C++ : Scope Class

Harimurti W
harmur@mailcity.com

Lisensi Dokumen:
Copyright © 2004 IlmuKomputer.Com
*Seluruh dokumen di **IlmuKomputer.Com** dapat digunakan, dimodifikasi dan disebarkan secara bebas untuk tujuan bukan komersial (nonprofit), dengan syarat tidak menghapus atau merubah atribut penulis dan pernyataan copyright yang disertakan dalam setiap dokumen. Tidak diperbolehkan melakukan penulisan ulang, kecuali mendapatkan ijin terlebih dahulu dari **IlmuKomputer.Com**.*

C++ memperluas pengertian scope (*visibility*) dalam bahasa C dengan masuknya class dan namespace. Artikel ini membahas scope sebuah class. Pembahasan scope tidak terlepas dari konsep yang saling berkaitan dalam C/C++ yaitu *lifetime* (*storage duration*) dan *linkage*. Lifetime menentukan kapan destructor sebuah class dipanggil.

Scope sebuah kaji ulang

Komputer, sebagai sebuah mesin, dirancang untuk bekerja mengolah angka. Komputer menyimpan data dan perintah di memori dalam bentuk angka. Manusia tidak menggunakan angka melainkan nama untuk membedakan suatu bentuk dengan bentuk lainnya. Manusia lebih mudah mengenali bentuk/benda melalui nama daripada angka. Sebagai contoh, dalam sebuah jaringan LAN lebih mudah mengenali server melalui nama server daripada alamat IP (sebuah angka) server tersebut. Nama juga sangat berarti bagi sebuah program, seseorang penulis program menggunakan nama untuk membedakan data (variabel), fungsi, atau entity lain yang dikenal dalam sebuah bahasa pemrograman.

Istilah formal sebuah nama, dalam bahasa C/C++ disebut *identifier*. Nama (*identifier*) digunakan untuk menyatakan,

- data
- fungsi (function)
- typedef
- structure/class, dan anggotanya
- enumeration, dan anggotanya
- union
- label

dalam sebuah program C/C++. Sebuah program mengenali nama obyek melalui deklarasi nama tersebut. Scope membatasi nama (identifier), artinya sebuah nama hanya dapat digunakan dalam scope nama tersebut. Scope sebuah nama sudah ditentukan pada saat nama tersebut dideklarasikan. Nama dalam sebuah scope dapat dikenali di scope yang berbeda apabila sesorang penulis program menghendaki demikian. Sebuah scope

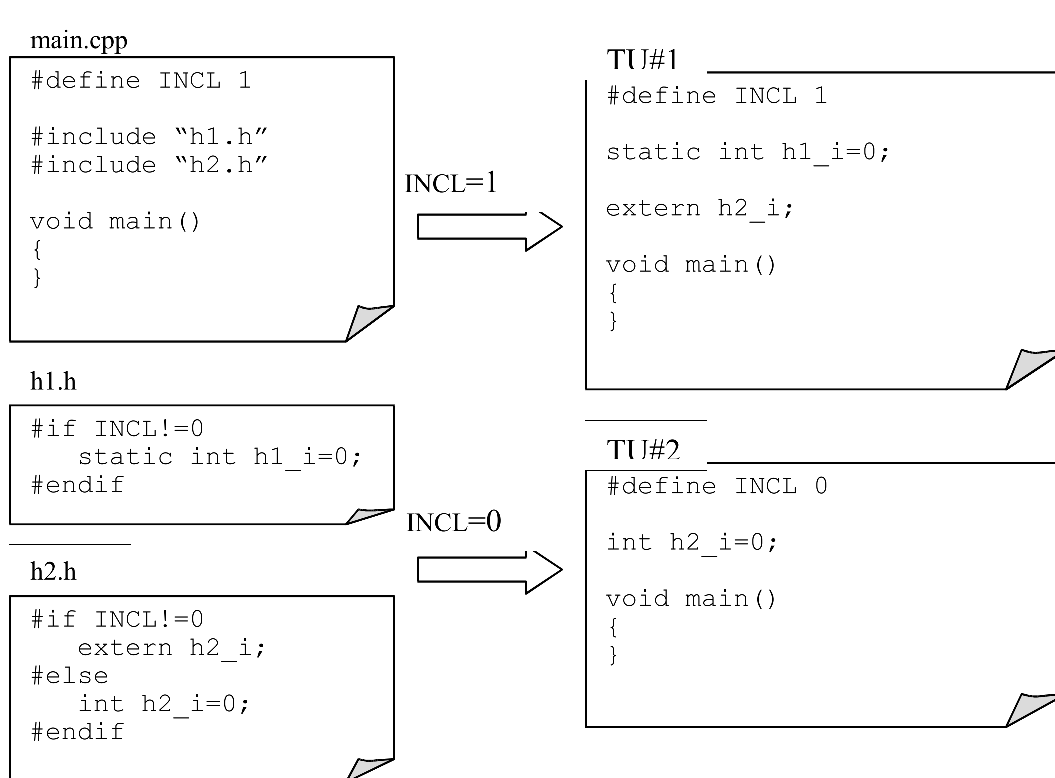
Copyright ©2004 Ilmukomputer.com

dapat mempunyai hubungan (*linkage*) dengan scope lain, karena menggunakan sebuah nama (identifier) yang sama, dengan kata lain nama (*identifier*) tersebut *visible* di scope yang berbeda.

Translation Unit

Ada perbedaan antara organisasi penulisan program C++, melalui file `.h` (berisi deklarasi) dan `.cpp` (berisi definisi), dengan proses pemilahan token (*parsing*) yang dilakukan compiler. Perbedaan tersebut tidak begitu menentukan dalam memahami konsep *scope*, *lifetime* dan *linkage* dalam C. C++ memiliki batasan scope yang lebih abstrak, oleh karena itu standard C++ memperkenalkan istilah *translation unit*.

Sebuah translation unit mungkin dibentuk dari beberapa file, karena umumnya sebuah program C++ menyertakan satu atau lebih file `.h`. Pada proses kompilasi, file berisi pernyataan program (source file) dan file lain yang dibutuhkan digabungkan menjadi satu kesatuan sebagai masukan proses pemilahan token.



Seperti ditunjukkan pada ilustrasi di atas [3], file `main.cpp`, `h1.h`, dan `h2.h` digabungkan menjadi sebuah translation unit. Compiler tidak mengikutsertakan baris program diantara `#ifndef` (`#if !defined`) atau `#else` jika syarat tidak terpenuhi. Sebagai contoh, jika INCL bernilai 1 maka dihasilkan translation unit TU#1, baris program diantara syarat kondisi pada file `h1.h` diikutsertakan. Apabila INCL bernilai 0, maka dihasilkan translation unit TU#2, baris program diantara syarat kondisional pada file `h1.h` tidak diikutsertakan dan baris program diantara syarat kondisi pada file `h2.h` yang diikutsertakan. Perlu diperhatikan juga bahwa compiler mengikutsertakan file yang dibutuhkan secara rekursif, jika file `h1.h` mencantumkan file *.h lainnya maka file tersebut termasuk dalam translation unit yang sama.

Proses pemilahan token menggunakan pernyataan program yang terdapat dalam translation unit dan bukan yang tertulis pada masing-masing file. Demikian halnya

Copyright ©2004 Ilmukomputer.com

dengan scope, lifetime dan linkage masing-masing identifer ditentukan berdasarkan translation unit.

Deklarasi dan Definisi

Sebuah deklarasi memperkenalkan sebuah nama dalam sebuah program. Deklarasi sebuah nama sekaligus menentukan scope nama tersebut. Definisi, selain memasukkan nama obyek[1] dalam sebuah program juga membentuk (*create*) obyek tersebut dan melakukan inisialisasi. Deklarasi dan definisi sering dipertukarkan, tetapi sebenarnya terdapat perbedaan deklarasi dan definisi.

Secara sederhana, sebuah program hanya mempunyai satu definisi terhadap satu obyek atau fungsi (function) dan sebuah program dapat mempunyai lebih dari satu deklarasi. Persyaratan tersebut dalam C++ dikenal dengan istilah aturan satu definisi (One Definition Rule(ODR)).

```
int i;                  //definisi
int i=0;                //definisi
extern int i;           //deklarasi
extern int i=0;         //definisi
static int j;           //definisi
static int j=0;         //definisi

void g(int k)           //deklarasi fungsi
void f(int k)           //definisi fungsi
{
   int l;
}
```

Dalam C++, seperti pada beberapa contoh di atas, sebuah deklarasi adalah sebuah definisi kecuali,

- sebuah deklarasi fungsi tanpa isi (*body*) fungsi tersebut, contoh fungsi `g`.
- sebuah deklarasi obyek dengan katakunci `extern` dan tanpa inisialisasi.
- sebuah deklarasi obyek (data member) `static` dalam scope class.

Jadi deklarasi menurut ODR merupakan bentuk pengecualian definisi, berbeda dengan C yang mengenal definisi tentatif (*tentative definition*). C++ menyarankan untuk tidak menggunakan “implicit `int`” dalam deklarasi. Dalam C, jika tidak ditulis type obyek maka dianggap obyek tersebut bertipe int,

```
static a;                   //a bertipe int
const b;                    //b bertipe int
void f(const c);            //c bertipe int
main(void) { return 0; }    //return type int
```

C++ menyarankan untuk menuliskan tipe obyek secara eksplisit, sebagai berikut,

```
static int a;
const int b=1;
void f(int const c);
int main(void) { return 0; }
```

[1] Pengertian obyek secara umum, semua obyek C yang menempati memori, tidak hanya obyek dari sebuah class.

Copyright ©2004 Ilmukomputer.com

Scope

Scope dan *lifetime* adalah dua konsep yang terkait erat. *Visibility* sebuah nama (*identifier*) dalam sebuah program C++ berbeda-beda, sebuah nama dengan scope global mempunyai *visibility* paling luas sedangkan sebuah nama dengan scope lokal mempunyai *visibility* lebih sempit misalkan sebatas eksekusi suatu fungsi. Sebuah scope menentukan batas *visibility* sebuah nama dalam program C++. Scope dapat dibentuk dengan sepasang tanda kurung {}, yang membentuk block scope, misalnya pada for-loop, while-loop, dll. Batasan scope dapat juga lebih abstrak seperti pada pembahasan translation unit.

Bahasa C mengenal beberapa bentuk scope, antara lain:

- block, scope nama adalah bagian program yang dibatasi oleh sepasang tanda kurung { }. Sebuah nama *visible* sejak deklarasi nama tersebut sampai dengan tanda kurung penutup } yang menandakan akhir masa pakai nama tersebut.
- prototipe fungsi (*function prototype*), scope nama hanya sebatas tanda kurung (dan) yang membentuk prototipe fungsi.
- fungsi, scope sebuah nama adalah seluruh badan fungsi tersebut.
- file, scope sebuah nama adalah keseluruhan file sejak sebuah nama dideklarasikan.

Scope block dan prototipe fungsi dikenal dengan scope lokal (*local scope*).

Bahasa C++ menambahkan dua scope lagi, yaitu class dan namespace. Deklarasi class (dan namespace) dibatasi oleh sepasang tanda kurung { }, walaupun definisi *member function* dapat ditulis di luar deklarasi class. Pada penulisan definisi seperti itu, *member function* tetap berada dalam scope class.

Scope minimal

Standar C++ mengubah sedikit aturan mengenai scope sebuah nama variabel dalam pernyataan *for-loop*, *while-loop*, atau *if-condition*. C++ mengijinkan deklarasi variabel sebagai bagian dari sebuah for-loop. Sebagai contoh deklarasi variabel i (*loop counter*) pada cuplikan baris program berikut ini,

```
for (int i=0; i<10; i++)       //deklarasi dan inisialisasi i
                               //dalam sebuah for-statement
{
    cout << i << endl;
}
int n=i;           //error: menggunakan i diluar scope
```

Pada contoh, scope i adalah badan for-loop tersebut yang dibatasi oleh pasangan tanda kurung { }. Penggunaan variabel i diluar scope adalah suatu pelanggaran terhadap aturan scope C++ yang terdeteksi pada saat kompilasi program. Perubahan aturan scope tersebut sejalan dengan teknik scope minimal (*minimal scoping*)[1]. Teknik scope minimal mempersempit jarak antara deklarasi sebuah nama dengan pemakaian nama tersebut. Teknik ini bertujuan untuk menambah *readability*[2] sebuah program. Scope minimal menunda alokasi obyek sampai obyek tersebut memang benar-benar dibutuhkan, seperti ditunjukkan pada contoh berikut

```
if (<kondisi>)
{
    int i,j;
```

[2] *Readability* adalah sebuah *quality characteristic* (*attribute*) sebuah produk perangkat lunak. Semakin tinggi *readability* semakin mudah *maintenance* perangkat lunak tersebut (semakin tinggi *maintainability*).

Copyright ©2004 Ilmukomputer.com

```
}
```

Jika <kondisi> tidak bernilai benar (.T.) maka tidak ada alokasi memori untuk obyek i dan j.

Tidak semua compiler C++ mengikuti aturan scope yang disempurnakan, salah satu yang paling menonjol adalah Microsoft Visual C++. Sampai dengan dengan versi 6, Visual C++ tidak mengikuti aturan scope yang baru dan hal ini adalah salah satu contoh ketidak sesuaian (*incompatibility*) VC++ dengan standar C++[6,7]. VC++ menggunakan konvensi pembentukan nama yang dikenal dengan notasi Hungarian (Hungarian notation) dengan mengikutsertakan tipe data dalam nama variabel. Dengan demikian pendekatan VC++ tidak memandang perlu memperpendek jarak antara deklarasi dengan pemakaian nama tersebut pertama kali.

Lifetime

Lifetime atau *storage duration* sebuah obyek adalah rentang waktu sejak obyek tersebut dibentuk sampai waktu saat obyek tersebut dihancurkan. Selama rentang waktu tersebut, nama tersebut mengacu ke lokasi memori yang ditempati oleh obyek tersebut, selepas rentang waktu tersebut, sebuah nama tidak lagi mengacu ke lokasi memori tersebut karena mungkin sudah ditempati oleh obyek lain.

C++ mengenal tiga jenis *storage duration* sebuah nama, yaitu: *static*, *automatic* (local), dan *dynamic*. Argument sebuah fungsi mempunyai *automatic storage duration*, artinya argumen tersebut dibentuk begitu eksekusi program memasuki fungsi tersebut (scope argumen tersebut) dan dihancurkan begitu eksekusi fungsi tersebut selesai (keluar dari scope fungsi). Sebuah obyek *automatic* menempati memori stack (*stack memory)*. Sebuah deklarasi obyek dengan scope block tanpa kata kunci `extern` atau `static` juga mempunyai *automatic storage duration* (seperti contoh scope minimal di atas). Sebuah obyek dengan *static storage duration* menempati memori sejak saat pertama eksekusi program, dengan demikian obyek static mempunyai alamat yang sama sepanjang eksekusi program. Obyek static menempati memori static (*static storage*). Sebuah obyek dengan *dynamic storage duration* menempati memori melalui fungsi alokasi (operator `new)` dan dilepas dengan fungsi dealokasi (operator `delete`). Obyek dinamik menempati memori heap atau memori dinamik (*free store*).

Storage specifier

C++ mengenal lima buah kata kunci yang menyatakan attribut *storage duration* sebuah nama (*storage class specifiers*), yaitu: `auto`, `register`, `extern`, `static` dan `mutable`. Sedikit perbedaan dengan C, `typedef` tidak termasuk *storage class specifier*.

Secara umum, dalam sebuah deklarasi tidak boleh menggunakan dua buah *storage specifier* secara bersamaan. Kata kunci `extern` dan `static` dapat menjadi bagian dari deklarasi obyek atau fungsi dalam scope namespace[3] dan scope block, akan tetapi `static` tidak dapat menjadi bagian deklarasi fungsi dalam scope block. `static` juga dapat menjadi bagian deklarasi dalam sebuah scope class.

`auto` dan `register` dapat menjadi bagian sebuah deklarasi dalam scope block maupun atau dalam deklarasi argumen fungsi (*formal parameter*), tetapi tidak dalam scope *namespace*, sebagai contoh:

```
register char *p;          //error – register dalam scope namespace
void f(register char *p);  //ok – register dalam formal parameter
```

[3] scope *namespace* termasuk scope file, di luar scope class dan scope block.

Copyright ©2004 Ilmukomputer.com

```
void f(auto int i);        //ok di C++, error di C
```

C++ mengijinkan penggunaan auto dalam argumen fungsi walaupun tidak ada pengaruh apapun, sedangkan C tidak memperbolehkannya.

`mutable` tidak berpengaruh terhadap linkage maupun *lifetime* sebuah nama, mutable hanya dapat digunakan dalam deklarasi obyek (*data member*) dalam scope class. `mutable` menyatakan bahwa sebuah obyek tidak pernah konstant.

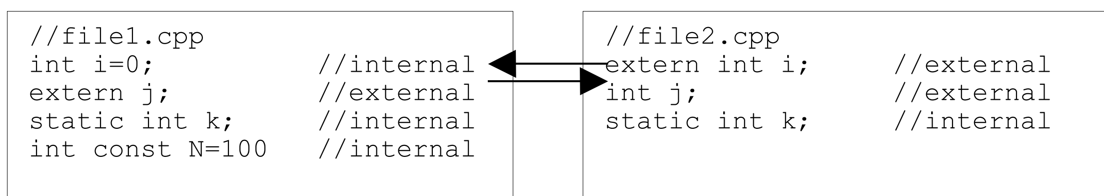
Linkage

Sebuah nama yang sama tetapi dalam translation unit yang berbeda dapat mengacu ke sebuah obyek yang sama. Hal ini, dalam C maupun C++, disebut *linkage*. Ada 3 macam linkage, masing-masing adalah:

1. *external linkage*, yaitu sebuah nama yang dapat terjangkau dari scope lain meskipun berbeda translation unit.
2. *internal linkage*, yaitu sebuah nama dalam sebuah translation unit yang dapat terjangkau di scope lain dalam translation unit yang sama.
3. tanpa (*no*) *linkage*, yaitu nama sebuah obyek dalam sebuah scope dan tidak terjangkau melalui nama tersebut dari scope lainnya.

Linkage sebuah obyek maupun fungsi ditentukan oleh kombinasi beberapa faktor, yaitu: scope, storage class specifier, dan linkage yang ditetapkan melalui deklarasi sebelumnya. Selain itu const juga mempunyai pengaruh terhadap hasil akhir linkage obyek atau fungsi.

Secara umum, `extern` pada deklarasi sebuah obyek atau fungsi dalam scope *namespace* menyatakan *external linkage*. Demikian halnya dengan deklarasi tanpa *storage class specifier* juga menyatakan *external linkage*. Sebaliknya `static` pada deklarasi obyek maupun fungsi menyatakan *internal linkage*. Sebagai contoh, definisi nama dalam file1.cpp dan file2.cpp berikut ini,



Pada contoh, definisi nama data tersebut mempunyai scope file, i mempunyai linkage external,sedangkan k mempunyai linkage internal. Hasil kompilasi (dan link) kedua file tersebut menghasilkan sebuah program dengan sebuah obyek i,j dan dua buah obyek k. Deklarasi j pada file1 mengacu ke obyek j di file2, deklarasi i pada file2 mengacu ke obyek i di file1. `static` pada deklarasi k mengacu ke obyek yang berbeda, walaupun kedua file menggunakan nama yang sama. Dalam C++, berbeda dengan C, const pada sebuah deklarasi menyatakan linkage internal, seperti deklarasi N pada contoh di atas. Namun demikian, `extern` pada deklarasi sebuah obyek const mengubah linkage obyek tersebut menjadi linkage eksternal.

Aturan linkage pada scope block berbeda dengan aturan linkage pada scope namespace (termasuk scope file). Dalam scope blok, nama obyek tidak mempunyai linkage (*no linkage*), kecuali terdapat `extern` pada deklarasi obyek tersebut. Nama parameter (formal parameter) juga tidak mempunyai linkage. Sebagai contoh,

```
void f(int i)      //f: eksternal
{                  //i:no linkage
   int j;          //no linkage
   static int k;   //no linkage
```

Copyright ©2004 Ilmukomputer.com

```
…
}
```

Pada contoh tersebut, nama fungsi f mempunyai linkage eksternal, nama parameter i tidak mempunyai linkage, nama variabel lokal j dan k tidak mempunyai linkage. Bagaimana dengan deklarasi yang menggunakan extern pada blok scope, seperti contoh berikut ini,

```
static int i=0;          //internal
extern int j;            //eksternal

void f()
{
   extern int i;         //internal
   extern float j;       //error
   extern int k;         //external
…
}
```

Seperti terlihat pada contoh, hasil akhir linkage obyek extern dalam scope blok bergantung kepada linkage obyek pada scope yang memuat scope blok tersebut. Obyek i dalam blok scope mempunyai linkage internal, mengikuti linkage obyek i dalam scope file. Obyek k mempunyai linkage eksternal, mengikuti linkage obyek k dalam scope file. Deklarasi obyek merupakan pelanggaran terhadap aturan linkage, karena tipe data j (float) dalam scope blok berbeda dengan tipe data j (int) dalam scope file.

Scope Class

Setiap definisi class membentuk sebuah scope baru. Scope class mencakup deklarasi class, masing-masing badan fungsi anggota class (*member function*), dan *constructor initializers*.

Akses terhadap anggota class dapat dilakukan menggunakan salah satu diantara beberapa cara dibawah ini,

a) operator .
b) operator ->
c) operator :: (*scope resolution operator*)
d) melalui fungsi anggota class tersebut (*member function*) atau derived class

Ketiga cara tersebut harus mengikuti batasan akses (*public*, *protected*, *private*) class tersebut. Keempat cara tersebut dapat digunakan untuk mengakses anggota class dengan batasan akses *public*. Cara keempat dapat digunakan untuk mengakses anggota class dengan batasan akses *protected* dan *public*.

Class dalam class (*nested class*) mempunyai scope terpisah, seperti contoh berikut ini,

```
struct s
{
   struct t
   {
      int i;
   } j;
}
```

Pada contoh di atas, struct t tidak terlihat di luar struct s sehingga akses ke struct t harus menggunakan operator ::, sebagai berikut:

Copyright ©2004 Ilmukomputer.com

```
s::t x;
```

Jika t tidak tersedia untuk akses public (protected atau private), maka operator :: tidak dapat digunakan untuk mengakses struct t karena batasan akses masih tetap berlaku.

Standar C++ sebelum dibakukan, memperkenalkan cara untuk mendefinisikan suatu besaran konstan (*compile-time constant*) yang mempunyai scope class. Sebagai contoh sebuah class C berikut ini,

```
class C
{
   int buf[10];
public:
   C();
};
```

mempunyai angota berupa array integer `buf` dengan jumlah elemen 10. Inisialisasi array `buf` dalam constructor perlu menggunakan operator `sizeof()` untuk mendapatkan jumlah elemen array, sebagai berikut:

```
C::C()
{
   for(int i=0;i<(sizeof(buf)/sizeof(&buf[0]));i++)
      buf[i]=i;
}
```

Cara yang lebih mudah untuk inisialisasi array `buf` adalah dengan mendefinisikan jumlah elemen array `buf`. Dua cara untuk mendefinisikan besaran konstan dengan scope class adalah,

`enum` **hack**	**Standar C++**

`class C`
`{`
`   enum { BUFSIZE=10 };`
`   int buf[BUFSIZE];`
`public:`
`   C();`
`};`
	`class C`
	`{`
	`   const int BUFSIZE=10;`
	`   int buf[BUFSIZE];`
	`public:`
	`   C();`
	`};`

Dengan demikian inisialisasi array `buf` dalam constructor tidak lagi menggunakan operator `sizeof()`, melainkan dengan nilai konstan `BUFSIZE`.

```
C::C()
{
   for(int i=0;i<BUFSIZE;i++)
      buf[i]=i;
}
```

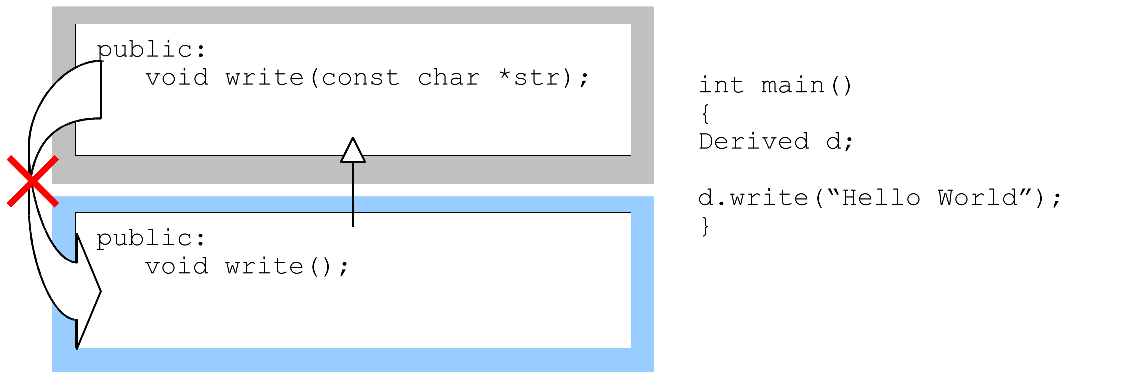
Cara ‘enum hack’ sering digunakan sebelum standar C++ memasukkan cara untuk mendefinisikan konstan dengan scope class. Cara ‘enum hack’ disebut demikian karena menggunakan enum tidak sesuai dengan penggunaan seharusnya. Namun demikian ‘enum hack’ masih lebih baik dari pada menggunakan *manifest constant* (`#define`) yang tidak mengenal batasan scope.

Overloading

Scope dan linkage adalah dua buah konsep dasar yang menjadi dasar *compiler* C++ dalam menguraikan penggunaan nama dalam sebuah program. Sebuah program dapat

Copyright ©2004 Ilmukomputer.com

menggunakan nama yang sama asalkan berbeda scope. C++ melonggarkan hal ini dengan memasukkan konsep *overloading* untuk nama fungsi. C++ mengijinkan sebuah program menggunakan nama yang sama untuk dua buah fungsi yang berbeda. *Overloading* dapat digunakan dalam scope file maupun scope class. Salah satu contoh adalah overloading anggota class berupa fungsi (*member function*), seperti pada contoh berikut ini,



Masing-masing class Base maupun class Derived membentuk scope tersendiri. Pada contoh, class Derived mempunyai fungsi write dengan signature yang berbeda. Hal ini tidak diperbolehkan dalam C++, karena *function overloading* tidak dapat menembus batas scope masing-masing class (*cross scope*). Cara pertama untuk mendapatkan hasil yang diinginkan adalah memanggil fungsi write class Base sebagai berikut,

forwarding function	`using` directive

`Class Derived : public Base`
`{`
`public:`
`    void write(const char *s)`
`    {`
`        Base::write(s);`
`    }`
`};`
	`Class Derived : public Base`
	`{`
	`public:`
	`    using Base::write;`
	`};`

Cara kedua adalah menggunakan `using` directive, namun hanya dapat dilakukan dengan compiler C++ yang sudah mengenal *namespace*.

Dalam menguraikan nama fungsi, linker C++ tidak hanya melihat nama fungsi melainkan *signature* fungsi tersebut. *Signature* merupakan rangkaian kata (token) yang dapat membedakan kedua fungsi, yaitu tipe masing-masing argumen (formal parameter) fungsi tersebut. Sebagai contoh, `char *strcpy(char *,const char *)`, mempunyai *signature* `{ char*,const char *}`. *Signature* hanya mengambil tipe masing-masing argumen, bukan nama argumen fungsi tersebut. Hal ini yang memungkinkan sebuah program C++ mempunyai deklarasi dua fungsi berbeda dengan nama yang sama dalam scope yang sama.

Class Linkage

Dalam hal nama obyek data maupun fungsi, external linkage dalam C++ dan C mempunyai pengertian yang sama, yaitu sebuah nama yang mengacu ke satu obyek dan nama tersebut dapat digunakan di translation unit yang berbeda. Sebagai contoh,

file1.cpp
```
void f() {}
```

Copyright ©2004 Ilmukomputer.com

```
int n;
```

file2.cpp

```
void f();          //mengacu ke fungsi f di file1.cpp
extern int n;      //mengacu ke obyek data n di file1.cpp
```

Linkage sebuah nama class berbeda dengan pengertian linkage pada nama obyek data maupun fungsi. Sebagai contoh,

Source File	Translation Unit

C.h
`class C`
`{`
`public:`
`  void f();`
`};`
 	
file1.cpp
`#include "C.h"`
 	
file2.cpp
`#include "C.h"`
	TU1
	`class C`
	`{`
	`public:`
	`   void f();`
	`};`
	
	TU2
	`class C`
	`{`
	`public:`
	`   void f();`
	`};`

Dalam C++, nama sebuah class, "C" pada contoh di atas, mempunyai *linkage external*. Pada contoh di atas kedua translation unit mempunyai definisi class C yang sama persis. Pengertian linkage eksternal pada nama obyek dan fungsi tidak dapat digunakan untuk menjelaskan pengertian linkage eksternal pada nama class "C" tersebut.

Hal inilah perbedaan yang muncul dengan masuknya konsep class dalam C++. Pengertian linkage eksternal secara umum, berlaku untuk nama data obyek, fungsi dan class, adalah setiap obyek dengan linkage eksternal harus mengikuti aturan "satu definisi" (ODR). Pada contoh, jika TU2 mempunyai definisi class C maka definisi tersebut harus sama dengan definisi class C pada TU1. ODR mensyaratkan, jika satu atau lebih translation unit mempunyai definisi sebuah obyek yang sama, maka definisi tersebut harus sama di semua translation unit (yang memuat definisi obyek tersebut). Salah satu cara praktis, sadar atau tidak sadar, setiap pemrogram memastikan hal ini dengan memasukkan definisi class dalam sebuah file *header* (.h) dan mengikutsertakan (melalui `#include`) obyek tersebut bilamana diperlukan, seperti kolom "source file" pada contoh di atas.

Static

Sebuah class dapat mempunyai anggota berupa data atau fungsi static. Tidak seperti obyek static pada scope file maupun blok, `static` dalam scope class mempunyai arti linkage eksternal. Selain operator . dan operator ->, anggota class berupa data dan fungsi static dapat diakses dengan operator ::.

Anggota class berupa data static harus dinisialisasi sekali saja, diluar scope class, yaitu dalam scope file tetapi tidak dalam file .h (header). Data static dalam scope class bukan merupakan bagian dari obyek class tersebut. Data static berlaku seperti halnya sebuah data global sebuah class, artinya hanya ada satu data static untuk semua obyek class tersebut.

Anggota class berupa fungsi static tidak mempunyai pointer `this`, karena hanya ada satu fungsi static untuk semua obyek class tersebut. Tanpa pointer `this`, fungsi static tidak dapat mengakses anggota class lainnya yang non-static. Dengan demikian fungsi

Copyright ©2004 Ilmukomputer.com

static umumnya digunakan untuk manipulasi anggota class berupa data statik. Salah satu contoh pemakaian fungsi static adalah pola desain (design pattern) Singleton.

Sebuah class dapat menempatkan data static dalam member function maupun sebagai anggota class seperti data member lainnya. Sebagai contoh,

```
class Base
{
public:
   int countCalls()
   {
       static int count=0;    //definisi static dalam member function
       return ++count;
   }
};
```

Semua class yang mewarisi (inherit) class Base tersebut juga mewarisi data static tersebut, sehingga `count` akan bertambah jika `countCalls()` dipanggil melalui obyek Base maupun Derived.

```
class Derived1 : public Base { };
class Derived2 : public Base { };

int main()
{
   Derived1 d1;
   Derived2 d2;

   int d1_count=d1.countCalls();    //d1_count=1
   int d2_count=d2.countCalls();    //d2_count=2
}
```

Penjelasan mengenai nilai obyek static pada contoh di atas merupakan latihan bagi para pembaca. Definisi data static sebagai anggota class lebih umum digunakan, seperti ditunjukkan pada contoh berikut,

```
class Base
{
private:
  static int i;
public:
  virtual int countCalls() {  return ++i; }
};
int Base::i;

class Derived1 : public Base
{
private:
  static int i; //menyembunyikan Base::i
public:
  int countCalls() {  return ++i; }
};
int Derived1::i;

class Derived2 : public Base
{
private:
  static int i; //menyembunyikan Base::i
public:
  int countCalls() {  return ++i; }
};
int Derived2::i;
```

Copyright ©2004 Ilmukomputer.com

```
int main()
{
  Derived1 d1;
  Derived2 d2;
  int d1_count = d1.countCalls(); //d1_count = 1
  int d2_count = d2.countCalls(); //d2_count = 1

  return 0;
}
```

Scope class Derived terpisah dengan scope Base maka class Derived tidak lagi mewarisi data static i class Base, sehingga countCalls memberikan nilai yang berbeda bila dipanggil dari obyek class Derived.

C++ mengenal class local, sebuah definisi class dalam scope blok, sebagai contoh:

```
#include <iostream>

int i=10;

int main()
{
   void f();
   f();
   return 0;
}

void f()
{
   static int j=20;

   class Local
   {
       int k;
    public:
      Local(int i) : k(i) {}
      void a() { std::cout << k+i << std::endl; }
      void b() { std::cout << k+j << std::endl; }
   };

   local l(30);
   l.a();
   l.b();
};
```

class lokal dapat digunakan (*visible*) hanya sebatas scope fungsi f. Definisi class lokal harus mencakup definisi semua anggota class berupa fungsi (*member function*), karena C maupun C++ tidak mengenal definisi fungsi dalam fungsi. Class local juga tidak mungkin mempunyai anggota berupa data static, karena data static memerlukan inisialisasi dalam scope file. Namun demikian scope class Local berbeda dengan scope fungsi f, artinya jika f mempunyai argumen maka argumen tersebut tidak dikenal dalam scope class Local.

Rangkuman

C++ seperti bahasa pemrograman tingkat tinggi (*high level language*) lainnya menggunakan nama untuk membedakan obyek yang berada dalam memori. Obyek

Copyright ©2004 Ilmukomputer.com

dalam C++ dapat berupa data, fungsi, structure /class, union dsb. Sebuah program C++ memperkenalkan nama obyek melalui deklarasi dan definisi. Secara umum, sebuah program hanya boleh mempunyai satu definisi obyek, sedangkan deklarasi dapat dilakukan lebih dari satu kali. Dalam C++, aturan ini dikenal dengan sebutan "One Definition Rule".

Sebuah program C++ tampak linear, tidak ada batas yang membatasi pemakaian nama obyek yang sudah diperkenalkan melalui deklarasi. Batas tersebut ada dan memang diperlukan, jika tidak maka konflik nama (*name conflict*) tidak dapat dihindarkan karena setiap nama harus unik dan berkorespondensi 1-1 dengan obyek yang berada dalam memori. Dalam C++, konsep scope menjelaskan aturan yang berlaku mengenai ruang lingkup sebuah nama dan pengaruhnya terhadap *visibility* sebuah nama. Sebuah nama dapat digunakan dalam perhitungan jika nama tersebut *visible*. Sebuah nama *visible* sebatas scope nama tersebut yang sudah tertentu saat pertama kali nama tersebut diperkenalkan melalui deklarasi. Jelas bahwa *scope* dan *visibility* adalah dua buah pengertian yang saling mempengaruhi.

Jika *visibility* sebuah nama ditentukan oleh scope demikian pula halnya dengan *lifetime* (*storage duration*) nama tersebut. Salah satu contoh yang paling mudah dipahami adalah variabel automatik dalam sebuah scope fungsi. Argumen fungsi (variabel automatik) terbentuk pada saat eksekusi program (*thread of control*) memasuki bagian awal scope fungsi dan variabel automatik hancur (*destroyed*) begitu eksekusi program meninggalkan scope fungsi. Artikel ini tidak membahas *lifetime* obyek temporer (*temporary object*). C++ mungkin membentuk obyek temporer, seperti dalam prefix dan postfix operator. Sebagai contoh, dalam `i++`, C++ membentuk obyek temporer, sedangkan dalam `++i` tidak. Hal ini dilakukan C++ di belakang 'layar'. Kiat (tip) dalam efisiensi program C++ sering menyarankan penggunaan operator prefix dibandingkan postfix karena alasan tersebut.

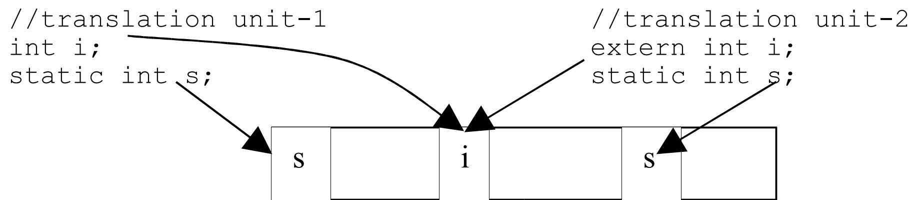
Scope sebuah nama dalam C++ berada dalam kendali (kontrol) seorang pemrogram. Sebuah nama tidak perlu mempunyai scope melebihi kebutuhan, tidak perlu menyimpan sesuatu obyek dalam memori terlalu lama. Seorang pemrogram perlu mengubah kebiasaan untuk menuliskan pernyataan program mengikuti scope minimal. Banyak pemrogram terbiasa menuliskan deklarasi variabel di awal fungsi, sesuatu yang lumrah dalam C maupun bahasa pemrograman lain. Dalam C++ deklarasi adalah definisi (kecuali obyek dengan linkage eksternal), dengan demikian sebuah obyek dibentuk dalam memori melalui sebuah definisi. Terlebih lagi jika sebuah definisi membentuk sebuah obyek dari sebuah class, maka semakin besar, semakin kompleks class tersebut semakin mahal 'biaya' pembentukan obyek tersebut. Terlepas dari gaya (*style*) penulisan program, notasi Hungarian bukanlah sesuatu yang tidak baik, kecuali jika penggunaan notasi Hungarian digunakan dengan mengesampingkan scope sebuah nama. Pemrogram tidak lagi mengkontrol *visibility* sebuah nama dengan membatasi scope nama tersebut. Notasi Hungarian tidak menggantikan kontrol *visibility* dan *lifetime* melalui penentuan scope nama obyek sesuai kebutuhan.

Abstraksi adalah suatu yang penting dalam setiap pemecahan persoalan pemrograman. Dalam terminologi OO, dikenal istilah *encapsulation*, suatu sarana untuk mewujudkan abstraksi tersebut. Dalam C++ *encapsulation* selalu dikaitkan dengan class. Encapsulation, secara umum, dapat dicapai melalui kontrol *visibility* melalui scope dan linkage. Sebagai contoh,

```
struct Point
{
   int x;
   int y;
}
```

Copyright ©2004 Ilmukomputer.com

adalah contoh abstraksi melalui *encapsulation* x,y dalam structure Point. Structure mewujudkan *encapsulation* melalui kontrol *visibility* menggunakan pembatasan scope, sehingga akses terhadap x dan y harus menggunakan operator . (dot). *Encapsulation* juga dapat dicapai menggunakan kontrol visibility melalui linkage, sebagai contoh



Translation unit 1 dan 2 menggunakan nama variabel i untuk mengacu ke obyek yang sama dalam memori. Translation unit 1 dan 2 menggunakan nama variabel s untuk mengacu ke dua buah obyek yang berbeda, karena static pada deklarasi menyatakan linkage internal. Dengan demikian akses terhadap s hanya terbatas di masing-masing translation unit, secara efektif akses s terbatasi melalui mekanisme linkage. Teknik ini memang lebih sering diterapkan menggunakan C.

Pengaruh storage specifier dan scope dalam menentukan hasil alkhir linkage fungsi dan obyek data disarikan dalam Tabel 1 dan Tabel 2[5].

Storage Class Specifier	**Linkage Fungsi**		
	Scope Namespace	**Scope Class**	**Scope Blok**
tanpa specifier	sama dengan `extern`	Linkage eksternal	sama dengan `extern`
`auto`			
`extern`	Linkage eksternal, kecuali deklarasi sebelumnya menyatakan linkage internal		Linkage eksternal, kecuali deklarasi sebelumnya dalam scope yang melingkupinya menyatakan linkage internal
`register`			
`static`	Linkage internal	Linkage eksternal	

Tabel 1. Linkage Fungsi berdasarkan scope dan storage specifier

Pada Tabel 1, tidak semua kombinasi scope dan storage specifier berlaku dalam menentukan hasil linkage fungsi.

Storage Class Specifier	**Linkage dan Lifetime Obyek**		
	Scope Namespace	**Scope Class**	**Scope Blok**
tanpa specifier	Linkage internal jika menggunakan (*qualifier*) `const`, lainnya linkage eksternal; storage static	tanpa linkage; storage bersatu dengan obyek class tersebut	sama dengan `auto`
`auto`			tanpa linkage; storage automatic

Copyright ©2004 Ilmukomputer.com

Storage Class Specifier	Linkage dan Lifetime Obyek		
	Scope Namespace	**Scope Class**	**Scope Blok**
`extern`	Linkage eksternal, kecuali deklarasi sebelumnya menyatakan linkage internal; storage static		Linkage eksternal, kecuali deklarasi sebelumnya dalam scope yang melingkupinya menyatakan linkage internal; storage static
			sama dengan `auto`
`static`	Linkage internal; storage static	Linkage eksternal; storage static	Tanpa linkage; storage static

Tabel 2. Linkage dan Storage Obyek berdasarkan Scope dan Storage Specifier

Seperti halnya Tabel 1, tidak semua kombinasi berlaku dalam menentukan hasil linkage obyek data. Selain linkage, Tabel 2, juga menunjukkan jenis storage yang digunakan. Artikel ini tidak membahas scope namespace, namun demikian Tabel 1 dan Tabel 2 dapat digunakan untuk menentukan linkage fungsi dan obyek data dalam scope file karena scope namespace juga mencakup scope file. Sebelum namespace menjadi bagian standar C++, deklarasi nama (identifier) di luar class atau fungsi mempunyai scope file. Setelah namespace menjadi bagian standar C++, scope file merupakan salah satu bentuk scope namespace dan disebut *global namespace scope*.

C++ mempunyai spesifikasi *linkage* berkaitan dengan penggunaan program yang ditulis dalam bahasa pemrograman lain. Spesifikasi tersebut dikenal dengan istilah ‘language linkage’. Language linkage juga dibutuhkan dalam menggabungkan (link) program C++ dan program C. Artikel ini tidak membahas *language linkage* secara lebih terperinci.

C++ adalah bahasa pemrograman yang kompleks, mempunyai aturan yang juga tidak sederhana untuk dipahami. Dalam komunitas C++ dikenal istilah ‘*language lawyer’*, sebuah istilah bagi orang yang mengetahui seluk beluk aturan C++ secara mendalam. Artikel ini tidak bertujuan ke arah pembahasan aturan C++. Artikel ini tetap berorientasi ke hal praktis pemrograman C++. Pengertian scope (dan konsep yang berkaitan) adalah dasar pemrograman C++, sesuatu yang penting dikuasai untuk menjadi pemrogram C++ yang baik. Di tangan seorang profesional, konsep atau prinsip dasar dapat menjadi alat yang efektif untuk memecahkan problem aktual. Salah satu contoh adalah idiom ScopeGuard[9]. Idiom tersebut memanfaatkan scope dan lifetime untuk memastikan destructor obyek class dipanggil jika terjadi *exception*.

Copyright ©2004 Ilmukomputer.com

Referensi

[1]. Chuck Allison, "Code Capsules: Visibility in C", C Users Journal vol. 12 no. 4.
[2]. Chuck Allison, "Code Capsules: Visibility in C++", C Users Journal vol. 12 no. 4.
[3]. Bobby Schmidt, "Completing the Foundation", C/C++ Users Journal vol. 13 no. 12
[4]. Dan Saks, "Storage Classes and Linkage", C/C++ Users Journal vol 15 no.11
[5]. Dan Saks, "Storage Classes and Language Linkage", C/C++ Users Journal vol 15 no.12
[6]. MSDN, Q243241 INFO: C++ Standard Noncompliance Issues with Visual C++ 6.0
[7]. MSDN, Q167748 PRB: Variable Scope in for-statement Extends Beyond Loop
[8]. G. Bowden Wise, "Statics: Schizophrenia for C++ Programmers", ACM
[9]. Andrei Alexandrescu and Petru Marginean, "Generic<Programming>: Simplify Your Exception Safe Code", C/C++ Users Journal C++ Experts Forum, December 1999.
[10].Bjarne Stroustrup, "The C++ Programming Language", 3rd Edition, Addison-Wesley, 1997

Copyright ©2004 Ilmukomputer.com

BIOGRAFI PENULIS

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).