MULTIPLE INHERITANCE UNTUK C++

Khoerul Anwar [*)]

ABSTRACT

Multiple inheritance is the ability of the class to provide more of a super class. Multiple inheritance in a programming language to support a program to be structured as a collection of inheritance lattice of a collection of inheritance tree. Multiple inheritance recognized to be an important structuring tool. It is as well as the recognition that multiple inheritance is a fairly difficult / complex in a programming language, is difficult to implement, and costly to run. Here the authors will try to explain that none of the three estimates are true.

***Keywords**: Multiple Inheritance, Super Class.*

PENDAHULUAN

1. Latar Belakang

Penelitian ini menjelaskan pengimplementasian dari mekanisme (mechanism) multiple inheritance C++[4,6]. Disini hanya menyajikan penjelasan paling mendasar bagaimana multiple inheritance secara umum dan bagaimana ia dapat digunakan dalam pemrograman. Variasi khusus dari penggunaan umum konsep multiple inheritance adalah penjelasan hal-hal utama dalam penggunaannya.

Pertama latar belakang multiple inheritance dan teknik-teknik implementasi C++ diperlihatkan, kemudian skema multiple inheritance untuk C++ dikenalkan dalam tiga tahapan.

1. Skema dasar untuk multiple inheritance, strategi dasar untuk resolusi yang rancu (ambiguity resolution), dan cara implementasi virtual function.
2. Penanganan dari penyertaan beberapa class (classes included) khusus dalam inheritance, programer mempunyai pilihan apakah menyertakan beberapa class induk akan menghasilkan sebuah atau beberapa sub object.
3. Detail dari pembangunan objek (construction of objects), perubahan objek (destruction of objects) dan pengendalian akses (access control).

KAJIAN TEORI

1. Multiple inheritance

Mempertimbangkan sebuah simulasi dari jaringan komputer. Setiap sambungan dalam jaringan adalah direpresentasikan dengan objek switch, setiap pengguna (user) komputer direpresentasikan dengan objek terminal dan setiap jalur komunikasi direpresentasikan sebagai objek line. Sebuah cara untuk memantau simulasi

[*)] Dosen STMIK PPKIA Pradnya Paramita Malang

tersebut akan ditampilkan oleh kedudukan objek dari variasi kelas-kelas pada screen (layar monitor). Setiap objek yang ditampilkan adalah merepresentasi dari objek kelas Display. Objek-objek kelas Display adalah dibawah kendali display manajer yang menentukan perubahan dilayar atau database.

Kelas-kelas Terminal dan Switch, keduanya adalah turunan dari kelas Task yang menyediakan fasilitas-fasilitas dasar untuk perilaku co-routine. Objek-objek dari Task dibawah kendali dari Task manajer yakni prosesor. Idealnya Task dan Display kelas dari standart library. Jika kita ingin menampilkan sebuah terminal dari kelas Terminal maka harus diturunkan dari kelas Display. Akan tetapi, kelas Terminal dapat diturunkan dari kelas Task. Dalam bahasa Inheritance, sebagaimana aslinya versi C++ atau Simula67, kita hanya mempunyai dua jalan pemecahan problem ini: yakni penurunan Task dari Display atau penurunan Display dari Task. Cara lain bisa saja ideal selama keduanya menciptakan sebuah ketergantungan antara dua fundamen dasar library dan konsep-konsep ketergantungan. Idealnya salah satu akan dapat memilih antara Task dan Display, Line adalah Display tetapi bukan Task; dan Switch adalah Task tetapi bukan Display.

Kemampuan untuk mengekspresikan penggunaan sebuah tingkatan kelas, kedalam turunan lebih dari satu kelas induk (base class), adalah sering direferensikan sebagai *multiple inheritance*. Contoh-contoh yang lain melibatkan representasi dari variasi macam-macam prosesor dan kompiler untuk berbagai mesin, bermacam debugger.

Pada umumnya multiple inheritance membolehkan pengguna mengkombinasikan konsep-konsep yang ditunjukkan sebagai kelas-kelas dalam sebuah panduan konsep yang direpresentasikan sebagai turunan sebuah kelas. Sebuah cara yang lazim dari penggunaan multi inheritance adalah untuk desain dengan memperhatikan kelas induk dengan tujuan untuk menciptakan kelas-kelas baru dengan pemilihan kelas induk dari setiap kumpulan kelas yang relevan. Dengan demikian seorang programer mampu menciptakan konsep-konsep baru menggunakan sebuah resep seperti “pick A atau pick B”. Contoh pada window, seorang pengguna (user) memilih model window dengan pilihan pada model yang ditampilkan. Sedangkan dalam debug misalnya seorang programer akan mengkhususkan debuger dengan memilih jenis prosesor dan kompiler.

Merepresentasikan seluruh konsep kombinasi. Penerapan ini hanya berlaku pada inheritance tunggal, kita memerlukannya untuk merepresentasikan informasi dan menyediakan pola kelas N+M+N*M. Heritance tunggal memperhatikan hal-hal dimana N==1 atau M==1. Penerapan multiple inheritance untuk menghindari ketergantungan replikasi pada suatu hal dimana nilai-nilai dari N & M keduanya lebih besar dari

*) Dosen STMIK PPKIA Pradnya Paramita Malang

pada 1. Hal ini nampak pada contoh N >=2 dan M >=2 keduanya bukanlah hal yang luar biasa, karena pada contoh diatas yakni window dan debuger mempunyai tipikal N & M yang lebih besar dari pada 2.

2. Strategi Implementasi dalam C++

Sebelum mendiskusikan tentang multiple inheritance dan mengimplementasikannya di C++, pertama kali sebaiknya kita deskripsikan hal utama tentang implementasi dari konsep turunan tunggal di C++.

Objek dari kelas C++ direpresentasikan dengan daerah bersebelahan dalam memori. Pointer objek dari kelas ditempatkan pada byte pertama di memori. Kompiler mengaktifkan pemanggilan anggota fungsi kedalam "ordinary" fungsi panggil dengan argumen "extra"; Argumen "extra" adalah pointerto objek untuk anggota fungsi yang dipanggil.

Perhatikan contoh kelas berikut:

```
Class A {
        Int a;
        Void f(int i);
}
```

Objek kelas A akan nampak seperti berikut:

```
--------------
|   int a;    |
--------------
```

Tidak ada keterangan penempatan dalam kelas A diatas, kecuali integer dispesifikasikan oleh user. Tidak ada informasi lokasi terhubung ke anggota fungsi dalam objek. Pemanggilan anggota fungsi A :: f diatas dapat dituliskan sebagai berikut:

```
A*pa;
Pa-> f(2);
```

Perubahan bentuk tampilan pemanggilan ke "ordinary function call" adalah sebagai berikut:

```
f_F1A(pa,2);
```

objek dari turunan tersusun secara jalin-menjalin antar anggota dari kelas yang terlibat.

```
Class A {int a; void f(int);};
Class B : A {int b; void g(int);};
Class C : B {int c; void h(int);};
```

Sekalilagi hal tersebut tidak ada keterangan "housekeeping" yang ditambahkan, sehingga objek dari kelas C dapat digambarkan sebagai berikut:

```
------------
|   int a   |
|   int b   |
|   int c   |
------------
```

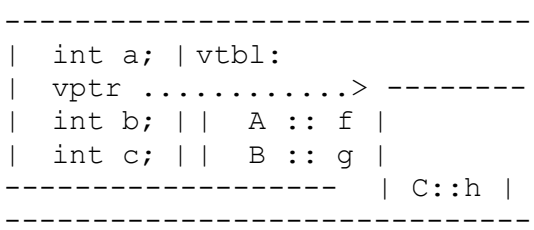
Kompiler mengetahui dengan pasti dari seluruh anggota-anggota objek turunan kelas, sebagaimana ia melakukan terhadap objek pada kelas tunggal dan turunannya begitu juga pada contoh kasusnya diatas.

Implementasi virtual function yang membentuk tabel fungsi, perhatikan berikut ini:

```
Class A {
  Int a;
  virtual void f (int;)
  virtual void g (int;)
  virtual void h (int;)
};
Class B : A {int b; void f(int);};
Class C : B {int c; void g(int);};
```


*) Dosen STMIK PPKIA Pradnya Paramita Malang

Dalam kasus ini, tabel dari virtual function misal kita beri identitas vtbl, berisi fungsi-fungsi yang cocok untuk pemberian kelas dan pointer ditempatkan pada setiap objek. Objek kelas C digambarkan seperti berikut ini:



Sebuah pemanggilan ke virtual function dilakukan perubahan bentuk terlebih dahulu kedalam bentuk pemanggilan tak langsung oleh kompiler. Perhatikan berikut ini:

```
C*pc;
Pc-> g(2);
```

Menjadi sesuatu yang baru seperti:

```
(C * (pc->vptr[1])) (pc,2);
```

Mekanisme multiple inheritance untuk C++ harus mempertahankan efisiensi dan keadaan pokok dari skema implementasi.

3. Multiple Base Classes

Berikut ini disajikan dua kelas:

```
Class A {---};
Class B {---};
```

Sebuah desain dengan tiga kelas sebagai kelas induk (base class) dan yang lain sebagai turunannya. Hal ini dapat dituliskan sebagai berikut:

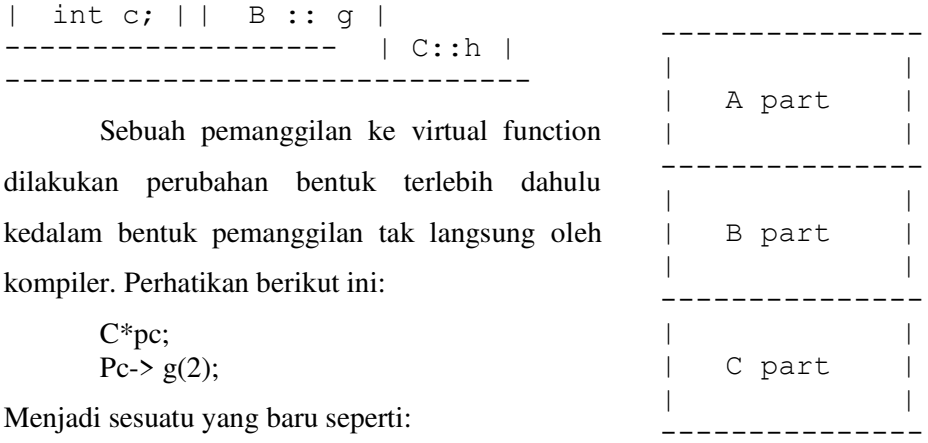
```
Class C : A,B {---};
```

Ini mempunyai arti bahwa C adalah bagian dari A dan B. penulisan tersebut akan sama artinya dengan penulisan sebagai berikut:

```
Class C : B,A {---};
```

4. Object Layout

Objek kelas C dapat ditampilkan sebagai berikut:



Penjelasan anggota dari kelas A, B atau C, perhatikan pada gambar diatas, kompiler mengetahui dengan benar letak objek dari tiap-tiap anggota dari turunannya yang sesuai.

5. Memanggil Anggota Fungsi

Pemanggilan anggota fungsi A atau c identik dengan apa yang telah dilakukan dalam kasus inheritance tunggal. Pemanggilan anggota fungsi dari B dengan pemberian C* lebih dari sekedar keterlibatan:

```
C*pc;
Pb-> bf(2);
```

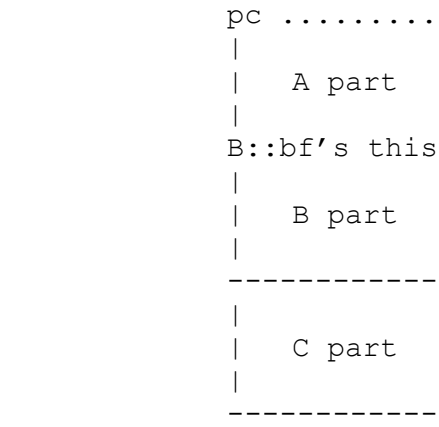
Naturalnya B :: bf(2) mengharapkan B*, untuk menyediakannya harus ditambahkan sebuah

*) Dosen STMIK PPKIA Pradnya Paramita Malang

konstanta ke pc. Konstanta delta(B), adalah posisi relatif terhadap B dari C. Delta ini menyatakan kompiler memanggil dengan merubah bentuk berikut:

bf_F1B((*B)((Char *) pc + delta(B)),2);

Penjelasan diatas setelah ada tambahan konstanta Delta(B) akan menjadi berbeda. Skema pemanggilan dari anggota fungsi B yakni pointer (B*) pada bagian tersebut menjadi sebagai berikut:



6. Kerancuan-kerancuan

Potensial yang dapat menimbulkan kerancuan adalah jika keduanya A dan B mempunyai anggota bersifat public dengan nama variabel yang sama:

```
Class A { int ii};
Class B { char * ii};
Class C : A, B { };
```

Dalam kelas ini C mempunyai anggota ii, sedangkan A ::ii dan B :: ii, sehingga kemudian menjadi sebagai berikut:

```
C*pc;
Pc-> ii
```

Disinilah letak kebingungan dari C, apakah ia memanggil ii bagian dari A atau ia memanggil ii bagian dari B.

```
pc-> A :: ii;
pc-> B :: ii
```

hal yang membingungkan bisa timbul jika keduanya A dan B mempunyai sebuah fungsi f()

```
Class A : { void f();};
Class B : { int f();};
Class C : A, B { };
        C*pc;
        pc -> f();          //error
        pc -> A :: f();
        pc -> B :: f();
```

Alternatif yang bisa dijadikan solusi dari kebingungan tersebut adalah dengan mengkhususkan pada saat setiap kelas induk memanggil fungsi (f()), salah satunya dengan mendefinisikan f() di C.

C :: f() akan memanggil kelas induk, contoh:

```
Class C : A,B {
    Int f() { A :: f(), B :: f();}
};
        C*pc;
        Pc-> f();                   // C :: f dipanggil
```

Solusi ini biasanya dianggap sebagai petunjuk yang terang bagi programer-progamer karena model ini menempatkan objek dari kelas turunan untuk dideklarasikan di kelas turunan.

7. Casting

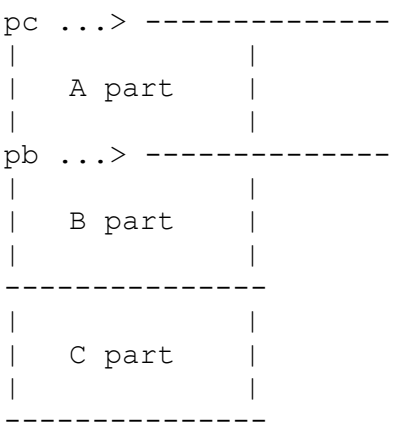
Ekplisit dan implisit casting mungkin juga meliputi modifikasi sebuah nilai ponter dengan Delta.

```
C * pc;
B * pb;
```


*) Dosen STMIK PPKIA Pradnya Paramita Malang

```
pb = (B*)pc;  // pb = (B*) ((char *)pc + Delta(B))
pb = pc;      // pb = (B*) ((char *)pc + Delta(B))
pc = pb;      // error cast needed
Pc = (C*) pb; // pb = (C*) ((char *)pc + Delta(B))
```

Casting pointer merepresentasikan bagian yang cocok pada objek yang sama



Perbandingan diinterpretasikan dengan cara yang sama:

```
pc == pb;     // pc == (C*) pb atau (B*)pc=pb
              // (B*)((Char *)pc + Delta(B))==pb
              // pc ==(C*)((Char *)pb– Delta(B))
```

8. Pointer bernilai Null

Pointer dengan nilai nul (zero) dapat mengurai masalah pada multiple base class. Perhatikan pada penulisan dibawah ini yang ditunjukkan zero nilai null pada pointer:

```
C * pc = 0;
B * pb = 0;
If (pb == 0) ---
pb = pc;              // pb=(B*)((Char*)pc +
Delta(B))
if (pb == 0) ---
```

pada pengujian yang kedua diatas akan gagal selama pb mempunyai nilai (B*)((char *) 0 + Delta(B)).

Solusinya mengkolaborasi operasi casting untuk mngetes pointer-values 0;

```
C * pc = 0;
B * pb = 0;
If (pb == 0) ---
pb = pc;          //pb=(pc==0)?0:(B)((Char *)pc
                  + Delta(B))
if (pb == 0) ---
```

9. Fungsi Maya

Naturalnya anggota fungsi mungkin bisa jadi virtual:

```
Class A : { virtual void f();};
Class B : { virtual void f(); virtual void g();};
Class C : A, B { void f();}
A * pa = New C;
B * pb = New C;
C * pc = New C;
pa ->f();
pb ->f();
pc ->f();
```

seluruh pemanggilan ini meminta C :: f(). Dibawah ini didefinisikan virtual function selama kelas C tersusun dari kelas A dan kelas B.

10. Implementasi

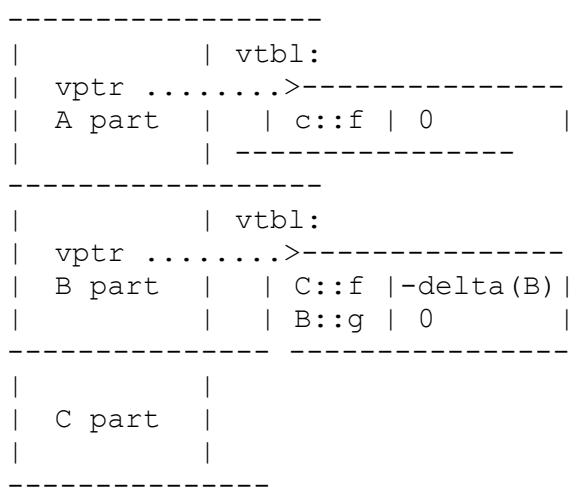
C :: f dalam hal ini pointer diletakkan diawal objek C. bagaimanapun hal ini tidak seperti pada umumnya pada saat compile B pointed to oleh pb yang merupakan bagian C, sehingga kompiler tidak dapat mengurangi konstanta. Konsekuansinya Delta (B) harus tersimpan sehingga pada saat run time ditemukan. Selama digunakan hanya ketika pemanggilan virtual function letak penyimpanan yang tepat pada tabel virtual function (vtbl). Untuk alasan-alasan

*) Dosen STMIK PPKIA Pradnya Paramita Malang

tersebut akan dijelaskan berikut, Delta disimpan dengan setiap fungsi dalam vtbl. Perhatikan penjelasan dibawah ini:

```
Struct vtbl_entry {
    Void (* fct)();
    Int delta;
};
```

Object kelas C akan nampak seperti berikut:




```
pb -> f();// memanggil C :: f
        // vtbl_entry*vt=& pb -> vtbl [index(f)];
        // (*vt->fct()((B*)((Char*)pb+vt->delta))
```

Catat objek pointer harus mengatur letak Sub-Objek dengan benar sebelum melihat peletakan anggota ke vtbl. Catat juga bahwa setiap komunikasi dari kelas induk dan turunannya ada di vtbl. Sebagai contoh, tabel vtbl untuk B di C adalah berbeda dari vtbl di B karena ada pemisah yang diletakkan. Implikasi pada umunya objek dari turunan kelas memerlukan vtbl untuk kelas induk plus satu untuk turunan kelas.

Bagaimanapun juga turunan tunggal dapat membagi (share) tabel vtbl dengan kelas induk sehingga contoh diatas hanya memerlukan dua vtbl, keduanya berlaku untuk objek bertipe di C. Gunakan int sebagai type data untuk menyimpan delta dalam ukuran terbatas pada objek tunggal. Cara seperti ini bukanlah pemikiran atau solusi yang jelek.

11. Kerancuan-kerancuan

```
Class A { virtual void f();};
Class B { virtual void f();};
Class C : A, B { void f();};
C * pc = New C;
pc-> f();
pc-> A::f();
pc-> B::f();
```

Meniadakan pernyataan virtual sehingga dua baris terakhir sungguh memohon pada fungsi kelas induk. Apakah ini suatu permasalahan? Biasanya tidak masalah. Salah satunya C mempunyai f() dan disini tidak diperlukan pernyataan dengan jelas atau C tidak mempunyai f() dan pernyataan tersebut adalah biasa dan itu dibetulkan. Masalah dapat terjadi ketika sebuah fungsi f() ditambahkan ke C dalam sebuah program sehingga berisi nama pernyataan yang jelas. Dalam kasus sebelumnya salah satu keajaiban mengapa seseorang menginginkan mendeklarasikan keduanya yakni sebuah fungsi dan juga pemanggilannya menggunakan pernyataan yang jelas. Jika f() adalah virtual, tambahkan f() kedalam kelas turunannya adalah cara yang nyata dalam menyelesaikan masalah yang membinggungkan.

Pada kasus dimana bukan pada C :: f pendeklarasian tidak dapat dikendalikan oleh

*) Dosen STMIK PPKIA Pradnya Paramita Malang

penyelesaian yang membingungkan pada saat pemanggilan. Perhatikan berikut ini:

```
Class A { virtual void f();};
Class B { virtual void f();};
Class C : A, B { };        // error: C::f needed
C * pc = New C;
pc-> f();         // ambiguous
A * pa=pc;     // implicit conversion of C* to A*
pa-> f();         // not ambiguous: calls A::f()
```

Potensial menjadikan kerancuan pada saat pemanggilan f() terdeteksi pada letak pada tabel dimana virtual fungsi A dan B dalam bangunan C. dengan kata lain pendeklarasian C diatas adalah rancu sebab hal tersebut akan berdampak pada saat pemanggilan fungsi, misal pa->f(), hal ini hanya akan membingungkan sebab bentuk informasi seperti ini akan hilang dengan paksaan. Jadi pemanggilan f() untuk objek type C adalah rancu.

12. Multiple Inclusions

Sebuah kelas bisa mempunyai beberapa kelas induk, sebagai contoh:

```
Class A : B1, B2, B3, B4, B5, b^ { … };
```

Pendefinisian kelas yang sama dua kali dalam kelas induk adalah tidak dianjurkan/dilarang. Contoh sebagai berikut;

```
Class A : B, B { … };    // error
```

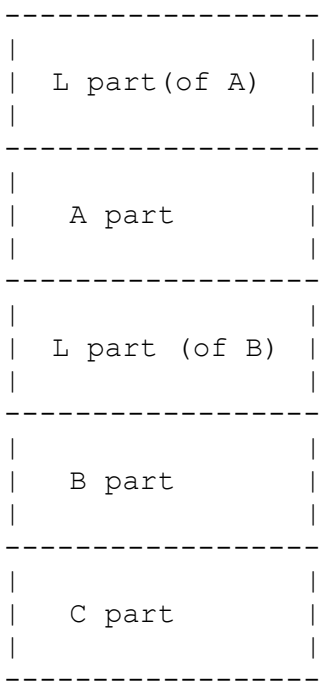
Alasan pelarangan adalah setiap akses ke anggota B akan menimbulkan kerancuan dan hal itu sebuah larangan. Hal ini sebuah rintangan dan juga sebuah penyederhanaan compiler.

13. Multiple Sub-objects

Sebuah kelas mungkin bisa mencakup lebih dari satu kelas induk, sebagai contoh:

```
Class L { … };
Class A : L { … };
Class B : L { … };
Class C : A, B { … };
```

Dalam kasus multiple objek dari kelas induk adalah bagian dari objek kelas turunan. Sebagai contohnya, objek kelas C terdiri dari 2 objek L satu dari kelas A dan satu lagi dari kelas B:



14. Pemberian Nama

Asumsinya bahwa kelas L dalam contoh diatas mempunyai member m. Bagaimana sebuah fungsi C :: f dapat mereferensikan ke L :: m?. Jawaban yang mudah dimengerti adalah dengan menulis rinci,

*) Dosen STMIK PPKIA Pradnya Paramita Malang

```
void C :: f() { A::m = B::m; }
```

hal ini akan berlaku juga pada A atau B yang mempunyai anggota m. Jika diperlukan, syntax C++ dapat dituliskan dengan lebih terperinci lagi:

```
void C :: f() { A::L::m = B::L::m; }
```

15. Casting

Simak sekali lagi contoh diatas. Fakta diatas adalah disana terjadi dua kali copy L membuat casting antara L* dan C* rancu, dan konsekuensinya itu adalah tidak dianjurkan atau larangan:

```
C* pc = New C;
L* pl = pc;        // error: ambiguous
pl = (L*)pc;       // error: still ambiguous
pl = (L*)(A*)pc;           // The L in C's A
pc = pl;           // error: ambiguous
pc = (L*)pl;       // error: still ambiguous
pc = (C*)(A*)pl;           // The C containing A's L
```

kita tidak menerapkan hal ini menjadi sebuah masalah. Pada tempatnya dimana hal ini akan nampak pada permukaan dalam kasus dimana A (atau B) dapat diatasi oleh pengharapan fungsi L; dalam kasus ini C akan dapat diterima meskipun C berisi sebuah A.

```
extern f(L*);      // some standard function
A aa;
C cc;
f(&aa);            // fine
f(&cc):            // error: ambiguous
f((A*)&cc);        //fine
```

Casting digunakan untuk menjelaskan agar tidak membingungkan.

16. Virtual Base Classes

Ketika kelas C mempunyai dua kelas A dan B hal ini menyebabkan pemisahan sub-objek sehingga tidak berelasi satu dengan lainnya dalam langkah yang berbeda dari selain objek A dan B. Saya menyebutnya *independent multiple inheritance*. Bagaimanapun banyak yang menduga tujuan dari penggunaan multiple inheritance tergantung kelas induk.

Sebagaimana ketergantungan dapat diekspresikan dalam waktu dari share objek antara berbagai kelas-kelas turunan. Dengan kata lain, secara khusus kelas induk harus memisahkan hanya satu objek pada akhir kelas turunan jika hal tersebut disebutkan beberapa kali. Untuk membedakan penggunaan dari multiple inheritance sebaiknya kelas-kelas induk dispesifikasikan kedalam virtual fungsi.

```
Class AW : virtual W { … };
Class BW : virtual W { … };
Class CW : AW, BW { … };
```

Objek tunggal dari kelas W adalah di share antara AW dan BW hal itu hanya satu objek W yang harus diikutkan dalam CW sebagai hasil dari turunan CW dari AW dan BW. Mengharap pada sebuah objek yang unik dalam penurunan kelas, virtual kelas induk bertindak tepat seperti kelas induk bukan turunan.

Penghilangan kata virtual dari W adalah ciri-ciri dari turunan khusus oleh AW dan BW dan bukan ciri-ciri dari W itu sendiri. Setiap virtual induk dalam heritance DAG mereferensikan ke objek yang sama.

*) Dosen STMIK PPKIA Pradnya Paramita Malang

Sebuah mungkin berciri sebuah kelas normal dan kelas virtual dalam heritance DAG:

```
Class A : virtual L { … };
Class B : virtual L { … };
Class C : A, B { … };
Class D : L, C { … };
```

Sebuah objek D akan mempunyai dua sub-objek dari kelas L, satu kelas normal dan satu lainnya kelas virtual.

Virtual base induk menyediakan cara untuk mensharing dalam lingkup heritance DAG tanpa memaksa mensharing informasi ke kelas induk terjauh, biarpun virtual base dapat digunakan untuk memperbaiki daerah reference. Cara lain dari penglihatan kelas turunan dari perintah virtual induk adalah sebagai alternative dan bisa terdiri dari interface ke virtual base.

17. Representation

Objek yang merupakan representasi dari sebuah kelas induk W objek tidak dapat diletakkan dalam posisi yang relatif pasti ke AW dan BW dalam semua objek. Konsekuensinya, *W harus disimpan pada seluruh objek yang aksesnya diarahkan ke objek W untuk kebebasan mengakses dari posisi relatif, sebagai contoh:

```
AW* paw = new AW;
BW* pbw = new BW;
CW* pcw = new CW;
-------------
paw ...>     | .........
|  AW part   | .
|            | v
-------------- .
|            | <........
|  W part    |
|            |
-------------
-------------
pbw ...>     | .........
|  BW part   | .
|            | v
-------------- .
|            | <........
|  W part    |
|            |
-------------
-------------
pcw ...>     | .........
|  AW part   | .
|            | v
-------------- .
|  .......   |
|  BW part   | .
|            | v
-------------- .
|            | .
|  BW part   | .
|            | v
-------------- .
|            | <.........
|  W part    |
|            |
-------------
```

Sebuah kelas dapat berdasarkan beberapa jumlah kelas dari kelas induk virtual. Satu dapat langsung dari sebuah kelas turunan ke sebuah kelas induk virtual, tetapi tidak dari kelas induk virtual ke kelas turunan.

Bentuk seperti itu melibatkan virtual base pointer. Yang kedua tidak dapat melakukan penyampaian informasi saat run time. Penyimpanan sebuah “back pointer” ke objek penutup pada umumnya adalah non-trivial dan mempertimbangkan ketidakcocokan untuk C++ seperti strategi alternatif dari dynamically keeping track object “for which” yang disetujui

*) Dosen STMIK PPKIA Pradnya Paramita Malang

menjalankan anggota fungsi bantuan. Penggunaan-penggunaan dari bac-pointer tidak berpengaruh pada implementasi yang komplek, penambahan permintaan ruang lebih dalam objek dan penambahan run-time dari inisialisasi.

Informasi sering langsung dari virtual base class ke kelas turunan adalah bagian dari apa yang diperlukan unjuk kerja pemanggilan maya dari sebuah deklarasi fungsi dalam virtual base dan menindihnya dengan kelas turunan. Ketiadaan dari "back-pointer" dapat diganti dengan pendefinisian dan pemanggilan fungsi virtual yang sesuai.

```
Class B { virtual void f(); … };
Class D : virtual B { void f(); … };
Void g(B* p)
{
// instead of casting B* to a D* here call f()
}
```

Sejak pencantuman secara eksplisit sering menghindari teknik seperti ini aktualnya menunjukkan ke arah yang lebih baik dalam banyak kasus.

18. Virtual Function

```
Consider;
class W {
virtual void f();
virtual void g();
virtual void h();
virtual void k();
…
};
Class AW : virtual W { void g() };
Class BW : virtual W { void f() };
Class CW : AW, BW { void h() };
CW* pcw = new CW;
pcw->f();          // BW::f()
pcw->g();          // AW::g()
pcw->h();          // CW::h()
((AW*)pcw->f();          // BW::f()
```

Objek CW mungkin nampak seperti ini:

```
-------------......|
| AW part  | v |    |
-------------......|
| BW part  | v |    |
-------------
|           |
| CW part  | v |    | vtbl:
---------------------...>|
vptr...>|BW::f|delta(BW)-delta(W)
|
|           |    |AW::g|-delta(W)|
| W part    |    |CW::h|-delta(W)|
|           |    |W::k |0|
--------------------------------
```

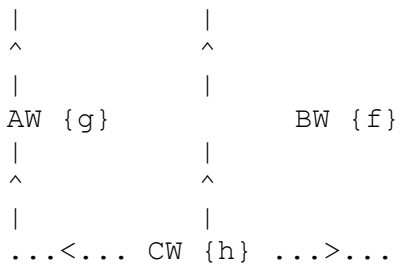
Pada umumnya, delta disimpan dengan sebuah pointer fungsi dalam vtbl adalah delta dari pendefinisian kelas dengan tanda minus adalah kelas untuk vtbl yang telah tersusun.

Jika W mempunyai sebuah virtual fungsi f itu adalah pendefinisian ulangdalam AW dan BW tetap tidak pada CW dalam keadaan rancu. Kerancuan tersebut mudah terdeteksi pada point dimana vtbl CW yang terbangun.

Peraturan untuk mendeteksi kerancuan dalam sebuah kisi-kisi kelas, atau lebih telitinya pada sebuah ireted Acyclic Graph (DAG) dari kelas-kelas, disana terdapat semua re-definitions dari kelas maya dari sebuah kelas-kelas induk maya harus terjadi pada akhir path tunggal DAG. Contoh diatas dapat digambarkan sebagai DAG seperti berikut ini:

```
....> W {f g h k} <...
```


*) Dosen STMIK PPKIA Pradnya Paramita Malang



Catatan bahwa sebuah pemanggil “up” diakhiri path dari DAG ke virtual function mungkin berhasil dalam memanggil dari fungsi (re-defined) dalam path yang lain (sebagaimana terjadi pada saat memanggil ((AW*)pcw->f() dalam contoh diatas).

19. Using Virtual Bases

Pemrograman dengan kelas virtual adalah ketangkasan dari pada pemrograman dengan kelas non virtual. Masalahnya adalah pada saat menghindari multiple pemanggilan dari fungsi dalam kelas maya ketika hal itu tidak diinginkan. Disini adalah gaya yang mungkin:

```
Class  W {
  // …
protected:
_f() { my stuff }
  // …
public:
f() { _f(); }
  // …
};
```

Setiap kelas menyediakan sebuah fungsi protected untuk mengerjakan “stuff”, _f(), untuk menggunakan dengan kelas-kelas turunan dan sebuah fungsi public f() sebagai antarmuka untuk digunakan dengan “general public”

```
Class A : public virtual W {
  // …
Protected:
_f() { my stuff }
  // …
public:
f() { _f(); W::_f(); }
  // …
};
```

Sebuah turunan kelas f() mengerjakan “own ftuff” dengan memanggil _f() dan kelas-kelas turunanya “own stuff” dengan memanggil _f().

```
Class B : public virtual W {
  // …
Protected:
_f() { my stuff }
  // …
public:
f() { _f(); W::_f(); }
  // …
};
```

Diluar kebiasaan, gaya seperti ini terdapat pada sebuah kelas bahwa turunan berdua dari kelas W ke W::f() hanya sekali saja:

```
Class C : public A, public B, public virtual W {
  // …
Protected:
_f() { my stuff }
  // …
public:
f() { _f(); A::_f(); B::_f(); W::_f(); }
  // …
};
```

Metode skema kombinasi, sebagaimana yang terdapat pada sistem Lisp dengan multiple inheritance yang dipertimbangkan sebagai sebuah cara untuk pengurangan yang berarti dari yang dibutuhkan seorang programeruntuk menuliskan kasus-kasus seperti contoh diatas. Namun demikian tak satupun dari skema-skema yang

*) Dosen STMIK PPKIA Pradnya Paramita Malang

nampak tersebut menampakkan kesederhanaan, umum dan cukup efisien untuk permasalahan yang komplek yang ditambahkan ke C++.

20. Constructors and Destructors

Konstruktor dan destruktor untuk kelas-kelas induk disebutkan sebelum konstruktor kelas turunan. Destruktor untuk kelas-kelas induk disebutkan setelah destruktor kelas turunan. Destruktor disebutkan dalam pesan pembalik dari yang telah dideklarasikan tentangnya.

Pernyataan konstruk kelas induk dapat dispesifikasikan sebagai berikut:

```
Class A { A(int); };
Class B { B(int); };
Class C : A, virtual B {
C(int a, int b) : A(a), B(b) { … }
```

Konstruktor-konstruktor dieksekusi dalam urutan yang nampak dalam daftar dari induk sehingga sebuah virtual base adalah selalu dikonstruksikan sebelum kelas turunan darinya. Kelas induk maya adalah selalu dikonstruksikan (once only) oleh kelas turunannya. Sbagai contoh:

```
Class V { V(); V(int); … };
Class A : virtual V { A(); A(int); … };
Class B : virtual V { B(); B(int); … };
Class C :  A, B { C(); C(int); … };
V v(1);          // use V(int)
A a(2);          // use V(int)
B b(3);          // use v()
C c(4);          // use v()
```

21. Access Control

Contoh-contoh diatasmengabaikan kontrol akses yang disarankan. Sebuah kelas mungkin bersifat public atau privat. Dalam ketentuannya hal itu mungkin maya. Sebagai contoh:

```
Class D {
  B1          // private (by default), non-virtual (by
              default)
  virtual B2  // private (by default), virtual
  public B3   // public, non-virtual (by default)
  public virtual B4 {
    // …
};
```

Catat bahwa sebuah akses atau aplikasi khusus virtual ke kelas induk tunggal hanya satu kali saja. Sebagai contoh,

```
Class C : public A, B { … };
```

Deklarasi kelas public A dan private untuk B. Kelas maya tersebut adalah mudah dicapai pada sebuah path jika path yang lain menggunakan turunan bertipe private.

22. Overheads

Keseluruhan diatas menggunakan skema berikut ini:

1. Sekali pengurangan dari sebuah konstanta untuk setiap menggunakan sebuah anggota dalam kelas induk tersebut adalah diikutkan kelas kedua atau kelas induk berikutnya.
2. Satu kata tiap fungsi dalam vtbl (to hold the delta).
3. Satu referensi memori dan sekali pengurangan untuk setiap memanggil virtual function.
4. Satu referensi memori dan sekali pengurangan untuk akses dari anggota kelas induk dari turunan kelas induk.

*) Dosen STMIK PPKIA Pradnya Paramita Malang

Perhatikan bahwa poin 1 dan 4 hanya terkena dimana multiple inheritance digunakan, tetapi poin 2 dan 3 terkena untuk setiap kelas dengan fungsi virtual dan untuk setiap fungsi virtual memanggil sesuatu ketika multiple inheritance tidak digunakan. Point 1 dan 4 hanya terkena ketika anggota-anggota dari induk yang kedua atau induk berikutnya di akses dari luar kelas,anggota fungsi dari virtual base class tidak melakukan hal khusus dari poin-poin tersebut diatas ketika pengaksesan anggota-anggota dari kelas.

Implikasi tersebut kecuali untuk 2 dan 3 anda tanggung hanya pada saat anda menggunakan 2 dan 3 mengenakan point yang kurang pentingpada mekanisme fungsi virtual dimana hanya digunakan inheritance tunggal. Point berikutnya dapat dilewatkan dengan menggunakan alternatif implementasi dari multiple inheritance, tetapi saya tidak tahu dari implementasinya, hal itu akrab juga dalam kasus multiple inheritance dan sebagai mana telah dideskripsikan.

Beruntung poin tersebut(1 s.d 4) tidak terlalu signifikan. Waktu ruang dan komlesitas yang dikenalkan pada kompiler ke implementasi inheritance tidak mudah dilihat oleh pengguna.

23. Kemudahan

Apa yang membuat sebuah fasilitas bahasa sulit digunakan?

1. Banyak aturan-aturan.
2. Sukar dipahami perbedaan-perbedaan antara peraturan yang satu dengan yang lain.
3. Tidak secara otomatismendeteksi kesalahan perintah.
4. Tidak mmpunyai keteraturantidak mempunyai keumuman.
5. Tidak mencukupi.

Dua kasus yang pertama mengunjukkan tingkat kesulitan memperlajhari dan pengingatan, disebabkan salah pnggunaan dan salah pemahaman.

Dua kasus yang terakhir dan menyebabkan kesalahan dan membingungkan programer mencoba mematahkan aturan-aturan dan simulasi kesalahan kedepannya. Point 3 menyebabkan keputusan programer untuk penemuan kesalahan yang dirasa jalan yang sulit.

Multiple inheritance skema yang ditunjukkan disini menyediakan dua jalan dari nama kelas lanjutan:

1. Kelas induk.
2. Virtual kelas induk.

Dua jalan penciptaan / penspesikasian kelas baru lebih disarankan dari pada dua jalan penciptaan yang berbeda. Aturan untuk penggunaan kelas-kelas tidak tergantung pada bagaimana nama berlanjut:

1. Kerancuan adalah tidak disarankan.
2. Aturan untuk menggunakan anggota-anggota apakah mereka untuk inheritance tunggal.
3. Kemunculan aturan-aturan apakah mereka untuk inheritance tunggal.

*) Dosen STMIK PPKIA Pradnya Paramita Malang

4. Inisialisasi aturan-aturan apakah mereka untuk inheritance tunggal.

Pelanggaran dari aturan –aturan ini terdeteksi oleh compiler. Dengan kata lain skema multi inheritance adalah hanya lebih lengkap digunakan dari pada skema inheritance tunggal yang ada:

1. Anda dapat lanjutkan nama kelas lebih dari sekali.
2. Anda dapat lanjutkan nama kelas dalam dua cara dari pada hanya satu cara.

Penambahan harus dibagi ke bagian dalam anggota kelas-kelas fungsi anggota pemrograman dari kelas-kelas dengan virtual base kelas; lihat bagian 7.

Pada beberapa hal itu mungkin juga menjadi masalah bahwa pointer ke sub-objek lebih sering digunakan. Ini akan berpengaruh pada program yang menaruh dengan jelas tipe non-objek dan "extra linguistic". Selain itu dengan penuh harapan normal adanya seluruh manipulasi dari objek pointer diikuti aturan-aturan yang konsisten pada sub bahasan 4,7 dan 8.

Kesimpulan

Multiple inheritance adalah sebuah pemikiran sederhana untuk ditambahkan ke C++ dalam jalan membuat mudah menggunakannya. Multiple inheritance tidaklah terlalu sulit untuk diimplementasikan selama hal itu permintaannya syntatic selanjutnya hanya sedikit dan fits dasarnya kedalam tipe struktur. Implementasinya adalah sangat efisiean dalam waktu dan ruang. Kesesuaiannya dengan C adalah nyata. Pembawaan ke berbagai program adalah nyata.

DAFTAR PUSTAKA

Tom Cargill: PI: *A Case Study in Object-Oriented Programming*. OOPSLA'86 Proceedings, pp 350-360, September 1986.

Stein Krogdahl: *An Efficient Implementation of Simula Classes With Multiple Prefixing*. Research Report No. 83 June 1984, University of Oslo, Institute of Informatics.

Stan Lippman and Bjarne Stoustrup: Pointers to Members in C++ Proc. USENIX C++ Conference, Denver, October 1988.

Bjarne Stoustrup: *The C++ Programming Language*. Addison-Wesley, 1986.

Bjarne Stoustrup: *What is "Object-Oriented Programming?*". Proc. ECOOP, Springer Verlag Lecture Notes in Computer Science, Vol 276, June 1987.

Bjarne Stoustrup: *The Evolution of C++: 1985-1989*. USENIX Computing System Vol 2 no 3, Fall 1989.

Daniel Weinreb and David Moon: *Lisp Machine Manual*. Symbolics, Inc. 1981.

*) Dosen STMIK PPKIA Pradnya Paramita Malang