

main() {
    GStack gs;
    funcVal(gs); // Stack::push() dipanggil
    funcPtr(&gs); // GStack::push() dipanggil
    funcRef(gs); // GStack::push() dipanggil
}
```

Virtual destructor

```
Stack* sp[MAX_ELEM];
// ... kode-kode lain
for (i=0; i<MAX_ELEM; i++)
    delete sp[i]; // tipe elemen: Stack/ GStack !
```

- › Destruktor mana yang akan dipanggil `delete sp[i];?`
- › Diperlukan *virtual destructor*

```
class Stack {
public:
    // ctor, dtor, cctor
    //...
    virtual ~Stack();
    //
};
```

Virtual destructor

- › Di dalam kelas turunan (GStack), destruktor **tidak perlu** dideklarasikan virtual karena otomatis mengikuti jenis dtor kelas dasar.

```
class Base {
public:
    virtual ~Base() { /* releases Base's resources */ }
};

class Derived: public Base {
    ~Derived() { /* releases Derived's resources */ }
};

int main() {
    Base* b = new Derived;
    delete b; // Makes a virtual function call to Base::~Base()
              // since it is virtual, it calls Derived::~Derived()
              // which can release resources of the derived class,
              // and then calls Base::~Base() following the usual
              // order of destruction
}
```

Abstract base class (ABC)

- › *Abstract Base Class* = kelas dasar untuk objek abstrak.
 - › Contoh: DataStore, Vehicle, Shape, dst.
- › Bagaimana implementasi `Shape::Draw()`?
 - › Tidak dapat diimplementasikan di dalam kelas Shape, namun harus dideklarasikan.
 - › Dideklarasikan sebagai method *pure virtual*.
- › ABC = kelas yang memiliki method *pure virtual*

Abstract base class

```
class DataStore {  
public:  
    // ...  
    virtual void Clear() = 0; // pure virtual  
    // ...  
};
```

- › Tidak ada objek yang dapat dibuat dari kelas dasar abstrak
- › Di dalam kelas non-abstrak (yang mewarisi kelas dasar abstrak) method *pure virtual* harus diimplementasikan.

```
// File: Stack.h
// Deskripsi: deklarasi kelas Stack yang diturunkan dari
//             DataStore

class Stack: public DataStore {
public:
    // ...
    void Clear ();
};

// File: Tree.h
// Deskripsi: deklarasi kelas Tree yang diturunkan dari DataStore
class Tree: public DataStore {
public:
    // ...
    void Clear ();
};
```



Bahasa C++: Keyword "virtual"

IF2210 – Semester II 2022/2023

Keyword virtual di C++

- › Keyword `virtual` memiliki dua kegunaan:
 - › *Specifier* untuk method
 - › *Specifier* untuk kelas dasar (*base class*)

Keyword virtual sebagai specifier untuk method

Virtual method

- › Virtual method adalah method yang perilakunya dapat di-override oleh *derived class*.
- › Jika objek dari *derived class* ditangani dengan ptr atau ref, maka pemanggilan method akan menginvokasi perilaku hasil *override*. (ingat kembali *polymorphism*)
- › Invokasi perilaku method asli (milik *base class*) dapat dilakukan dengan menggunakan scope (::) *base class*.
- › Jika method tidak virtual, pemanggilan method pada *derived class* akan menginvokasi perilaku milik *base class*.

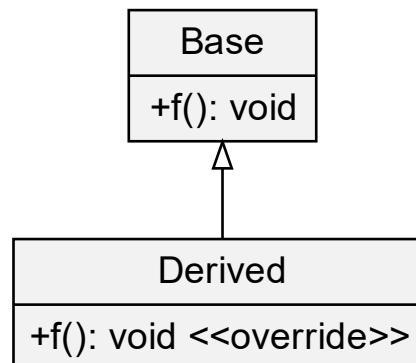
```

#include <iostream>

class Base {
    virtual void f() {
        std::cout << "base\n";
    }
};

class Derived: Base {
    void f() override { // keyword 'override' tidak wajib
        std::cout << "derived\n";
    }
};

```



Notasi pada kuliah ini:
diagram kelas (UML)
yang dimodifikasi untuk
memudahkan ilustrasi
(jangan ditiru)

```
int main() {
    Base b;
    Derived d;

    // non-ptr, non-ref: non-virtual function call
    Base b1 = b;      // the type of b1 is Base
    Base d1 = d;      // the type of d1 is Base as well
    b1.f();           // prints "base"
    d1.f();           // prints "base" as well

    // virtual function call through reference
    Base& br = b;    // the type of br is Base&
    Base& dr = d;    // the type of dr is Base& as well
    br.f();           // prints "base"
    dr.f();           // prints "derived"

    // virtual function call through pointer
    Base* bp = &b;   // the type of bp is Base*
    Base* dp = &d;   // the type of dp is Base* as well
    bp->f();         // prints "base"
    dp->f();         // prints "derived"

    // non-virtual function call
    br.Base::f();    // prints "base"
    dr.Base::f();    // prints "base"
}
```

In detail... (1)

- › Jika:
 - › method **f** dideklarasikan sebagai **virtual** di kelas **Base**,
 - › terdapat kelas **Derived**, yang diturunkan (langsung maupun tidak) dari kelas **Base**, memiliki deklarasi method dengan kesamaan: **nama**, **signature**, **cv-qualifiers**, dan **ref-qualifiers**,
- › maka method tsb. di kelas **Derived**:
 - › juga bersifat **virtual** (meski tanpa **keyword virtual**), dan
 - › meng-**override** **Base** :: **f** (meski tanpa **keyword override**)

In detail... (2)

- › Base::f tidak perlu *visible* (dapat dideklarasikan **private**, atau diwariskan dengan mode akses **private**) untuk bisa di-*override*.

```
class B {
private:
    virtual void do_f(); // private member
public:
    void f() { do_f(); } // public interface
};

class D: public B {
public:
    void do_f() override; // overrides B::do_f
};

int main() {
    D d;
    B* bp = &d;
    bp->f(); // internally calls D::do_f();
}
```

Final overrider

- › *Final overrider* adalah method **sebenarnya** yang dieksekusi saat *run-time* ketika method virtual dipanggil.
 - › Saat *compile-time*, *compiler* tidak tahu implementasi mana yang akan dieksekusi
- › Method virtual *f* milik kelas Base adalah *final overrider* kecuali kelas turunannya mendeklarasikan atau mewariskan method lain yang meng-*override* *f*.

```

struct A { virtual void f(); };      /* A::f is virtual          */
struct B: A { void f(); };           /* B::f overrides A::f in B */
struct C: virtual B { void f(); }; /* C::f overrides A::f in C */
struct D: virtual B {};             /* D does not introduce an overrider,
                                         B::f is final in D */
struct E: C, D {                  /* E does not introduce an overrider,
                                         C::f is final in E */
    using A::f; /* not a function declaration,
                   just makes A::f visible to lookup */
};

int main() {
    E e;
    e.f(); /* virtual call calls C::f, the final overrider in e */
    e.E::f(); /* non-virtual call calls A::f, which is visible in E */
}

// struct = class dengan hak akses member default adalah public

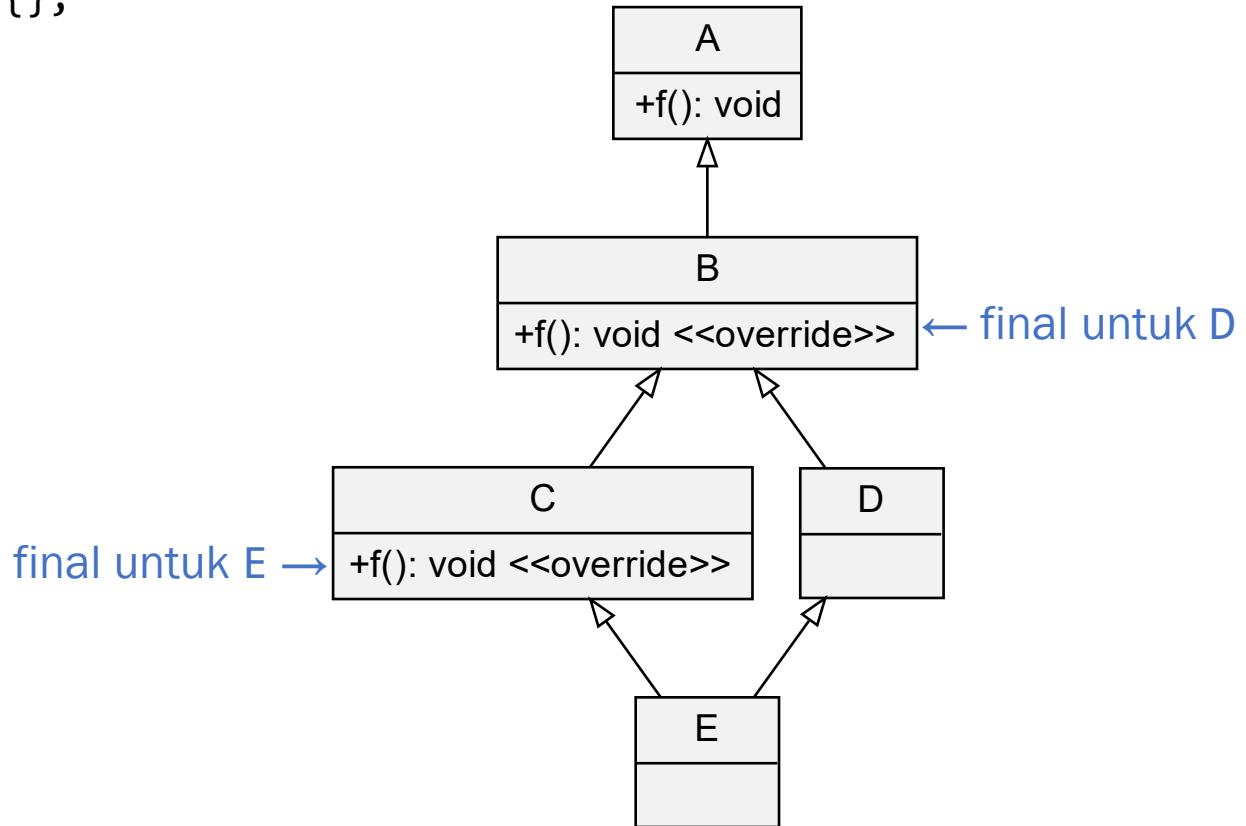
```

```

struct A { virtual void f(); };
struct B: A { void f(); };
struct C: virtual B { void f(); };
struct D: virtual B {};
struct E: C, D {
    using A::f;
};

int main() {
    E e;
    e.f();
    e.E::f();
}

```



Final overrider: conflict

```
struct A {
    virtual void f();
};

struct VB1: virtual A {
    void f(); // overrides A::f
};

struct VB2: virtual A {
    void f(); // overrides A::f
};

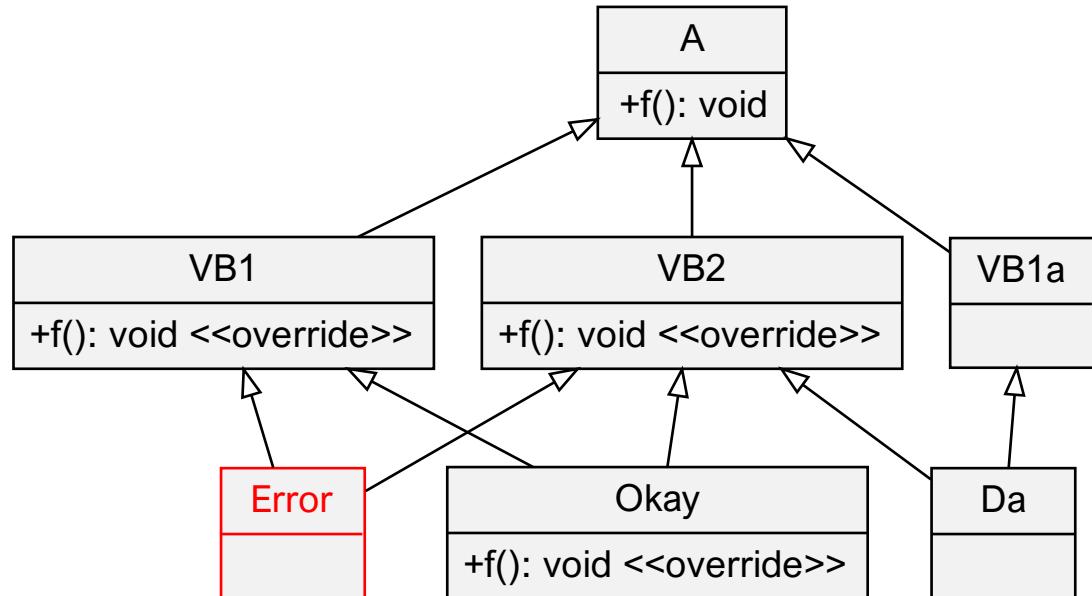
// struct Error: VB1, VB2 {
//     // Error: A::f has two final overriders in Error
// };

struct Okay: VB1, VB2 {
    void f(); // OK: this is the final overrider for A::f
};

struct VB1a: virtual A {}; // does not declare an overrider
struct Da: VB1a, VB2 {
    // in Da, the final overrider of A::f is VB2::f
};
```

Final overrider: conflict

```
struct A {  
    virtual void f();  
};  
struct VB1: virtual A {  
    void f();  
};  
struct VB2: virtual A {  
    void f();  
};  
// struct Error: VB1, VB2 {  
//  
// };  
struct Okay: VB1, VB2 {  
    void f();  
};  
struct VB1a: virtual A {};  
struct Da: VB1a, VB2 {  
};
```



Hiding virtual function

- Method dengan nama yang sama tapi memiliki signature yang berbeda tidak meng-override method kelas dasar, tapi menyembunyikannya.

```

struct B {
    virtual void f();
};

struct D: B {
    void f(int) // D::f hides B::f (wrong parameter list)
};

struct D2: D {
    void f(); // D2::f overrides B::f (doesn't matter that it's not visible)
};

int main() {

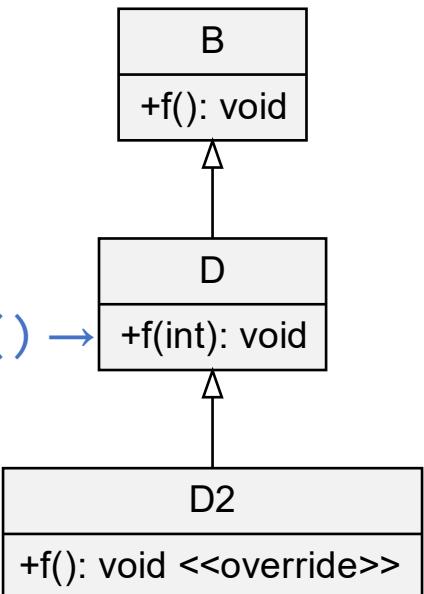
    B b;    B& b_as_b = b;
    D d;    B& d_as_b = d;    D& d_as_d = d;
    D2 d2; B& d2_as_b = d2;    D& d2_as_d = d2;

    b_as_b.f(); // calls B::f()
    d_as_b.f(); // calls B::f()
    d2_as_b.f(); // calls D2::f()

    d_as_d.f(); // Error: lookup in D finds only f(int)
    d2_as_d.f(); // Error: lookup in D finds only f(int)
}

```

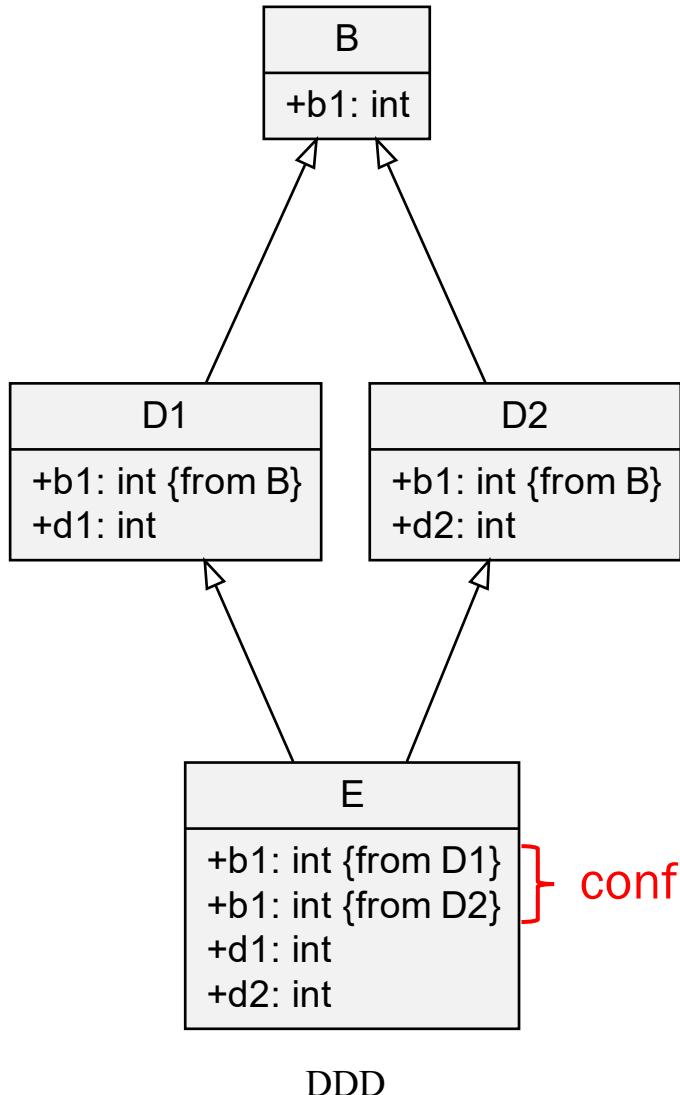
Menyembunyikan B::f() →



Keyword virtual sebagai specifier kelas dasar

Mengingat kembali: *inheritance* & DDD

- › Jika D turunan dari B, maka D mempunyai seluruh anggota B ditambah anggota D sendiri. Contoh:
 - › Jika B memiliki anggota b1,
 - › ... dan D1 adalah turunan B dengan tambahan anggota d1, maka D1 memiliki anggota: b1 dan d1.
 - › ... dan D2 adalah turunan B dengan tambahan anggota d2, maka D2 memiliki anggota: b1 dan d2.
 - › Jika E adalah turunan dari D1 dan D2, maka E memiliki anggota: (dari D1) **b1**, d1, (dari D2) **b1**, dan d2 → DDD.



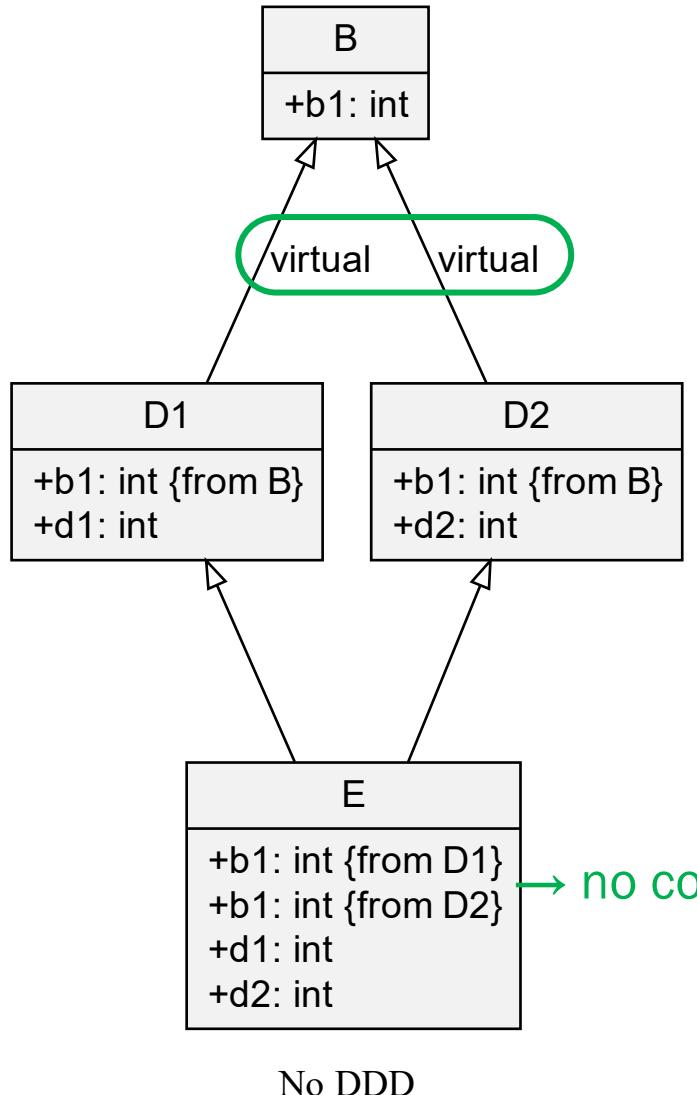
```

E e;
e.b1 = 1;      // error
e.B::b1 = 2;   // error
e.D1::b1 = 3; // ok
e.D2::b1 = 4; // ok
  
```

DDD

Kelas dasar virtual

- › Kelas dasar virtual menghindari terjadinya DDD:
- › Untuk setiap *base class* yang diturunkan secara virtual, kelas turunan level berikutnya hanya memiliki satu subobjek dari kelas tersebut.
- › Pada contoh sebelumnya, D1 dan D2 harus diturunkan secara virtual dari B
- › Sintaks: `class D1: virtual public B { ... };`
atau `class D1: public virtual B { ... };`
(sama saja)



```
E2 e;  
e.b1 = 1;      // ok  
e.B::b1 = 2;  // ok  
e.D1::b1 = 3; // ok  
e.D2::b1 = 4; // ok  
// all calls access the same b1
```

Contoh lain...

```
struct B { int n; };
class X: public virtual B {};
class Y: virtual public B {};
class Z: public B {};

struct AA: X, Y, Z {
    // every object of type AA has one X, one Y, one Z, and two B's:
    // one that is the base of Z and one that is shared by X and Y
    AA() {
        X::n = 1; // modifies the virtual B subobject's member
        Y::n = 2; // modifies the same virtual B subobject's member
        Z::n = 3; // modifies the non-virtual B subobject's member

        std::cout << X::n << Y::n << Z::n << '\n'; // prints 223
    }
};
```

Contoh lain...

```
struct B { int n; };
class X: public virtual B {};
class Y: virtual public B {};
class Z: public B {};
```

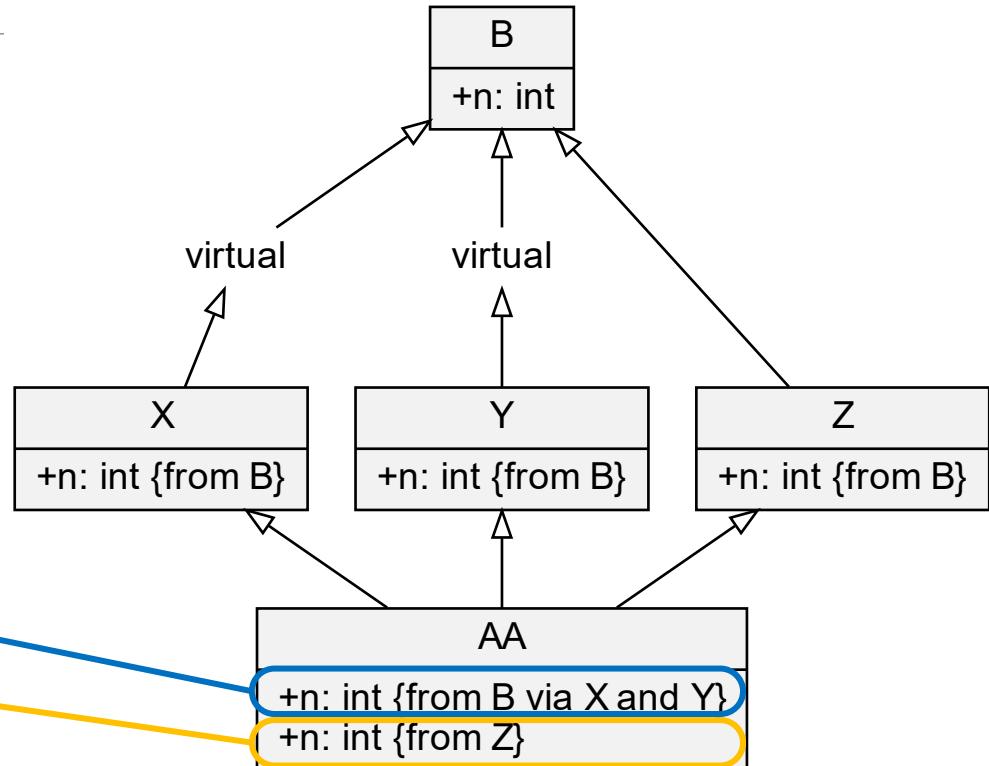
```
struct AA: X, Y, Z {
```

```
AA() {
```

```
    X::n = 1;
    Y::n = 2;
    Z::n = 3;
```

```
};
```

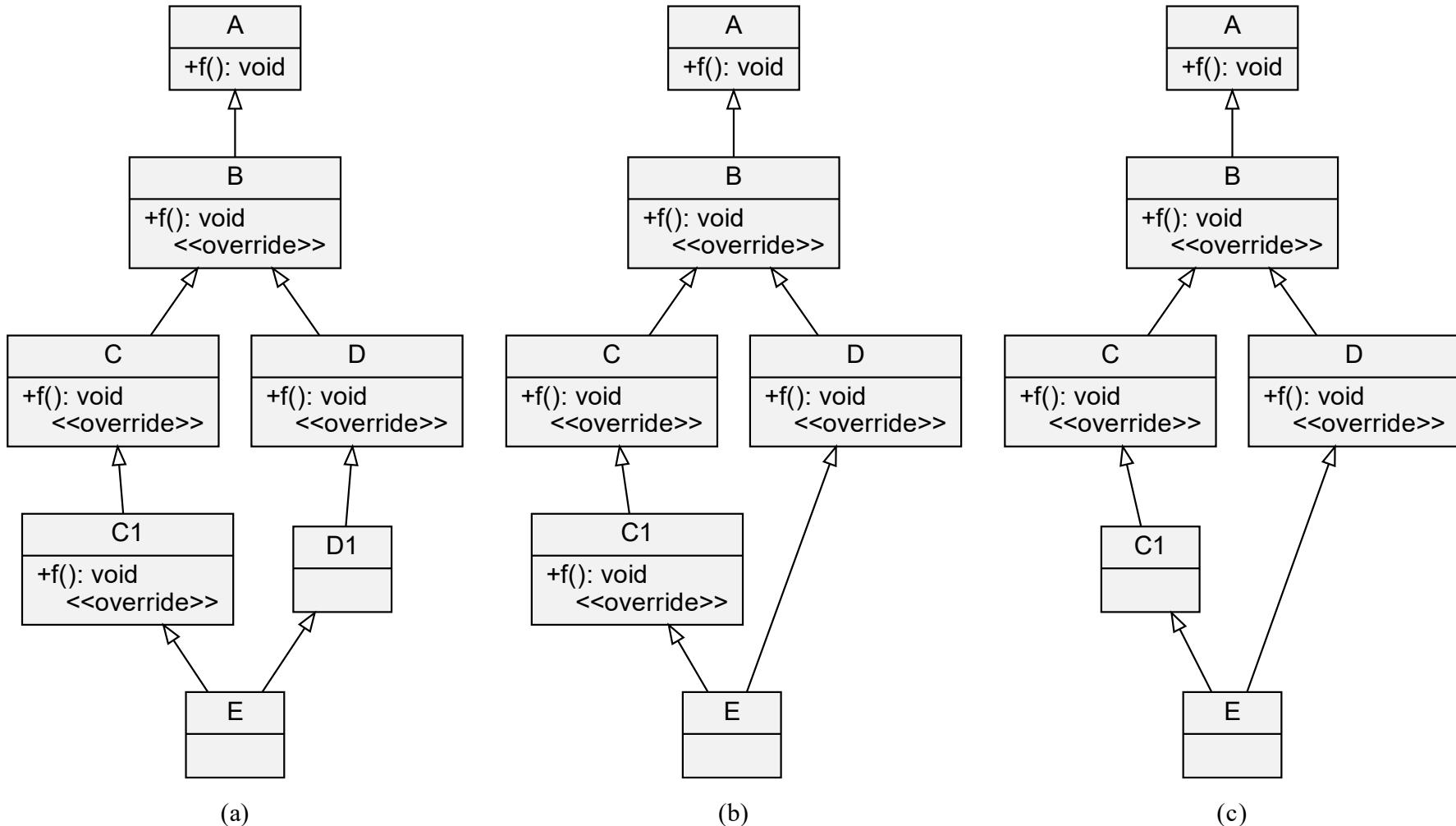
```
    std::cout << X::n << Y::n << Z::n << '\n';
```



Latihan

Method `f()` mana yang menjadi *final* *overrider* di kelas E pada hierarki-hierarki berikut? Apakah terjadi konflik?

[asumsi A::`f()` adalah virtual]



Sumber

- › www.cppreference.com



Bahasa C++: Function Template dan Kelas Generik

IF2210 -Semester II 2022/2023

Sumber: Diktat Bahasa C++ oleh Hans Dulimarta

Function Template

Latar belakang

- › Seringkali kita membutuhkan suatu operasi yang sejenis terhadap tipe yang berbeda-beda.
- › Contoh: fungsi `min()` dapat diterapkan untuk `int` maupun `float`.

```
int min(int a, int b) {  
    return a < b ? a : b;  
}  
  
float min(float a, float b) {  
    return a < b ? a : b;  
}
```

- › Untuk setiap tipe yang akan dimanipulasi oleh fungsi `min()`, harus ada sebuah fungsi untuk tipe tersebut.

Bagaimana mengatasinya?

- › Trik yang biasa digunakan adalah dengan definisi makro:

```
#define mmin(a,b) ((a) < (b) ? (a) : (b))
```

- › Namun makro tersebut dapat memberikan efek yang tidak diinginkan pada contoh *statement* berikut:

```
if (mmin (x++, y) == 0) printf ("...");
```

yang akan diekspansi sebagai:

x nya jd bertambah 2 kali

```
if (((x++) < (y) ? (x++) : (y)) == 0) printf ("...");
```

- › Subtitusi makro ≠ pemanggilan fungsi.

Solusi: function template

- › Deklarasi: menggunakan *prefix* “`template <class XYZ>`” sebelum nama fungsi. Contoh:

```
template <class T>
T min (T a, T b) {
    return a < b ? a : b;
}
```

- › Dipanggil dengan cara:

```
int a, b, c;
c = min <int> (a, b);
float x, y, z;
z = min <float> (x, y);
```

Catatan-catatan (1)

- › Banyaknya nama tipe (kelas) yang dicantumkan di antara ‘<’ dan ‘>’ dapat lebih dari satu. Setiap nama tipe harus didahului oleh kata kunci `class`. Contoh:

```
template <class T1, class T2>
```

- › Nama tipe yang dicantumkan di antara ‘<’ dan ‘>’ harus tercantum sebagai *function signature*. Contoh:

```
template <class T1, class T2, class T3>
T1 myFunc (T2 a, T3 b) {...} /* error: T1 bukan bagian dari
signature */
```

Catatan-catatan (2)

- Definisi template fungsi dapat disertai oleh definisi "non-template" dari fungsi tersebut.

```
template <class T>
T min_arr (const T x[], int size) {...} /* template */
Complex min_arr (const Complex x[], int size) {...} /* non-
template */
```

- Apa yang akan terjadi pada pemanggilan berikut?

```
Complex c[5], d;
d = min_arr(c, 5);
```

Kelas Generik

Generic Class

- › Kelas generic: kelas yang masih “umum”, belum spesifik ketika didefinisikan
- › Pada saat deklarasi objek, hal yang umum harus dibuat spesifik
 - › Setelah dibuat spesifik, baru bisa dipakai
- › Biasanya yang “umum” adalah “type”-nya, dipakai untuk membungkus “operasi” yang sama
- › Dalam bahasa C++ menjadi *template*

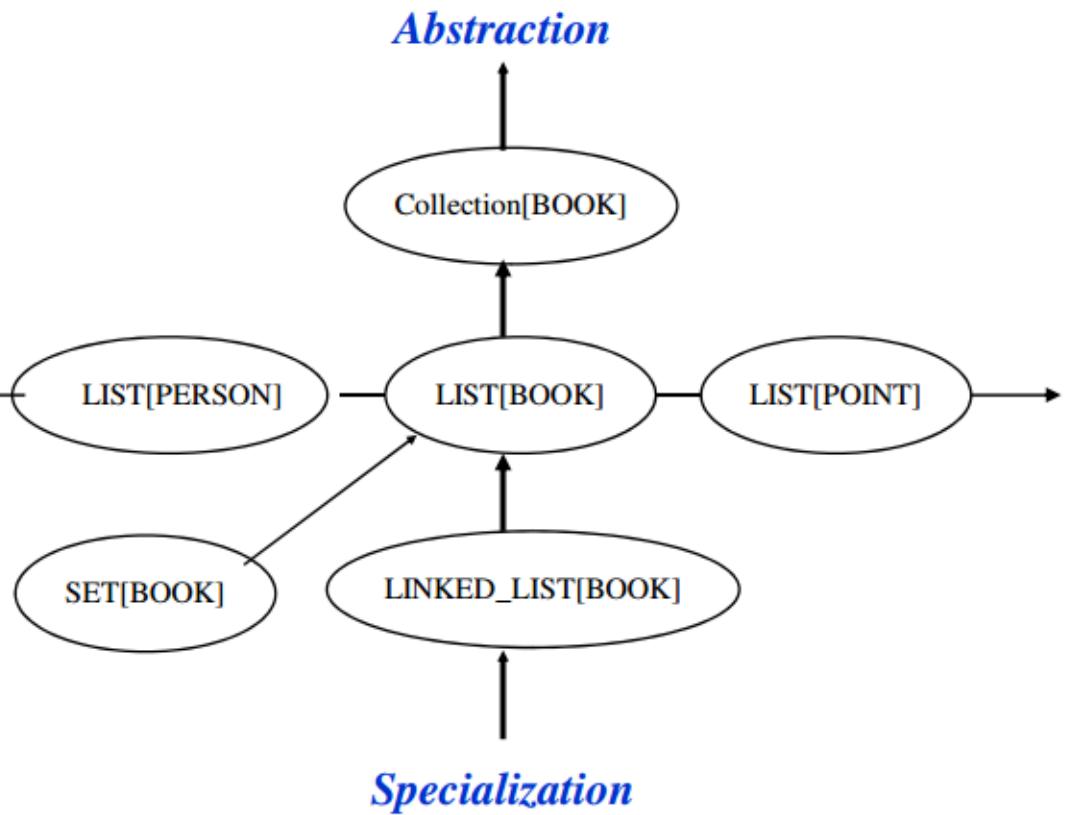
Generic vs Inheritance

Horizontal : Generic

List of “something”

Type parametrization

Vertical : Inheritance



Konsep template pada kelas

- › Kelas generik merupakan penerapan konsep template pada kelas.
- › Dengan demikian cukup mendefinisikan satu kelas generik (misal kelas **Stack** generik) untuk dapat diinstansiasikan sebagai **Stack of int**, **Stack of float**, **Stack of char**, **Stack of Complex**, **Stack of String**, dst.
- › Untuk menciptakan kelas generik, perancang kelas harus dapat mengidentifikasi parameter-parameter mana yang menentukan sifat kelas.
 - › Dalam contoh **Stack**, parameter yang menentukan kelas adalah jenis **int** yang berkaitan dengan data yang disimpan di dalam **Stack**.

Contoh Kelas Generik

- › Kelas Point yang masih generic:
 - › **GPoint <Numeric>**: kelas Point dengan absis dan ordinat ber-type numerik
 - › Saat dideklarasi:
 - › **GPoint <integer> P**; maka absis dan ordinat akan ber-type integer
 - › **GPoint <float> P**; maka absis dan ordinat akan ber-type float
- › Kelas **G_Array <a_type>**: array of elemen bertype a_type
 - › Pada saat dideklarasikan dapat menjadi: array of <**integer**>, array of <**float**>, array of <**Point**>, ...

```
template <class T>
class Stack {
public:
    // ctor-cctor-dtor
    Stack();           // default ctor
    Stack(int);        // ctor dengan ukuran max stack
    Stack(const Stack&); // cctor
    ~Stack();

    // services
    void Push (T);      // <== parameter generik
    void Pop (T&);      // <== parameter generik
    int isEmpty() const;
    int isFull() const;

    // operator
    Stack& operator= (const Stack&);
    void operator<< (T); // <== parameter generik
    void operator>> (T&); // <== parameter generik

private:
    const int defaultStackSize = 500; // ANSI: tidak boleh inisialisasi
    int topStack;
    int size;
    T *data;           // <== parameter generik
};
```

Contoh: kelas Stack generik

Catatan-catatan (1)

- › Penciptaan objek dilakukan sebagai berikut:

```
Stack<int> a;           // Stack of integer
Stack<double> b(30);   // Stack of double, kapasitas maks = 30
Stack<Complex> c;     // Stack of Complex
```

- › Nama Stack<int>, Stack<double>, ... dapat dipandang sebagai nama tipe baru.
- › Definisi fungsi anggota harus dituliskan sebagai fungsi template dan scope yang semula dituliskan sebagai Stack:: harus dituliskan sebagai Stack<T>::::.

```
template <class T>
Stack<T>::Stack() {
    size = defaultStackSize;
    topStack = 0;
    data = new T[size];
} /* konstruktor */
```

```
template <class T>
void Stack<T>::Push(T item) {
    // ...
} /* fungsi anggota */
```

Catatan-catatan (2)

- › Deklarasi kelas generik maupun definisi fungsi generik dituliskan dalam file header (.h).
 - › Ada cara-cara lain, lihat <https://stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file>,
<https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl>
- › Di luar konteks definisi kelas generik, nama tipe yang dapat digunakan (misalnya oleh fungsi, deklarasi variabel/objek, dsb.) adalah nama tipe hasil instansiasi, seperti Stack<int>, Stack<double>, atau Stack<Complex>. Termasuk di dalam definisi fungsi, contoh:

```
template <class T>
void Stack<T>::Reverse() {
    Stack<T> stemp; // objek lokal yang generik
    // ...algoritma dari Reverse()...
}
```

Spesialisasi kelas generik

- › Kita dapat mendefinisikan perilaku spesial untuk kelas generik yang memiliki tipe tertentu sebagai *template parameter*.

```
// class template:  
template <class T>  
class mycontainer {  
    T element;  
public:  
    mycontainer (T arg) {element=arg;}  
    T increase () {return ++element;}  
};  
  
// class template specialization:  
template <>  
class mycontainer <char> {  
    char element;  
public:  
    mycontainer (char arg) {element=arg;}  
    char uppercase () // perilaku khusus untuk mycontainer <char>  
    {  
        if ((element>='a')&&(element<='z'))  
            element+='A'-'a';  
        return element;  
    }  
};
```



Bahasa C++: Exception Handling

IF2210 – Semester II 2022/2023

Tujuan

- › Di akhir sesi, peserta diharapkan mampu untuk:
 - › Mendefinisikan *exception*
 - › Menangani *exception* dengan menggunakan blok try-catch
 - › Membuat kelas yang dilengkapi dengan class Exception sendiri

Penanganan Kesalahan pada Program

- › Penambahan kode khusus untuk menangani kesalahan dalam eksekusi program membuat program menjadi rumit

```
int LakukanAksi() {  
    // ...  
    if ( ... ) // periksa kondisi kesalahan  
        return NILAI_KHUSUS;  
    else  
        return NILAI_FUNGSI;  
}  
  
-----  
  
if (LakukanAksi() == NILAI_KHUSUS) {  
    // ... jalankan instruksi untuk menangani kesalahan  
} else {  
    // instruksi-1  
    // instruksi-2  
    // ...  
    // instruksi-n  
}
```

Exception

- › Kesalahan pada eksekusi program = exception
- › Penanganan exception pada C++: **throw**, **catch**, and **try**
- › **return** = kembali dari fungsi secara normal;
throw = kembali fungsi secara abnormal (exception)

```
int LakukanAksi() {  
    // ...  
    if ( ... ) // periksa kondisi kesalahan  
        throw "Ada kesalahan";  
    else  
        return NILAI_FUNGSI;  
}
```

Exception

- › *Error* tidak harus ditangani dengan *exception handling* (namun *exception* mempermudah penanganan *error*)
- › *Exception* bekerja dengan cara mengubah alur eksekusi program sambil melempar suatu objek tertentu sebagai informasi untuk alur yang baru
- › Sebuah *event* yang akan menginterupsi alur proses program normal dari sebuah program

Membuat Exception

- › Objek yang diciptakan disebut *exception object*
- › *Exception object* mengandung informasi tentang *error* tersebut (termasuk tipe dan state program ketika *error* terjadi)
- › Menciptakan *exception object* dan melempar ke *runtime system* disebut *throwing an exception* (keyword `throw` di C++)
- › Setelah *method* melempar *exception*, *runtime system* akan mencari sesuatu untuk *meng-handle* itu = disebut *exception handler*
- › *Exception handler* akan melakukan *catch the exception*
 - › Jika tidak *di-handle* (*di-catch*) akan mengakibatkan program *terminate abnormally*

Exception Handling

- › **try block** (keyword `try` di C++)
 - › Berisi kode yang mungkin memunculkan exception
 - › *Try block* bisa dibuat untuk setiap kode yang mungkin menimbulkan exception
 - › Bisa juga dengan mengumpulkan banyak kode dalam sebuah *try block*
- › **catch block** (keyword `catch` di C++)
 - › Berisi kode yang merupakan *exception handler*
 - › Menangani exception dengan tipe yang sesuai dengan tipe yang ditunjukkan pada argumen
- › Jika sebuah *method* ditulis menangani exception, sebaiknya dilakukan invokasi dalam sebuah blok `try ... catch`

Exception Handling

- › **catch** dituliskan setelah sebuah blok **try** untuk menangkap exception yang di-**throw**

```
try {
    // instruksi-1
    // instruksi-2
    LakukanAksi();
    // instruksi-4
    // ...
    // instruksi-n
}
catch (const char*) {
    // ... jalankan instruksi untuk menangani kesalahan
}

// eksekusi berlanjut pada bagian ini...
```

Exception Handling

- › Perintah dalam blok **catch** = *exception handler*
- › Sebuah blok try dapat memiliki > 1 *exception handler*, masing-masing untuk menangani tipe exception yang berbeda
- › Tipe *handler* ditentukan “*signature*”-nya. catch-all handler memiliki “*signature*”

```
try {  
    // instruksi  
}  
catch (StackExp&) {  
    // handler untuk tipe StackExp  
}  
catch (...) { // literally pakai tiga tanda titik  
    // handler untuk semua jenis exception lainnya  
}
```

Pemilihan Handler

- › Jika terjadi *exception*, hanya satu *handler* yang dipilih berdasarkan “*signature*”-nya
- › Instruksi di dalam *handler* yang terpilih diajarkan
- › Eksekusi dari blok **try** tidak dilanjutkan
- › Eksekusi berlanjut ke bagian yang dituliskan setelah bagian **try-catch** tersebut.

Contoh Kelas untuk Stack Exception

```
const int STACK_EMPTY = 0;
const int STACK_FULL = 1;

class StackExp {
public:
    // ctor, cctor, dtor, operator=
    // services
    void DisplayMsg() const;
    static int NumException();
private:
    // static member, shared by all objects of this case
    static int num_ex; // pencacah jumlah exception
    static char* msg[]; // tabel pesan kesalahan
    const int msg_id; // nomor kesalahan
}
```

Fungsi Anggota StackExp

```
#include <iostream>
using namespace std;
#include "StackExp.h"

int StackExp::num_ex = 0;
char* StackExp::msg[] = { "Stack is empty!", "Stack is full!" };

StackExp::StackExp (int x) : msg_id(x) {
    num_ex++; // increase the exception counter
}

StackExp::StackExp (const StackExp& s) : msg_id (s.msg_id) {}

void StackExp::DisplayMsg() const {
    cerr << msg[msg_id] << endl;
} console error

int StackExp::NumException() {
    return num_ex;
}
```

Modifikasi Kelas Stack

```
// File: Stack.cpp
// Deskripsi: Kelas Stack dengan exception handling

#include "StackExp.h"

void Stack::Push(int x) {
    if (isFull()) throw (StackExp (STACK_FULL)); // Raise exception
    else {
        // algoritma Push
    }
}

void Stack::Pop(int& x) {
    if (isEmpty()) throw (StackExp (STACK_EMPTY)); // Raise exception
    else {
        // algoritma Pop
    }
}
```

Pemakaian Kelas Stack

```
#include "Stack.h"
#include <iostream>
using namespace std;

int main () {
    Stack s;
    int n;
    try {
        // instruksi
        s << 10;
        // instruksi
    }
    catch (StackExp& ) {
        s.DisplayMsg();
    }
    n = StackExp::NumException();
    if (n > 0) { cout << "Muncul " << n << " stack exception" << endl; }

    return 0;
}
```

Chain Exception

- Method bisa merespon terjadinya exception dengan melempar exception lagi

```
try {
    // ...
}
catch (IOException e) {
    throw new SampleException("Other IOException", e);
} bisa lgsg throw e
```

CPP Standard Exception

exception	description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when it fails in a dynamic cast
bad_exception	thrown by certain dynamic exception specifiers
bad_typeid	thrown by typeid
bad_function_call	thrown by empty function objects
bad_weak_ptr	thrown by shared_ptr when passed a bad weak_ptr

exception	description
logic_error	error related to the internal logic of the program
runtime_error	error detected during runtime

```
// Example: bad_alloc standard exception
#include <iostream>
#include <exception>
using namespace std;
int main () {
    try {
        int* myarray= new int[1000];
    } catch (exception& e) {
        cout << "Standard exception: " << e.what() << endl;
    }
    return 0;
}
```



Konsep Collection

IF2210 – Semester II 2022/2023

SAR

Collection

- › Di dunia nyata maupun dalam memrogram, seringkali kita berurusan dengan sekumpulan benda yang sejenis.
- › Contoh:
 - › Lapangan parkir berisi sekumpulan mobil
 - › Sekumpulan barang yang dibeli dari supermarket di dalam kantong belanja
 - › Menu di resto merupakan sekumpulan *tuple* {nama makanan, harga}
 - › Daftar semua mahasiswa yang mengambil mata kuliah IF2210
 - › Deretan kursi di ruang bioskop
 - › dll.
- › Dalam konteks pemrograman (khususnya OOP) kumpulan tersebut disebut dengan *collection*.

Jenis-jenis collection

- › Setiap *collection* memiliki karakteristik tertentu tergantung kegunaannya:
 - › *Ordered collection* memiliki suatu keterurutan
 - › Misalnya list/array yang keterurutannya ditentukan oleh indeks, e.g., ada elemen ke-0, elemen ke-1, dst
 - › *Sorted collection* seperti *ordered collection*, dengan keterurutan yang merupakan hasil suatu kalkulasi, i.e., terurut berdasarkan value
 - › *Set* elemennya tidak boleh berulang
 - › *Dictionary/Map* merupakan sekumpulan *value* yang diakses melalui *key*
 - › *Bag*, *collection* yang berisi daftar item dan berapa kali item tersebut muncul, seperti kantong belanja yang (misal) berisi 12 telur, 2 susu, dan 3 kentang → map<item,int>
 - › *String* dapat dipandang sebagai sebuah *ordered collection of chars*
 - › Terkadang dibutuhkan *collection* yang jumlah elemennya tetap (misal: pemain basket dalam satu tim yang sedang bermain), terkadang jumlah elemen harus fleksibel

Apa yang bisa dilakukan dengan collection? (1)

- › **Akses:** kapasitas & jumlah elemen yg terisi (umum), mengakses sebuah elemen dalam *collection* (tergantung jenis *collection*)
- › **Enumerasi:**
 - › **For each:** melakukan hal yang sama kepada setiap elemen
 - › **Select/filter:** menghasilkan sub-collection berdasarkan suatu kriteria (misal: mahasiswa yang IPK-nya > 3)
 - › **Find/detect:** ambil satu saja dari hasil select
 - › **Collect/map:** menghasilkan *collection* baru dengan jumlah elemen yang sama, setiap elemen yang baru merupakan hasil transformasi/operasi terhadap elemen yang lama (contoh: perkalian skalar dengan vektor adalah sebuah collect/map, di mana $3 * [1, 2, 3] = [3*1, 3*2, 3*3]$)
 - › **Reduce, fold:** menghasilkan satu nilai dari semua elemen (misal: sum, average, max)

Apa yang bisa dilakukan dengan collection? (2)

- › **Testing:** memeriksa apakah elemen-elemen *collection* memenuhi suatu kondisi tertentu
 - › Is empty, is full
 - › Any: apakah **ada** elemen yang memenuhi suatu kondisi
 - › All: apakah **semua** elemen memenuhi suatu kondisi
- › **Menambah & mengurangi elemen:**
 - › Menambah & mengurangi 1 elemen
 - › Menambah semua elemen dari sebuah *collection* ke *collection* lain
 - › Mengurangi elemen yang memenuhi kriteria
 - › *Collection* dapat memiliki batasan spesifik dalam penambahan & pengurangan elemen, e.g., Queue dapat dipandang sebagai *collection* (sekumpulan objek) yang hanya bisa ditambah di belakang dan dikurangi di depan

Collection dalam OOP

- › Bahasa OO biasanya menyediakan kelas-kelas untuk menangani *collection*, baik *built-in* maupun dalam bentuk *library* tambahan
 - › Java: Collection API or Collection Framework (JCF), C++: STL, dst.
- › Dalam pemrograman prosedural, aksi terhadap *collection* (seperti pada slide sebelumnya) dilakukan dengan iterasi/loop terhadap semua elemen
 - › jika dilakukan di OOP, itu artinya meng-expose isi dan cara kerja internal *collection* → menyalahi enkapsulasi
- › Pada OOP, *collection* adalah objek juga, bisa dikirim message seperti contoh pada slide berikutnya
- › Iterasi/loop seharusnya hanya boleh terjadi **di dalam** kelas-kelas *collection*; kelas yang memanfaatkan *collection* **hanya menggunakan method** yang di-expose kelas-kelas *collection*
 - › Terkadang tidak dimungkinkan karena keterbatasan desain bahasa pemrograman yang digunakan

Contoh (1) dalam *Java-like syntax*

- › **For-each:** cetak semua isi array:

- › Prosedural:

```
for (i=0; i<10; i++) print(array[i]);
```

- › OOP:

```
array.forEach(elmt -> print(elmt));
```

- › **Select/filter:** isi arr2 dengan elemen arr1 yang genap saja:

- › Prosedural:

```
j = 0;  
for (i=0; i<10; i++)  
    if (arr1[i]%2 == 0)  
        arr2[j++] = arr1[i];
```

- › OOP:

```
arr2 = arr1.select(elmt -> elmt%2 == 0);
```

Perhatikan dengan OO untuk banyak hal jadi tidak perlu peduli jumlah elemen.

Contoh (2) dalam Java-like syntax

- › Collect/map: perkalian skalar dengan vektor, $\mathbf{v}_2 = 3 \cdot \mathbf{v}_1$:

- › Prosedural:

```
for (i=0; i<3; i++)
    v2[i] = 3 * v1[i];
```

- › OOP:

```
v2 = v1.map(component -> 3*component);
```

- › Testing: “Boleh masuk jika ada anggota rombongan yang dewasa”:

- › Prosedural:

```
bolehMasuk = false;
for (i=0; i<10; i++)
    if (rombongan[i].usia > 17)
        { bolehMasuk = true; break; }
```

- › OOP:

```
bolehMasuk = rombongan.any(anggota -> anggota.usia > 17);
```

Takeaways

- › Dalam desain kelas/objek akan ditemukan banyak sekali kelas/objek yang naturnya bersifat sebagai *collection*
- › *Levels of abstraction* di OOP, pada tingkat tinggi biasanya tidak perlu peduli jumlah elemen sebuah *collection* maupun tiap elemen secara individu
 - › Kode menjadi lebih *readable*
- › Tidak semua bahasa OO menyediakan kelas-kelas *collection* yang “true OO” secara *built-in*
 - › Contoh: Collection API di Java tidak “true OO”, bisa gunakan library seperti Eclipse Collections

```
boolean anyPeopleHaveCats =  
    this.people  
        .anySatisfy(person -> person.hasPet(PetType.CAT));  
  
int countPeopleWithCats =  
    this.people  
        .count(person -> person.hasPet(PetType.CAT));  
  
MutableList<Person> peopleWithCats =  
    this.people  
        .select(person -> person.hasPet(PetType.CAT));
```

<https://www.eclipse.org/collections/>



C++ Standard Template Library (STL)

IF2210 – Semester II 2022/2023

Motivasi

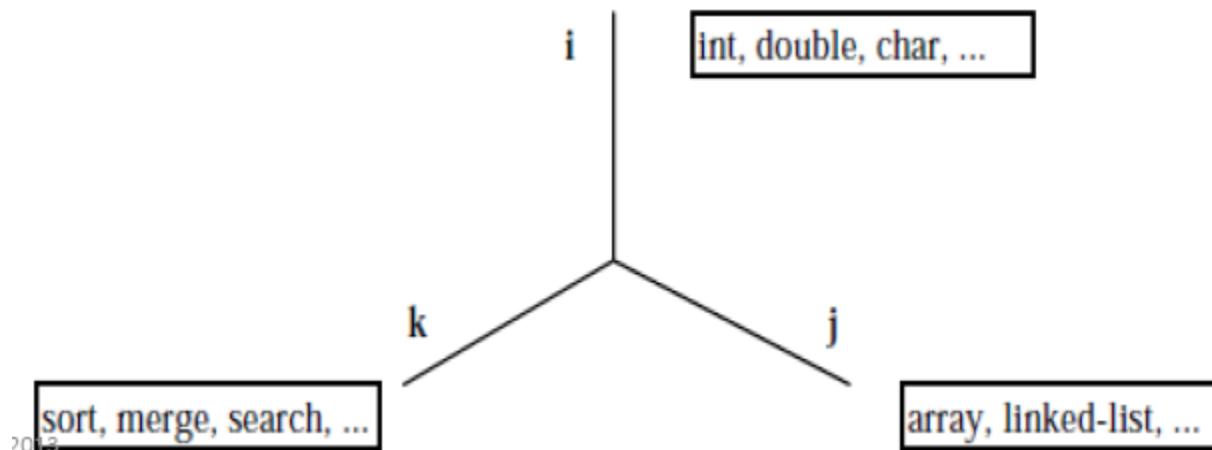
- › Alexander Stepanov (1970an):
 - › “*some algorithms do not depend on some particular implementation of a data structure, but only a few fundamental semantic properties of the structure*”
 - › “*fundamental semantic properties of the structure*”: e.g. how to get one element, how to get the next, how to step through the beginning to the end, ...

Sejarah

- › 1985: Stepanov built generic Ada library, and was asked if he could do in C++ as well
- › 1987: Template has yet implemented in C++
- › 1988: Stepanov moved to the HP Labs
- › 1992: Stepanov was appointed as manager of an algorithm projects:
 - › He and Meng Lee wrote STL, to build algorithms defined as generically as possible without losing efficiency

Intro to STL

- › STL (Standard Template Library) is a component library, described in a clean and formally sound concepts
- › The idea is “the orthogonal decomposition of the component space”



STL Components

- › Containers
 - › Template of data structures
 - › Object that can keep and administer objects
- › Iterators
 - › Like pointers, access elements of containers
- › Algorithms
 - › Computational processor that can work on different containers

Containers

Intro to Containers

- › Tiga jenis *container*:
 - › Sequence container
 - › Struktur data linear (*vector*, *linked list*)
 - › First-class container
 - › Associative container
 - › Tidak linear, pencarian elemen lebih cepat
 - › Pasangan *key/value*
 - › First-class container
 - › Container adapter
- › Near/partial container: mirip *container*, dengan fungsionalitas terbatas
- › Container memiliki fungsi-fungsi yang sama (common)

Kelas-kelas Container pada STL

- › *Sequence container:*
 - › `vector`, `deque`, `list`
- › *Associative container:*
 - › `set`, `multiset`, `map`, `multimap`
- › *Container adapter:*
 - › `stack`, `queue`, `priority_queue`

Fungsi anggota STL

- › Fungsi anggota untuk semua *container*:
 - › ctor, cctor, dtor
 - › `empty`, `max_size`, `size`, `= < <= > >= == !=`, `swap`
- › Fungsi untuk *first-class container*:
 - › `begin`, `end`
 - › `rbegin`, `rend`
 - › `erase`, `clear`

typedef umum di STL

- › **typedef untuk first-class container:**
 - › `value_type`
 - › `reference`
 - › `const_reference`
 - › `pointer`
 - › `iterator`
 - › `const_iterator`
 - › `reverse_iterator`
 - › `const_reverse_iterator`
 - › `difference_type`
 - › `size_type`

Iterators

Intro to Iterators (1)

- › *Iterator* mirip dengan *pointer*
 - › Menunjuk ke elemen pertama sebuah *container*
 - › Operator untuk iterator sama pada semua *container*:
 - › * *dereference*
 - › ++ menunjuk ke elemen berikutnya
 - › `begin()` mengembalikan *iterator* ke elemen pertama
 - › `end()` mengembalikan *iterator* ke elemen terakhir
 - › Iterator digunakan terhadap sekuens (rentang/ranges):
 - › *Containers*
 - › *Input sequences*: `istream_iterator`
 - › *Output sequences*: `ostream_iterator`

Intro to Iterators (2)

› Penggunaan:

- › `std::istream_iterator <int> inputInt(cin)`
 - › Membaca input dari `cin`
 - › `*inputInt`: *dereference* ke `int` pertama dari `cin`
 - › `++inputInt`: pindah ke `int` berikutnya pada *stream*
- › `std::ostream_iterator <int> outputInt(cout)`
 - › Menulis `int` ke `cout`
 - › `*outputInt = 7`: menulis 7 ke `cout`
 - › `++outputInt`: memajukan *iterator* supaya dapat menulis `int` berikutnya

```

1 // Fig. 21.5: fig21_05.cpp
2 // Demonstrating input and output with iterators.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13     cout << "Enter two integers: ";
14
15     // create istream_iterator for reading int values from cin
16     std::istream_iterator< int > inputInt( cin );
17
18     int number1 = *inputInt;    // read int from standard input
19     ++inputInt;               // move iterator to next input value
20     int number2 = *inputInt;   // read int from standard input
21

```

Note creation of **istream_iterator**. For compilation reasons, we use **std::** rather than a **using** statement.

Access and assign the iterator like a pointer.



Outline

fig21_05.cpp
(1 of 2)

```
22 // create ostream_iterator for writing int values to cout
23 std::ostream_iterator< int > outputInt( cout );
24
25 cout << "The sum is: ";
26 *outputInt = number1 + number2; // output result to cout
27 cout << endl;
28
29 return 0;
30
31 } // end main
```

Enter two integers: 12 25
The sum is: 37

Create an
ostream_iterator is
similar. Assigning to this
iterator outputs to **cout**.



Outline

fig21_05.cpp
(2 of 2)

fig21_05.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Jenis-jenis iterator

- › **Input:** membaca elemen dari *container*, hanya bisa maju
- › **Output:** menulis elemen ke *container*, hanya bisa maju
- › **Forward:**
 - › gabungan *input* dan *output*, mempertahankan posisi
 - › *multi-pass* (dapat melewati sekuens dua kali)
- › **Bidirectional:** seperti *forward*, tapi bisa mundur juga
- › **Random access:** seperti *bidirectional*, tapi bisa lompat ke elemen manapun

Jenis iterator yang didukung container

- › *Sequence container*
 - › vector, deque: random access
 - › list: bidirectional
- › *Associative container*
 - › set, multiset, map, multimap: bidirectional
- › *Container adapter*
 - › stack, queue, priority_queue: tidak mendukung iterator

Operasi pada iterator (1)

- › Semua:
 - › `++p, p++`
- › Iterator input:
 - › `*p`
 - › `p = p1`
 - › `p == p1, p != p1`
- › Iterator output:
 - › `*p`
 - › `p = p1`
- › Iterator forward:
 - › Memiliki fungsionalitas iterator input dan output

Operasi pada iterator (2)

- › Iterator bidirectional:
 - › `--p, p--`
- › Iterator random access:
 - › `p+i, p+=i`
 - › `p-i, p-=i`
 - › `p[i]`
 - › `p<p1, p<=p1`
 - › `p>p1, p>=p1`

Algorithms

Intro to Algorithms

- › STL memiliki algoritma-algoritma yang digunakan secara generik untuk setiap *container*
 - › Beroperasi terhadap elemen (secara tidak langsung, melalui iterator)
 - › Beroperasi pada sekuens elemen
 - › Didefinisikan oleh pasangan iterator (elemen pertama dan terakhir)
 - › Algoritma biasanya mengembalikan iterator
 - › Contoh: `find()` mengembalikan iterator yg menunjuk ke elemen yang dicari, atau mengembalikan `end()` jika tidak ketemu
 - › Algoritma premade menghemat waktu & usaha pemrogram

Algoritma

- › Sebelum STL
 - › *Library* kelas tidak saling kompatibel
 - › Algoritma tertanam di kelas-kelas container
- › STL memisahkan algoritma dari *container*
 - › Lebih mudah untuk menambah algoritma baru
 - › Lebih efisien, menghindari pemanggilan fungsi **virtual**
 - › <algorithm>

Algoritma dasar *searching & sorting*

- › `find(iter1, iter2, value)`: mengembalikan iterator ke kemunculan pertama `value` pada rentang `iter1` sampai sebelum `iter2`
- › `find_if(iter1, iter2, function)`: seperti `find`, tapi mengembalikan iterator ketika `function` mengembalikan `true`
- › `sort(iter1, iter2)`: mengurutkan elemen secara menaik (*ascending*)
- › `binary_search(iter1, iter2, value)`: mencari elemen pada sekuens yang terurut menaik, dengan algoritma pencarian biner

Outline

fig21_31.cpp
(1 of 4)

```
1 // Fig. 21.31: fig21_31.cpp
2 // Standard library search and sort algorithms.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // algorithm definitions
9 #include <vector>      // vector class-template definition
10
11 bool greater10( int value ); // prototype
12
13 int main()
14 {
15     const int SIZE = 10;
16     int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
17
18     std::vector< int > v( a, a + SIZE );
19     std::ostream_iterator< int > output( cout, " " );
20
21     cout << "Vector v contains: ";
22     std::copy( v.begin(), v.end(), output );
23
24     // locate first occurrence of 16 in v
25     std::vector< int >::iterator location;
26     location = std::find( v.begin(), v.end(), 16 );
```



Outline

fig21_31.cpp
(2 of 4)

```
27
28     if ( location != v.end() )
29         cout << "\n\nFound 16 at location "
30             << ( location - v.begin() );
31     else
32         cout << "\n\n16 not found";
33
34 // locate first occurrence of 100 in v
35 location = std::find( v.begin(), v.end(), 100 );
36
37     if ( location != v.end() )
38         cout << "\nFound 100 at location "
39             << ( location - v.begin() );
40     else
41         cout << "\n100 not found";
42
43 // locate first occurrence of value greater than 10 in v
44 location = std::find_if( v.begin(), v.end(), greater10 );
45
46     if ( location != v.end() )
47         cout << "\n\nThe first value greater than 10 is "
48             << *location << "\nfound at location "
49             << ( location - v.begin() );
50     else
51         cout << "\n\nNo values greater than 10 were found";
52
```

```
53 // sort elements of v
54 std::sort( v.begin(), v.end() );
55
56 cout << "\n\nVector v after sort: ";
57 std::copy( v.begin(), v.end(), output );
58
59 // use binary_search to locate 13 in v
60 if ( std::binary_search( v.begin(), v.end(), 13 ) )
61     cout << "\n\n13 was found in v";
62 else
63     cout << "\n\n13 was not found in v";
64
65 // use binary_search to locate 100 in v
66 if ( std::binary_search( v.begin(), v.end(), 100 ) )
67     cout << "\n\n100 was found in v";
68 else
69     cout << "\n\n100 was not found in v";
70
71 cout << endl;
72
73 return 0;
74
75 } // end main
76
```



Outline

fig21_31.cpp
(3 of 4)

```
77 // determine whether argument is greater than 10
78 bool greater10( int value )
79 {
80     return value > 10;
81
82 } // end function greater10
```



Outline

fig21_31.cpp
(4 of 4)

Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4
100 not found

The first value greater than 10 is 17
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

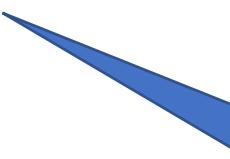
13 was found in v
100 was not found in v

fig21_31.cpp
output (1 of 1)

Contoh-contoh: vector dan stack

Sequence Container

- › Ada tiga sequence container
 - › **vector** – berbasis array
 - › **deque** – berbasis array
 - › **list** – *linked list* yang robust



We will only
discuss vector

vector (1)

- › **vector**
 - › **Header <vector>**
 - › Struktur data dengan lokasi memori kontigu
 - › Akses elemen dengan []
 - › Digunakan jika data harus diurutkan dan data harus mudah diakses
- › Ketika memori yang teralokasi penuh:
 - › Alokasikan area memori kontigu yang lebih besar
 - › Salin isi ke area memori baru tsb
 - › Dealokasi memori yang lama
- › Memiliki iterator *random access*

vector (2)

- › Deklarasi:

- › `std::vector <type> v;`
 - › `type`: int, float, etc.

- › Iterator:

- › `std::vector<type>::const_iterator iterVar;`
 - › tidak dapat memodifikasi elemen
- › `std::vector<type>::reverse_iterator iterVar;`
 - › Iterasi elemen dari belakang (mundur)
 - › Starting point: `rbegin`
 - › Ending point: `rend`

Fungsi-fungsi pada vector (1)

- › `v.push_back(value)`: menambah elemen di akhir *vector* (dimiliki oleh semua *container* sekuens)
- › `v.size()`: ukuran *vector* saat ini
- › `v.capacity()`: jumlah elemen yang dapat ditampung sebelum realokasi. Realokasi menggandakan ukuran
- › `vector<type> v(a, a+SIZE)`: membuat *vector* *v* dengan elemen dari array *a* sebanyak *SIZE*

Fungsi-fungsi pada vector (2)

- › `v.insert(iterator, value)`: menambahkan elemen `value` di depan lokasi `iterator`
- › `v.insert(iterator, array, array+SIZE)`: menambahkan elemen `array` sejumlah `SIZE` ke `v`
- › `v.erase(iterator)`: hapus elemen dari `container`
- › `v.erase(iter1, iter2)`: hapus elemen pada `iter1` hingga sebelum `iter2`
- › `v.clear()`: kosongkan `container`

Fungsi-fungsi pada vector (3)

- › `v.front()`, `v.back()`: mengembalikan elemen pertama dan terakhir
- › `v[elementNumber] = value;` : meng-assign `value` ke sebuah elemen
- › `v.at(elementNumber) = value;` : sama dengan sebelumnya, tapi dengan pemeriksaan indeks. Melempar exception `out_of_bounds`

Iterator ostream

- › `std::ostream_iterator <type> Name(outputStream, separator);`
 - › `type`: jenis tipe data yang dikeluarkan
 - › `outputStream`: iterator lokasi keluaran
 - › `separator`: karakter yang memisahkan keluaran
- › Contoh:
 - › `std::ostream_iterator <int> output(cout, "");`
 - › `std::copy(iter1, iter2, output);`
 - › Menyalin elemen dari posisi `iter1` sampai sebelum `iter2` ke `output`, sebuah `ostream_iterator`



Outline

fig21_14.cpp
(1 of 3)

```

1 // Fig. 21.14: fig21_14.cpp
2 // Demonstrating standard library vector class template.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <vector> // vector class-template definition
10
11 // prototype for function template printVector
12 template < class T >
13 void printVector( const std::vector< T > &integers2 );
14
15 int main()
16 {
17     const int SIZE = 6;
18     int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
19
20     std::vector< int > integers;
21
22     cout << "The initial size of integers is: "
23         << integers.size()
24         << "\nThe initial capacity of integers is: "
25         << integers.capacity();
26

```

Create a **vector** of **ints**.

Call member functions.

```
27 // function push_back is in every sequence collection
28 integers.push_back( 2 );
29 integers.push_back( 3 );
30 integers.push_back( 4 );
31
32 cout << "\nThe size of integers is: " << integers.size()
33     << "\nThe capacity of integers is: "
34     << integers.capacity();
35
36 cout << "\n\nOutput array using pointer notation: ";
37
38 for ( int *ptr = array; ptr != array + SIZE; ++ptr )
39     cout << *ptr << ' ';
40
41 cout << "\nOutput vector using iterator notation: ";
42 printVector( integers );
43
44 cout << "\nReversed contents of vector integers: ";
45
```

Add elements to end of
vector using **push_back**.

fig21_14.cpp
(2 of 3)



Outline

```

46     std::vector< int >::reverse_iterator reverseIterator;
47
48     for ( reverseIterator = integers.rbegin();
49           reverseIterator!= integers.rend();
50           ++reverseIterator )
51     cout << *reverseIterator << ' ';
52
53     cout << endl;
54
55     return 0;
56
57 } // end main
58
59 // function template for outputting vector elements
60 template < class T >
61 void printVector( const std::vector< T > &integers2 )
62 {
63     std::vector< T >::const_iterator constIterator;
64
65     for ( constIterator = integers2.begin();
66           constIterator != integers2.end();
67           constIterator++ )
68     cout << *constIterator << ' ';
69
70 } // end function printVector

```

Walk through **vector** backwards using a **reverse_iterator**.

Template function to walk through **vector** forwards.

```
The initial size of v is: 0
The initial capacity of v is: 0
The size of v is: 3
The capacity of v is: 4

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2
```



Outline

fig21_14.cpp
output (1 of 1)



Outline

fig21_15.cpp
(1 of 3)

```

1 // Fig. 21.15: fig21_15.cpp
2 // Testing Standard Library vector class template
3 // element-manipulation functions.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 #include <vector>      // vector class-template definition
10 #include <algorithm> // copy algorithm
11
12 int main()
13 {
14     const int SIZE = 6;
15     int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
16
17     std::vector< int > integers( array, array + SIZE );
18     std::ostream_iterator< int > output( cout, " " );
19
20     cout << "Vector integers contains: ";
21     std::copy( integers.begin(), integers.end(), output );
22
23     cout << "\nFirst element of integers: " << integers.front()
24         << "\nLast element of integers: " << integers.back();
25

```

Create **vector** (initialized using an array) and **ostream_iterator**.

Copy range of iterators to **output (ostream_iterator)**.

```

26     integers[ 0 ] = 7;           // set first element to 7
27     integers.at( 2 ) = 10;      // set element at position 2 to 10
28
29     // insert 22 as 2nd element
30     integers.insert( integers.begin() + 1, 22 );
31
32     cout << "\n\nContents of vector integers after changes: ";
33     std::copy( integers.begin(), integers.end(), output );
34
35     // access out-of-range element
36     try {
37         integers.at( 100 ) = 777;
38
39     } // end try
40
41     // catch out_of_range exception
42     catch ( std::out_of_range outOfRange ) {
43         cout << "\n\nException: " << outOfRange.what();
44
45     } // end catch
46
47     // erase first element
48     integers.erase( integers.begin() );
49     cout << "\n\nVector integers after erasing first element: ";
50     std::copy( integers.begin(), integers.end(), output );
51

```



Outline

More **vector** member functions.

at has range checking, and can throw an exception.

```
52 // erase remaining elements
53 integers.erase( integers.begin(), integers.end() );
54 cout << "\nAfter erasing all elements, vector integers "
55     << ( integers.empty() ? "is" : "is not" ) << " empty";
56
57 // insert elements from array
58 integers.insert( integers.begin(), array, array + SIZE );
59 cout << "\n\nContents of vector integers before clear: ";
60 std::copy( integers.begin(), integers.end(), output );
61
62 // empty integers; clear calls erase to empty a collection
63 integers.clear();
64 cout << "\nAfter clear, vector integers "
65     << ( integers.empty() ? "is" : "is not" ) << " empty";
66
67 cout << endl;
68
69 return 0;
70
71 } // end main
```



Outline

fig21_15.cpp
(3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
Vector integers contains: 1 2 3 4 5 6
```

```
First element of integers: 1
```

```
Last element of integers: 6
```

```
Contents of vector integers after changes: 7 22 2 10 4 5 6
```

```
Exception: invalid vector<T> subscript
```

```
Vector integers after erasing first element: 22 2 10 4 5 6
```

```
After erasing all elements, vector integers is empty
```

```
Contents of vector integers before clear: 1 2 3 4 5 6
```

```
After clear, vector integers is empty
```



Outline

fig21_15.cpp
output (1 of 1)

```

#include <iostream>
using namespace std;
#include <vector>
#include <stdio.h>

int main() {
    vector<int> v;
    vector<int>::const_iterator CI;
    vector<int>::reverse_iterator RI;

    /* push five elements into the vector */
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    v.push_back(5);
    cout << "Size of vector: " << v.size() << endl;
    cout << "Initial capacity vector: " << v.capacity() << endl;

    /* print the vector */
    for (CI = v.begin(); CI != v.end(); CI++) {
        cout << *CI << " ";
    }
    cout << endl;

    /* print the vector backward */
    for (RI = v.rbegin(); RI != v.rend(); RI++) {
        cout << *RI << " ";
    }

    return 0;
}

```



Container Adapter

- › Container adapter
 - › stack, queue, dan priority_queue
 - › Bukan *first-class container*
 - › Tidak mendukung iterator
 - › Tidak menyediakan struktur data yang sebenarnya
 - › Pemrogram dapat memilih implementasi yang diinginkan
 - › Fungsi anggota: push dan pop



We will
discuss only
stack

stack

- › stack
 - › Header <stack>
 - › Tambah & hapus data dari salah satu ujung saja: LIFO
 - › Secara internal dapat menggunakan vector, list, atau deque (*default*)
 - › Deklarasi:

```
stack <type, vector<type>> myStack;
stack <type, list<type>> myOtherStack;
stack <type> anotherStack; // default deque
```
 - › struktur data internal tidak mempengaruhi *behavior*, hanya kinerja (deque dan vector lebih cepat daripada list)



Outline

fig21_23.cpp
(1 of 3)

```
1 // Fig. 21.23: fig21_23.cpp
2 // Standard library adapter stack test program.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <stack>    // stack adapter definition
9 #include <vector>   // vector class-template definition
10 #include <list>     // list class-template definition
11
12 // popElements function-template prototype
13 template< class T >
14 void popElements( T &stackRef );
15
16 int main()
17 {
18     // stack with default underlying deque
19     std::stack< int > intDequeStack;
20
21     // stack with underlying vector
22     std::stack< int, std::vector< int > > intVectorStack;
23
24     // stack with underlying list
25     std::stack< int, std::list< int > > intListStack;
```

Create stacks with various implementations.

```
27 // push the values 0-9 onto each stack
28 for ( int i = 0; i < 10; ++i ) {
29     intDequeStack.push( i );
30     intVectorStack.push( i ); ←
31     intListStack.push( i );
32
33 } // end for
34
35 // display and remove elements from each stack
36 cout << "Popping from intDequeStack: ";
37 popElements( intDequeStack );
38 cout << "\nPopping from intVectorStack: ";
39 popElements( intVectorStack );
40 cout << "\nPopping from intListStack: ";
41 popElements( intListStack );
42
43 cout << endl;
44
45 return 0;
46
47 } // end main
48
```



Outline

Use member function **push**. 21_23.cpp
(2 of 3)

```
49 // pop elements from stack object to which stackRef refers
50 template< class T >
51 void popElements( T &stackRef )
52 {
53     while ( !stackRef.empty() ) {
54         cout << stackRef.top() << ' '; // view top element
55         stackRef.pop(); // remove top element
56     } // end while
57 }
58 // end function popElements
```

```
Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```



Outline

[fig21_23.cpp](#)
(3 of 3)

[fig21_23.cpp](#)
output (1 of 1)

```
#include <iostream>
using namespace std;
#include <stack>
#include <stdio.h>

int main() {
    stack<int> st;
    /* push three elements into the stack */
    st.push(1);
    st.push(2);
    st.push(3);
    /* pop & print 2 elements from the stack */
    cout << st.top() << " ";
    st.pop();
    cout << st.top() << " ";
    st.pop();
    /* modify top element */
    st.top() = 77;
    /* push two new elements */
    st.push(4);
    st.push(5);
    /* pop 1 element without processing it */
    st.pop();
    /* pop and print remaining elements */
    while (!st.empty()) {
        cout << st.top() << " ";
        st.pop();
    }
    cout << endl;
    return 0;
}
```

References

- › H.M. Deitel, P.J. Deitel: “*How to Program in C++*”, Prentice Hall (...)
- › <http://en.cppreference.com/w/cpp/container>



Object-Oriented Design

IF2210 – Pemrograman Berorientasi Objek

Semester II – 2021/2022

Tim Dosen

Bahan Bacaan: URL dan juga tautan di dalamnya

- › <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOOD>
- › [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))
- › <http://www.ecs.syr.edu/faculty/fawcett/handouts/cse687/Presentations/principles.pdf>
- › <http://www.odesign.com/design-principles.html>
- › <https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/>

Bad design

- › What makes a design bad? Robert Martin suggests:
 - › **Rigidity:** It is hard to change because every change affects too many other parts of the system.
 - › **Fragility:** When you make a change, unexpected parts of the system break.
 - › **Immobility:** It is hard to reuse in another application because it cannot be disentangled from the current application.
- › The design principles discussed in the following are all aimed at preventing “bad” design.



S.O.L.I.D. Principles

- › Single responsibility principle
- › Open closed principle
- › Liskov substitution principle
- › Interface segregation principle
- › Dependency inversion principle

Single Responsibility Principle

“A class should have one, and only one, reason to change.”

i.e.

“A class should have only one job.”



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Open Closed Principle

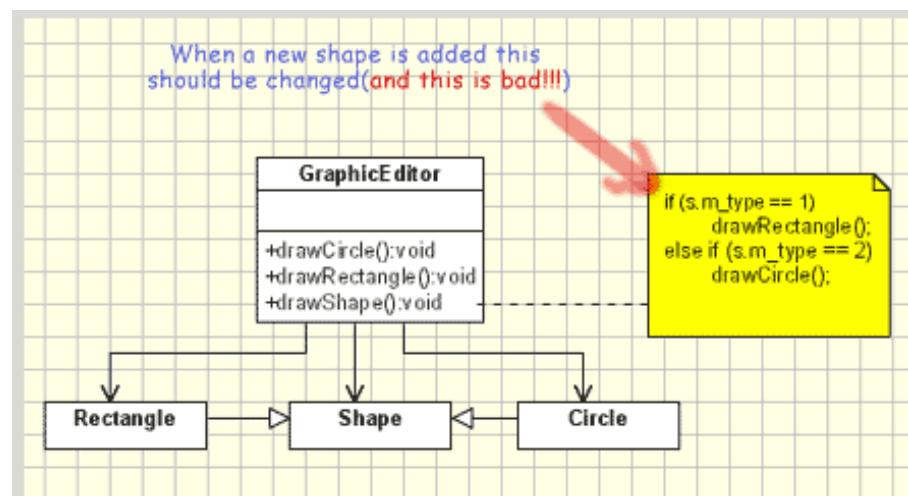
*“Objects or entities should be open for extension,
but closed for modification.”*



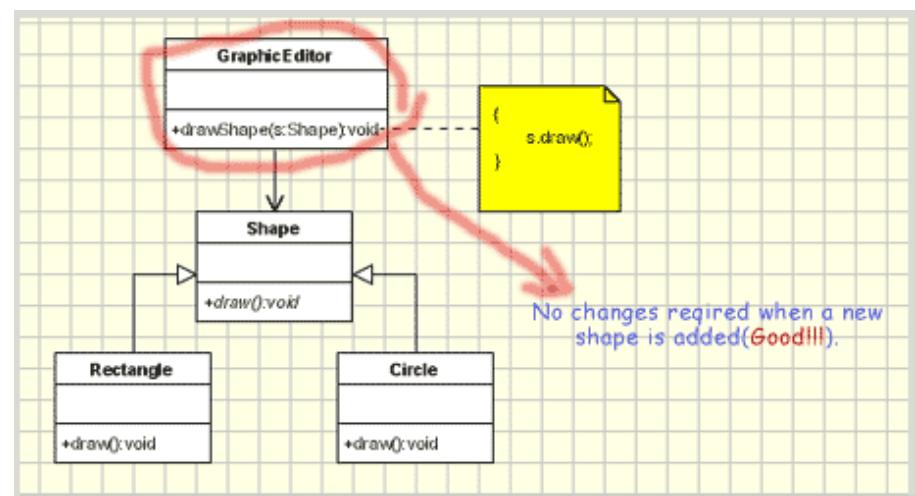
OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

✗ Bad Example



✓ Good Example

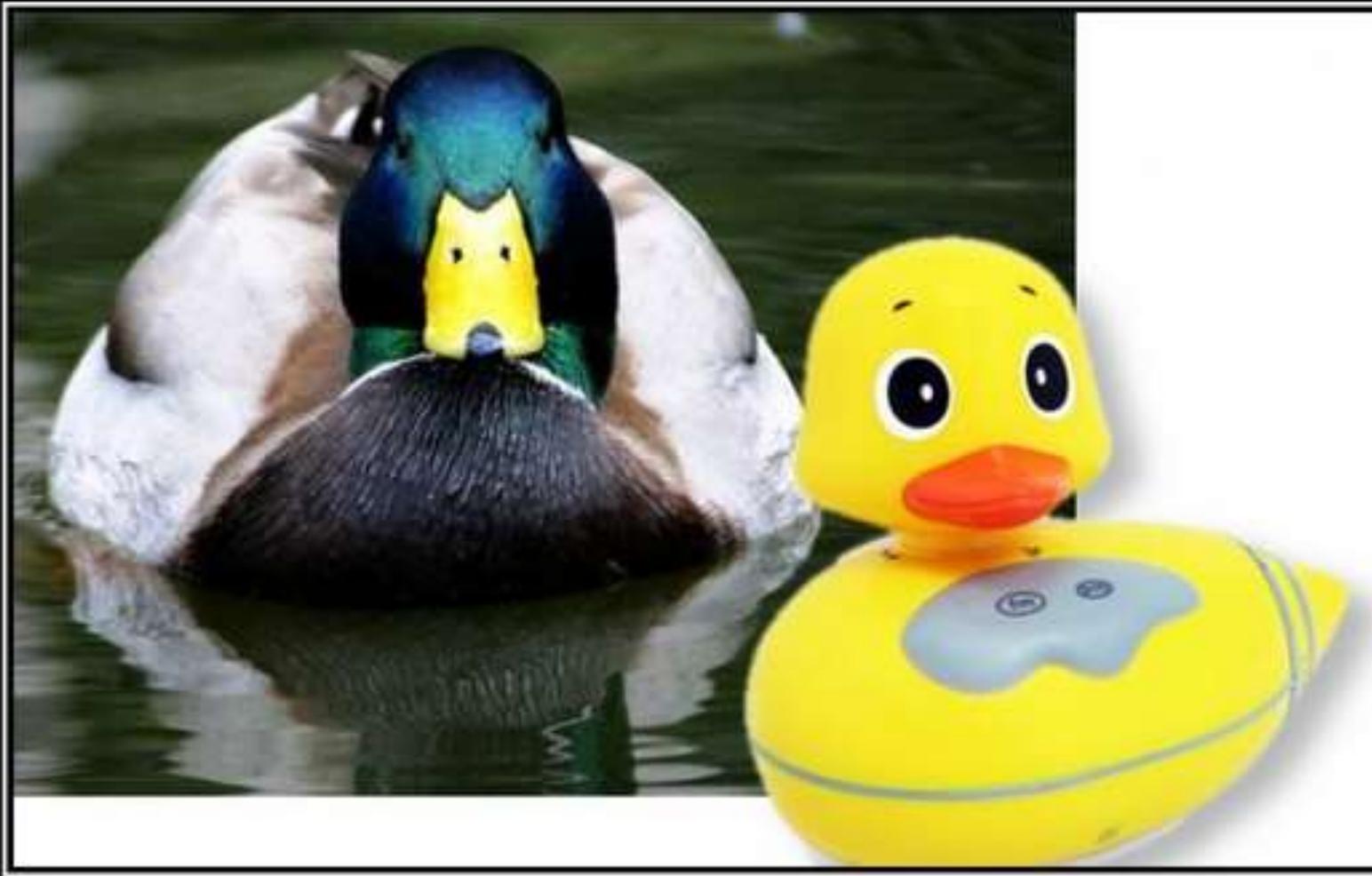


Liskov Substitution Principle

“Let $q(x)$ be a property provable about objects of x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .”

i.e.

“Every subclass should be substitutable for their base class.”



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You
Probably Have The Wrong Abstraction

- › A great example illustrating LSP (given by Uncle Bob in a podcast I heard recently) was how sometimes something that sounds right in natural language doesn't quite work in code.
- › In mathematics, a **Square** is a **Rectangle**. Indeed it is a specialization of a rectangle. The "is a" makes you want to model this with inheritance. However if in code you made **Square** derive from **Rectangle**, then a **Square** should be usable anywhere you expect a **Rectangle**. This makes for some strange behavior.
- › Imagine you had **SetWidth()** and **SetHeight()** methods on your **Rectangle** base class; this seems perfectly logical. However if your **Rectangle** reference pointed to a **Square**, then **SetWidth()** and **SetHeight()** doesn't make sense because setting one would change the other to match it. In this case **Square** fails the Liskov Substitution Test with **Rectangle** and the abstraction of having **Square** inherit from **Rectangle** is a bad one.

Interface Segregation Principle

“A client should never be forced to implement an interface that it doesn't use.”

Or,

“Clients shouldn't be forced to depend on methods they do not use.”

i.e.

“Make fine grained interfaces that are client specific.”



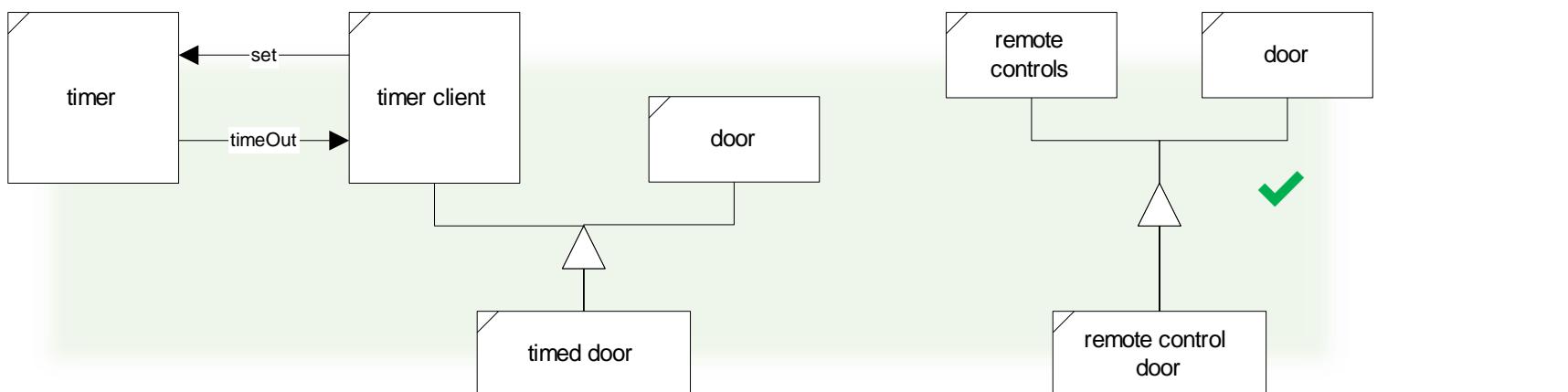
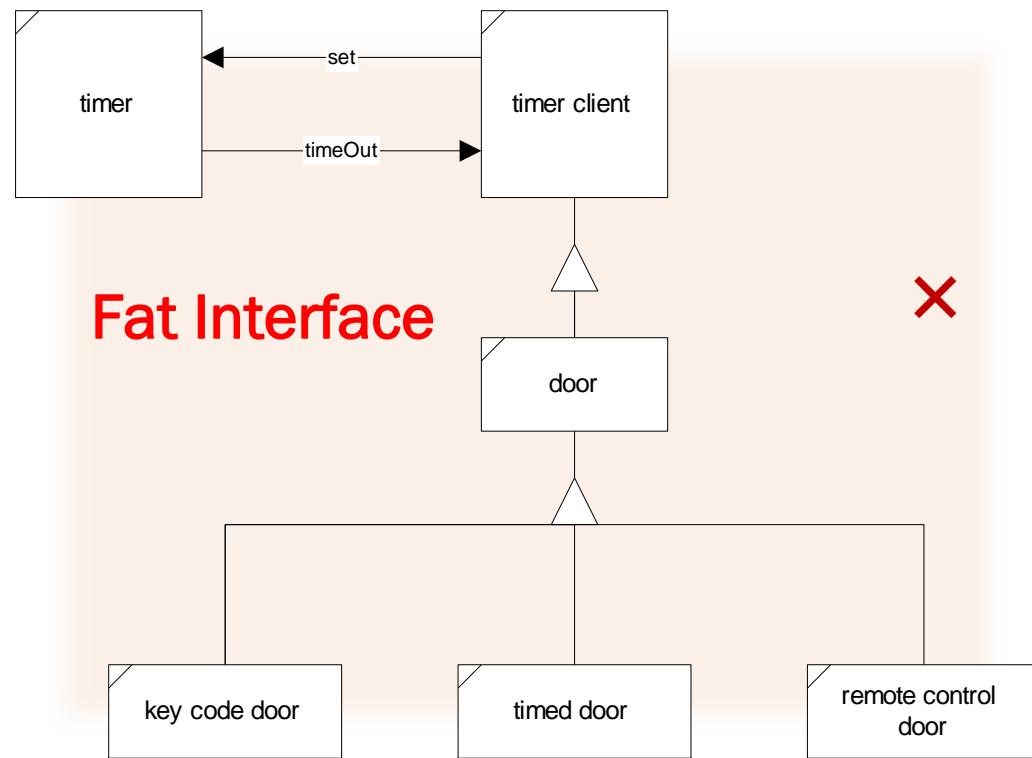
INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

The Interface Segregation Principle

states that:

- fat interfaces lead to inadvertent couplings between clients that ought to be isolated
- fat interfaces can be segregated, through multiple inheritance, into abstract base classes that break unwanted coupling between components.
- clients simply mix-in the appropriate interfaces for their activities.



Dependency Inversion Principle

“Entities must depend on abstractions not on concretions.”

i.e.

“The high level module must not depend on the low level module, but they should depend on its abstractions.”

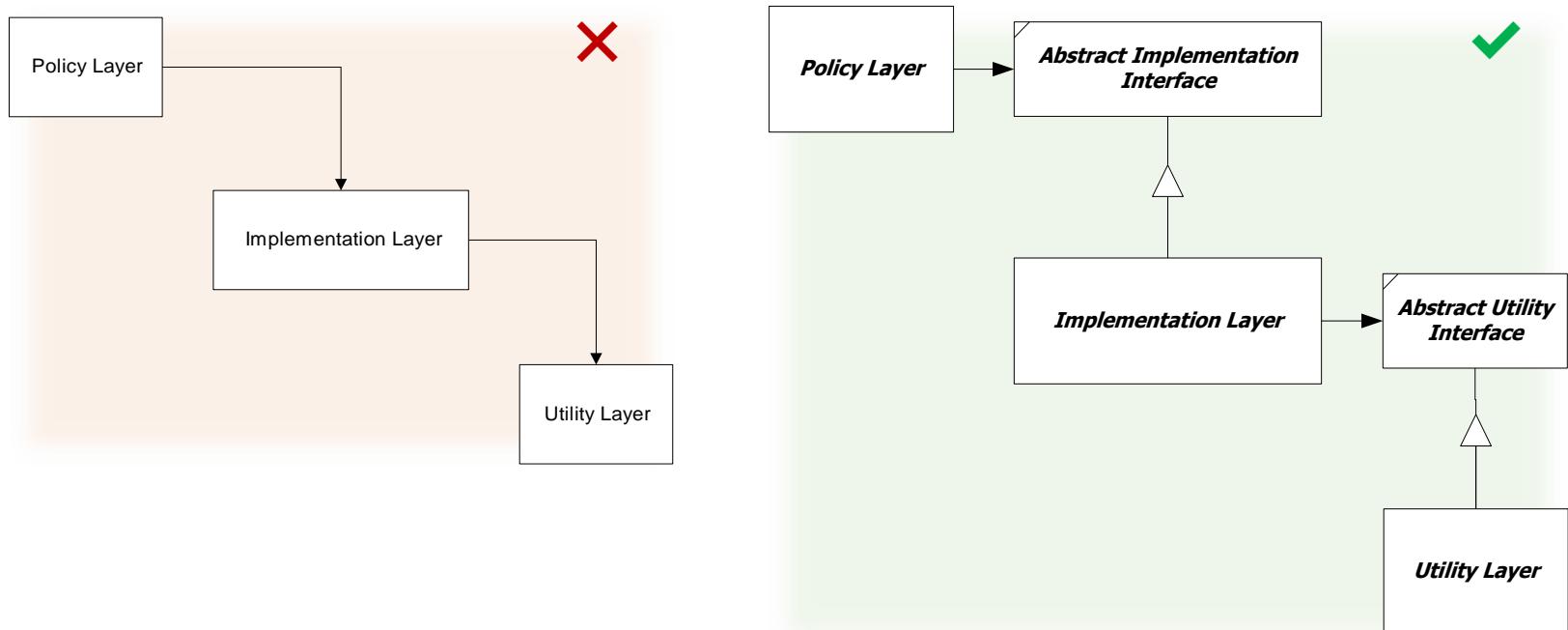


DEPENDENCY INVERSION PRINCIPLE

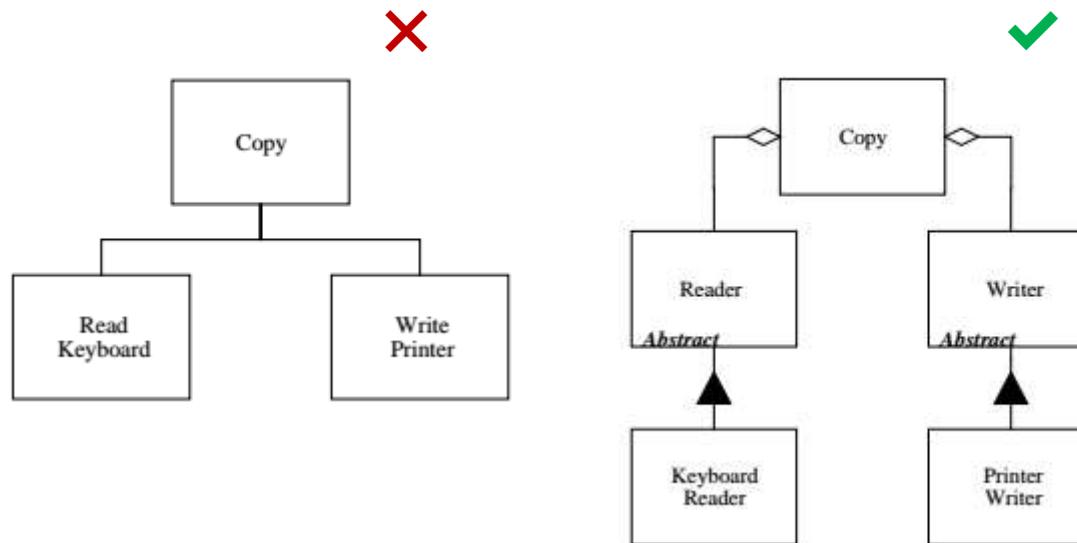
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

› Dependency Inversion Principle:

- › High level components should not depend upon low level components. Instead, both should depend on abstractions.
- › Abstractions should not depend upon details. Details should depend upon the abstractions.



- › We all can agree that complex systems need to be structured into layers. But if that is not done carefully the top levels tend to depend on the lower levels.
- › On the next page we show a “standard” architecture that appears to be practical and useful.
- › Unfortunately, it has the ugly property that policy layer depends on implementation layer which depends on utility layer, e.g., dependencies all the way down.



```
// Dependency Inversion Principle -  
// Bad example  
class Worker {  
    public void work() { /* some work */ }  
}  
  
class Manager {  
    Worker worker;  
  
    public void setWorker(Worker w) { worker = w; }  
  
    public void manage() { worker.work(); }  
}  
  
class SuperWorker {  
    public void work() { /* some heavy work */ }  
}
```

```
// Dependency Inversion Principle -  
// Good example  
interface IWorker {  
    public void work();  
}  
  
class Worker implements IWorker {  
    public void work() { /* some work */ }  
}  
  
class SuperWorker implements IWorker {  
    public void work() { /* some heavy work */ }  
}  
  
class Manager {  
    IWorker worker;  
  
    public void setWorker(IWorker w) { worker = w; }  
  
    public void manage() { worker.work(); }  
}
```

Partitioning into Packages

- › As software becomes large and complex we need to enforce some form of partitioning that is larger than the class and smaller than a program.
- › Packages represent a grouping of classes into a cohesive structure that represents a single high-level abstraction.
- › Packages allow us to reason about and reuse software on a large scale without being swamped with detail.

Package Principles (1)

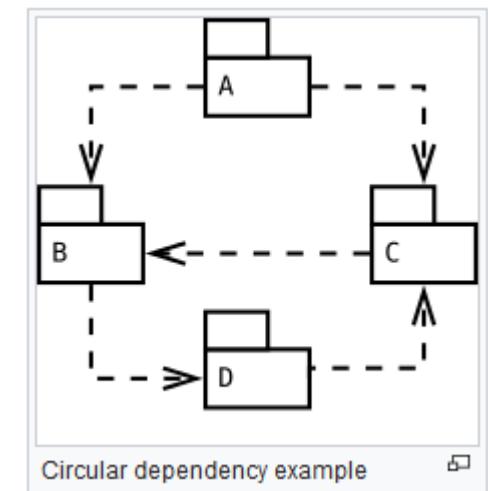
- › The first three package principles are about package cohesion, they tell us what to put inside packages:
 - › REP ([The Release Reuse Equivalency Principle](#)): *The granule of reuse is the granule of release.*
 - › CCP ([The Common Closure Principle](#)): *Classes that change together are packaged together.*
 - › CRP ([The Common Reuse Principle](#)): *Classes that are used together are packaged together.*



Package Principles (2)

Coupling between packages:

- › ADP (The Acyclic Dependencies Principle): *The dependency graph of packages must have no cycles.*
- › SDP (The Stable Dependencies Principle):
Depend in the direction of stability.
- › SAP (The Stable Abstractions Principle):
Abstractness increases with stability.



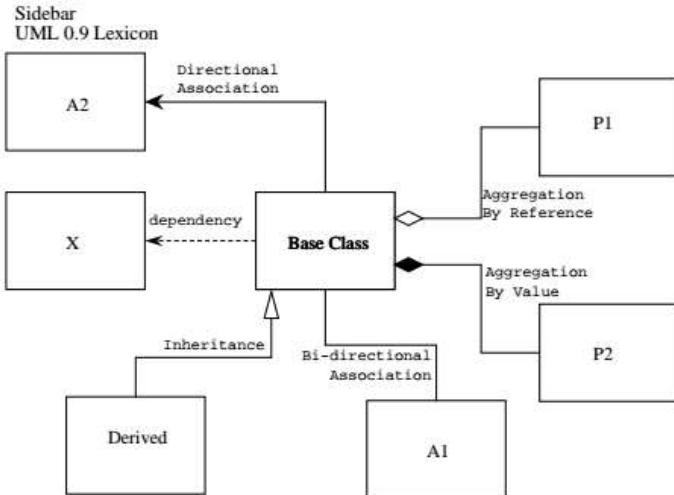


Figure 3
Package Diagram with Cycles

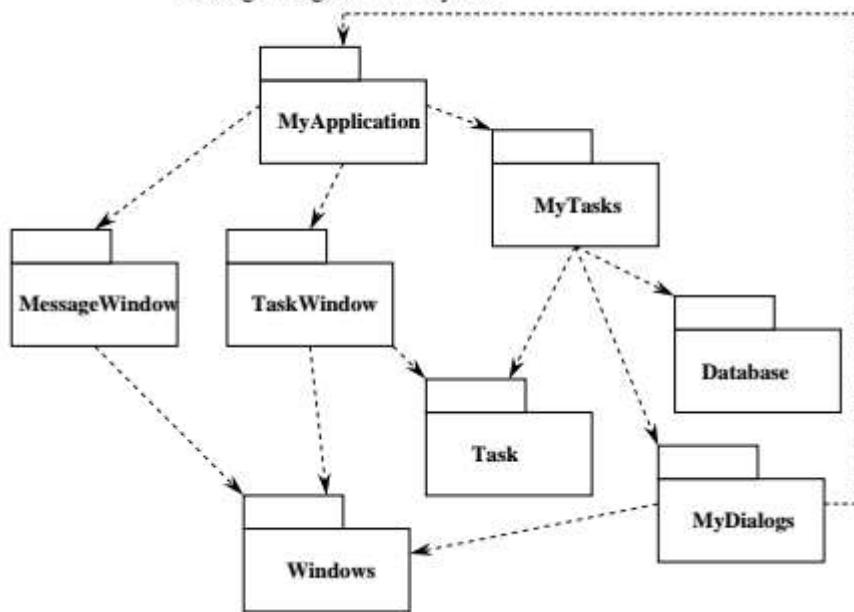
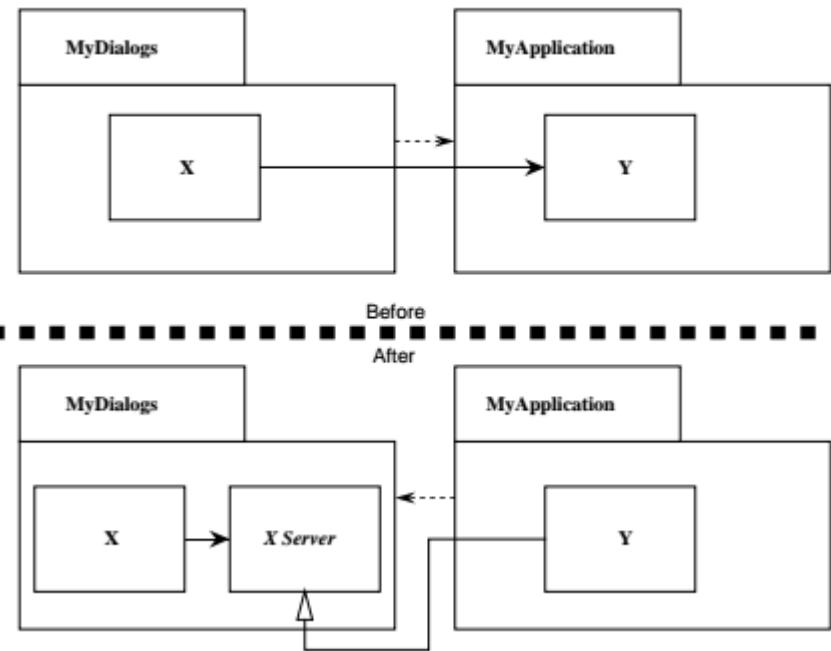


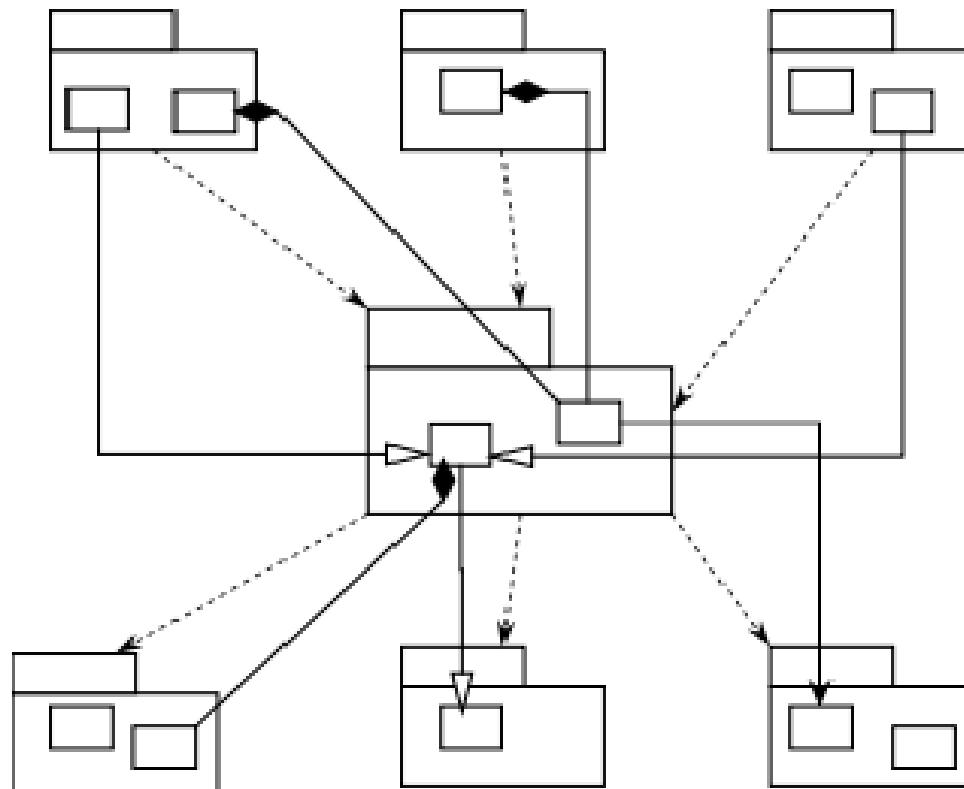
Figure 4
Breaking the Cycle with Dependency Inversion



Measuring Dependency: Positional Stability

- › Positional stability is based on the number of dependencies that enter and leave a package:
 - › **Afferent Couplings:**
 C_a = number of classes outside package depending on classes inside package
 - › **Efferent Couplings:**
 C_e = number of classes inside package that depend on outside classes
 - › **Instability:**
 $I = C_e / (C_a + C_e)$
 $I \in [0,1]$, $I=0 \Rightarrow$ maximum stability, $I=1 \Rightarrow$ minimum stability
- › If we are careful only to `#include` files that we depend on and we isolate one class per file, then we can compute I by counting `#includes`.

Measuring Dependency (2)



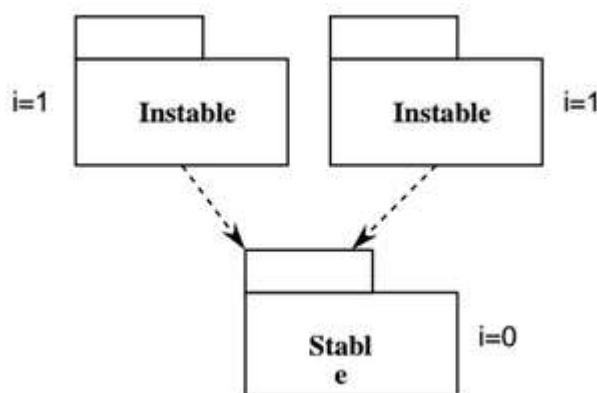
$$Ca=4$$

$$Ce=3$$

$$I=3/7$$

Not all packages should be stable

- › If all the packages in a system were maximally stable, the system would be unchangeable.
 - › This is not a desirable situation.
- › We'd want to design our package structure so that some packages are instable and some are stable, e.g.:



*"The changeable packages are on top and depend on the stable package at the bottom. Putting the *instable* packages at the top of the diagram is my own convention. By arranging them this way then any arrow that puts up is **violating the SDP.**"*

Measuring Abstraction: Stable Abstractions Principle

- › A measure of abstraction is the ratio of the number of abstract classes to total number of classes:

$$A = \text{number of abstract classes} / \text{total number of classes}$$

$$A \in [0,1].$$

$A = 1 \Rightarrow$ maximum abstraction, $A = 0 \Rightarrow$ minimum abstraction

- › The distance of a design from the balanced state is measured by:

$$D = A + I - 1$$

This is the perpendicular distance of the design from the balanced line on the following chart.

Measuring Abstraction (2)

