Bahasa C++

Hans Dulimarta

Daftar Isi

Daftar Tabel

Daftar Contoh

1 Latar Belakang C++

Bahasa C++ diciptakan oleh Bjarne Stroustrup di AT&T Bell Laboratories pada awal 1980an sebagai pengembangan dari bahasa C. Pada mulanya bahasa ini dikenal sebagai "C with Classes" (nama C++ digunakan sejak 1983, setelah diusulkan oleh Rick Mascitti). Pada tahun 1985 bahasa ini mulai disebarluaskan oleh AT&T dengan mengeluarkan perangkat lunak `cfront` yang berfungsi sebagai C++ *translator* (`cfront` menerima masukan program bahasa C++ dan menghasilkan kode bahasa C).

Perancangan bahasa C++ didasarkan pada bahasa C, Simula67, Algol68, dan Ada. Sebagai contoh, konsep "class" diambil dari bahasa Simula67, konsep *operator overloading* dan kemungkinan penempatan deklarasi di antara instruksi diambil dari bahasa Algol68, konsep *template* dan *exception* diambil dari bahasa Ada.

Bahasa C++ memperluas kemampuan bahasa C dalam beberapa hal yaitu: (1) memberikan dukungan untuk menciptakan dan memanfaatkan abstraksi data, (2) memberikan dukungan untuk *object-oriented programming*, dan (3) memperbaiki beberapa kemampuan yang sudah ada pada bahasa C.

Upaya pembakuan terhadap bahasa C++ sudah dilakukan sejak beberapa tahun terakhir dan pada bulan Oktober/November 1998, ANSI telah mengeluarkan standard untuk bahasa C++.

Beberapa buku yang dianggap sebagai acuan baku bagi penulis program C++ tercantum pada daftar pustaka.

2 Perbandingan C++ dengan C

2.1 Komentar

Komentar di dalam C++ dapat juga dituliskan setelah simbol `//`. Jika komentar yang dituliskan di antara `/*` dan `*/` bersifat *block-oriented*, komentar yang dituliskan setelah tanda `//` bersifat *line-oriented*.

2.2 Masukan dan Keluaran

Dalam bahasa C, penulis program biasanya menggunakan perintah `scanf() atau printf()` untuk keperluan operasi keluaran/masukan dari stream. Bahasa C++ memiliki operasi ma-

sukan/keluaran melalui objek `cin`, `cout`, dan `cerr` sebagai pasangan dari `stdin`, `stdout`, dan `stderr`.

Contoh 2.1 Stream dalam C dan C++

```
#include <stdio.h>                  #include <stream.h>

main() {                            main() {
   printf ("Hello world!\n");          cout << "Hello world!" << endl;
}                                   }
```

2.3 Deklarasi variable

Selain di awal blok, variable/objek dapat dideklarasikan/ didefinisikan di antara instruksi.[1] Dalam bahasa C, deklarasi variabel harus selalu dilakukan di luar atau di awal blok.

```
void main()
{
   int x = 1;   // contoh baris komentar pertama
   // contoh baris  komentar kedua

   printf("x = %d\n",x);

   float r;   // didefinisikan di antara instruksi

   r = 5.0;
}
```

2.4 Perubahan Tipe

Perubahan tipe (*typecasting*) dalam C++ dapat dipandang sebagai pemanggilan fungsi dengan nama tipe yang digunakan dalam *casting*.

```
   int a;
   float r = 2.5;

   a = (int) r;
   a = int (r);   // dalam \C++: dianggap fungsi dengan nama 'int'
```

[1] Dalam catatan kuliah ini, contoh-contoh program kadangkala dituliskan tanpa komentar seperti untuk menyatakan bagian `/* kamus */`, `/* algoritma */` atau keterangan lainnya, dengan maksud agar contoh dapat disajikan dalam bentuk yang singkat dan perhatian pembaca dapat tertuju pada hal yang sedang menjadi topik pembicaraan.

2.5 Reference

Dalam hal pengelolaan variabel dan parameter, C++ juga menyediakan *reference variable*, dan *call by reference*. Reference ke suatu variabel adalah nama alias terhadap variabel tersebut. Reference berbeda dengan pointer. Jika sudah digunakan untuk mengacu suatu objek/variabel, reference tidak dapat direset untuk mengacu objek/variabel lain. Fasilitas ini dapat dimanfaatkan untuk memberikan alias terhadap suatu variabel yang mempunyai nama yang panjang (misalnya karena berada dalam struktur yang berlapis-lapis).

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

Penggunaan reference yang lain adalah untuk *call by reference* dan *return value* dari sebuah fungsi. Dengan demikian, dalam bahasa C++ simbol `&` digunakan dengan dua makna, yaitu sebagai *address-of* dan *reference*[2]. Setiap pendefinisian variabel referensi harus selalu diinisialisasi oleh variabel lain. Dalam contoh di atas, variabel `xr` **tidak** berisi alamat dari `x`, seperti halnya pada

```
int *py;
int &yr; // error (tidak diinisialisasi)
int y;
py = &y; // py akan berisi alamat dari y
```

Pada fungsi `swap()` dalam Contoh 2.2, parameter formal dideklarasikan sebagai *reference*.

2.6 Function Overloading

Nama fungsi yang sama dapat dideklarasikan dengan *function signature* yang berbeda. Fasilitas ini sering disebut sebagai *function name overloading*. *Function signature* adalah jumlah dan tipe parameter formal sebuah fungsi. Contoh 2.2 menunjukkan fungsi `swap()` yang mengalami *overloading*.

Dalam C++, pemanggilan fungsi tidak hanya ditentukan oleh nama fungsi, tetapi juga oleh jenis dan banyaknya parameter aktual. Fasilitas yang berkaitan dengan fungsi yang ada di C++ lainnya adalah *template function*, *operator function*, *inline function*.

2.7 Nilai *default* parameter formal

Dalam C++ parameter formal dapat diberi nilai default. Dalam Contoh 2.3 ditunjukkan fungsi `MoveWindow()` yang memiliki 3 parameter formal, dua di antaranya diberi nilai *default*. Pemanggilan `MoveWindow()` (seperti pada baris 13 dan 14) dapat dilakukan dengan memberikan satu, dua, atau tiga parameter aktual.

[2]Perhatikanlah bahwa keduanya dituliskan dengan cara yang berbeda

Contoh 2.2 *Function overloading*

```
1   void swap(int &x, int &y) { /* swap integer */
2      int tmp;
3
4      tmp = x;
5      x = y;
6      y = tmp;
7   }
8
9   void swap(float &x, float &y) {   /* swap float */
10     float tmp;
11
12     tmp = x;
13     x = y;
14     y = tmp;
15  }
16
17  void main() {
18     int x=5, y=10;
19     float v=5.3, w=4.2;
20
21     swap(x,y);   // otomatis memanggil swap integer
22     swap(v,w);   // memanggil swap float
23  }
```

Contoh 2.3 Fungsi dengan parameter formal *default*

```
1   // prototype dari MoveWindow
2   void MoveWindow(int, int = 10, int = 5);
3
4   // pedefinisian dilakukan tanpa nilai inisialisasi
5   // karena sudah dituliskan pada prototype/deklarasi
6   void MoveWindow (int wid, int dx, int dy) {
7      // ...
8   }
9
10  main() {
11     int id;
12
13     MoveWindow (id);      // identik dengan MoveWindow (id, 10, 5);
14     MoveWindow (id,18);   // identik dengan MoveWindow (id, 18, 5);
15  }
```

2.8 Operator-operator Baru

C++ juga mendefinisikan beberapa operator baru seperti global scope (unary `::`), class scope (binary `::`), `new`, `delete`, member pointer selectors (`->*`, `.*`) dan kata kunci baru seperti: `class`, `private`, `operator`, dsb.

Operator scope digunakan untuk menegaskan ruang lingkup dari sebuah nama. Pada Contoh 2.4 ditunjukkan beberapa cara menggunakan operator ini di dalam fungsi `Print()`.

Contoh 2.4 Pemanfaatan scope

```
1   int x;
2   class List
3   {
4       int x;
5       List *next;
6     public:
7       void Print()
8       {
9          int x;
10
11         x = 5;          // local (baris 9)
12         List::x = 10;   // anggota data (baris 4)
13         ::x = 23;       // global (baris 1)
14      }
15  };
```

2.9 *Tag name*

Nama kelas atau enumerasi (tag name) adalah nama tipe (baru)

```
enum TOption {OP_READ, OP_WRITE, OP_CREATE};

TOption file_op; // dalam \C++: 'TOption' otomatis menjadi nama tipe
enum TOption file_op; // dalam C
```

2.10 *Anonymous union*

Nama tag sebuah `union` dapat dihilangkan.

```
struct Ident {
   char *nama;
   char type;
   union /* tanpa tagname */ {   /* anonymous union */
      char   *str_value;
```

```
      int      int_val;
   };
};

struct Ident r;
r.int_val = 0;
```

2.11 Kompatibilitas antara C++ dan C

Program yang dituliskan dalam bahasa C seharusnya dapat dikompilasi oleh kompilator C++. Namun demikian, ada beberapa hal yang harus diperhatikan

- Program tidak dapat menggunakan kata kunci dari C++ sebagai nama identifier
- Dalam C++ deklarasi fungsi "`f()`" berarti bahwa, `f` tidak memiliki parameter formal satupun, dalam C ini berarti bahwa f dapat menerima parameter dari jenis apapun
- Dalam C++, tipe dari konstanta karakter adalah `char`, sedangkan dalam C, tipe tersebut adalah `int`. Akibatnya `sizeof('a')` memberikan nilai 1 di C++ dan 4 di C pada mesin yang memiliki representasi integer 4 byte.
- 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++ */
```

2.12 Saran untuk C programmers

- Hindarilah penggunaan (`#define`) dalam program C++ seperti untuk mendefinisikan konstanta seperti pada

  ```
  #define MAXBUFF 1500
  ```

  Gunakanlah `const` untuk mendefinisikan konstanta
- Gunakan inline untuk menghindari *function-calling overhead*
- Deklarasikan setiap fungsi (procedure) dan spesifikasikan semua tipe parameter formalnya
- Jangan gunakan `malloc()` maupun `free()`. Sebagi gantinya, gunakanlah `new` dan `delete`. Kedua operator baru ini tidak hanya sekedar mengalokasikan memori melainkan juga secara otomatis melibatkan *constructor* dan *destructor*.
- Union pada umumnya tidak memerlukan tag name, gunakanlah *anonymous union*

3 Class

Konsep kelas dalam C++ ditujukan untuk menciptakan tipe data baru. Sebuah tipe terdiri dari kumpulan bit yang merepresentasikan nilai abstrak dari instansiasi tipe tersebut serta kumpulan operasi terhadap tipe tersebut. Sebagai contoh `int` adalah sebuah tipe karena memiliki representasi bit dan kumpulan operasi seperti "penambahan dua variabel bertipe `int`", "perkalian dua variabel bertipe `int`", dsb.

Dengan cara yang sama, sebuah kelas juga menyediakan sekumpulan operasi (biasanya public) dan sekumpulan data bit (biasanya non-public) yang menyatakan nilai abstrak objek dari kelas tersebut.

Hubungan antara kelas dengan objek dapat dinyatakan dengan analogi berikut:

class vs. object = type vs. variable

Pendeklarasian kelas (terutama fungsi anggotanya) menentukan perilaku objek dalam operasi penciptaan, pemanipulasian, pemusnahan objek dari kelas tersebut. Dalam pemrograman dengan bahasa yang berorientasi objek dapat dilihat adanya peran perancang kelas dan pengguna kelas. Perancang kelas menentukan representasi internal objek yang berasal kelas yang dirancangnya.

Pendeklarasian kelas dilakukan seperti pendefinisian sebuah struktur namun dengan mengganti kata kunci `struct` dengan `class`. Kata kunci `class` dalam C++ dapat dipandang sebagai perluasan dari kata kunci `struct`, hanya perbedaannya nama kelas (*tag-name*) dalam C++ sekaligus merupakan tipe baru.

Contoh 3.1 Sebuah `struct` yang memiliki prosedur/fungsi

```
1   struct Stack {  // nama tag "Stack" sekaligus menjadi tipe baru
2      /*******************
3       * function member *
4       *******************/
5      void Pop(int&);
6      void Push (int);
7      int isEmpty();
8      // ... definisi fungsi lainnya
9
10     /***************
11      * data member *
12      ***************/
13     int topStack;
14     int *data;
15     // ... definisi data lainnya
16  };
```

Sebuah kelas memiliki satu atau lebih *member* (analog dengan *field* pada `struct`). Ada dua jenis member:

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

Setiap *field* yang dimiliki sebuah `struct` dapat secara bebas diakses dari luar struktur tersebut. Hal ini berbeda dibandingkan dengan pengaksesan terhadap anggota kelas. Hak akses dunia luar terhadap anggota (data dan fungsi) diatur melalui tiga kata kunci `private`, `public`, dan `protected`. Setiap fungsi anggota kelas selalu dapat mengakses data dan fungsi anggota kelas tersebut (dimanapun data tersebut dideklarasikan: private, public, protected). Sedangkan fungsi bukan anggota kelas hanya dapat mengakses anggota yang berada di bagian `public`. Hak akses terhadap fungsi dan data anggota kelas dinyatakan dalam Tabel 1.

Wilayah member di deklarasikan	Makna
`public`	dapat diakses oleh fungsi di luar kelas (fungsi bukan anggota kelas tersebut) dengan menggunakan operator selektor (`.` atau `->`)
`private`	hanya dapat diakses oleh fungsi anggota kelas tersebut
`protected`	hanya dapat diakses oleh fungsi anggota kelas tersebut dan fungsi-fungi anggota kelas turunan

Tabel 1: Pengaturan hak akses melalui `public`, `private`, dan `protected`.

Dalam deklarasi "kelas" `Stack` pada Contoh 3.1, semua anggota bersifat *public*, karena hal ini sesuai dengan sifat sebuah `struct`. Namun jika, kata kunci "`struct`" diganti menjadi "`class`", maka semua anggota otomatis bersifat *private*.

Dalam contoh tersebut, fungsi-fungsi anggota `Pop()`, `Push()`, dsb hanya **dideklarasikan** namun belum **didefinisikan**. Pendefinisian anggota fungsi dapat dilakukan dengan dua cara:

- Di dalam class body, otomatis menjadi inline function
- Di luar class body, nama fungsi harus didahului oleh class scope

Contoh 3.2 menyajikan deklarasi kelas `Stack` dengan menggunakan kata kunci `class` [3]

[3]Dalam implementasi yang sebenarnya, kelas `Stack` selayaknya memiliki fungsi layanan untuk mengetahui apakah stack penuh atau tidak. Layanan ini misalnya dapat diimplementasikan sebagai fungsi dengan prototype `int isFull();`

Contoh 3.2 Deklarasi kelas `Stack` beserta fungsi anggota

```
1   class Stack {
2     public:
3      // function member
4      void Pop(int& );   // deklarasi (prototype)
5      void Push (int);   // deklarasi (prototype)
6      /*--- pendefinisian di dalam class body ---*/
7      int isEmpty() {
8         return topStack == 0;
9      }
10    private:
11
12     // data member
13     int topStack;   /* posisi yang akan diisi berikutnya */
14     int *data;
15  };  // PERHATIKAN TITIK KOMA !!!
16
17  // pendefinisian member function Pop di luar
18  // class body
19  void Stack::Pop(int& item) {
20     if (isEmpty()) {
21        // error message
22     }
23     else {
24        topStack--;
25        item = data [topStack];
26     }
27  }  // TIDAK PERLU TITIK KOMA !!!
28
29  void Stack::Push (int item) {
30     if (isFull()) {
31        // error message
32     }
33     else {
34        data [topStack] = item;
35        topStack++;
36     }
37  }
```

3.1 Pointer implisit `this`

Setiap objek dari suatu kelas memiliki sendiri salinan anggota data dari kelas tersebut. Namun, hanya ada **satu** salinan anggota fungsi untuk objek-objek dari kelas tersebut. Dengan

kata lain, jika ada dua objek dari suatu kelas yang memanggil salah satu fungsi anggota kelas tersebut maka kedua objek akan menjalankan rangkaian instruksi yang terletak pada **lokasi memori yang sama**, tetapi anggota data yang diakses oleh fungsi anggota tersebut terletak pada dua **lokasi memori yang berbeda**.

Untuk menangani hal di atas, setiap function member secara implisit memperoleh argumen (parameter aktual) tersembunyi berupa pointer ke objek (implicit this pointer). Jika pointer `this` ini akan digunakan di dalam fungsi anggota `Push` di atas, setiap pengaksesan terhadap data anggota (maupun fungsi anggota) kelas `Stack` dapat diawali dengan '`this->`'.

```
1   void Stack::Push (int item) {
2      // . . .
3      this->data [this->topStack] = item;
4      this->topStack++;
5      // . . .
6   }
```

Dalam contoh berikut di atas, perhatikanlah bahwa parameter formal `item` **tidak dapat** dituliskan sebagai `this->item` karena bukan merupakan anggota kelas `Stack`.

Pointer `this` merupakan sebuah *rvalue* sehingga ekspresi assignment terhadap `this` dalam contoh berikut **tidak diijinkan**:

```
this = ...;  // ruas kanan diisi suatu ekpresi
```

Mengapa ada this pointer?

- Pointer implisit this untuk kelas `X`, dideklarasikan sebagai `X* this`, dan digunakan untuk mengakses member di dalam kelas tersebut
- Pointer `this` dapat juga digunakan memberikan return value yang berjenis kelas tersebut (misalnya fungsi operator). Hal ini dibahas lebih lanjut pada Bagian 4.

3.2 Objek dari Kelas

Pendeklarasian kelas tidak mengakibatkan alokasi memory untuk kelas tersebut. Memory dialokasikan jika ada objek yang didefinisikan dengan tipe kelas tersebut. Dengan menggunakan kelas `Stack` di atas, berikut ini beberapa contoh pendefinisian variabel (objek) yang berasal dari kelas `Stack` di atas:

```
1   Stack myStack;
2   Stack OprStack [10];
3   Stack * pts = new Stack;
4   Stack ns = myStack; // definition & initialization
5
6      // inisialisasi di atas sama dengan instruksi berikut:
7      // ns.topStack = myStack.topstack
8      // ns.data = myStack.data
```

3.3 Pengaksesan *public member*

Anggota yang publik dapat diakses melalui objek seperti layaknya pengaksesan field pada sebuah `struct`. Pengaksesan terhadap *data member*: jika berperan sebagai lvalue maka berarti diacu alamatnya, dan jika berperan sebagai rvalue maka berarti diacu isinya. Pengaksesan terhadap *function member* berarti pemanggilan terhadap fungsi tersebut.

```
1   int x;
2
3   // constructor Stack harus sudah menjamin inisialisasi stack
4   // dengan benar
5
6   myStack.Push (99);
7   OprStack[2].Pop(x);
8   pts->Push(x);
9
10  if (myStack.isEmpty()) {
11     printf ("Stack masih kosong . . . ");
12  }
```

3.4 Constructor, Destructor, dan Copy Constructor

Untuk tipe-tipe primitif (`int`, `float`, `char`, `double`, dsb.) kompilator mengetahui bagaimana mengalokasikan, menginisialisasi, dan mendealokasikan kumpulan bit yang merepresentasikan tipe tersebut. Untuk tipe data yang lebih kompleks, proses ini mungkin harus dilakukan sendiri oleh perancang kelas. Untuk keperluan tersebut, C++ menggunakan konsep **constructor** dan **destructor**. Untuk selanjutnya, dalam penulisan "ctor" akan digunakan untuk menyatakan constructor dan "dtor" untuk menyatakan destructor.

Constructor (destructor) adalah fungsi anggota (khusus) yang secara otomatis dipanggil pada saat penciptaan (pemusnahan) objek. Dalam sebuah kelas, ctor dan dtor adalah fungsi yang memiliki nama yang sama dengan nama kelas. Sebuah kelas mungkin **tidak** memiliki ctor atau memiliki **lebih dari satu** ctor. Tugas utama konstruktor adalah untuk menginisialisasi nilai-nilai dari anggota data yang dimiliki kelas. Konstruktor dapat dibedakan menjadi dua jenis:

1. *Default constructor*: konstruktor yang menginisialisasi objek dengan nilai(-nilai) default yang ditentukan oleh perancang kelas. Dalam deklarasi kelas, ctor ini **tidak** memiliki parameter formal.

2. *User-defined constructor*: konstruktor yang menginisialisasi objek dengan nilai(-nilai) yang diberikan oleh pemakai kelas pada saat objek diciptakannya. Dalam deklarasi kelas, ctor ini memiliki satu atau lebih parameter formal.

Destructor adalah fungsi yang namanya sama dengan nama kelas dan didahului tanda '~' (tilde). Sebuah kelas dapat memiliki **paling banyak satu** destructor

Sebuah objek dapat pula diciptakan dengan cara menginisialisasinya dengan objek lain yang sudah ada. Dalam hal ini, objek tersebut akan diciptakan melalui **copy constructor**. Untuk selanjutnya "cctor" akan digunakan untuk menyatakan copy constructor.

```
Stack ns = myStack; // create & init
```

Dengan cara di atas, inisialisasi objek dilakukan oleh "default cctor" yang melakukan *bitwise copy*. **Hal ini dapat mengakibatkan kesalahan untuk kelas yang memiliki anggota data berupa pointer**. Dengan contoh kelas `Stack` yang diberikan pada Contoh 3.2, anggota `data` dari objek `ns` dan `myStack` akan mengacu ke lokasi memori yang sama, padahal kedua objek tersebut seharusnya tidak memiliki lokasi memori yang sama.

Jika "default cctor" tidak dikehendaki, perancang kelas harus mendefinisikan sebuah *copy constructor*.

Dalam penulisan kode sebuah kelas akan terdapat dua bagian berikut:

1. Interface / specification yang merupakan **deklarasi** kelas, dan
2. Implementation / body yang berisi **definisi** dari fungsi-fungsi anggota dari kelas tersebut.

Agar kelas dapat digunakan oleh pengguna, hanya bagian deklarasi kelas yang perlu disertakan di dalam program pengguna. Untuk itu, deklarasi kelas dituliskan ke dalam file `X.h` (`X` adalah nama kelas). Untuk mencegah penyertaan header lebih dari satu kali, deklarasi kelas dituliskan di antara `#ifdef XXXX_H` dan `#endif` (atau `#endif XXXX_H`). Perhatikan Contoh 3.3.

> PERHATIAN: Di dalam header file, **JANGAN** menuliskan **DEFINISI** objek/variabel karena akan mengakibatkan kesalahan "*multiply defined name*" pada saat *linking* dilakukan.

Implementasi (definisi fungsi-fungsi anggota) seperti yang terlihat pada Contoh 3.4 dituliskan ke dalam file `X.cc`, `X.cpp`, `X.cxx` atau file `X.C`.

Dalam contoh pada Contoh 3.4, cctor tidak didefinisikan sehingga jika dilakukan penciptaan objek lewat inisialisasi, *data member* `data` dari dua objek yang berbeda akan menunjuk ke lokasi yang sama. Selain itu, contoh kelas tersebut juga mendefinisikan dua konstruktor: satu default constructor (`Stack::Stack()`) dan satu konstruktor yang memungkinkan pemakai kelas `Stack` menyatakan ukuran maksimum stack yang akan digunakannya (`Stack::Stack (int)`).

Perhatikanlah pula bahwa ctor, dtor, maupun cctor merupakan fungsi yang **tidak** memiliki tipe kembalian (*return type*) karena fungsi-fungsi tersebut tidak dapat dipanggil secara eksplisit oleh pengguna, melainkan secara implisit oleh kompilator. Penjelasan lebih jauh mengenai hal ini dapat dilihat pada bagian 3.6.

Contoh 3.3 Deklarasi kelas `Stack`

```
1  /*--------------------------------------*
2   * Nama file: Stack.h                   *
3   * Deskripsi: interface dari kelas Stack *
4   *--------------------------------------*/
5  #ifndef STACK_H
6  #define STACK_H
7  class Stack {
8    public:
9     // ctor -- dtor
10    Stack();     // constructor
11    Stack (int); // constructor dengan ukuran stack
12    ~Stack();    // destructor
13
14    // fungsi-fungsi layanan
15    void Pop(int&);
16    void Push (int);
17    int isEmpty()
18    { // pendefinisian di dalam class body
19       return topStack == 0;
20    }
21   private:
22    // data member
23    const int defaultStackSize = 500; // ANSI: tidak boleh inisialisasi
24    int topStack;
25    int size;
26    int *data;
27 };
28 #endif STACK_H
```

Contoh 3.4 Implementasi / Body dari kelas `Stack`

```
1  /*-----------------------------------------------*
2   * Nama file: Stack.cc                           *
3   * Deskripsi: definisi function members dari kelas *
4   *            Stack (implementation)             *
5   *-----------------------------------------------*/
6  #include <Stack.h>
7
8  // Stack constructor
9  Stack::Stack () {
10    data = new int [defaultStackSize];
11    topStack = 0;
12    size = defaultStackSize;
13 }
14
15 // constructor dengan ukuran stack
16 Stack::Stack (int s) { /* parameter s = ukuran stack */
17    data = new int [s]; /* alokasi array integer dengan
18                         * index 0 .. s-1 */
19    topStack = 0;
20    size = s;
21 }
22
23 Stack::~Stack () { // destructor
24    delete [] data; // dealokasi array integer
25    size = 0;
26    data = 0;
27 }
28
29 void Stack::Pop(int& item) {
30    if isEmpty()
31       // error message
32    else {
33       topStack--;
34       item = data [topStack];
35    }
36 }
37
38 void Stack::Push (int item) {
39    // . . .
40    data [topStack] = item;
41    topStack++;
42    // . . .
43 }
```

Contoh 3.5 Program yang menggunakan `Stack`

```
1   /*--------------------------*
2    * Nama file: main.cc       *
3    * Deskripsi: Uji coba stack *
4    *--------------------------*/
5   #include <Stack.h>
6   #include ... // header file lain yang diperlukan
7
8   main ()
9   {
10     // kamus
11     Stack s1;        // constructor Stack()
12     Stack s2 (20);   // constructor Stack (int)
13
14     // algoritma ...
15     // kode program dituliskan di sini
16  }
```

3.5 Penciptaan/Pemusnahan Objek

Setelah `Stack.h` didefinisikan dan `Stack.cc` dikompilasi menjadi `Stack.o` maka pengguna kelas dapat menuliskan program berikut yang kemudian dilink dengan `Stack.o` (atau melalui pustaka tertentu). Contoh program yang menggunakan kelas `Stack` di atas ditunjukkan pada Contoh 3.5.

Ada beberapa jenis objek yang dapat digunakan di dalam program C++:

- Automatic object: diciptakan jika ada deklarasi objek di dalam blok eksekusi dan dimusnahkan (secara otomatis oleh kompilator) pada saat blok yang mengandung deklarasi tersebut selesai eksekusi
- Static object: diciptakan satu kali pada saat program dimulai dan dimusnahkan (secara otomatis oleh kompilator) pada saat program selesai
- Free store object: diciptakan dengan operator `new` dan dimusnahkan dengan operator `delete`. Kedua hal ini harus secara **eksplisit** dilakukan oleh pengguna kelas/objek.
- Member object: sebagai anggota dari kelas lain penciptaannya dilakukan melalui memberwise initialization list. Bagian 3.7 membahas hal ini.

Contoh 3.6 menunjukkan sebuah penggalan program yang berisi tiga dari empat jenis objek di atas.

3.5.1 Array of Objects

Untuk memberikan kemungkinan pada pemakai kelas mendeklarasikan array dari objek, kelas tersebut harus memiliki constructor yang dapat dipanggil tanpa argumen (*default con-*

structor).

Jika array diciptakan melalui operator `new`, *destructor* harus dipanggil (melalui `delete`) untuk setiap elemen array yang dialokasikan.

```
1   #include <Process.h>
2
3      Process *pa1, *pa2;
4
5      pa1 = new Process [3];
6      pa2 = new Process [5];
7
8      // ... kode yang menggunakan pa1 & pa2
9
10     delete pa1;   // not OK
11     delete [] pa2;   //   OK
```

3.6 Copy Constructor

Pada bagian 3.4 telah dijelaskan sekilas mengenai salah satu manfaat dari cctor. Pada bagian ini akan dijelaskan lebih lanjut manfaat lain dari cctor. Perhatikan Contoh 3.7. Dengan menggunakan deklarasi `Stack.h` yang sudah diberikan pada bagian tersebut, penciptaan objek `s3` melalui inisialisasi oleh `s1` akan mengakibatkan terjadinya proses penyalinan bit per bit dari objek `s1` ke `s3`. Hal ini terjadi karena kelas `Stack` belum memiliki *copy constructor*.

Dalam kelas `Stack` seperti yang terlihat pada Contoh 3.3, proses penyalinan bit per bit ini akan mengakibatkan efek yang tidak diinginkan. Nilai dari data anggota `topStack` dan `size` dapat disalin bit per bit tanpa masalah, tetapi jika nilai data anggota `data` disalin bit per bit akan terjadi adanya dua objek berjenis `Stack` yang mengacu ke lokasi memori yang sama, padahal seharusnya keduanya mengacu ke lokasi memori yang berbeda. Untuk menghindari hal ini, kelas `Stack` harus mendefinisikan sebuah *copy constructor*.

Contoh 3.6 Penciptaan / pemusnahan objek

```
1   #include <Stack.h>
2
3   Stack s0; /* global (static) */
4
5   int reverse() {
6      static Stack tstack = ...; /* local static */
7
8      // kode untuk fungsi reverse() di sini
9   }
10
11  main () {
12     Stack s1;         // automatic
13     Stack s2 (20);    // automatic
14     Stack *ptr;
15
16     ptr = new Stack(50);   /* free store object */
17     while (...) {
18        Stack s3;       // automatic
19
20        /* assignment dgn automatic object */
21        s3 = Stack (5); // ctor Stack(5) is called
22        /* dtor Stack(5) is called */
23
24        // ... instruksi lain ...
25     }
26     /* dtor s3 is called */
27
28     delete ptr; /* dtor *ptr is called */
29  }
30  /* dtor s2 is called */
31  /* dtor s1 is called */
32
33  /* dtor s0 is called */
```

Contoh 3.7 Peranan *copy constructor*

```
1   #include <Stack.h>
2
3   void f1 (const Stack& _) { /* instruksi tidak dituliskan */}
4
5   void f2 (Stack _) { /* instruksi tidak dituliskan */ }
6
7   Stack f3 (int) {
8      /* instruksi tidak dituliskan */
9      return ...; // return objek bertipe "Stack"
10  }
11
12  main ()
13  {
14     Stack s2 (20);   // constructor Stack (int)
15
16     /* s3 diciptakan dengan inisialisasi oleh s2 */
17     Stack s3 = s2;   // BITWISE COPY, jika
18                      // tidak ada cctor yang didefinisikan
19     f1 (s2);         // tidak ada pemanggilan cctor
20     f2 (s3);         // ada pemanggilan cctor
21     s2 = f3 (-100);  // ada pemanggilan cctor dan assignment
22  }
```

Copy constructor (cctor) dipanggil pada saat penciptaan objek yang dilakukan melalui:

- Deklarasi variabel dengan inisialisasi
- Pemberian parameter aktual ke parameter formal yang dilakukan secara "*pass by value*". Dalam Contoh 3.7, parameter aktual `s2` diberikan ke fungsi $\widehat{\texttt{f1}}()$ **tanpa** adanya pemanggilan *copy constructor* sedangkan parameter aktual `s3` diberikan ke fungsi `f2()` dengan adanya pemanggilan *copy constructor*.
- Pemberian nilai kembalian fungsi yang nilai kembaliannya bertipe kelas tersebut (bukan *pointer* atau *reference*). Dalam Contoh 3.7 hal ini terjadi pada saat instruksi `return` dijalankan, *bukan* pada saat nilai kembalian di*assign* ke variabel `s2`.

Copy constructor untuk kelas `MyClass` dideklarasikan sebagai fungsi dengan nama `MyClass` dan memiliki sebuah parameter formal berjenis *const reference* dari kelas `MyClass`.

```
MyClass(const MyClass&);
```

Parameter aktual yang diberikan pada saat eksekusi adalah objek (yang akan diduplikasi) yang digunakan untuk menginisialisasi objek yang sedang diciptakan oleh cctor.

Deklarasi kelas `Stack` harus ditambahkan dengan deklarasi cctor yang sesuai seperti terlihat pada Contoh 3.8.

Contoh 3.8 Penambahan cctor pada kelas `Stack`

```
1   class Stack {
2     public:
3      Stack(); // constructor
4      Stack (int); // constructor dengan ukuran stack
5      Stack (const Stack&); // copy constructor
6      ~Stack(); // destructor
7     // ...anggota-anggota lain tidak dituliskan...
8   };
```

Yang harus dituliskan dalam **definisi** cctor adalah kode yang membuat penciptaan objek secara inisialisasi menjadi benar. Dalam contoh `Stack` di atas, yang harus dilakukan adalah mengalokasikan tempat untuk *data member* `data` agar setiap objek yang berasal dari kelas `Stack` memiliki lokasi memori terpisah untuk menyimpan datanya. Perhatikan Contoh 3.9.

Contoh 3.9 Definisi cctor untuk kelas `Stack`

```
1   Stack::Stack (const Stack& s)
2   {
3      int i;
4
5      size = s.size;
6      topStack = s.topStack;
7      data = new int [size];  // PERHATIKAN: data member "data" harus di
8                              // alokasi ulang, tidak disalin dari
9                              // "s.data".
10     for (i=0; i<size; i++)
11        data[i] = s.data[i];
12  }
```

Untuk memahami pemanggilan ctor, cctor, dtor perhatikan bagian berikut. Misalkan pada deklarasi kelas `Stack` di tambahkan beberapa instruksi seperti terlihat pada Contoh 3.10 yang dipanggil oleh program pada Contoh 3.11.

Contoh 3.10 Deklarasi ctor, cctor, dan dtor dari `Stack`

```
1   /* Nama file: Stack.h                              */
2   /* Deskripsi: ujicoba constructor, destructor, */
3
4   #ifndef STACK_H
5   #define STACK_H
6   class Stack {
7   public:
8      Stack() { printf ("ctor %x\n", this); }
9      Stack (const Stack&) { printf ("cctor %x\n", this); }
10     ~Stack() { printf ("dtor %x\n", this); }
11  };
12  #endif STACK_H
```

Contoh 3.11 Program untuk observasi ctor, dtor, cctor

```
1   /*-----------------------------------------------*
2    * Nama file: test.cc                            *
3    * Deskripsi: pengamatan pemanggilan ctor, dtor, *
4    *            cctor                              *
5    *-----------------------------------------------*/
6   #include <Stack.h>
7
8   f (Stack _) {
9      printf ("ffffffff\n");
10  }
11
12  Stack ll;
13
14  main() {
15     printf ("11111111\n");
16     Stack x;
17     printf ("22222222\n");
18     Stack y = x;
19     printf ("33333333\n");
20     f(x);
21     printf ("44444444\n");
22  }
```

Hasil eksekusi dari Contoh 3.11 ditunjukkan pada Gambar 1.

```
ctor  0x198a8
11111111
ctor  0x5f514
22222222
cctor 0x5f510
33333333
cctor 0x5f50c
ffffffff
dtor  0x5f50c
44444444
dtor  0x5f510
dtor  0x5f514
dtor  0x198a8
```

Gambar 1: Hasil eksekusi program pada Gambar 3.11

3.7 Constructor Initialization List

Misalkan ada sebuah kelas `Parser` yang memiliki data member yang bertipe `Stack` seperti yang dideklarasikan di atas dan salah satu ctor dari `Parser` memberikan kemungkinan pengguna objek untuk menciptakan objek `Parser` dengan sekaligus menyatakan ukuran `Stack` yang dapat digunakan `Parser` tersebut. Perhatikan Contoh 3.12.

Contoh 3.12 Kelas `Parser` yang memiliki data member bertipe `Stack`

```
1   #include <Stack.h>
2
3   class Parser {
4     public:
5        Parser(int);
6      // ...
7     private:
8      Stack sym_stack, op_stack;
9      // ...
10  };
```

Pada saat *constructor* `Parser::Parser(int)` dipanggil, anggota data `sym_stack` akan diciptakan dan diinisialisasi melalui konstruktor *default* `Stack::Stack()`. Bagaimana jika inisialisasi ingin dilakukan melalui *user-defined constructor* `Stack::Stack(int)`? Ada dua cara untuk melakukan hal ini:

- Member `sym_stack` diciptakan melalui *default ctor*, lalu ctor `Parser::Parser()` melakukan operasi *assignment*. Dengan cara seperti ini terjadi pemanggilan konstruktor `Stack::Stack()` (tanpa parameter) serta fungsi yang menangani operasi assignment (dua kali pemanggilan).

- ctor `Parser::Parser()` melakukan inisialisasi `Stack` melalui *member initialization list* seperti ditunjukkan pada Contoh 3.13. Dengan cara ini, hanya ada satu kali pemanggilan konstruktor yaitu `Stack::Stack (int)`

Manfaat *constructor initialization list* adalah performansi yang lebih baik. Oleh karena itu, anggota data yang bertipe non-primitif sebaiknya diinisialisasi melalui cara ini.

Constructor initialization list dapat digunakan untuk menginisialisasi beberapa member sekaligus. Untuk melakukan hal ini, nama anggota data yang diinisialisasi dituliskan setelah parameter formal konstruktor dan setiap nama anggota diikuti oleh sejumlah argumen yang sesuai dengan *user defined constructor* yang ada. Dalam Contoh 3.13, anggota data `sym_stack` dan `op_stack` diinisialiasi dengan menggunakan satu parameter aktual karena kelas `Stack` dalam contoh yang sudah disajikan memiliki *user defined constructor* dengan satu parameter formal.

Contoh 3.13 Memberwise initialization

```
1  Parser::Parser(int x) :  sym_stack (x), op_stack (x)
2  {
3     // ...
4  }
```

3.8 Const Member

Pada deklarasi variabel, atribut `const` menyatakan bahwa variabel tersebut bersifat konstan dan nilainya tidak dapat diubah oleh siapapun. Anggota data (data member) maupun anggota fungsi (function member) dapat juga memiliki atribut `const`. Makna dari atribut ini adalah:

- Anggota data yang memiliki atribut `const` berarti bahwa nilai anggota data tersebut akan tetap sepanjang waktu hidup objeknya. Standard ANSI mengharuskan pengisian nilai awal terhadap anggota data `const` dilakukan pada saat objek tersebut diciptakan. Bandingkanlah dengan definisi konstan yang juga memanfaatkan pengisian nilai awal pada saat penciptaan berikut:

  ```
  const int max_size = 5000;
  ```

  Untuk sebuah objek, pengisian tersebut harus dilakukan melalui *constructor initialization list* seperti pada Bagian 3.7.

- Anggota fungsi yang memiliki atribut `const` berarti bahwa fungsi tersebut tidak akan mengubah objek yang memanggilnya.

  Object yang ditandai sebagai `const` tidak boleh memanggil fungsi anggota yang tidak memiliki atribut `const` karena hal tersebut dapat mengakibatkan perubahan status objek tersebut.

Dalam contoh `Stack` di atas, fungsi yang dapat mendapatkan atribut `const` adalah `isEmpty()`, sedangkan data yang dapat mendapatkan atribut `const` adalah `size`. Pada Contoh 3.14 ditunjukkan bagaimana penulisan deklarasi fungsi dan data tersebut setelah mendapatkan atribut `const`.

Contoh 3.14 Pemanfaatan `const` pada anggota fungsi

```
1   class Stack {
2      // ...
3     public:
4      Stack ();
5      Stack (int s);
6      Stack (const Stack&);
7      int isEmpty() const; /* keyword 'const' dituliskan pada
8                            * deklarasi maupun definisi
9                            * member function */
10    private:
11     const int size;
12  };
13
14  int Stack::isEmpty () const { // <== PERHATIKAN "const"
15     //...
16  }
17
18  Stack::Stack () : size (defaultStacksize) {
19  }
20
21  Stack::Stack (int p) : size (p) {
22  }
23
24  Stack::Stack (const Stack& s) : size (s.size) {
25  }
```

3.9 Static Member

Setiap objek di dalam kelas memiliki sendiri member datanya. Dalam keadaan tertentu diperlukan anggota data yang digunakan bersama oleh seluruh objek dari satu kelas objek tersebut. Hal ini misalnya dapat digunakan untuk menghitung jumlah objek yang sudah diciptakan.

```
1   class Stack {
2    public:
3      // ... fungsi lain
4
```

```
5    private:
6      static int n_stack;      // static data member!!
7      // ... data & fungsi lain
8    };
```

Inisialisasi anggota statik **tidak dapat** dilakukan di dalam constructor, melainkan **di luar** deklarasi kelas dan di luar fungsi anggota. Inisialisasi anggota data yang statik dilakukan di file implementasi (`X.cc`), **jangan** di dalam file *header*.

```
// inisialisasi anggota data yang statik
// di dalam file Stack.cc
int Stack::n_stack = 0;
```

Anggota fungsi yang **hanya** mengakses anggota (data maupun fungsi) statik dapat dideklarasikan sebagai *static function*

```
1    class Stack {
2       // ...
3     public:
4       static int NumStackObj ();
5    };
6
7    int Stack::NumStackObj() {
8      // kode yang mengakses hanya data member statik
9    }
```

Untuk memahami anggota statik (fungsi maupun data) bandingkanlah dengan deklarasi variabel lokal statik berikut:

```
1    void SuatuFungsi ()
2    {
3       static int v = -1;
4
5       // ... instruksi ...
6    }
```

Jika atribut `statik` tidak digunakan, maka umur hidup dan keberadaan variabel `v` sepenuhnya bergantung pada umur hidup dan keberadaan fungsi `SuatuFungsi`. Dengan dituliskannya atribut `static` maka umur hidup dan keberadaan variabel `v` **tidak lagi** bergantung pada `SuatuFungsi`. Hanya *visibility* `v` yang ditentukan oleh `SuatuFungsi`. Demikian juga dengan anggota yang dideklarasikan dengan atribut `static`. Umur hidup dan keberadaan mereka tidak ditentukan oleh kelas yang melingkupinya.

Sebagai akibatnya, pada anggota fungsi / data yang statik berlaku sifat-sifat berikut:

- Anggota fungsi statik dapat dipanggil tanpa melalui objek dari kelas tersebut, misalnya:

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

- Anggota fungsi statik tidak memiliki pointer implisit `this`
- Data member yang statik diinisialisasi tanpa perlu adanya objek dari kelas tersebut

3.10 Friend

Dalam C++, sebuah kelas (A) atau fungsi (F) dapat menjadi *friend* dari kelas lain (B). Dalam keadaan biasa, kelas A maupun fungsi F tidak dapat mengakses anggota (data/fungsi) non public milik B. Dengan adanya hubungan *friend* ini, A dan F dapat mengakses anggota *non public* dari B. Deklarasi `friend` dituliskan dari pihak yang memberikan ijin. Pemberian ijin ini tidak bersifat dua arah, yang berarti dalam kode berikut, kelas `B` **tidak** memiliki hak untuk mengaskses anggota *non-public* dari kelas `A`. Dalam contoh ini, realisasinya adalah:

```
1  class B { // kelas "pemberi ijin"
2     friend class A;
3     friend void F (int, char *);
4
5   private:
6     // ...
7   public:
8     //...
9  };
```

Fungsi yang dideklarasikan dengan atribut `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, melainkan salinan dari objek tersebut. Fungsi anggota merupakan fungsi di dalam kelas dan operasi yang dilakukannya selalu berpengaruh pada objek sesungguhnya.

Kriteria penggunaan atribut `friend`:

- Sedapat mungkin hindari penggunaan `friend`. Penggunaan `friend` di antara kelas menunjukkan perancangan kelas yang kurang baik. Jika kelas `A` menjadikan kelas `B` sebagai `friend` maka kemungkinan besar kelas `A` dan `B` seharusnya tidak dipisahkan
- Jika operasi yang dijalankan oleh sebuah fungsi `friend` mengubah status dari objek, operasi tersebut harus diimplementasikan sebagai fungsi anggota
- Gunakan `friend` untuk *overloading* pada operator tertentu. Hal ini dibahas lebih lanjut di Bagian 4.

3.11 *Nested Class*

Dalam keadaan tertentu, perancang kelas membutuhkan pendeklarasian kelas di dalam deklarasi suatu kelas tertentu. Sebagai contoh pada deklarasi kelas `List` yang merupakan list dari integer, kita mungkin membutuhkan deklarasi kelas `ListElem` untuk menyatakan elemen list tersebut. Operasi-operasi terhadap list didefinisikan di dalam kelas `List`, namun demikian ada kemungkinan operasi-operasi ini membutuhkan pengaksesan terhadap bagian non-publik dari kelas `ListElem` sehingga penggunaan `friend` dituliskan di dalam kelas `ListElem` seperti yang terlihat pada Contoh 3.15.

Contoh 3.15 Kelas `List` dan `ListElem`

```
1   class List;
2
3   class ListElem {
4      friend class List;
5    public:
6      //
7    private:
8   };
9
10  class List {
11   public:
12   //
13   private:
14   //
15  };
```

Sesungguhnya, pemakai kelas `List` **tidak perlu** mengetahui keberadaan kelas `ListElem`. Yang perlu ia ketahui adalah adanya layanan untuk menyimpan nilai (integer) ke dalam list tersebut maupun untuk mengambil nilai dari list tersebut.

Dalam keadaan di atas, kelas `ListElem` dapat dijadikan sebagai *nested class* di dalam kelas `List` dan deklarasinya dapat dituliskan seperti pada Contoh 3.16.

Contoh 3.16 Deklarasi `ListElem` di dalam `List`

```
1   class List {
2      //
3      //
4      class ListElem {
5         //
6         //
7      };
8   };
```

Namun demikian, ada pertanyaan yang mungkin muncul: “Dimanakah kelas `ListElem`

dideklarasikan? Di bagian publik atau non-publik?". Jika ditempatkan pada bagian `public` dari kelas `List`, maka bagian publik dari kelas `ListElem` akan tampak ke luar kelas `List` sebagai anggota yang juga publik. Sebaliknya, jika ditempatkan pada bagian non-publik, maka bagian publik dari kelas `ListElem` akan tersembunyi terhadap pihak luar kelas `List`, namun akan tetap terlihat oleh anggota-anggota kelas `List`. Efek terakhir inilah yang diinginkan, sehingga dengan demikian deklarasi kelas `ListElem` ditempatkan di bagian non-publik seperti yang ditunjukkan pada Contoh 3.17. Dalam keadaan ini juga kemungkinan besar kelas `ListElem` tidak perlu memiliki bagian non-publik.

Contoh 3.17 Deklarasi `ListElem` di dalam `List`

```
1   class List {
2   public:
3   // bagian public kelas List
4   // ...
5   private:
6      class ListElem {
7           public:
8           // semua anggota ListElem berada pada bagian publik
9      };
10
11     // definisi anggota private kelas List
12  };
```

3.11.1 Fungsi Anggota *Nested Class*

Dalam contoh `List` dan `ListElem` di atas, pendefinisian kelas `ListElem` berada di dalam lingkup kelas `List` sehingga nama *scope* yang harus digunakan dalam mendefinisikan kelas `ListElem` adalah "`List::ListElem`" bukan hanya sekedar "`ListElem::`". Jika kelas `List` merupakan kelas generik yang memiliki parameter generik `Type`, maka nama *scope* menjadi "`List<Type>::ListElem`". Perhatikan Contoh 3.18 dan Contoh 3.19.

Penggunaan nama parameter generik `Type` di dalam *nested class* `ListElem` pada baris 12 mengakibatkan kelas `ListElem` menjadi kelas generik. Perhatikanlah pula penulisan nama *scope* pada baris 19, 24, dan 29, serta jenis kembalian fungsi "`operator=`" dari kelas `ListElem`.

4 Operator Overloading dan Fungsi `operator`

Function overloading adalah fasilitas yang memungkinkan sebuah nama fungsi yang sama dapat dipanggil dengan jenis & jumlah parameternya (*function signature*) yang berbeda-beda. Sebenarnya hal ini sudah sering kita gunakan, secara tidak langsung, tanpa sadar. Misalnya, untuk penambahan dua bilangan kita menggunakan menggunakan simbol `+` baik untuk penambahan integer maupun real.

Contoh 3.18 Deklarasi anggota `ListElem` di dalam `List`

```
1   // File: List.h
2   template <class Type>
3   class List {
4    public:
5      //... bagian ini tidak dituliskan ...
6    private:
7      class ListElem {
8           public: // seluruh anggota bersifat publik
9         ListElem (const Type&);
10        ListElem (const ListElem&);
11        ListElem& operator (const ListElem&);
12        Type info; // <-- perhatikan parameter generik
13        ListElem* next;
14     };
15  };
```

Contoh 3.19 Definisi fungsi anggota *nested class* `ListElem`

```
16  // File: List.h
17
18  template <class Type>
19  List<Type>::ListElem::ListElem (const Type& v) : info (v) {
20     next = 0;
21  }
22
23  template <class Type>
24  List<Type>::ListElem::ListElem (const ListElem& x) : info (x.info) {
25     next = x.next;
26  }
27
28  template <class Type>
29  List<Type>::ListElem& List<Type>::ListElem::operator= (const ListElem& x) {
30     info = x.info;
31     next = x.next;
32     return *this;
33  }
```

```
int a, b, c;
float x, y, z;

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

Jika penambahan dituliskan sebagai fungsi `tambah()`, maka kedua operasi tersebut dapat dituliskan sebagai:

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

dengan sebelumnya mendeklarasikan **dua** prototipe fungsi `tambah` sbb:

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

Fasilitas *function name overloading* dalam C++ dapat juga diterapkan pada simbol-simbol operasi. Hal ini dapat dimanfaatkan oleh perancang kelas untuk menggunakan simbol-simbol operator dalam pemanipulasian objek, seperti pada contoh berikut:

```
Matrix A, B, C;
C = A * B; /* perkalian matrix */

Complex x, y, z;
x = y / z; /* pembagian bilangan kompleks */

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

Hal ini dapat diwujudkan oleh perancang kelas dengan menggunakan fungsi anggota yang namanya terdiri kata kunci `operator` dan diikuti oleh simbol operator yang digunakan. Fasilitas ini disebut sebagai fungsi operator (*operator function*). Dalam contoh-contoh di atas, perancang kelas `Matrix`, `Complex`, dan `Process` harus menuliskan deklarasi kelas seperti pada Contoh 4.1.

Dari contoh di atas terlihat dua bentuk pendeklarasian fungsi operator: (a) sebagai non-anggota[4] atau (b) sebagai fungsi anggota biasa. Dalam contoh-contoh di atas, seluruh operator merupakan operator biner (menerima dua argumen), namun terdapat perbedaan jumlah argumen ketika diimplementasikan sebagai fungsi non-anggota atau sebagai fungsi anggota. Jika diimplementasikan sebagai fungsi anggota, maka fungsi tersebut memiliki pointer implisit `this` sehingga pada saat operasi dilakukan pointer `this` tersebut akan menunjuk ke alamat objek pada "ruas kiri" dari operasi. Dalam contoh kelas `Complex` di atas, operasi pembagian dua bilangan kompleks diwakili oleh fungsi anggota `operator/` yang menerima satu parameter. Parameter tersebut mewakili objek pada ruas kanan operasi.

Misalkan dalam contoh `Stack` yang sudah dibahas sejauh ini akan ditambahkan fungsi operator untuk push, dan pop dengan menggunakan simbol operator `<<`, dan `>>`. Pada kelas `Stack` harus ditambahkan tiga prototipe fungsi berikut seperti terlihat pada Contoh 4.2.

[4]Harus dideklarasikan sebagai `friend` jika mengakses anggota yang non-public

Contoh 4.1 Deklarasi kelas yang memanfaatkan `operator`

```
1   class Matrix {
2     public:
3     // fungsi-fungsi operator
4     friend Matrix& operator* (const Matrix&, const Matrix&);
5     // ...
6   };
7
8   class Complex {
9     // ...
10    public:
11     Complex& operator/ (const Complex&);
12    // ...
13  };
14
15  class Process {
16    // ...
17    public:
18     void operator<< (int);
19    // ...
20  };
```

Contoh 4.2 Deklarasi kelas `Stack` yang memanfaatkan fungsi `operator`

```
1   class Stack {
2      //
3      public:
4      void operator<< (int);    // untuk push
5      void operator>> (int&);    // untuk pop
6      // ...
7   };
```

Definisi dari kedua fungsi operator ini adalah sama seperti definisi dari prosedur `Push()` dan `Pop()`. Bahkan jika juga didefinisikan, prosedur `Push()` dan `Pop()` dapat dipanggil dari dalam kedua fungsi operator tersebut seperti tertera pada Contoh 4.3.

Contoh 4.3 Pendefinisian fungsi `operator` pada kelas `Stack`

```
1   void Stack::operator<<(int item) {
2      Push (item);
3   }
4
5   void Stack::operator>>(int& item) {
6      Pop (item);
7   }
```

Bandingkanlah prototipe `void operator<< (int)` dengan `void Push (int)`. Dari kedua prototipe ini tampak bahwa pengguna kelas `Stack` dapat memanfaatkan fungsi dengan nama `Push` dan `operator<<`. Demikian juga `void operator>> (int&)` dengan `void Pop (int&)`. Dengan sudut pandang ini, maka pengguna kelas `Stack` dapat menuliskan kode seperti pada Contoh 4.4.

Contoh 4.4 Pemanfaatan fungsi `operator`

```
Stack s, t;

s.Push (100);
t.operator<< (500);
```

Selain dengan cara di atas, fungsi `operator` dapat dipanggil seperti layaknya operator-operator lainnya, sehingga operasi "`t.operator>>(500)`" di atas dapat juga dituliskan sebagai:

```
t << 500;
```

Jika `operator<<` dan `operator>>` diimplementasikan sebagai non-anggota (*friend*)[5] maka deklarasi kelas dan definisi kedua fungsi tersebut menjadi seperti terlihat pada Contoh 4.5.

Dengan didefinisikannya kedua operator sebagai fungsi non-anggota maka pengguna dapat memanfaatkan operasi tersebut **hanya** sebagai operator `<<` atau `>>`, dan **tidak** sebagai fungsi dengan nama `operator<<` atau `operator>>`. Pemanfaatan yang benar diberikan pada Contoh 4.6.

4.1 Fungsi anggota atau non-anggota

Terdapat perbedaan antara implementasi fungsi operator sebagai fungsi anggota atau sebagai fungsi non-anggota. Beberapa perbedaan tersebut berkaitan dengan:

[5]Dalam Contoh 4.5 pendeklarasian `friend` tidak diperlukan karena kedua fungsi memanfaatkan hanya fungsi anggota `public` `Push()` dan `Pop()`.

Contoh 4.5 Fungsi operator sebagai fungsi non-anggota

```
1   class Stack {
2      friend void operator<< (Stack&, int);
3      friend void operator>> (Stack&, int&);
4      //...
5      public:
6      //...
7      private:
8      //...
9   };
10
11  void operator<< (Stack& s, int v) {
12    s.Push (v);
13  }
14
15  void operator>> (Stack& s, int& v) {
16    s.Pop (v);
17  }
```

Contoh 4.6 Pemanfaatan fungsi `operator` yang dideklarasikan sebagai non-anggota

```
Stack s;

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

1. Jumlah parameter formal. Dalam implementasi sebagai fungsi non-anggota banyaknya parameter formal sama dengan banyaknya operand yang dibutuhkan oleh operator tersebut. Sedangkan dalam implementasi sebagai fungsi anggota, banyaknya parameter formal berkurang satu dari banyaknya operand yang dibutuhkan operator. Hal ini dapat terlihat dari contoh `Stack` di atas yang menggunakan operator biner `>>` dan `<<`.

2. Jenis parameter pertama. Karena fungsi anggota memiliki pointer implisit `this` maka fungsi operator yang diimplementasikan sebagai fungsi anggota hanya dapat dipanggil jika "ruas kiri" operator adalah objek dari kelas yang mendefinisikan operator tersebut. Misalkan pada contoh `Stack` perancang kelas menyediakan fungsi `operator+ (int)` yang dapat digunakan pemakai kelas untuk memperbesar kapasitas maksimum stack. Perhatikan Contoh 4.7.

Contoh 4.7 Pengaruh ruas kiri pada operator

Deklarasi kelas `Stack`

```
1   class Stack {
2     //...
3     void operator+ (int);
4     //
5   };
6
7   void Stack::operator+ (int dsize)
8   {
9      // kode untuk mengubah kapasitas stack sebesar "dsize"
10  }
```

dengan pemanggilan

```
1   Stack s, t;
2   int inc = 4;
3
4   s + 4; // OK: ekivalen dengan t = s.operator+ (4);
5   inc + s; // ERROR: tipe data tidak cocok
```

3. Sifat komutatif. Jika perancang kelas bermaksud untuk memanfaatkan operator biner secara komutatif dan salah satu operand bukan berasal dari kelas tersebut maka kelas tersebut harus mendeklarasikan dua fungsi dengan nama yang sama, dan karena sifat nomor (2) di atas, salah satu fungsi operator tersebut harus diimplementasikan sebagai fungsi non-anggota. Misalkan pada kelas `Stack` diinginkan `operator +` dapat dipanggil secara komutatif seperti pada Contoh 4.7, maka perancang kelas wajib mendeklarasikan kelas seperti pada Contoh 4.8. Untuk keseragaman penulisan, dalam contoh tersebut kedua fungsi diimplementasikan sebagai fungsi `friend`.

Kriteria penentuan implementasi sebagai fungsi anggota atau non-anggota suatu kelas:

Contoh 4.8 Implementasi operasi komutatif

```
1   class Stack {
2     //...
3     friend void operator+ (Stack&, int);
4     friend void operator+ (int, Stack&);
5     //
6   };
7
8   void operator+ (Stack& s, int m)
9   {
10     s.UbahKapasitas (m); // misalkan memanggil fungsi "private"
11  }
12
13  void operator+ (int m, Stack& s)
14  {
15     s.UbahKapasitas (m); // misalkan memanggil fungsi "private"
16  }
```

- 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 standard) memerlukan operand lvalue seperti *assignment* dan *arithmetic assignment* (`=`, `+=`, `++`, `*=`, dst) sebaiknya diimplementasikan sebagai fungsi anggota
- Operator yang (dalam tipe standard) tidak memerlukan lvalue (`+`, `-`, `&&`, dst) operand sebaiknya diimplementasikan sebagai fungsi non-anggota

Beberapa manfaat fungsi operator:

1. Operasi aritmatika terhadap objek-objek matematik lebih terlihat alami dan mudah dipahami oleh pembaca program. Bandingkanlah:

   ```
   c = a*b + c/d + e;
   ```

   dengan

   ```
   c = tambah (tambah (kali (a,b), bagi(c,d)), e);
   ```

2. Dapat menciptakan operasi input/output yang seragam dengan memanfaatkan *stream I/O* dari C++. Penjelasan lebih lanjut untuk *stream I/O* dapat dilihat pada Bagian 9.

3. Pengalokasian dinamik dapat dikendalikan perancang kelas melalui fungsi operator `new` dan `delete`.

4. Batas index pengaksesan array dapat dikendalikan lewat operator `[]`.

4.2 Perancangan *operator overloading*

- Operator yang secara otomatis didefinisikan untuk setiap kelas adalah assignment ('=') dan address of ('`&`').

- Dalam menerapkan operasi terhadap objek melalui fungsi operator, pilihlah operator yang memiliki makna yang paling mendekati makna aslinya

- *Overloading* **tidak dapat** dilakukan terhadap operator berikut: '`::`' (scope), '`.*`' (member pointer selector), '`.`' (member selector), '`?:`' (arithmetic if), dan `sizeof()`.

- Urutan presedensi operator tidak dapat diubah. Tabel 2 menunjukkan seluruh operator yang didefinisikan di dalam C++ beserta sifat asosiatif dan urutan presedensinya.

- Sintaks (arity, banyaknya operand) dari operator tidak dapat diubah

- Operator baru tidak dapat diciptakan. Jadi misalnya tidak dapat simbol '`<>`' digunakan untuk menyatakan operasi "tidak sama dengan" karena simbol ini tidak dikenal di dalam bahasa C++

- Operator `++` dan `--` tidak dapat dibedakan antara versi *postfix* dan *infix*

- Fungsi operator harus merupakan fungsi anggota atau paling sedikit salah satu argumen berasal dari kelas yang dioperasikan

4.3 Operator =

Fungsi `operator=` termasuk sebagai operator yang didefinisikan otomatis oleh kompilator, yaitu diimplementasikan sebagai operasi penyalinan bit-per-bit. Kadang kala efek ini tidak diinginkan oleh perancang kelas dan perancang kelas menuliskan secara eksplisit perilaku operasi ini.

Secara sepintas operasi *assignment* mirip dengan *copy constructor* seperti yang dijelaskan pada Bagian 3.4, namun kedua operasi tersebut memiliki keadaan awal yang berbeda. Pada operasi *assignment* a = b, kedua objek sudah tercipta sebelumnya sedangkan pada *copy constructor* seperti misalnya di dalam penciptaan dengan inisialisasi Stack s = t;, hanya satu objek (`t`) yang **sudah** tercipta, sedangkan objek `s` sedang dalam proses penciptaan.

Pada saat assignment, kedua objek mungkin memiliki atribut yang berbeda. Misalnya dalam contoh `Stack` kapasitas kedua stack dapat berbeda. Dalam keadaan seperti ini, perancang kelas dapat membatalkan operasi `operator=` atau mengubah atribut dari objek di

Precedence	Associativity	`Operator`	Function
17	R	`::`	global scope (unary)
	L	`::`	class scope (binary)
16	L	`->,.`	member selectors
	L	`[]`	array index
	L	`()`	function call
	L	`()`	type construction
15	R	`sizeof`	size in bytes
	R	`++.--`	increment, decrement
	R	`~`	bitwise NOT
	R	`!`	logical NOT
	R	`+,-`	unary minus, plus
	R	`*,&`	dereference, address-of
	R	`()`	type conversion (cast)
	R	`new,delete`	storage management
14	L	`->*,.*`	member pointer selectors
13	L	`*,/,%`	multiplicative operators
12	L	`+,-`	arithmetic operators
11	L	`<<,>>`	bitwise shift
10	L	`<,<=,>,>=`	relational operators
9	L	`==,!=`	equality, inequality
8	L	`&`	bitwise AND
7	L	`^`	bitwise XOR
6	L	`|`	bitwise OR
5	L	`&&`	logical AND
4	L	`||`	logical OR
3	L	`?:`	arithmetic if
2	R	`=,`	assignment operators
2	R	`*=,/=,%=`	
2	R	`+=,-=`	
2	R	`<<=,>>=`	
2	R	`&=,|=,^=`	
1	L	`,`	comma operator

Tabel 2: Operator Precedence and Associativity [Lippman, 1989]

ruas kiri. Dalam menyediakan layanan fungsi `operator=`, perancang kelas harus memperhatikan hal ini dan menentukan aturan yang akan ditetapkan pada `operator=`. Dalam contoh berikut ini, ditetapkan aturan bahwa objek `Stack` dapat di*assign* satu sama lain walaupun misalnya ukuran maksimum kedua stack tersebut berbeda.

```
Stack& Stack::operator= (const Stack& s)
   /* assign stack "s" ke stack "*this" */
{
   int i;

   delete [] data; // bebaskan memory 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;
}
```

4.3.1 Return value dari fungsi `operator=()`

Dalam bahasa C pemrogram dapat menuliskan perintah *assignment* berantai sbb:

```
int a, b, c;

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

Tabel 2 menunjukkan bahwa operator = memiliki sifat asosiatif kanan, sehingga aksi eksekusinya adalah seperti yang terlihat pada komentar program di atas.

Jika perancang kelas menginginkan operasi assignment berantai dapat juga diterapkan pada kelas yang dirancangnya, maka nilai kembali dari `operator=` harus ditulis dengan tepat. Jika dipandang sebagai `operator=` pada kelas, operasi berantai di atas dapat dituliskan sebagai:

```
ObjType a, b, c;

a.operator= ( b.operator= ( c.operator= (4) ) )
```

Agar `operator=` dapat dipanggil berantai maka `operator=` haruslah memiliki nilai kembali bertipe `ObjType` atau `ObjType&`. Jika `operator=` memiliki nilai kembali lain, misalnya `void`, maka operasi berantai tersebut di atas akan dideteksi sebagai kesalahan oleh kompilator. Jika perancang kelas berkeputusan untuk menggunakan nilai kembali bertipe `ObjType` atau `ObjType&`, maka yang harus diberikan sebagai nilai kembali adalah objek yang **sudah** mengalami operasi *assignment*, dalam hal ini adalah objek yang ditunjuk oleh pointer implisit `this`. Dengan demikian, di akhir alur kode fungsi `operator=` akan terlihat perintah `return *this`.

Setelah mengetahui manfaat fungsi `operator=` perancang kelas memahami bahwa untuk memperoleh operasi inisialisasi dan assignment yang benar, setiap kelas yang dirancangnya harus memperhatikan definisi dari **empat** fungsi[6] anggota berikut:

1. Constructor (*default constructor* maupun *user-defined constructor*),
2. Destructor,
3. Copy constructor, dan
4. Operasi *assignment*

Perhatikahlah Contoh 4.9.

Contoh 4.9 Fungsi anggota minimal yang harus dimiliki sebuah kelas

```
class XC {
  public:
   XC (...);                      // constructor
   XC (const XC&);                // copy constructor
   XC& operator= (const XC&);       // assignment
   ~XC();                         // destructor
   //... member public yang lain
   private:
   // ...
};
```

4.4 Operator[]

Dapat digunakan untuk melakukan subscripting terhadap kelas objek. Parameter kedua (index/subscript) dapat berasal dari tipe data apapun: integer, float, character, string, maupun tipe/kelas yang didefinisikan user.

5 Template Function

Seringkali kita membutuhkan suatu operasi yang sejenis terhadap tipe yang berbeda-beda. Sebagai contoh penentuan minimum dari dua objek. Definisi sederhana dari fungsi `min()` terhadap integer

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

[6]Keempat anggota ini dapat dianggap sebagai anggota yang "wajib" harus ada di dalam kelas. Selain itu, kelas juga "wajib" memiliki fungsi anggota untuk menangani masukan/keluaran melalui kelas `stream`, yang dibahas lebih lanjut di Bagian 9

Jika akan diterapkan juga terhadap `float`

```
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. Untuk mengatasi hal ini, "trick" 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 ekspansi berikut:

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

Akan diekspansi sebagai kode:

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

Untuk mengatasi hal ini, C++ menyediakan fasilitas fungsi template. Pendeklarasian fungsi template dituliskan dengan cara menuliskan prefix "`template <class XYZ>`" sebelum nama fungsi.

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

Contoh lain misalnya untuk menghitung nilai minimum dari sebuah array.

```
template <class Type>
Type min_arr (const Type x[], int size)
/* mencari minimum dari array */
{
   Type min_val;
   int i;

   min_val = x[0];
   for (i=1; i<size; i++) {
      if (x[i] < min_val)
         min_val = x[i];
   }
   return min_val;
}
```

Deklarasi fungsi template seperti di atas **belum** merupakan definisi fungsi (belum ada instruksi yang dihasilkan oleh kompilator pada saat deklarasi template di atas dibaca). Dalam hal ini dikatakan bahwa fungsi tersebut belum diinstansiasi. Instansiasi dilakukan pada saat kompilator mengetahui ada pemanggilan fungsi pada nama generik tersebut. Sebagai akibatnya, deklarasi fungsi template **harus** dituliskan di dalam file *header*.

Dengan adanya template, C++ menyediakan deklarasi fungsi generik / algoritma generik yang dapat diinstansiasi bergantung pada tipe yang diperlukan pemakai. Standard Template Library (STL) adalah pustaka C++ baru yang hampir seluruhnya didasarkan pada konsep fungsi generik ini. Keterangan lebih lanjut mengenai STL dapat ditemukan pada Bagian 10.

Beberapa hal yang harus diperhatikan dalam penggunaan fungsi template:

- Banyaknya nama tipe (kelas) yang dicantumkan di antara '<' dan '>' dapat lebih dari satu. Dalam penulisannya setiap nama tipe harus didahului oleh kata kunci `class`.

```
template <class T1, T2>  // SALAH: seharusnya <class T1, class T2>
void .....
```

- Nama tipe yang dicantumkan di antara '<' dan '>' harus tercantum sebagai *function signature*.

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

- Definisi fungsi `template` dapat disertai oleh definisi "non-template" dari fungsi tersebut. Dalam contoh berikut fungsi `min_arr` dideklarasikan sebagai fungsi generik namun untuk tipe `Complex` yang akan digunakan adalah fungsi non-generik yang dideklarasikan.

```
template <class Type>
Type min_arr (const Type x[], int size)
{
   // ...
}

Complex min_arr (const Complex x[], int size)
/* specialized instantiation dari min_arr
   untuk kelas Complex */
{
   // fungsi yang akan digunakan untuk
   // tipe Complex
}
```

6 Kelas Generik

Selain pada fungsi, konsep template dapat diterapkan juga pada kelas. Misalnya dengan menggunakan contoh `Stack` yang sudah diberikan dan dengan fasilitas template ini, perancang kelas dapat dengan mudah menciptakan Stack of integer, Stack of double, Stack of character, Stack of Complex, Stack of Process, Stack of String, dsb.

Fasilitas template C++ memberikan kemungkinan untuk membangkitkan kelas-kelas di atas melalui instansiasi dari kelas generik. Untuk menciptakan kelas generik, perancang kelas harus dapat mengidentifikasi parameter-parameter mana yang menentukan sifat kelas. Dalam contoh `Stack` yang diberikan parameter yang menentukan kelas adalah jenis `int` yang berkaitan dengan data yang disimpan di dalam `Stack`. Deklarasi kelas `Stack` yang ditunjukkan pada Contoh 3.2 dapat diubah menjadi deklarasi kelas `Stack` generik seperti pada Contoh 6.1

Contoh 6.1 Deklarasi kelas `Stack` generik

```
1   template <class Type>
2   class Stack {
3     public:
4      // ctor-cctor-dtor
5      Stack();       // default ctor
6      Stack (int);   // ctor dengan ukuran max stack
7      Stack (const Stack&); // cctor
8      ~Stack();
9
10     // services
11     void Push (Type);           // <=== parameter generik
12     void Pop (Type&);           // <=== parameter generik
13     int isEmpty() const;
14     int isFull() const;
15     // operator
16     Stack& operator= (const Stack&);
17     void operator<< (Type);     // <=== parameter generik
18     void operator>> (Type&);    // <=== parameter generik
19
20    private:
21     const int defaultStackSize = 500; // ANSI: tidak boleh inisialisasi
22     int topStack;
23     int size;
24     Type *data;                 // <=== parameter generik
25  };
```

Untuk menciptakan objek dari kelas generik, pemrogram menuliskan deklarasi objek dengan sintaks:

kls-generik < *tipe-instansiasi* > *objek* ;

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

CATATAN:

- 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<Type>::`. Hal ini harus dilakukan untuk seluruh fungsi anggota kelas tersebut. Sebagai contoh, konstruktor dan fungsi anggota `Push()` dituliskan sebagai fungsi *template* berikut ini:

```
template <class Type>
Stack<Type>::Stack ()
{
   size = defaultStackSize;
   topStack = 0;
   data = new TYPE [size];
}

template <class Type>
void Stack<Type>::Push (Type item)
{
   // ...
   if (!isFull()) {
      data [topStack] = item;
      topStack++;
   }
   // ...
}
```

  Sebelumnya dijelaskan bahwa dalam penulisan kelas ke dalam file, bagian deklarasi kelas dituliskan ke file `X.h` dan bagian definisi fungsi-fungsi anggota dituliskan ke file `X.cc`. Jika kelas generik digunakan untuk mendeklarasikan kelas, maka baik **deklarasi kelas generik** maupun **definisi fungsi generik** dituliskan ke dalam file `X.h`. Sehingga untuk contoh `Stack` di atas, keduanya dituliskan di dalam file `Stack.h`.

- Di luar konteks definisi kelas generik, nama tipe yang dapat digunakan (misalnya oleh fungsi, deklarasi variabel/objek, dsb.) adalah nama tipe hasil instansiasi. Dalam contoh di atas tipe hasil instansiasi adalah `Stack<int>`, `Stack<double>`, and `Stack<Complex>`.

  Hal ini juga berlaku pada fungsi anggota kelas. Jika misalkan ada fungsi anggota kelas generik `Stack::Reverse()` yang memerlukan objek lokal bertipe `Stack` yang **generik** maka deklarasinya adalah:

```
template <class Type>
void Stack<Type>::Reverse() {
   Stack<Type> stemp;   // objek lokal yang generik

   // ...algoritma dari Reverse()...
}
```

Untuk efisiensi pemanfaatan kelas generik, sebaiknya digunakan *member initialization list* di dalam constructor maupun copy constructor.

7 Pewarisan & Penurunan Kelas

Konsep-konsep yang berkaitan erat dengan pemrograman berorientasi objek adalah: objek, kelas, pewarisan (*inheritance*), *polymorphism*, dan *dynamic binding*. Di antara konsep-konsep tersebut, pewarisan adalah konsep yang merupakan ciri unik dari model pemrograman berorientasi objek. Konsep-konsep lainnya juga ditemukan dalam model pemrograman lainnya. Tanpa inheritance, pemanfaatan bahasa C++ hanyalah sekedar *abstract data type programming*, bukan *object-oriented programming*.

Konsep pewarisan memungkinkan perancang kelas untuk mendefinisikan dan mengimplementasikan sebuah kelas berdasarkan kelas-kelas yang sudah ada. Konsep ini jugalah yang mendukung fasilitas penggunaan ulang (*reuse*). Jika sebuah kelas A mewarisi kelas lain B, maka A merupakan kelas turunan (*derived class/ subclass*) dan B merupakan kelas dasar (*base class/superclass*). Seluruh anggota (data & fungsi) di B akan berada juga di kelas A, **kecuali** constructor, copy constructor, destructor, dan `operator=`, karena setiap kelas memiliki ctor, cctor, dtor, dan `operator=` sendiri. Akibat dari pewarisan, kelas A akan memiliki dua bagian: bagian yang diturunkan dari B dan bagian yang didefinisikan sendiri oleh kelas A dan bersifat spesifik terhadap A. Dalam konteks ini, objek B yang muncul di dalam A dapat dipandang sebagai sub-objek dari.

Penurunan kelas dituliskan dalam C++ sebagai berikut:

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

Mode-pewarisan[7] mempengaruhi tingkat pengaksesan setiap anggota (fungsi/data) kelas dasar jika diakses melalui fungsi **di luar** kelas dasar maupun di luar kelas turunan. Bagi fungsi di dalam kelas turunan, semua anggota (fungsi/data) di dalam bagian `public` atau `protected` selalu dapat diakses. Perubahan tingkat pengaksesan akibat pewarisan/penurunan ini ditunjukkan pada Tabel 3. Pada tabel tersebut tingkat akses `private` dari kelas dasar **tidak ditunjukkan** karena walaupun diwariskan dari kelas dasar ke kelas turunan, anggota yang berada di bagian `private` **tidak dapat diakses** oleh kelas turunan, walaupun anggota tersebut diwariskan ke kelas turunan. Tabel 3 di atas menunjukkan

[7]Jika tidak dituliskan, mode-pewarisan dianggap sebagai `private`

bahwa, misalnya, anggota di kelas dasar yang berada pada bagian `public` jika diturunkan secara `protected` akan menjadi anggota yang bersifat `protected` di kelas turunan.

Tingkat-Akses di kelas dasar	Mode-pewarisan		
	`private`	`protected`	`public`
private	private	private	private
protected	private	protected	protected
public	private	protected	public

Tabel 3: Tingkat pengaksesan pada kelas turunan

Makna `private:`, `protected:`, dan `public:` dalam deklarasi kelas sudah dijelaskan pada Table 1. Dalam suatu kelas, jika tingkat pengaksesan `private:` dipandang sebagai "sangat tertutup" (hanya fungsi anggota kelas tersebut yang dapat mengakses anggota yang disebutkan pada bagian ini) dan `public:` sebagai "sangat terbuka" (fungsi manapun, di dalam atau di luar kelas dapat mengakses anggota dalam bagian ini), maka `protected:` dapat dipandang sebagai "setengah terbuka" atau "setengah tertutup" (hanya kelas turunan yang dapat mengakses anggota pada bagian ini).

Dengan memanfaatkan contoh `Stack` yang sudah ada, misalkan perancang kelas akan membuat kelas baru `GStack` ("*growing stack*") yang berperilaku seperti stack namun memiliki kemampuan untuk memperbesar dan memperkecil kapasitasnya secara otomatis. Dengan kemampuan ini, jika operasi `Push()` dilakukan pada stack yang sudah penuh, kapasitas stack tersebut akan diperbesar secara otomatis dengan sejumlah memori tertentu (ditentukan oleh perancang). Demikian juga jika operasi `Pop()` dilakukan dan kapasitas yang tidak terpakai melebihi jumlah tertentu maka secara otomatis kapasitas stack tersebut diperkecil.

Dari penjelasan di atas terlihat bahwa kelas `GStack` dapat diwariskan dari kelas `Stack`. Untuk menciptakan perilaku seperti dijelaskan di atas, perancang kelas `GStack` harus melakukan hal-hal berikut:

1. Mengubah perilaku `Pop()` dan `Push()` yang ada pada kelas `Stack`

2. Menambahkan anggota data yang digunakan untuk menyimpan faktor penambahan/penciutan kapasitas stack

Karena ada anggota yang harus ditambahkan, perancang kelas turunan `GStack` harus mendefinisikan anggota di dalam kelas tersebut. Misalkan faktor penambahan/penciutan dicatat dalam anggota `gs_unit` (*grow/shrink unit*). Deklarasi kelas `GStack` ditunjukkan pada Contoh 7.1.

Misalkan perancang kelas `GStack` menetapkan `GStack::Push()` akan memeriksa apakah stack penuh atau tidak. Jika ya, berarti kapasitas stack harus diperbesar. Untuk keperluan ini `GStack::Push()` cukup memanfaatkan `Stack::isFull()`. Sebaliknya, `GStack::Pop()` harus memeriksa apakah setelah operasi "pop" terjadi kekosongan kapasitas. Jika ya, berarti kapasitas stack harus diperkecil. Untuk mengetahui hal ini, `GStack::Pop()` harus mengetahui nilai `size` dan `topStack`, padalah kedua data tersebut berada pada bagian `private:`. Sebagai akibatnya, kedua data tersebut harus dipindahkan ke bagian `protected:`.

Contoh 7.1 Deklarasi kelas `GStack` yang diturunkan dari `Stack`

```
1   // File GStack.h
2   // Deklarasi kelas GStack
3
4   #ifndef GSTACK_H
5   #define GSTACK_H
6
7   #include "Stack.h"
8
9   class GStack : public Stack {
10    public:
11     // ctor, cctor, dtor, oper=
12     GStack();
13     GStack(const GStack&);
14     ~GStack();
15     GStack& operator=(const GStack&);
16
17     // redefinition of Push & Pop
18     void Push (int);
19     void Pop (int&);
20     void operator<<(int);
21     void operator>>(int&);
22
23    private:
24     int gs_unit;
25
26     // fungsi untuk mengubah kapasitas
27     void Grow();
28     void Shrink();
29
30  };
31  #endif GSTACK_H
```

Contoh 7.2 Definisi fungsi anggota `GStack`

```
1  // File GStack.cc
2  // Definisi fungsi-fungsi anggota kelas GStack
3
4  #include <stdio.h>
5  #include "GStack.h"
6
7  GStack::GStack() {
8     gs_unit = 10;
9  }
10
11 GStack::GStack(const GStack& s) : Stack (s) {
12    gs_unit = s.gs_unit;
13 }
14
15 GStack::~GStack() {
16    /* tidak ada yang perlu dilakukan */
17 }
18
19 GStack& GStack::operator=(const GStack& s) {
20    Stack::operator= (s);
21    gs_unit = s.gs_unit;
22    return *this;
23 }
24
25 void GStack::Push (int x) {
26    if (isFull()) Grow();
27    Stack::Push(x);
28 }
29
30 void GStack::Pop (int& x) {
31    Stack::Pop(x);
32    if (size - topStack > gs_unit) Shrink();
33 }
34
35 void GStack::operator<<(int x) { Push(x); }
36
37 void GStack::operator>>(int& x) { Pop(x); }
38
39 void GStack::Grow() {
40    // ... kode untuk memperbesar kapasitas
41 }
42
43 void GStack::Shrink() {
44    // ... kode untuk memperkecil kapasitas
45 }
```

Akibat pewarisan, objek yang diciptakan dari kelas `GStack` juga memiliki data anggota dari kelas `Stack` seperti `topStack`, `data`, dsb. Demikian juga dengan fungsi anggota `Stack`. Kelas `GStack` juga secara otomatis memiliki fungsi anggota `isEmpty()` dan `isFull()`. Pemilikan anggota dari kelas dasar **belum tentu** berarti kelas dasar dapat mengakses anggota tersebut karena hal ini bergantung pada hak akses yang diterapkan di kelas dasar.

7.1 Ctor, dtor, cctor, dan `operator=` Kelas Dasar

Bagi kelas turunan, komponen kelas dasar merupakan subobjek yang dimiliki kelas turunan. Sebagai akibatnya, pada penciptaan objek dari kelas turunan, konstruktor kelas dasar akan diaktifkan **sebelum** konstruktor kelas turunan. Sebaliknya pada saat pemusnahan objek kelas turunan, destruktor kelas dasar dipanggil **setelah** destruktor kelas turunan.

Penanganan *copy constructor* pada kelas turunan, dibedakan menjadi 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

Pada kasus (1) cctor kelas dasar akan dipanggil, inisialisasi kelas turunan dilakukan dengan *default* cctor (*bitwise copy*). Pada kasus (2) dan (3), cctor dari kelas dasar **tidak dipanggil**, inisialisasi kelas dasar menjadi tanggung jawab kelas turunan.

Penginisialisasian kelas dasar oleh kelas turunan melalui ctor atau cctor dilakukan melalui *constructor initialization list* seperti yang dijelaskan pada Bagian 3.7. Dalam contoh `GStack` di atas, cctor harus didefinisikan dengan menggunakan nama kelas dasar. Perhatikanlah cctor di dalam Contoh 7.2 di baris 11–13.

Assignment ditangani dengan cara yang sama seperti inisialisasi, yaitu jika kelas turunan **tidak** mendefinisikan `operator=` maka `operator=` dari kelas dasar akan dipanggil (jika ada). Sebaliknya, jika kelas turunan mendefinisikan `operator=` maka operasi *assignment* dari kelas dasar menjadi tanggung jawab kelas turunan. Perhatikanlah baris 19–23 dari Contoh 7.2.

7.2 *Polymorphism* dan Fungsi Anggota Virtual

Karena ciri-ciri yang ada di kelas `Stack` juga muncul di kelas `GStack`, objek-objek yang berasal dari kelas `GStack` memiliki sifat sebagai `GStack` dan **sekaligus** sebagai `Stack`. Karakteristik terakhir ini berkaitan dengan *polymorphism* (*poly* = banyak, *morph* = bentuk). Dalam C++ *reference* (untuk selanjutnya disingkat "ref") dan *pointer* (disingkat "ptr") dapat bersifat polimorfik. Sifat polimorfik mengakibatkan sebuah ref/ptr memiliki tipe statik dan tipe dinamik. Tipe statik adalah tipe pada saat ref/ptr tersebut dideklarasikan di dalam program. Tipe dinamik adalah tipe yang dapat berubah pada saat eksekusi program, bergantung pada objek yang ditunjuk/diacu ptr/ref tersebut. Berdasarkan tipe dinamiknya ini, ptr/ref dalam C++ dapat memanggil anggota kelas yang berasal dari kelas berbeda-beda. Penentuan anggota mana yang dipanggil, dilakukan pada saat eksekusi (*dynamic binding*).

Hal ini yang membedakan antara pemberian parameter secara *value* dengan *pointer/reference*. Dalam contoh, berikut, perintah `s.Push()` yang dipanggil oleh `FuncVal` adalah `Stack::Push`

walaupun parameter `s` bertipe `GStack`. Karena parameter formal bukan berupa sebuah *pointer* maupun *reference*.

```
#include <GStack.h>

void FuncVal (Stack s) {
   s.Push (10);
}
```

Jika digunakan *pointer* atau *reference* seperti pada Contoh 7.3, maka parameter formal dapat bersifat polimorfik. Dalam contoh tersebut, parameter formal memiliki tipe statik dan dinamik seperti ditunjukkan pada Tabel 4. Untuk memanfaatkan sifat polimorfik fungsi yang dipanggil melalui ptr/ref harus dideklarasikan sebagai fungsi anggota *virtual*. Pendeklarasian *virtual* ini harus dicantumkan pada kelas dasar. Dalam hal ini, pada kelas `Stack`. Modifikasi deklarasi kelas `Stack` diberikan pada Contoh 7.4.

Pendefinisian ulang fungsi anggota *virtual* di kelas turunan harus menggunakan nama, jenis nilai kembali, dan *function signature* yang sama dengan di kelas dasar.

Parameter Formal	Tipe Statik	Tipe Dinamik
`t`	`Stack*`	`Stack*` dan `GStack*`
`u`	`Stack&`	`Stack&` dan `GStack&`

Tabel 4: Tipe statik dan dinamik *pointer* dan *reference*

.

7.2.1 Virtual Destructor

Misalkan ada sebuah array dari pointer ke `Stack` dan kelas `GStack` diturunkan dari `Stack` seperti pada contoh di atas. Setiap elemen dalam array tersebut dapat berisi objek bertipe `Stack` maupun `GStack`. Dalam contoh berikut ini:

```
Stack* sp [MAX_ELEM];

// ... kode-kode lain

for (i=0; i<MAX_ELEM; i++)
  delete sp[i];
```

destruktor mana yang akan dipanggil oleh `delete sp[i];`? Dalam contoh yang sudah diberikan, destruktor yang akan dipanggil adalah `Stack:: Stack()`. Seharusnya, jika elemen array `sp` menunjuk ke objek berjenis `GStack` destruktor yang harus dipanggil melalui elemen tersebut adalah `GStack:: GStack()`. Untuk mengatasi hal ini, destruktor dari kelas dasar (`Stack`) harus dideklarasikan sebagai `virtual` seperti dalam contoh berikut:

```
class Stack {
 public:
```

Contoh 7.3 Sifat Polimorfik melalui *pointer* atau *reference*

```
1   #include <GStack.h>
2
3   void FuncPtr (Stack *t) {
4      t->Push (10);
5   }
6
7   void FuncRef (Stack& u) {
8      u.Push (10);
9   }
10
11  main()
12  {
13     Stack s;
14     GStack gs;
15
16     FuncRef (s);     // u.Push akan memanggil Stack::Push()
17     FuncRef (gs);    // u.Push akan memanggil GStack::Push()
18
19     FuncPtr (&s)     // t->Push akan memanggil Stack::Push();
20     FuncPtr (&gs);   // t->Push akan memanggil GStack::Push()
21  }
```

Contoh 7.4 Modifikasi deklarasi kelas `Stack`

```
1   // File: Stack.h
2   // Deklarasi kelas Stack
3
4   #ifndef STACK_H
5   #define STACK_H
6   #include "StackExp.h"
7
8   class Stack {
9    public:
10     // ctor, cctor, dtor, & oper=
11     Stack();                // default ctor
12     Stack(int);             // user-defined ctor
13     Stack(const Stack&);    // cctor
14     ~Stack();               // dtor
15
16   public:
17     // services
18     virtual void Push (int);         // <=== penambahan "virtual"
19     virtual void Pop (int&);         // <=== penambahan "virtual"
20     int isEmpty() const;
21     int isFull() const;
22     void Cetak() const;
23
24     // operator
25     Stack& operator= (const Stack&);
26     virtual void operator<< (int);  // "Push" <=== penambahan "virtual"
27     virtual void operator>> (int&); // "Pop"  <=== penambahan "virtual"
28
29   private:
30     const int defaultStackSize = 1000;
31     int *data;
32
33   protected:   // dipindahkan dari private ke protected
34     int topStack;
35     int size;
36  };
37  #endif STACK_H
```

```
  // ctor, dtor, cctor
  //...
  virtual ~Stack();
  //
};
```

Di dalam kelas turunan (`GStack`), destruktor **tidak perlu** dideklarasikan `virtual` karena destruktor kelas yang diturunkan dari kelas lain yang memiliki dtor virtual otomatis bersifat virtual.

7.2.2 Abstract Base Class (ABC)

Abstract Base Class adalah kelas dasar yang berkaitan dengan konsep abstrak. Misalnya kelas `Stack`, `Queue`, `List`, `Tree` dapat dikelompokkan menjadi sebuah kelas abstrak `DataStore` yang merepresentasikan kelas penyimpan data. Kelas tersebut abstrak karena jika ada orang yang meminta anda

> "Tolong ciptakan sebuah `DataStore`!"

anda tidak dapat melakukannya karena tidak tahu harus menciptakan objek yang berasal dari jenis mana.

Misalkan dalam kelas `DataStore` tersebut dideklarasikan prosedur `Clear()` untuk menghapus seluruh item yang tersimpan di dalam `DataStore`. Agar seluruh kelas turunan `DataStore` mewarisi prosedur ini dan pemanggilannya dilakukan secara dinamik, maka `Clear()` dideklarasikan sebagai fungsi virtual di dalam kelas `DataStore`. Operasi penghapusan ini tidak dapat diimplementasikan di dalam kelas `DataStore` karena cara penghapusan `List` misalnya berbeda dengan cara penghapusan `Tree`. Sebagai akibatnya kelas `DataStore` hanya menyatakan keberadaan `Clear()` **tanpa** mendefinisikan aksi yang dilakukannya. Dalam bahasa C++, hal ini dinyatakan sebagai *pure virtual function* sebagai berikut:

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

Sebuah kelas dasar yang memiliki *pure virtual function* disebut sebagai kelas dasar abstrak (*Abstract Base Class = ABC*). Pemakai kelas **tidak dapat** mendeklarasikan objek yang berasal dari sebuah ABC, melainkan dari kelas non-abstrak yang diturunkan dari kelas dasar abstrak tersebut.

Di dalam kelas non-abstrak yang mewarisi kelas dasar abstrak, perancang kelas wajib mendefinisikan fungsi-fungsi *pure virtual* yang ada di kelas dasar abstrak. Sebagai contoh jika kelas-kelas `Stack` dan `Tree` diturunkan dari kelas `DataStore` maka perancang kelas harus mendefinisikan fungsi `Clear()` seperti dalam contoh berikut:

```
// 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 ();
};
```

Di dalam file `Stack.cc` dan `Tree.cc` definisi dari `Stack::Clear()` dan `Tree::Clear()` harus dituliskan. Isi instruksi kedua fungsi tersebut mungkin berbeda.

8 Exception

Seringkali penanganan kesalahan yang muncul pada saat eksekusi program membuat penulis program harus menambahkan instruksi-instruksi tertentu di dalam programnya yang dapat membuat program menjadi rumit. Sebagai contoh jika ada sebuah fungsi yang memiliki nilai kembalian bertipe integer dan dalam eksekusi fungsi tersebut terjadi kesalahan maka biasanya penulis program akan memberikan nilai kembalian "khusus" yang harus diperiksa oleh instruksi pemanggil fungsi tersebut. Dalam hal ini dalam badan fungsi tersebut akan mengandung instruksi yang kurang lebih seperti ditunjukkan pada Contoh 8.1. Sehingga untuk menangani kesalahan yang mungkin muncul pemanggil `LakukanAksi()` harus menuliskan instruksi seperti ditunjukkan pada Contoh 8.2.

Contoh 8.1 Penyampaian informasi kesalahan melalui `return`

```
1   int LakukanAksi ()
2   {
3      // ...
4      if ( .... ) // periksa kondisi kesalahan
5         return NILAI_KHUSUS;
6      else
7         return NILAI_FUNGSI;
8   }
```

Kesalahan dalam eksekusi program seperti di atas seringkali disebut sebagai *exception*. C++ menyediakan fasilitas khusus untuk menangani kesalahan melalui `throw`, `catch` dan

Contoh 8.2 Penanganan kesalahan setelah pemanggilan `LakukanAksi()`

```
1   if (LakukanAksi () == NILAI_KHUSUS) {
2      // ... jalankan instruksi untuk menangani kesalahan
3   }
4   else {
5      // instruksi-1
6      // instruksi-2
7      // ...
8      // instruksi-n
9   }
```

`try`. Untuk memudahkan pengertian, `throw` dapat dianggap sebagai "`return` dengan exception". Dengan kata lain, `return` menyatakan keadaan bahwa fungsi selesai dengan normal, sedangkan `throw` menyatakan keadaan bahwa fungsi "selesai" dengan **tidak** normal. Dengan penjelasan di atas, sebuah fungsi dapat dianggap memiliki dua "pintu keluar" (untuk keadaan normal dan abnormal). Contoh 8.3 menunjukkan pemanfaatan `throw`. Perhatikanlah bahwa jenis ekspresi yang dinyatakan dalam `throw` **tidak harus sama** dengan jenis nilai kembali fungsi. Dalam hal ini jenis "pintu keluar" normal berbeda dengan jenis "pintu keluar" abnormal. Tipe ekspresi yang dinyatakan pada `throw` dapat berupa tipe apapun yang dideklarasikan di dalam program (tipe primitif: int, float, dsb. maupun tipe kelas yang ada).

Contoh 8.3 Penyampaian informasi kesalahan melalui `throw`

```
1   int LakukanAksi ()
2   {
3      // ...
4      if ( .... ) // periksa kondisi kesalahan
5         throw "Ada kesalahan";
6      else
7         return NILAI_FUNGSI;
8   }
```

Untuk menerima nilai dari "pintu keluar abnormal", program harus menggunakan `catch` yang dituliskan setelah sebuah blok `try`. Dengan fasilitas ini, Contoh 8.2 dapat dituliskan menjadi instruksi pada Contoh 8.4. Terlihat bahwa instruksi menjadi lebih mudah dibaca.

Perintah `catch` yang dituliskan setelah blok `try` disebut juga sebagai *exception handler* untuk blok `try` tersebut. Secara umum, sebuah blok `try` dapat memiliki lebih dari satu *exception handler*, masing-masing untuk menangani tipe *exception* yang berbeda. Setiap *handler* dapat dianggap sebagai fungsi yang memiliki *signature* tertentu seperti yang dituliskan setelah kata kunci `catch`. Jika penulis program ingin menuliskan sebuah *handler* yang dapat menerima *semua* jenis *exception*, `catch` dapat dituliskan dengan tipe ellipsis ("`...`"). Hal ini mirip dengan prototipe fungsi yang dapat menerima parameter dalam berbagai jenis.

Contoh 8.4 Pemanfaatan `try` dan `catch`

```
1   try {
2      LakukanAksi ();
3      // instruksi-1
4      // instruksi-2
5      // ...
6      // instruksi-n
7   }
8   catch (const char*) {
9      // ... jalankan instruksi untuk menangani kesalahan
10  }
11
12  // eksekusi berlanjut pada bagian ini ...
```

```
try {
   // instruksi...
}
catch (StackErr&) {
   // handler untuk tipe "StackErr"
}
catch (...) {
   // handler untuk semua jenis exception lainnya
}
```

Jika pada saat instruksi di dalam blok `try` dijalankan terjadi *exception* maka salah satu *handler* yang tersedia akan dipilih berdasarkan kecocokan tipe dari *exception* (seperti yang dinyatakan dalam ekspresi `throw`) dan tipe *handler* (seperti yang dinyatakan dalam `catch`) yang ada. Jika ada *handler* yang cocok, maka instruksi di dalam *handler* tersebut dijalankan dan handler yang lain tidak akan dipilih. Setelah instruksi di dalam *handler* dijalankan eksekusi dari blok `try` `tidak dilanjutkan` tetapi dilanjutkan dengan eksekusi yang dituliskan setelah bagian `try-catch` tersebut.

Berikut ini akan diberikan contoh pemanfaatan *exception* dalam kelas `Stack`. Misalkan perancang kelas membuat kelas `StackExp` untuk menunjukkan *exception* yang terjadi di dalam kelas `Stack`. Di dalam kelas tersebut didefinisikan sejumlah pesan kesalahan (dalam tabel) serta pencacah (*counter*) yang dapat digunakan untuk mencatat berapa kali muncul *exception*. Deklarasi kelas `StackExp` ditunjukkan pada Contoh 8.5.

Contoh 8.5 Deklarasi kelas `StackExp`

```
1   #ifndef STACKEXP_H
2   #define STACKEXP_H
3
4   const int STACK_EMPTY = 0;
5   const int STACK_FULL = 1;
6
7   class StackExp {
8    public:
9      // ctor, cctor, dtor, oper=
10     StackExp (int);      // ctor: initialize msg_id
11     StackExp (const StackExp&);
12     // dalam contoh ini, StackExp tidak memerlukan dtor dan oper=
13     //   ~StackExp();
14     //   StackExp& operator= (const StackExp&);
15
16     // services
17     void DisplayMsg () const;
18     static int NumException ();
19
20   private:
21     // static member, shared by all objects of this class
22     static int num_ex;  // pencacah jumlah exception
23     static char* msg[]; // tabel pesan kesalahan
24
25     const int msg_id; // nomor kesalahan
26  };
27  #endif STACKEXP_H
```

Definisi fungsi-fungsi anggota `StackExp` serta inisialisasi anggota-anggota data statik dapat dilihat pada Contoh 8.6. Definisi fungsi-fungsi anggota `StackExp` serta inisialisasi

Dengan kelas `StackExp` di atas, perancang kelas `Stack` dapat mengubah fungsi anggota `Push()` dan `Pop()` untuk menggunakan `throw` seperti pada Contoh 8.7.

Contoh 8.6 Definisi fungsi anggota dan inisialisasi data statik `StackExp`

```
1   #include <stdio.h>
2   #include "StackExp.h"
3   #include <stdio.h>
4
5   int StackExp::num_ex = 0;
6   char* StackExp::msg [] = { "Stack is empty!", "Stack is full!" };
7
8   StackExp::StackExp (int x) : msg_id (x) {
9     num_ex++; // increase the exception counter
10  }
11
12  StackExp::StackExp (const StackExp& s) : msg_id (s.msg_id) { }
13
14  void StackExp::DisplayMsg() const {
15     printf ("%s\n", msg[msg_id]);
16  }
17
18  int StackExp::NumException () {
19     return num_ex;
20  }
```

Setelah perancang kelas menambahkan pembangkit *exception* melalui perintah `throw`, pemakai kelas `Stack` dapat menangani kesalahan operasi `Push()` dan `Pop()` dengan menggunakan `try-catch` seperti dalam Contoh 8.8. Perhatikanlah bahwa nama parameter "`s`" yang ada pada baris 14 independen terhadap nama objek "`s` yang dideklarasikan pada baris 6. Kedua nama tersebut *tidak harus* sama.

9 C++ I/O Library

Selain menggunakan `printf()`, program dalam bahasa C++ dapat menggunakan kelas `stream` untuk perintah masukan/keluaran. Sama seperti program C yang secara otomatis memiliki tiga variabel `stdin`, `stdout`, dan `stderr` yang bertipe `FILE *`, program bahasa C++ juga otomatis memiliki objek `cin`, `cout`, dan `cerr` yang bertipe `stream`.

Mengapa C++ mendefinisikan lagi fasilitas masukan/keluaran ini? Andaikan di dalam kelas stack generik ada fungsi anggota `Cetak()` yang dapat digunakan pemakai kelas untuk mencetak seluruh isi stack. Bagaimana fungsi anggota Cetak() ini diimplementasikan? Jika elemen stack berasal dari tipe primitif (int, char, float, dsb.) perancang kelas dapat menggunakan `printf()`, tetapi jika elemen stack bertipe non-primitif `printf()` tidak dapat

Contoh 8.7 Hasil pengubahan kelas `Stack` setelah memanfaatkan `StackExp`

```
1  // file: Stack.cc
2  // deskripsi: kelas Stack dengan exception handling
3
4  #include "StackExp.h"
5
6  void Stack::Push (int x) {
7     if (isFull()) throw (StackExp (STACK_FULL));   // raise exception
8     else {
9        /* algoritma Push() */
10    }
11 }
12
13 void Stack::Pop (int& x) {
14    if (isEmpty()) throw (StackExp (STACK_EMPTY));  // raise exception
15    else {
16       /* algoritma Pop() */
17    }
18 }
```

Contoh 8.8 Pemanfaatan kelas `StackExp`

```
1  #include "Stack.h"
2  #include <stdio.h>
3
4  main()
5  {
6     Stack s;
7     int n;
8
9     try {
10       // ...
11       s << 10;
12       // ...
13    }
14    catch (StackExp& s) {
15       s.DisplayMsg();
16    }
17
18    n = Stack::NumException();
19    if (n > 0)
20       printf ("Muncul %s stack exception\n", n);
21 }
```

digunakan untuk mencetak tipe tersebut. Dalam keadaan inilah kelas `stream` diperlukan oleh perancang kelas.

Contoh 9.1 Definisi fungsi `Cetak()` pada kelas `Stack` generik

```
1   template <class Type>
2   void Stack<Type>::Cetak () const {
3     if (isEmpty())
4        cout << "Stack tidak memiliki elemen" << endl;
5     else {
6        int i;
7
8        cout << "Stack berisi " << topStack
9             << " elemen berikut:" << endl;
10       for (i = 0; i < topStack; i++)
11          cout << data [i] << " ";
12       cout << endl;
13    }
14  }
```

Dengan memanfaatkan kelas `stream`, definisi dari fungsi anggota `Cetak()` adalah seperti ditunjukkan pada Contoh 9.1. Misalkan perancang kelas menuliskan jumlah elemen dan lalu setiap elemen yang ada di dalam stack tersebut ke `cout`. Untuk tipe primitif (int, char, float, dsb.) `cout` tahu bagaimana mencetaknya. Bagaimana `cout` harus mencetak objek yang non-primitif, misalnya kelas `Process`. Jika stack generik tersebut diinstansiasi menjadi objek `Stack<Process>` maka pada saat `Cetak()` mencetak objek yang bertipe `Process` melalui `cout << data[i]` pada baris 10, kompilator akan mencari fungsi anggota dari kelas `cout`

```
cout.operator<< (const Process&);
```

atau fungsi non-anggota

```
operator<< (stream&, const Process&);
```

Salah satu dari fungsi tersebut **harus** didefinisikan oleh **perancang kelas** `Process`. Untuk menambah fungsi anggota (yang pertama) pada `stream`, perancang kelas harus mengubah deklarasi kelas `stream` yang tidak mungkin dilakukan. Jadi yang dapat dilakukan adalah mendefinisikan fungsi non-anggota (yang kedua). Agar fungsi tersebut dapat mengakses anggota private dari kelas `Process` maka, fungsi tersebut dideklarasikan sebagai `friend` dari kelas `Process`. Untuk memungkinkan *cascading* operasi keluaran ke `cout`, maka nilai kembali fungsi tersebut harus bertipe `stream&`. Definisi fungsi non-anggota untuk mencetak isi kelas `Process` tersebut ditunjukkan pada Contoh 9.2.

Dengan demikian, fungsi "wajib" sebuah kelas bertambah satu lagi menjadi:

1. *Constructor*

Contoh 9.2 Fungsi non-anggota `operator<<` untuk mencetak `Process`

Prototipe dalam kelas `Process`

```
1   #include <stream.h>
2
3   class Process {
4     // ...
5
6     // overload oper<< untuk pencetakan kelas Process
7     friend stream& operator<< (stream&, const Process&);
8
9     // ...
10  };
```

```
1   stream& operator<< (stream& s, const Process& p)
2   {
3      // pencetakan isi Process "p" ke stream "s"
4   }
```

2. *Destructor*
3. *Copy Constructor*
4. Operator *assignment*
5. Operator masukan/keluaran `stream`

Ada beberapa keuntungan yang diperoleh dari penggunaan kelas `stream` dibandingkan dengan fungsi-fungsi di `stdio.h`:

1. *Type safety*: tipe objek yang dimanipulasi diketahui oleh kompilator pada saat kompilasi
2. *Extensible*: fungsi `operator<<` dan `operator>>` terhadap kelas `stream` dapat di-overload untuk objek yang diciptakan pemakai.
3. Mengurangi kesalahan pemakaian karena tidak adanya format (seperti `%d, %c, %s`) yang harus digunakan pemakai.

9.1 Input/Output dengan kelas `stream`

Stream I/O dalam C++ mendeklarasikan tiga kelas berikut:

1. `istream`: menangani stream input. `cin` merupakan objek dari kelas `istream` yang diasosiasikan dengan standard input
2. `ostream`: menangani stream output.

(a) `cout` merupakan objek dari kelas `ostream` yang diasosiasikan dengan standard output

(b) `cerr` merupakan objek dari kelas `ostream` yang diasosiasikan dengan standard error

3. `iostream`: menangani stream dua arah (input/output) (diturunkan dari `istream` dan `ostream`)

Untuk manipulasi terhadap file disediakan kelas-kelas berikut:

1. `ifstream` yang diturunkan dari kelas `istream`
2. `ofstream` yang diturunkan dari kelas `ostream`
3. `fstream` yang diturunkan dari kelas `iostream`

9.2 Output

Penulisan keluaran dilakukan dengan menggunakan operator "`<<`" seperti pada contoh-contoh yang sudah diberikan. Fungsi `endl` adalah untuk menyatakan *end of line* sehingga stream akan menghasilkan kombinasi karakter *carriage return, line-feed* atau *line-feed* saja.

9.3 Input

Pembacaan input, pada dasarnya mirip dengan penulisan output hanya menggunakan operator "`>>`". Pembacaan input yang umum dilakukan melalui stream masukan adalah melalui loop sebagai berikut:

```
1   #include <stream.h>
2
3   main ()
4   {
5     char ch;
6
7     while ( cin >> ch ) {  // FALSE jika mendapatkan EOF
8       // ...
9     }
10  }
```

Pembacaan karakter seperti yang ditunjukkan di atas akan mengabaikan karakter *white space* seperti: *newline, tab, form feed*, dsb.

10 Standard Template Library

Pada tahun 1995, Alexander Stepanov dan Meng Lee mengeluarkan dokumen resmi mengenai Standard Template Library (STL) [Stepanov and Lee, 1995]. Pustaka ini terutama

menyediakan sejumlah kelas penampung (*container class*) dan algoritma generik (*template function*). Kelas penampung yang disediakan termasuk: *vector, list, deque, set, multiset, map, multimap, stack, queue*, dan *priority queue*. Algoritma generik yang disediakan termasuk algoritma-algoritma dasar untuk melakukan pencarian (*searching*), pengurutan (*sorting*), penggabungan (*merging*), penyalinan (*copying*), dan pengubahan (*transforming*).

Pustaka ini didasarkan dari hasil penelitian mengenai pemrograman generik (*generic programming*) dan pustaka perangkat lunak generik (*generic software libraries*) yang sudah dilakukan beberapa tahun oleh Stevanov, Lee, dan Musser dalam bahasa Scheme, Ada, dan C++. Penyertaan STL ke dalam pustakan baku C++ sudah diusulkan sebelumnya oleh ANSI/ISO C++ Standards Committe, yaitu pada bulan Juli 1994.

STL terdiri dari lima jenis komponen:

1. Penampung (*container*): mengelola sekumpulan lokasi memori
2. Iterator: menyediakan sarana agar sebuah algoritma dapat menjelajah sebuah penampung
3. Objek fungsi: membungkus sebuah fungsi di dalam objek yang digunakan komponen lain
4. Algoritma: mendefinisikan prosedur komputasi
5. Adaptor: mengadaptasikan komponen agar menyediakan interface yang berbeda

Algoritma di dalam STL bersifat generik, dapat digunakan terhadap berbagai penampung dan bahkan pada array C++ biasa. Salah satu ciri penting dari perancangan pustaka ini adalah penggunaan iterator yang sangat konsisten sebagai perantara antara algoritma dan penampung.

10.1 Penampung

Contoh `Stack` generik yang sudah diberikan pada bagian awal dari diktat ini merupakan contoh sebuah penampung. Penampung (*container*) di dalam STL dibedakan antra penampung urutan (*sequence container*) dan penampung asosiatif (*associative container*).

- Penampung urutan menyimpan kumpulan objek dalam susunan linear. Termasuk ke dalam kategori ini adalah `vector<T>`, `deque<T>`, dan `list<T>`.
- Penampung asosiatif menyediakan fasilitas pengambilan kembali objek (retrieval) dari suatu kumpunan berdasarkan nilai kunci. Ukuran kumpulan dapat berubah pada saat eksekusi. Termasuk ke dalam kategori ini adalah `set<T>`, `multiset<T>`, `map<T>`, dan `multimap<T>`,

Pada Contoh 10.1 ditunjukkan penggunaan penampung generik `vector`. Metoda `begin()` dan `end()` memberikan iterator yang menunjuk pada posisi awal (elemen pertama) dan "akhir" (element *setelah* terakhir) dari vektor. Pada contoh tersebut, baris 1–3 dapat diganti dengan `#include <stl.h>`.

Contoh 10.1 Penggunaan kelas penampung `vektor`

```
1   #include <algo.h>
2   #include <stream.h>
3   #include <vector.h>
4
5   main ()
6   {
7      vector<float> s (10);
8      float sum;
9      int k;
10
11     for (k=0; k < s.size(); k++)
12        s[k] = float(k);
13
14     sum = accumulate (s.begin(), s.end(), 0.0);
15     cout << "Sum is " << sum << endl;
16  }
```

Sebuah penampung memiliki metoda-metoda seperti yang ditunjukkan pada Tabel 5. Jika iterator yang dimiliki penampung berjenis *bidirectional* atau *random access* maka penampung disebut sebagai penampung *reversible*. Selain metoda ini, penampung urutan (*sequence*) memiliki metoda tambahan seperti ditunjukkan pada Tabel 6.

Metoda	Makna	Keterangan
`begin()`	iterator yang menunjuk ke posisi awal	
`end()`	iterator yang menunjuk ke posisi "akhir"	
`size()`	jumlah elemen di dalam penampung	
`max_size()`	jumlah elemen terbesar	
`empty()`	menyatakan kosong/tidak	
`rbegin()`	iterator yang menunjuk ke posisi akhir	hanya untuk reversible
`rend()`	iterator yang menunjuk ke posisi "awal"	hanya untuk reversible

Tabel 5: Metoda pada penampung

10.2 Iterator

Iterator merupakan pointer C++ yang digeneralisasi sehingga memungkinkan penulis program memanipulasi penampung dengan cara yang seragam. Karena merupakan generalisasi dari pointer, maka fungsi generik yang memanfaatkan iterator akan dapat digunakan juga terhadap pointer biasa.

Secara konsep, iterator adalah objek yang mendefinisikan fungsi `operator*`[8] yang mengembalikan nilai dari suatu kelas `T` yang disebut sebagai *value type* dari iterator tersebut.

[8]sebagai operasi *dereference*, bukan sebagai operasi perkalian!

Metoda	Makna
`insert(p, t)`	menyisipkan salinan `t` sebelum iterator `p`
`insert(p, n, t)`	menyisipkan `n` salinan `t` sebelum `p`
`insert(p, i, j)`	menyisipkan salinan elemen dalam wilayah iterator masukan `[i,j)`sebelum `p`
`erase(p)`	hapus elemen yang ditunjuk `p`
`erase(p1,p2)`	hapus elemen dalam wilayah `[p1,p2)`
`front()`	reference objek yang ditunjuk oleh `begin()`
`back()`	reference objek yang ditunjuk oleh `--end()`
`push_front(t)`	`insert (begin(), t)`
`push_back(t)`	`insert (end(), t)`
`pop_front(t)`	`erase (begin())`
`pop_back(t)`	`erase (--end(), t)`

Tabel 6: Metoda tambahan pada penampung urutan

Selain itu, setiap operator memiliki tipe korespondensi yang disebut *distance type* dari operator tersebut.

Untuk memahami konsep ini perhatikanlah kasus pointer biasa. Seandainya diketahui sebuah pointer (biasa) seperti contoh di bawah ini:

```
float *p;
```

maka dari variabel `p` dapat dibentuk ekspresi "`*p`" yang bertipe `float` yang dalam hal ini merupakan *value type* dari `p`. Sedangkan jika seandainya representasi tipe `float` memerlukan 8 byte, maka *distance type* adalah tipe integral (bulat) yang nilainya kelipatan 8. Dalam kasus ini, misalnya, ekspresi `p + 2` akan menambah nilai `p` dengan 16.

Dalam STL, iterator berfungsi sebagai perantara algoritma generik dgn penampung. Iterator dibagi ke dalam lima kategori:

- Forward: menyediakan penjelajahan satu arah dari sebuah urutan. Operasi ini dinyatakan dengan `++`.
- Bidirectional: menyediakan penjelajaahan dua arah. Operasi ini dinyatakan dengan `++` dan `--`.
- Random access: berperan seperti iterator *bidirectional* dengan tambahan kemampuan berikut:
  - Melompat jauh (maju/mundur) iterator `r` yang dinyatakan dengan operasi `r += n` atau `r -= n` (`n` memiliki jenis "*distance*")
  - Penambahan / pengurangan dengan integer dengan operasi `r + n` atau `r - n` yang menghasilkan iterator
  - Pengurangan dua iterator `r - s` yang menghasilkan integer ("*distance*")
  - Pembandingan nilai iterator, dinyatakan dengan operasi `r < s`, `r > s`, `r <= s`, dan `r >= s`.

Ketiga iterator di atas dapat dibandingkan dengan operator `==` dan `!=`.

- Input dan Output: berperan seperti interator maju (*forward iterator*) namun dengan beberapa pengecualian:
  - Tidak ada jaminan bahwa nilai iterator (input/output) dapat disimpan dan lalu digunakan untuk maju dari posisi terakhir yang disimpan tersebut jika iterator tersebut digunakan kedua kalinya
  - Tidak ada jaminan bahwa objek `*ri`, yaitu objek yang diacu oleh iterator masukan `ri`, dapat di*assign*
  - Tidak ada jaminan bahwa objek `*ro`, yaitu objek yang diacu oleh iterator keluaran `ro`, dapat dibaca
  - Tidak ada jaminan bahwa nilai iterator dapat dibandingkan dengan operator `==` maupun `!=`

Jenis iterator yang dimiliki oleh setiap penampung ditunjukkan pada Tabel 7. Catatan: semua iterator dari kelas penampung asosiatif merupakan *bidirectional*.

Iterator	Jenis
`vector<T>::iterator`	*random access*
`deque<T>::iterator`	*random access*
`list<T>::iterator`	*bidirectional*

Tabel 7: Jenis Iterator Untuk Penampung Urutan

10.3 Iterator Stream

Iterator *stream* diciptakan agar algoritma-algoritma generik dapat langsung memanfaatkan *stream* masukan/keluaran. STL mendefinisikan kelas generik `istream_iterator<T>` untuk membaca data dan `ostream_iterator<T>` untuk menuliskan data.

10.4 Objek Fungsi

Selain mendefinisikan sejumlah fungsi generik, STL juga mendefinisikan **objek fungsi**, yaitu objek yang mendefinisikan fungsi `operator()`. Seringkali, dalam program C atau C++ penulis program mendeklarasikan pointer ke fungsi sebagai parameter sebuah fungsi, seperti parameter kedua dalam contoh berikut:

```
typedef void (*MyFuncType) (int, char*);

int MyTestFunc (float x, MyFuncType f) {
   (*f)(x); // panggil "f" dengan parameter "x"
}
```

Dalam STL, parameter kedua dalam konteks seperti ini dituliskan sebagai sebuah objek fungsi. Sehingga, algoritma dapat memanfaatkan baik pointer ke fungsi maupun objek fungsi. Sebagai dasar didefinisikan "kelas" `unary_function`, dan `binary_function` dengan definisi seperti pada Contoh 10.2. Dengan dasar ini, misalnya kelas **generik** `plus`[9] didefinisikan sebagai kode dalam Contoh 10.3.

Contoh 10.2 Deklarasi "kelas" `unary_function` dan `binary_function`

```
template <class Arg, class Result>
struct unary_function {
   typedef Arg argument_type;
   typedef Result result_type;
}

template <class Arg1, class Arg2, class Result>
struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
}
```

Contoh 10.3 Implementasi fungsi objek "`plus`"

```
template <class T>
struct plus : binary_function<T,T,T> {
   T operator() (const T& x, const T& y) const {
      return x + y;
   }
};
```

Dengan cara yang mirip, **kelas-kelas** objek lain di definisikan. Selain kelas dasar fungsi di atas, terdapat tiga kelompok lainnya:

1. Fungsi-fungsi aritmatika `minus`, `times`, `divides`, `modulus`, `negate`,
2. Fungsi-fungsi pembadingan `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, `less_equal`.
3. Fungsi-fungsi logika `logical_and`, `logical_or`, dan `logical_not`.

Pemanfaatan objek fungsi ini diberikan pada Contoh 10.4.

[9]Perhatikanlah bahwa `plus` adalah sebuah "**kelas**" bukan sebuah fungsi

Contoh 10.4 Pemanfaatan objek fungsi

```
1  #include <stl.h>
2
3  main () {
4     plus<float> op;
5
6     cout << op (.3, 12.) << endl;
7  }
```

10.5 Algoritma

Semua algoritma di dalam STL dipisahkan dari implementasi struktur data tertentu serta diparameterisasikan oleh tipe iterator. Berbeda dari fungsi objek yang merupakan objek, algoritma-algoritma ini adalah fungsi generik.

Untuk algoritma-algoritma tertentu baik versi *in-place* maupun versi *copying* didefinisikan. Untuk membedakannya, nama algoritma dituliskan sebagai *algoritma* untuk versi *in-place* dan *algoritma*_copy untuk versi *copying*.

Beberapa algoritma juga mendeklarasikan parameter formal berjenis Predicate atau BinaryPredicate jika algoritma ini memanfaatkan objek fungsi dalam operasinya. Untuk membedakannya, algoritma yang seperti ini diberi nama *algoritma*_if.

Beberapa algoritma yang didefinisikan STL di antaranya ditunjukkan pada Tabel 8.

Algoritma	Keterangan
for_each ()	
find (), find_if()	
adjacent_find ()	
count (), count_if()	
mismatch()	
equal()	
search()	
copy(), copy_backward()	
swap(), iter_swap(), swap_ranges()	
transform()	
replace(), replace_if()	
replace_copy(), replace_copy_if()	
fill(), fill_n()	
generate(), generate_n()	
remove(), remove_if()	

Tabel 8: Algoritma yang didefinisikan STL

10.6 Adaptor

11 Panduan Pemanfaatan C++

- Dalam merancang sebuah kelas perhatikanlah bahwa lima jenis fungsi berikut ada di dalam kelas tersebut:

  1. *Constructor*: *default* dan/atau *user-defined*
  2. *Copy Constructor*
  3. *Destructor*: deklarasikanlah sebagai `virtual` untuk mengantisipasi kemungkinan kelas ini diturunkan menjadi kelas lain
  4. Operasi assignment (`operator =`)
  5. Operasi masukan/keluaran yang melibatkan kelas `stream`. Kedua fungsi ini harus didefinisikan sebagai non-anggota sehingga kemungkinan besar dideklarasikan sebagai `friend`.

```
1   class MyClass {
2      MyClass ();                              // ctor
3      virtual ~MyClass ();                     // virtual dtor
4      MyClass (const MyClass&);                // cctor
5      MyClass& operator= (const MyClass&); // assignment
6
7      // masukan/keluaran
8      friend stream& operator << (stream&, const MyClass*);
9      friend stream& operator >> (stream&, const MyClass*);
10  }
```

- Hindari penggunaan data global
- Hindari penggunaan fungsi non-anggota global
- Hindari penggunaan anggota data yang ditempatkan pada bagian `public`
- Hindari penggunaan `friend`, kecuali untuk menghindari tiga "aturan" di atas
- Jangan mengakses anggota data dari objek lain secara langsung
- Jangan menggunakan "*type field*", gunakanlah *virtual function*
- Dalam melakukan peralihan dari C ke C++
  - Gunakan feature dari C++ secara bertahap
  - Penggunaan C++ yang baik memerlukan perancangan yang baik. C++ hanyalah merupakan alat untuk mengimplementasikan rancangan tersebut

- Demi efisiensi kode, deklarasikan objek sedekat mungkin dengan pemakaian pertamanya
- Untuk menginisialisasi subobjek, gunakan *memberwise initialization*, jangan gunakan assignment!!!

Pustaka

[Ellis and Stroustrup, 1990] Ellis, M. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA.

[Lippman, 1991] Lippman, S. (1991). *C++ Primer. 2nd Edition*. Addison-Welsy, Reading, MA.

[Stepanov and Lee, 1995] Stepanov, A. and Lee, M. (1995). *The Standard Template Library*. Hewlett-Packard Company.

[Strouptrup, 1997] Strouptrup, B. (1997). *The C++ Programming Languange. 3rd Edition*. Addison-Wesley.

[Stroustrup, 1994] Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.