Class C++ Dasar

Harimurti Widyasena
harmur@mailcity.com

Pemrograman C++ memerlukan pemahaman yang memadai untuk menterjemahkan desain ke dalam bentuk implementasi, terutama untuk desain yang menggunakan abstraksi class. Fokus pembahasan pada aspek pembentukan obyek (*construction*) sebuah class, dan proses sebaliknya pada saat obyek tersebut sudah tidak digunakan lagi (*destruction*).

Deklarasi dan Definisi

Deklarasi dan definisi adalah langkah awal dalam setiap penulisan program tidak terkecuali dalam bahasa C++. Deklarasi dan definisi diperlukan untuk semua tipe data termasuk tipe data bentukan user (*user-defined type*).

Bentuk sederhana deklarasi class adalah sebagai berikut,

```
class C { }; atau
struct C { };
```

dalam bahasa C++ `struct` dan `class` mempunyai pengertian yang sama. Deklarasi class dengan struct mempunyai anggota dengan akses public kecuali jika dinyatakan lain.

```
struct C
{
   int i;
   void f();
}
```

```
class C
{
public:
   int i;
   void f();
}
```

Kedua deklarasi tersebut mempunyai arti yang sama.

Hal ini adalah pilihan desain yang diambil oleh desainer C++ (Bjarne Stroustrup) untuk menggunakan C sebagai basis C++ ketimbang membuat bahasa yang sama sekali baru. Tentunya ada konsekuensi atas pilihan desain ini, salah satu contoh adalah kompatibilitas terhadap bahasa C.

Dalam bahasa C deklarasi,

```
struct C { … };
```

menyatakan C sebagai nama tag. Nama tag berbeda dengan nama tipe, sehingga C (nama tag) tidak dapat dipergunakan dalam deklarasi yang membutuhkan C sebagai suatu tipe obyek. Kedua contoh deklarasi berikut ini tidak valid dalam bahasa C,

```
C c;     /* error, C adalah nama tag */
C *pc;   /* error, C adalah nama tag */
```

Dalam bahasa C, kedua deklarasi tersebut harus ditulis sebagai berikut,

```
struct C c;
struct C *pc;
```

atau menggunakan `typedef` sebagai berikut,

```
struct C { … };
typedef struct C C;

C c;
C *pc;
```

C++ memperlakukan nama class, C sebagai nama tag sekaligus nama tipe dan dapat dipergunakan dalam deklarasi. Kata class tetap dapat dipergunakan dalam deklarasi, seperti contoh berikut ini,

```
class C c;
```

Dengan demikian C++ tidak membedakan nama tag dengan nama class, paling tidak dari sudut pandang pemrogram (*programmer*), dan tetap menerima deklarasi *structure* seperti dalam bahasa C. Kompatibilitas C++ terhadap tidak sebatas perbedaan nama tag dan nama tipe, karena standar C++ masih perlu mendefinisikan tipe POD (Plain Old Data). POD type mempunyai banyak persamaan dengan structure dalam C. Standar C++ mendefinisikan POD type sebagai obyek suatu class yang tidak mempunyai user-defined constructor, anggota protected maupun private, tidak punya base class, dan tidak memiliki fungsi virtual.

Dalam desain suatu aplikasi terdiri atas banyak class, dan masing-masing class tidak berdiri sendiri melainkan saling bergantung atau berhubungan satu sama lain. Salah satu

contoh hubungan tersebut adalah hubungan antara satu class dengan satu atau lebih *base class* atau *parent class*. Jika class C mempunyai base class B, dikenal dengan *inheritance*, maka deklarasi class menjadi,

```
class C : public B {}; atau
class C : protected B {}; atau
class C : private B {};
```

akses terhadap anggota base class B dapat bersifat public, protected, maupun private, atau disebut dengan istilah *public*, *protected* atau *private inheritance*. Class C disebut dengan istilah *derived class*. Jika tidak dinyatakan bentuk akses secara eksplisit, seperti dalam deklarasi berikut:

```
class C : B
```

maka interpretasinya adalah private inheritance (*default*), tetapi jika menggunakan `struct` maka tetap merupakan public inheritance.

Jika desainer class C tersebut menginginkan hubungan *multiple inheritance* (MI) terhadap class B dan A, maka deklarasi class C menjadi,

```
class C : public B, public A { };
```

Sebuah class, seperti halnya class C mempunyai anggota berupa data maupun fungsi (*member function*). Isi class tersebut berada diantara tanda kurung `{ }` dan dipilah-pilah sesuai dengan batasan akses yang ditentukan perancang (desainer) class tersebut.

```
class C : public B
{
 public:
   (explicit) C()(:member-initializer);
   C(const C& );
   C& operator=(const C&);

   (virtual)~C();
   statement lain
 (protected: statement)
 (private: statement)
};
```

Secara ringkas batasan akses (*access specifiers*) mempunyai arti seperti ditunjukkan pada table berikut ini,

Batasan Akses	**Arti**
public	Semua class atau bebas
protected	Class itu sendiri, *friend*, atau *derived class*
private	Class itu sendiri, *friend*

Sebuah class dapat memberikan ijin untuk class lain mengakses bagian protected maupun private class tersebut melalui hubungan *friendship* (dinyatakan dengan *keyword* `friend`).

Sebuah class mempunyai beberapa fungsi khusus, yaitu *constructor*, *copy constructor*, *destructor* dan *copy assignment operator*.

Constructor

`C()` adalah anggota class yang bertugas melakukan inisialisasi obyek (*instance*) dari suatu class C. Constructor mempunyai nama yang sama dengan nama class, dan tidak mempunyai *return value*. Sebuah class dapat mempunyai lebih dari satu constructor. Constructor yang tidak mempunyai argumen, disebut *default constructor*, sebaliknya constructor yang mempunyai lebih dari satu argumen adalah *non-default consructor*. Constructor dengan satu default argument tetap merupakan sebuah default constructor,

```
class C
{
public:
   C(int count=10) : _count(count) {}
…
private:
   int _count;
};
```

Compiler C++ dapat menambahkan *default constructor* bilamana diperlukan, jika dalam definisi class

- tidak tertulis secara eksplisit sebuah default constructor dan tidak ada deklarasi constructor lain (*copy constructor*).
- tidak ada anggota class berupa data `const` maupun *reference*.

Sebagai contoh definisi class C sebagai berikut,

```
class C {…};

C c1;          // memerlukan default constructor
C c2(c1);      // memerlukan copy constructor
```

Compiler C++ memutuskan untuk menambahkan default dan copy construtor setelah menemui kedua baris program tersebut, sehingga definisi class secara efektif menjadi sebagai berikut,

```
class C
{
public:
   C();                // default costructor
   C(const C& rhs);    // copy constructor

   ~C();               // destructor
```

```
   C& operator=(const C& rhs);   // assignment operator
   C* operator&();    // address-of operator
   const C* operator&(const C& rhs) const;
};
```

compiler menambahkan public constructor, dan destructor. Selain itu, compiler juga menambahkan *assignment operator* dan *address-of operator*. Constructor (default dan non-default) tidak harus mempunyai akses public, sebagai contoh adalah pola desain (*design pattern*) Singleton.

```
class Singleton
{
public:
   static Singleton* instance();
protected:
   Singleton();
private:
   static Singleton* _instance;
};
```

obyek (*instance*) *singleton* tidak dibentuk melalui constructor melainkan melalui fungsi instance. Tidak ada obyek *singleton* lain yang dapat dibentuk jika sudah ada satu obyek *singleton*.

Umumnya default constructor bentukan compiler (*generated default constructor*) menggunakan default constructor anggota bertipe class, sedangkan anggota biasa (*built-in type*) tidak diinisialisasi. Demikian halnya dengan obyek yang dibentuk dari obyek lain (copy), maka copy constructor bentukan compiler (*generated copy constructor*) menggunakan copy constructor dari anggota bertipe class pada saat inisialisasi. Sebagai contoh deklarasi class C berikut ini,

```
class C
{
public:
   C(const char* aName);
   C(const string& aName);
…
private:
   std::string name;
};
```

copy constructor bentukan compiler menggunakan copy constructor class `string` untuk inisialisasi `name` dari `aName`. Jika class C tidak mempunyai constructor, maka compiler menambahkan juga default constructor untuk inisialisasi `name` menggunakan default constructor class `string`.

Inisialisasi obyek menggunakan constructor (non-default) dapat dilakukan dengan *member initializer* maupun dengan *assignment* sebagai berikut,

member initialization

```
class C
{
   int i,j;
public:
   C() : i(0),j(1) {}
…
};
```

assignment

```
class C
{
   int i,j
public:
   C()
   {
      i=0;j=0;
   }
…
};
```

Kedua cara tersebut memberikan hasil yang sama, tidak ada perbedaan signifikan antara kedua cara tersebut untuk data bukan tipe class. Cara member initializer **mutlak** diperlukan untuk data `const` maupun *reference*, seperti kedua contoh berikut ini:

```
class C   //:1
{
public:
   C(int hi,int lo) : _hi(hi),_lo(lo) {}
…
private:
   const int _hi,_lo;   // const member
};

class C   //:2
{
public:
   C(const string& aName) : name(aName) {}
…
private:
   std::string& name;   // reference member
};
```

Cara *member initialization* **sebaiknya** dilakukan untuk anggota bertipe class (*user-defined type*) seperti ditunjukkan pada contoh berikut ini,

```
class C
{
public:
   C(const string& aName) : name(aName) { }
private:
   std::string name;   // bukan reference member
};
```

Pertimbangan menggunakan cara *member initialization* terletak pada efisiensi eksekusi program. Hal ini berkaitan dengan cara kerja C++ yang membentuk obyek dalam dua tahap,

- pertama, inisialisasi data
- kedua, eksekusi constructor (assignment)

Dengan demikian jika menggunakan cara assignment sebenarnya eksekusi program dilakukan dua kali, pertama inisialisasi kemudian assignment, sedangkan menggunakan *member initialization* hanya memanggil sekali constructor class `string`. Semakin kompleks class tersebut (lebih kompleks dari class `string`) semakin mahal (tidak efisien) proses pembentukan obyek melalui cara assignment.

Constructor dengan satu argumen berfungsi juga sebagai *implicit conversion operator*. Sebagai contoh deklarasi class A dan B berikut ini,

```
class A
{
public:
   A();
};

class B
{
public:
   B(const A&);
};
```

pada cuplikan baris program di bawah ini terjadi konversi tipe obyek A ke B secara implisit melalui copy constructor class B.

```
A a
B b=a;   // implicit conversion
```

explicit

C++ menyediakan satu sarana, menggunakan keyword `explicit`, untuk mengubah perilaku constructor dengan satu argumen agar tidak berfungsi sebagai *conversion operator*. Jika class B menyatakan explicit pada copy constructor sebagai berikut,

```
class B
{
public:
   explicit B(const A& a);   // explicit ctor
};
```

maka konversi A ke B secara implisit tidak dapat dilakukan. Konversi A ke B dapat dilakukan secara eksplisit menggunakan *typecast*,

```
A a;
```

```
B b=static_cast<B>(a);
```
atau
```
B b=(B)a;
```

Konversi secara implisit dapat terjadi melalui argumen fungsi f dengan tipe B

```
void f(const B& );
```

tetapi f diakses dengan variabel tipe A, `f(a)`. Apabila class B menghalangi konversi secara implisit maka argumen fungsi f menjadi,

```
f((B)a);
```
atau
```
f(static_cast<B>(a));
```

Konversi tipe obyek secara implisit sebaiknya dihindari karena efeknya mungkin lebih besar terhadap aplikasi program secara keseluruhan dan tidak dapat dicegah pada saat kompilasi, karena construcor dengan argumen tunggal adalah suatu pernyataan program yang sah dan memang dibutuhkan.

Copy Constructor dan Copy Assignment

Sejauh ini sudah dibahas mengenai copy constructor sebagai anggota class yang berperan penting pada saat pembentukan obyek. Apabila sebuah class tidak menyatakan secara tegas copy constructor class tersebut, maka compiler menambahkan copy constructor dengan bentuk deklarasi,

```
C(const C& c);
```

Bentuk lain copy constructor adalah sebagai berikut,

```
C(C& c);
```
atau
```
C(C volatile& c);
```
atau
```
C(C const volatile& c);
```

Copy constructor class C adalah constructor yang mempunyai satu argumen. Sebuah copy constructor boleh mempunyai lebih dari satu argumen, asalkan argumen tersebut mempunyai nilai default (*default argument*).

```
C(C c);   // bukan copy constructor
C(C const& c,A a=b);   //copy constructor
```

Constructor dengan argumen bertipe C saja (tanpa *reference*) bukan merupakan copy constructor.

Copy constructor juga dibutuhkan pada saat memanggil suatu fungsi yang menerima argumen berupa obyek suatu class,

```
void f(C x);
```

memerlukan copy onstructor class C untuk mengcopy obyek c bertipe C ke obyek x dengan tipe yang sama, yaitu pada saat memanggil fungsi `f(c)`(*pass-by-value*).

Hal serupa terjadi pada saat fungsi f sebagai berikut,

```
C f()
{
   C c;
…
   return c;
}
```

mengirim obyek c ke fungsi lain yang memanggil fungsi f() tersebut.

Copy assignment operator class C adalah `operator=`, sebuah fungsi yang mempunyai satu argumen bertipe C. Umumnya deklarasi copy assignment mempunyai bentuk,

```
C &operator=(const C &c);
```

Bentuk lain yang mungkin adalah,

`C &operator=(C &c);` atau
`C &operator=(C volatile &c);` atau
`C &operator=(C const volatile &c);`

Copy assignment boleh mempunyai argumen dengan tipe C (bukan reference), tetapi tidak boleh mempunyai argumen lebih dari satu walaupun argumen tersebut mempunyai nilai *default* (default argument). Seperti halnya copy constructor, compiler akan menambahkan copy assignment jika suatu class tidak mempunyai fungsi tersebut. Copy assignment dibutuhkan untuk membentuk obyek melalui assignment, seperti contoh berikut

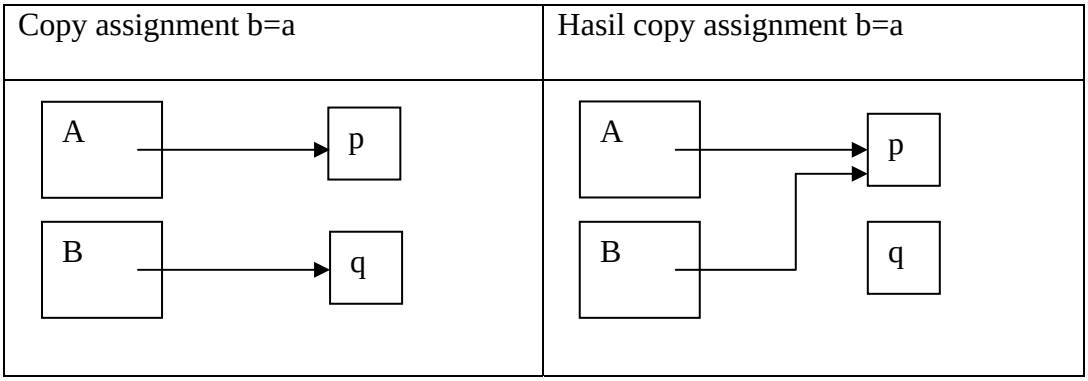
```
class C
{
public:
   C();   //ctor
   ~C();  //dtor
…
};

C c1;
C c2=c1;   //copy constructor
C c3;
c3=c1;     //copy assignment
```

Class C tidak mempunyai copy constructor maupun copy assignment operator, maka pembentukan obyek c2, dan c3 menggunakan copy constructor dan copy assignment yang ditambahkan oleh compiler ke class C tersebut.

Suatu class yang mempunyai data dengan alokasi dinamik (pointer) sebaiknya tidak mengandalkan copy constructor maupun copy assignment operator yang ditambahkan compiler. Copy assignment hasil tambahan compiler mengcopy (*memberwise copy*)

pointer dari obyek satu (yang dicopy) ke obyek lainnya (hasil copy), sehingga kedua obyek mengacu ke lokasi memori yang sama. Masalah timbul jika kedua obyek mempunyai masa pakai (*lifetime*[1]) yang berbeda. Jika salah satu obyek sudah habis masa pakainya maka destructor obyek tersebut mengembalikan memori (*dynamic memory*) yang digunakan obyek tersebut, padahal copy obyek tersebut masih mengacu ke lokasi memori yang sama.



Pada contoh hasil copy assignment b=a (*shalow copy*), menunjukkan kedua obyek a dan b mengacu ke lokasi memori p. Apabila obyek a melepas memori p (melalui destructor), maka obyek b mengacu ke lokasi memori yang sudah tidak valid lagi. Lokasi memori p dapat digunakan obyek lain jika obyek a melepasnya. Demikian pula halnya dengan lokasi memori q, apabila obyek b habis masa pakainya (keluar scope, dihapus dll) maka destructor class B tidak melepas memori q. Akibatnya terjadi pemborosan memori (*memory leak*).

Salah satu jalan keluar adalah dengan menyatakan secara tegas copy constructor dan copy assignment yang dibutuhkan suatu class sehingga compiler tidak membuat copy constructor dan copy assignment ke class tersebut. Alternatif lain adalah menempatkan deklarasi copy constructor dan copy assignment operator `private` sebagai berikut,

```
class C
{
…
private:
   C(const C&);
   C &operator=(const C&);
};
```

definisi copy constructor dan copy assignment operator class C pada contoh di atas tidak perlu ada, karena tujuannya adalah menghalangi proses penggandaan (copy) menggunakan kedua fungsi tersebut. Pada tahap kompilasi penggunaan assignment, `b=a` masih dapat diterima karena deklarasi asignment operator tersebut tersedia. Pada saat link akan gagal karena *linker* tidak dapat menemukan definisi copy assignment operator. Teknik ini masih mempunyai kelemahan, karena class lain masih mungkin

[1] *Lifetime* atau *storage duration* adalah waktu sejak pembentukan (construction) sampai penghancuran (destruction) obyek.

mempunyai akses ke private copy constructor dan copy assignment operator tersebut (melalui hubungan *friendship*).

Destructor

Destructor adalah anggota class (*member function*) yang berfungsi melepas memori pada saat suatu obyek sudah tidak diperlukan lagi. Fungsi destructor kebalikan constructor. Destructor tidak mempunyai atau memerlukan argumen. Destructor juga tidak mengembalikan nilai apapun (tidak mempunyai *return type*). Seperti halnya constructor, compiler dapat menambahkan sebuah destructor jika sebuah class tidak mempunyai destructor.

`virtual` Destructor

Sebuah destructor dapat berupa fungsi virtual. Hal ini menjadi keharusan jika class B,

- merupakan base class.
- class D yang menggunakan B sebagai base class mempunyai anggota berupa data dengan alokasi memori dinamik (pointer).

```
class B
{
public:
   B();
   ~B();
};

class D : public B
{
public:
   D() : p(new char[256]) {}
   ~D()
   {
      delete[] p;
   }
…
private:
   char *p;
};
```

Pada contoh tersebut destructor base class B bukan fungsi virtual. Dalam C++ umumnya obyek class D digunakan secara *polimorphic* dengan membentuk obyek class D (*derived class*) dan menyimpan alamat obyek tersebut dalam pointer class B (*base class*) seperti pada contoh berikut ini,

```
void main(void)
{
   B *pB=new D();

   delete pB;
}
```

Dalam standar C++ menghapus obyek D (derived class) melalui pointer class B (base class) sedangkan destructor base class non-virtual mempunyai efek yang tidak menentu (*undefined behaviour*). Apabila standard C++ tidak menetapkan apa yang seharusnya berlaku, maka terserah kepada pembuat compiler menentukan perilaku program pada kondisi semacam ini. Umumnya pembuat compiler mengambil langkah untuk tidak memanggil destructor class D (derived class). Dengan demikian, pada saat menjalankan perintah delete, destructor class D tidak dieksekusi karena destructor base class B non-virtual. Akibatnya lokasi memori dinamik yang digunakan class D tidak pernah dilepas. Hal ini adalah contoh lain terjadinya pemborosan memori (*memory leak*) oleh suatu program. Jalan keluarnya adalah membuat destructor base class B virtual,

```
class B
{
public:
   B();
   virtual ~B();
}
```

Tidak seperti destructor, tidak ada *virtual constructor* atau *virtual copy constructor*. Pada saat membentuk obyek, tipe obyek harus diketahui terlebih dahulu, apakah membentuk obyek class A, B, C dsb. Tidak ada aspek bahasa C++ untuk mewujudkan virtual constructor secara langsung, menempatkan `virtual` pada deklarasi constructor merupakan kesalahan yang terdeteksi pada proses kompilasi. Efek virtual constructor bukan tidak mungkin dicapai, C++ memungkinkan membuat idiom *virtual constructor* yang bertumpu pada fungsi virtual dalam kaitannya dengan hubungan antara sebuah class dengan *base classnya*.

Ringkasan

Sejauh ini pembahasan artikel masih belum menyentuh aspek praktis pemrograman, namun demikian dalam menterjemahkan suatu desain maupun memahami program yang ditulis orang lain sangatlah penting mengetahui aturan dasar sesuai standarisasi C++.

Butir-butir pembahasan dalam artikel ini antara lain,

- Fokus pembahasan adalah aspek pembentukan obyek. Tidak membahas aturan (*rule*) berkaitan dengan class dalam C++ secara komprehensif.
- Constructor merupakan anggota class yang berperan dalam pembentukan obyek. Compiler menambahkan constructor bilamana diperlukan ke class yang tidak mempunyai constructor. Constructor tidak harus mempunyai akses public. Inisialisasi data menggunakan constructor dapat dilakukan dengan cara member initialization dan assignment. Keduanya tidak mempunyai perbedaan signifikan untuk data biasa (*built-in type* seperti `char`, `int`, `float`, dll). Cara member initialization lebih efisien untuk data berupa class (*user-defined type*).
- Constructor dengan satu argumen dapat digunakan untuk konversi tipe data secara implisit. C++ menyediakan `explicit` untuk mengubah perilaku ini, karena hal tersebut melonggarkan janji C++ sebagai bahasa yang mengutamakan *strict type (type safe)*.

- Sebuah class membutuhkan copy constructor dan copy assignment operator untuk menggandakan obyek suatu class. Hal ini terjadi juga pada saat memanggil suatu fungsi dengan cara *pass-by-value*. Apabila suatu class tidak mempunyai copy constructor dan copy assignment maka compiler menambahkannya. Copy constructor dan copy assignment hasil tambahan compiler bekerja dengan cara *memberwise copy* dan menghasilkan *shalow copy* untuk data dengan alokasi memori dinamik.
- Destructor merupakan anggota class yang berfungsi pada saat *lifetime* suatu obyek habis. Destructor sebuah base class sebaiknya virtual.
- Constructor selalu merupakan fungsi non-virtual. Efek *virtual constructor* dan *virtual copy constructor* mungkin diperlukan dalam suatu desain. Efek virtual constructor dapat diwujudkan melalui sifat polimorphisme class. Efek virtual copy constructor dapat diwujudkan memanfaatkan aspek *covariant return type* sebuah hirarki class. Kedua hal tersebut memerlukan pembahasan khusus.
- Pembahasan pembentukan obyek belum dikaitkan dengan jenis scope yang ada dalam C++. C++ mempunyai jenis scope yang lebih kaya dibandingkan bahasa C, selain file scope, function scope, dan block scope C++ memiliki *class scope* dan `namespace` scope. Salah satu panduan praktis bahkan menyarankan untuk menunda (*lazy initialization*) pembentukan obyek selagi belum diperlukan.
- Pembentukan suatu obyek mungkin saja gagal. Artikel ini tidak membahas mengenai kegagalan pembentukan obyek, karena pembahasan tersebut berkaitan pembahasan *exception* dalam C++. Pembahasan exception C++ (*exception safety*) merupakan topik tersendiri.

Desain dan implementasi class C++ bukanlah hal yang mudah, masih banyak aspek lain yang belum terjangkau pembahasan artikel ini. Pada artikel selanjutnya akan dibahas scope (*visibility*) dalam C++, batasan akses (access specifier) C++, abstract class, function overloading, class relationship, template, dll.

Referensi

1. Bjarne Stroustrup, “The C++ Programming Language”, 3rd edition, Addison-Wesley 1997
2. Scott Meyers,”Effective C++”, 2nd edition, Addison-Wesley
3. Scott Meyers,”More Effective C++”, Addison-Wesley
4. Q&A dalam C/C++ Users Journal.
5. GOF,”Design Pattern”, Addison-Wesley.

Harimurti W. *programmer* freelance tinggal di Cimahi. Komentar, koreksi, kritik, saran maupun pertanyaan mengenai artikel ini dapat dikirim ke alamat email: **harmur@mailcity.com**.

Biografi

Harimurti Widyasena. Lahir di Jakarta 20 April 1962. Lulus SMA di Jakarta tahun 1981, kemudian melanjutkan kuliah di Institut Teknologi Bandung jurusan Teknik Mesin. Lulus S1 tahun 1989 kemudian bekerja di PT. Industri Pesawat Terbang Nusantara.

Belajar pengetahuan komputer secara umum sejak kuliah dan melalui pengalaman kerja, mulai dari mainframe seperti IBM 3031, kemudian mini computer seperti PDP 11/44, workstation (DEC Alpha) sampai dengan era PC pada akhir 80-an dengan sistem operasi (a.l: RSX-11, OSF/1, Windows, DOS) dan bahasa pemrograman yang berbeda-beda (a.l: FORTRAN, C++, VB). Pengalaman selama ini adalah dalam pembuatan aplikasi untuk analisa data hasil uji terbang, dan beberapa simulator antara lain: simulator pembangkit listrik (Steam Powerplant), simulator RADAR maritim, dan simulator ATC. Pada tahun 1997-1998 membantu (sebagai tutor) proyek kerjasama antara PT. IPTN-ITB-UT(Universite Thomson) dalam peningkatan sumberdaya manusia bidang rekayasa perangkat lunak (software egineering).

Saat ini mempunyai minat pada bidang software engineering secara umum, terutama aspek analisis dan desain suatu aplikasi dengan teknik OO. Selain aspek teknis dalam proses rekayasa (*lifcycle*) suatu produk system/software, aspek manajemen menjadi perhatian penulis antara lain: software configuration management, software testing dan bentuk proses rekayasa software (a.l: extreme programming).