Modul 3 Programming

Object-Oriented Programming (OOP)

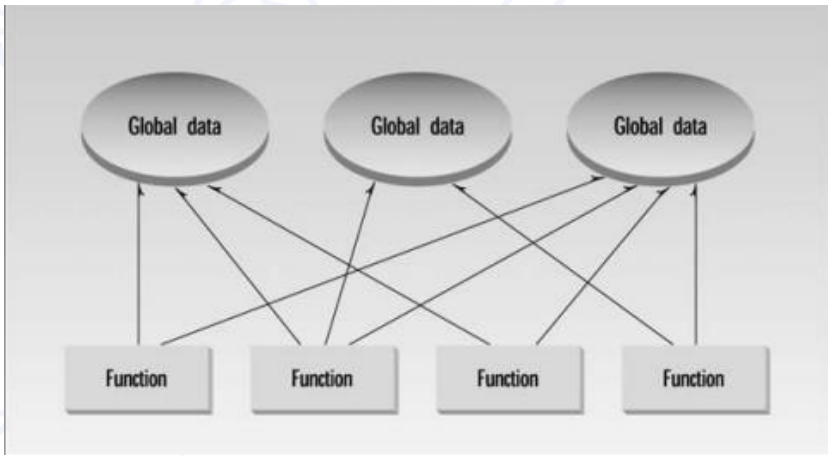
Paradigma Pemrograman

Paradigma pemrograman adalah suatu strategi atau metodologi tertentu untuk menyelesaikan suatu masalah dengan cara pemrograman. Paradigma pemrograman juga dapat diartikan sebagai sebuah *styles* atau cara mengorganisir suatu program. Paradigma bukanlah suatu *tools* atau software, melainkan sebuah gagasan atau *guidline* dalam menulis program. Terdapat berbagai paradigma dalam pemrograman, seperti *imperative programming*, *procedural programming*, *functional programming*, *declarative programming*, dan, *object-oriented programming*. Pada modul ini, tidak akan dibahas masing-masing paradigma, hanya paradigma *object-oriented programming* saja yang menjadi fokus.

Why Do We Need Object-Oriented Programming?

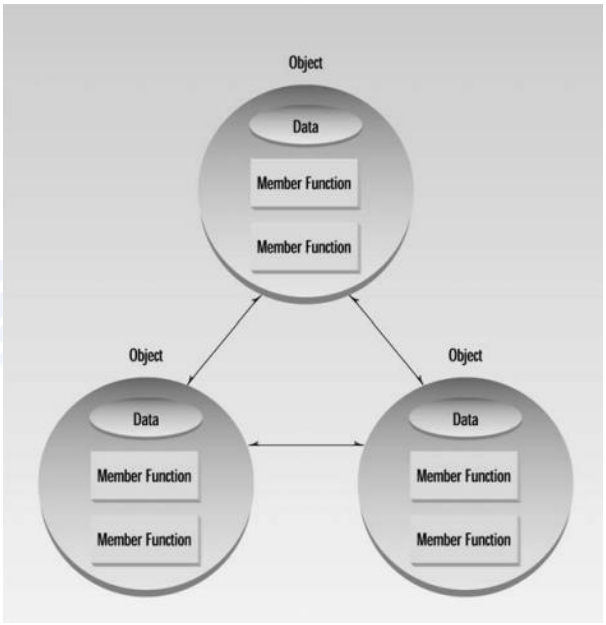
Untuk program yang kecil, faktor *readability* dan *mantainability* dari suatu program umumnya tidak akan menjadi masalah. Akan tetapi, jika skala dan kompleksitas dari program kita semakin besar maka *readability* dan *mantainability* dari program akan menjadi masalah besar jika program tidak dibuat dengan baik, terutama, jika Anda bekerja dalam suatu tim yang mengharuskan antarindividu saling bekerja sama.

Salah satu strategi untuk menghadapi permasalahan di atas adalah dengan memecah program menjadi fungsi yang kemudian dapat dikelompokkan kembali dari kumpulan fungsi tersebut menjadi suatu entitas disebut modul (biasanya dalam bentuk file). Akan tetapi, saat kompleksitas program semakin tinggi, paradigma prosedural seperti itu pun tetap menunjukkan gejala kewalahan. Mau sebagus apapun implementasi *code* secara prosedural, terdapat kelemahan dari paradigma tersebut, yaitu pertama, yaitu (1) sebuah fungsi punya akses tidak dibatas dalam mengakses global variabel dan (2) tidak secara eksplisit menunjukkan hubungan antara fungsi dan data sehingga basis dari paradigma prosedural tidak memberikan model representasi yang baik terhadap objek di dunia nyatanya.



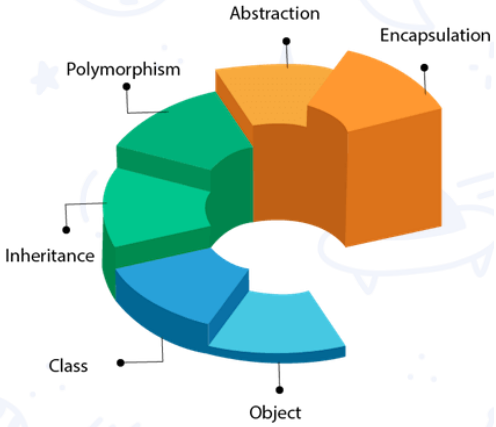
Gambar 1 *Procedural Programming*

Saat skala program membesar koneksi antara global data dengan fungsi yang terlalu banyak akan menimbulkan masalah karena jika salah satu global data diubah maka semua fungsi yang terkait harus diubah. *Object-oriented programming* merupakan paradigma yang hadir untuk menyelesaikan masalah dari *procedural programming*. Ide Fundamental dari *object-oriented programming* adalah menggabungkan antara data dengan fungsi menjadi satu unit yang kemudian unit tersebut disebut **objek**. *Object-oriented programming* juga mengadopsi konsep yang merepresentasikan objek di dunia nyata, yaitu dengan menambahkan **atribut** dan ***behavior***. Contoh, pada mobil terdapat atribut, seperti lampu rem, lampu mundur, *steering wheel*, dll. Kemudian contoh *behavior* pada mobil, seperti ketika Anda menekan pedal rem maka mobil akan mengerem dan menyalakan lampu remnya sehingga analogi dari atribut adalah variabel atau data dan analogi dari *behavior* adalah fungsi.



Gambar 2 *Object Oriented Structure*

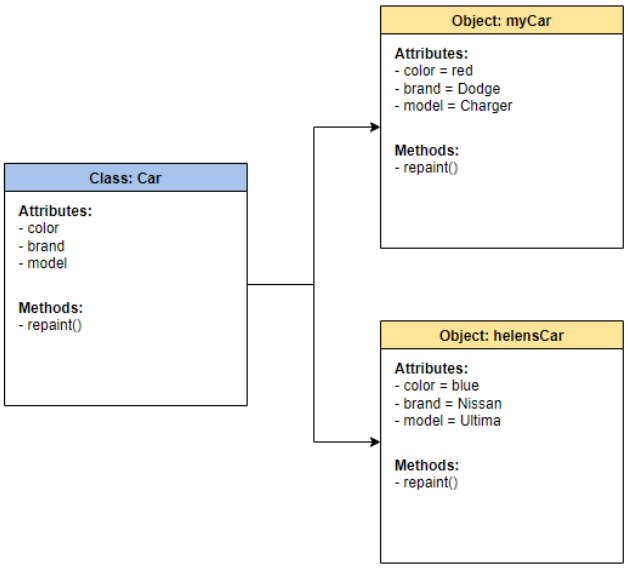
Major Elements and Concept of OOP



Gambar 3 Konsep OOP

Objek dan Class

Class adalah sebuah blueprint dari objek. Class juga biasanya merepresentasikan suatu kategori, salah satu contohnya seperti mobil. Terdapat banyak sekali tipe mobil di dunia ini, misal seperti mobil mercedes benz, atau mobil bmw, dan banyak lainnya. Dari contoh yang diberikan, mobil adalah sebuh class dan *instance* dari class, yaitu mobil mercedes benz dan mobil nissan merupakan objek. Setiap class akan memiliki **attribut** dan **method**. Attribut merujuk pada sifat atau variabel yang dimiliki class, sedangkan method adalah sebuh fungsi yang dimiliki class.



Class blueprint being used to create two Car type objects, myCar and helensCar

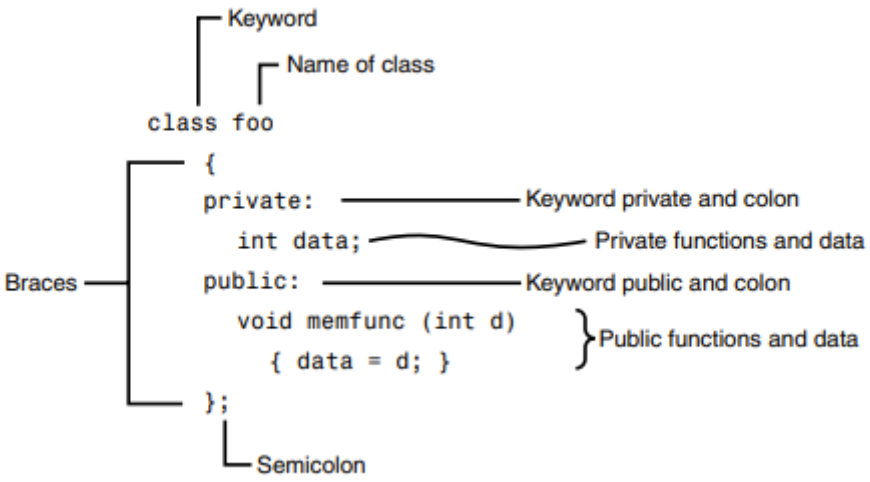
Pada contoh diagram di atas, class mobil memiliki atribut color, brand, dan model. Kemudian class tersebut memiliki method, repaint(), yaitu untuk mengecat ulang warna mobil. Berikut adalah contoh implementasi code untuk diagram di atas

```
#include <iostream>
#include <string>

class Car                                   // define a class
{
private:
    std::string color;                      // attribute
    std::string model;
    std::string brand;

public:
    Car (std::string warna, std::string model, std::string brand) { // Constructor
        this->color = warna;
        this->model = model;
        this->brand = brand;
    }
    Car() = default;                        // Default Constructor
    ~Car() {                                // Destructor
        std::cout << "Destructor dijalankan\n";
    }

    void repaint(std::string warnaCat) {    // Member function to set data
        this->color = warnaCat;
    }
    void setModel(std::string model) {
        this->model = model;
    }
    void setBrand(std::string brand) {
        this->brand = brand;
    }
    std::string getColor() {               // Member function to get data
        return this->color;
    }
    std::string getModel() {
        return this->model;
    }
    std::string getBrand() {
        return this->brand;
    }
};
```

```cpp
int main()
{
    // define two objects of class Car
    Car myCar;
    Car helensCar("blue", "Ultima", "Nissan"); // set data dengan constructor

    myCar.setBrand("Dodge");                // call member function to set data
    myCar.setModel("Charger");
    myCar.repaint("red");

    std::cout << "My car is a " << myCar.getColor() << ' '
              << myCar.getBrand() << ' ' << myCar.getModel() << '\n';
    std::cout << "Helens car is a " << helensCar.getColor() << ' '
              << helensCar.getBrand() << ' ' << helensCar.getModel() << '\n';

    return 0;
}
```

Mari kita bedah kode di atas!

Secara umum, berikut adalah syntax untuk mebuat definisi class



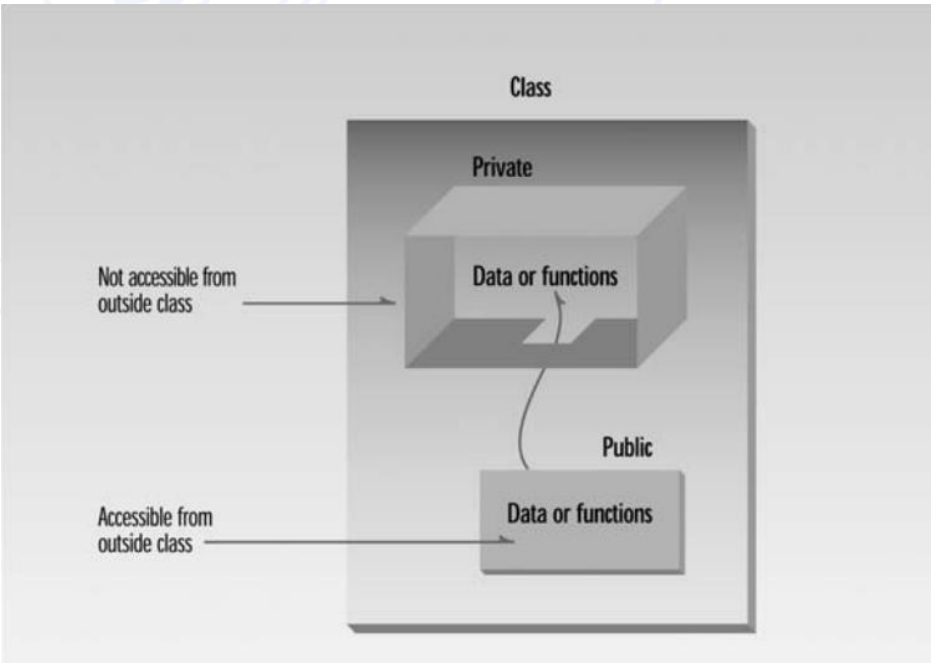
Access Modifier

Salah satu fitur fundamental dari OOP adalah penyembunyian data atau biasa disebut enkapsulasi. Penyembunyian data dalam konteks OOP adalah menyimpan data dalam class

dan data tidak dapat secara langsung diakses dari luar class. Untuk melakukan hal tersebut, terdapat tiga keyword untuk access modifier.

- Public
- Private
- Protected

Member dari class yang dideklarasikan di dalam keyword private hanya bisa diakses oleh fungsi di dalam class, sedangkan setiap member dalam keyword public dapat langsung diakses dari luar class. Secara default (Jika Anda tidak men-specify keyword private atau public) oleh C++ akan dianggap sebagai private. Keyword terakhir adalah protected. Keyword protected memungkinkan untuk member data bersifat private, tetapi data tersebut bisa diakses secara langsung oleh class turunannya. Penjelasan mengenai class turunan akan dijelaskan lebih rinci dalam materi inheritance.



Gambar 4 public dan private

Class Data

Class Car memiliki tiga item data, yaitu color, model, dan brand yang semuanya bertipe data string. Data item pada class biasanya disebut *data members*. Kemudian fungsi yang berada dalam class biasanya disebut *member function*. Pada class Car, terdapat beberapa member function, yaitu repaint(), setModel(), dan setBrand() ketiga fungsi tersebut adalah fungsi setter karena tugasnya adalah mengubah value dari data member. Ingat bahwa data member merupakan private sehingga diperlukan fungsi yang berada di dalam class untuk mengubah valuenya. Ketiga fungsi lainnya getColor(), getBrand(), dan getModel() merupakan fungsi getter yang tugasnya adalah untuk mengakses nilai data member dari luar class.

Functions Are Public, Data Is Private

Biasanya, data dalam class bersifat private dan fungsi-fungsi bersifat publik. Data disembunyikan agar aman dari manipulasi yang tidak disengaja, sedangkan fungsi-fungsi yang beroperasi pada data bersifat publik agar dapat diakses dari luar class. Namun, tidak ada aturan yang mengatakan bahwa data harus bersifat private dan fungsi bersifat publik; dalam beberapa situasi, Anda mungkin menemukan bahwa Anda perlu menggunakan fungsi yang bersifat private dan data yang bersifat publik.

Inisialisasi Objek

Perhatikan pada contoh kode di atas, ditunjukkan dua cara untuk menginisialisasi sebuh objek

```
Car myCar;
Car helensCar("blue", "Ultima", "Nissan");
```

Cara pertama, yaitu menginisialisasi objek myCar dengan syntax seperti sebuah variabel biasa. Cara kedua, yaitu dengan menginisialisasi objek helensCar sembari memberikan argumen untuk mengisi nilai dari member data. Hal tersebut dapat terjadi karena kita meletakkan *constructor* dalam class kita.

Constructor

Constructor adalah fungsi yang pertama kali akan dijalankan ketika suatu objek dibuat. Jadi, setiap kali Anda membuat objek, pastilah constructor yang pertama kali akan dijalankan. Secara default kalau Anda tidak mencantumkan constructor maka oleh C++ akan dibuatkan *default constructor* secara otomatis, di belakang layar tanpa ditunjukkan kepada Anda. Fungsi utama dari constructor biasanya adalah untuk menginisialisasi member data dalam class tersebut.

```
Car (std::string warna, std::string model, std::string brand) { // Constructor
    this->color = warna;
    this->model = model;
    this->brand = brand;
}
Car() = default;                        // Default Constructor
```

Seperti pada contoh di atas, syntax untuk constructor sama seperti membuat fungsi pada umumnya hanya saja *nama dari fungsi tersebut haruslah sama dengan nama dari class*.

Destructor

Jika constructor adalah fungsi yang dijalankan pertama kali ketika objek dibuat, Destructor adalah fungsi yang dijalankan paling terakhir, yaitu ketika suatu objek akan dihapus. Destructor akan terpanggil dalam kondisi-kondisi berikut

- Ketika objek sudah keluar dari scope pendefinisiannya (ini dilakukan secara otomatis oleh C++)
- Jika suatu objek dibuat dengan keyword *new* dan kita ingin mendealokasikan (membebaskan) memory objek tersebut dengan keyword *delete*.
- Program berakhir dan terdapat objek yang didefinisikan secara global atau statik
- Destructor dipanggil secara eksplisit dengan memanggil nama lengkap fungsi destructor-nya

Penggunaan destructor sangat lah berguna jika salah satu diantara member data adalah sebuah pointer yang memory-nya harus dialokasikan (dibebaskan) jika sudah tidak digunakan. Syntax dari destructor adalah sama seperti constructor, tetapi ditambahkan tilda (~) didepan nama destructornya.

```
~Car() {                        // Destructor
    std::cout << "Destructor dijalankan\n";
}
```

Beberapa hal yang perlu diperhatikan dalam mendefinisikan destructor adalah destructor tidak memiliki parameter dan tidak men-return-kan sesuatu (bersifat void).

Calling Member Functions

Dalam main() telah diberikan contoh untuk memanggil method dari sebuah class

```
myCar.setBrand("Dodge");                // call member function to set data
myCar.setModel("Charger");
```

Syntax-nya berupa nama dari class kemudian diikuti tanda titik dan nama method-nya.

Keyword this

Keyword this sebenarnya adalah pointer yang merujuk pada address dari class-nya sendiri sehingga saya bisa gunakan keyword ini untuk mengakses member dari class.

```
void repaint(std::string warnaCat) {    // Member function to set data
    this->color = warnaCat;
}
```

Contoh dari code di atas, statement `this->color = warnaCat` merupakan ekivalen dengan `color = warnaCat`.

Penggunaan Static Data Dalam Class

Pertama-tama, data bertipe static dalam C++ artinya data tersebut hanya akan *terdeklarasi sekali dan akan hidup sampai akhir dari program*. Jika suatu class memiliki member static maka pada setiap objek dari class yang sama akan memiliki akses pada member static yang digunakan secara bersama. Hal ini akan sangat berguna jika setiap objek harus mengetahui sudah seberapa banyak objek dari class sama yang telah terbuat.

Contohnya, pada suatu tempat atau lembaga, ingin mengetahui seberapa banyak orang yang masuk dan berapa banyak orang yang keluar dari tempat tersebut, maka kita bisa buat sebuah class yang tugasnya adalah menyimpan suatu nilai. Nilai tersebut merujuk pada banyaknya orang di dalam tempat tersebut sehingga saat kita buat objek baru, kita menginginkan nilai tersebut bertambah sesuai dengan nilai banyak orang pada objek sebelumnya. Static member data dapat menyelesaikan persamalahan tersebut karena ketika objek baru dibuat, nilai static member akan tetap sama dengan nilai pada objek sebelumnya. Perhatikan bahwa hal tersebut benar karena static member data hanya akan *terdeklarasi sekali dan akan hidup sampai akhir dari program*. Berikut adalah contoh code yang menggunakan static data member

```
#include <iostream>

class Foo
{
private:
    static int count; // only one data item for all objects
                      // note: "declaration" only!

public:
    Foo() {
        count++; // increments count when object created
    }
    int getCount() { // return count
        return count;
    }
};

int Foo::count = 0;

int main()
{
    Foo f1, f2, f3;
```

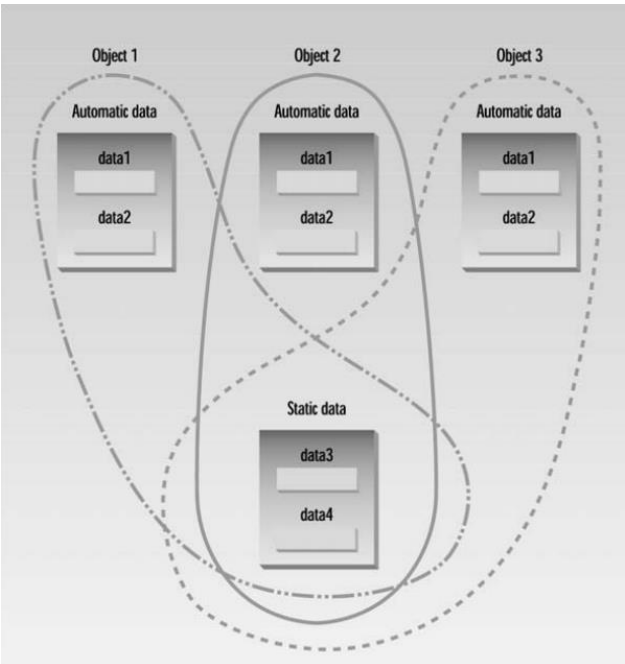
```
    std::cout << "count is " << f1.getCount() << '\n';
    std::cout << "count is " << f2.getCount() << '\n';
    std::cout << "count is " << f3.getCount() << '\n';
    return 0;
}
```

Perhatikan jika member count merupakan static data maka seharusnya akan didapatkan hasil seperti ini

```
count is 3
count is 3
count is 3
```

Jika Anda tidak menggunakan data static maka hasilnya akan seperti ini

```
count is 1
count is 1
count is 1
```



Gambar 5 Static versus automatic (nonstatic) member variable

Perhatikan pemberian definisi atau inisialisasi pada contoh di atas adalah di luar dari class.

```
int Foo::count = 0;
```

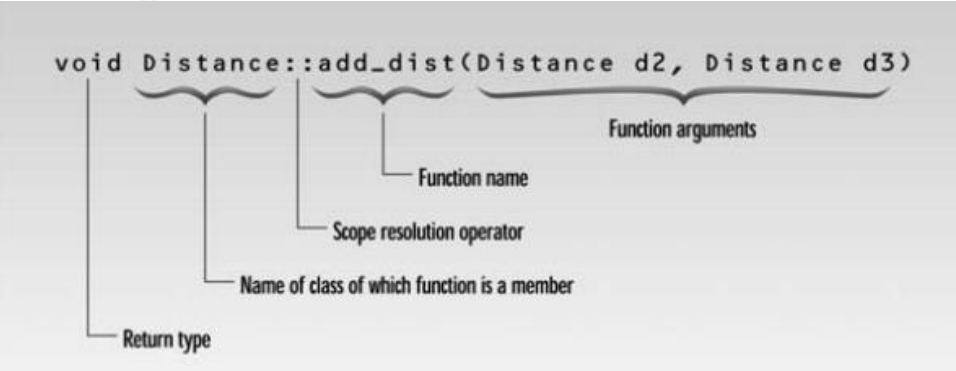
Jika inisialisasi static data berada di dalam class maka akan melanggar ide bahwa class adalah sebuah *blueprint*. Meletakkan definisi static member data di luar class juga bertindak untuk menekankan bahwa memory yang dialokasi buat static data hanya dialokasi sekali, yaitu saat diberikan definisi (inisialisasi awal) dan akan hidup sampai program selesai. Satu hal yang harus Anda perhatikan, yaitu static member dapat diakses oleh semua objek yang terbuat dari class yang sama.

Syntax Pendefinisian Member, di Luar Class

Tak hanya pendefinisian member data yang dapat dilakukan di luar class seperti pada contoh di atas, member function juga dapat diletakkan di luar class. Hal ini sangat penting untuk diketahui karena biasanya pendefinisian suatu fungsi tidak langsung di dalam class, tetapi diletakkan di file yang berbeda. Hal tersebut dilakukan untuk mengorganisir code menjadi lebih baik dan jelas. Misal saya ingin membuat method untuk menambahkan pembacaan distance

```
#include <iostream>

class Distance
{
    float inches;
    float feet;

public:
    Distance() : inches(0.0), feet(0.0)
    {  }
    void add_dist(Distance d2, Distance d3);
};

// add lengths d2 and d3
void Distance::add_dist(Distance d2, Distance d3)
{
    inches = d2.inches + d3.inches; // add the inches
    feet = 0;                       //(for possible carry)
    if (inches >= 12.0)             // if total exceeds 12.0,
    {                               // then decrease inches
        inches -= 12.0;             // by 12.0 and
        feet++;                     // increase feet
    }                               // by 1
    feet += d2.feet + d3.feet;      // add the feet
}
```

Simbol double titik dua (::) adalah simbol yang disebut *scope resolution operator*. Simbol tersebut merupakan cara untuk men-specify class apa yang akan diasosiasikan. Pada contoh di atas syntax `Distance::add_dist()` artinya "member fungsi add_dist() dari class Distance". Hal ini akan sangat penting terutama ketika kita membahas *inheritance*. Inheritance akan di bahas selanjutnya pada modul ini



Gambar 6 Syntax definisi member fungsi di luar class

Summary Object and Classes

Sebuah class adalah spesification atau blueprint dari objek. Objek terdiri atas data dan fungsi yang akan mengubah data tersebut. Dalam class terdapat *access modifier*, yaitu *private public*, dan *protected*. Private artinya hanya bisa diakses oleh member fungsi dari class tersebut, sedangkan public artinya data dapat diakses oleh semua fungsi di dalam program. Secara default access modifier dari class adalah private.

Specifiers	Same Class	Derived Class	Outside Class
`public`	Yes	Yes	Yes
`private`	Yes	No	No
`protected`	Yes	Yes	No

Sebuah member function adalah fungsi yang merupakan member dari class. Member function memiliki akses ke dalam objek private data, sementara nonmember function tidak punya akses tersebut.

Sebuah constructor adalah member function, dengan nama yang sama dengan nama class-nya. Constructor pasti yang pertama kali dieksekusi tiap kali membuat objek. Constructor tidak memiliki return type, tetapi dapat menerima argumen. Constructor sering digunakan

untuk memberikan nilai awal kepada member data dari objek. Constructor dapat di-overload (bentuknya bisa berbeda-beda), sehingga objek dapat diinisialisasi dengan cara yang berbeda.

Sebaliknya, sebuah destruktor adalah member function dengan nama yang sama dengan kelasnya, tetapi diawali dengan tilda (~). Destructor dipanggil ketika objek dihancurkan. Destruktor tidak mengambil argumen dan tidak memiliki nilai pengembalian.

Salah satu alasan untuk menggunakan OOP adalah korespondensi yang erat antara objek di dunia nyata dengan class dalam OOP. Menentukan bagaimana objek dan class yang akan digunakan dalam suatu program dapat rumit tergantung permasalahan yang dihadapi sehingga untuk program-program kecil, metode trial and error mungkin sudah cukup. Namun, untuk program-program besar, pendekatan yang lebih sistematis biasanya diperlukan.

Operator Overloading

Operator overloading dalam C++ adalah compile-time polymorphism, yaitu ide atau gagasan untuk memberikan makna spesial terhadap operator yang telah ada dalam C++. Hal ini menarik karena kita bisa membuat operasi seperti berikut

```
d3.addobjects(d1,d2);
```

atau operasi yang seperti berikut

```
d3 = d1.addObjects(d2);
```

menjadi operasi yang lebih mudah dibaca dan intuitif, seperti berikut

```
d3 = d1 + d2;
```

Dengan operator overloading, kita tidak terbatas pada data tipe apa yang bisa digunakan untuk operasi di atas. Kita bisa membuat user-defined types menjadi legal menggunakan operator +, *, <=, dll.

Mari kita coba menggunakan operator overloading untuk meng-increment suatu variabel

```
#include <iostream>

class Counter
{
private:
    unsigned int count; // count

public:
    Counter() : count(0) {} // Constructor
    unsigned int getCount() { // return count
        return count;
    }
```

```
    void operator++() { // increment (prefix)
        ++count;
    }
};

int main()
{
    Counter c1, c2; // define and initialize
    std::cout << "c1 = " << c1.getCount(); // display
    std::cout << "\nc2 = " << c2.getCount();
    ++c1; // increment c1
    ++c2; // increment c2
    ++c2; // increment c2
```

Syntax Operator Overloading

`void operator++()`

Syntax untuk operator overloading pertama-tama adalah return type-nya (pada kasus ini void) kemudian diikuti keyword *operator* dan kemudian jenis operatornya (pada kasus ini ++). Satu-satunya cara C++ compiler dapat membedakan antara operator overloaded dengan operator biasa adalah dengan melihat daata type dari operands-nya. Jika operand tipe datanya adalah *int* biasa maka contoh seperti berikut

```
++intVar;
```

oleh compiler akan secara otomatis digunakan built-in routine (fungsi bawaan dari C++) untuk meng-increment *int*. Akan tetapi, jika data yang dilakukan operasi merupakan user-define data type maka compiler akan tau untuk menggunakan `operator++()` dan bukan increment biasa.

Operator Return Values

Perhatikan bahwa fungsi `operator++()` memiliki sedikit kelemahan. Anda akan menemukan jika Anda menggunakan statement tersebut dalam main(), seperti berikut

```
c1 = ++c2;
```

Compiler akan komplain. Mengapa? Hal tersebut terjadi karena kita mendefinisikan ++ operator untuk men-return type void. Agar memungkinkan kita menggunakan syntax di atas, terdapat beberapa perbaikan yang bisa dilakukan

```
#include <iostream>

class Counter
{
private:
    unsigned int count; // count

public:
    Counter() : count(0) {} // Constructor
    unsigned int getCount() // return count
    {
        return count;
    }
    Counter operator++() // increment (prefix)
    {
        ++count;            // increment count
        Counter temp;       // make a temporary Counter
        temp.count = count; // give it same values as this obj
        return temp;        // return the copy
    }
};


int main()
{
    Counter c1, c2; // define and initialize

    std::cout << "c1 = " << c1.getCount(); // display
    std::cout << "\nc2 = " << c2.getCount();

    ++c1;      // increment c1, c1 = 1
    c2 = ++c1; // c1 = 2 dan c2 = 2

    std::cout << "\nc1 = " << c1.getCount(); // display again
    std::cout << "\nc2 = " << c2.getCount() << '\n';
    return 0;
}
```

Pada contoh di atas, fungsi operator++() membuat objek baru bertipe Counter, yang disebut temp, untuk digunakan sebagai return value. Fungsi ini menambahkan data count dalam objeknya sendiri seperti sebelumnya, kemudian membuat objek sementara yang diberi nama temp. Objek temp ini kemudian diberikan value count yang telah di-increment, setelah itu,

objek temp dikeluarkan dengan keyword return. Hal ini memiliki efek yang diinginkan seperti ekspresi berikut

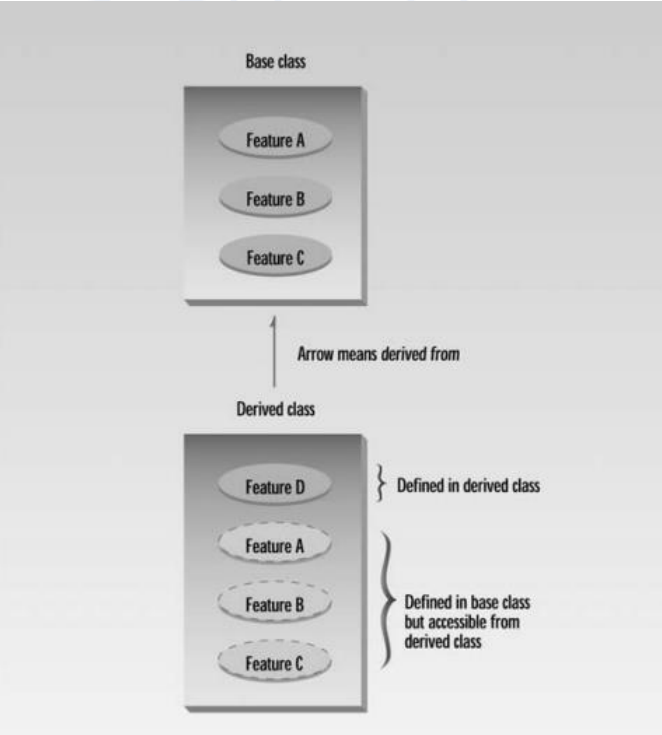
```
c2 = ++c1
```

Perhatikan hasil ouput program main() adalah sesuai yang kita inginkan

```
c1 = 0
c2 = 0
c1 = 2
c2 = 2
```

Inheritance

Inheritance adalah salah satu fitur powerful dalam OOP. Inheritance adalah proses membuat classes baru, disebut *derived classes*, dari class yang sudah ada atau *base classes*. Perhatikan bahwa derived class akan memiliki method dan struktur data yang sama dengan base class. Kemudian untuk tambahan kostumisasi, Anda tetap bisa menambahkan fitur-fitur (method dan data) lain ke dalam derived class dengan tidak merubah struktur dari base class. Berikut adalah diagram hubungan antara derived class dengan base class



Gambar 7 diagram hubungan base class dengan derived class

Inheritance adalah salah satu bagian penting dari OOP karena dengan inheritance memungkinkan kita untuk menggunakan ulang code yang sudah ada. Perhatikan bahwa jika

sebuah base class telah dibuat, dengan menggunakan inheritance kita dapat mengadaptasi pekerjaaan sebelumnya (base class). Menggunakan dan mengadaptasi code sebelumnya akan sangat menghemat waktu dan meningkatkan reliabilitas dari program.

Derived Class dan Base Class

Pada contoh sebelumnya, kita sudah membuat class Counter sebagai general-purpose counter variable. Misal kita ingin menambahkan sebuah method untuk decrement, tetapi kita tidak ingin menambahkannnya secara langsung pada class Counter. Pada kasus yang simple, method decrement bisa saja langsung diletakkan dalam class Counter. Akan tetapi, bayangkan jika class yang Anda buat telah melalui berbagai pengujian dan mengahabiskan waktu berjam-jam untuk debugging maka pendekatan melalui inheritance bisa menjadi salah satu solusi. Berikut adalah code, menambahkan fitur decrement dengan mengaplikasikan inheritance

```
#include <iostream>

class Counter
{
protected:
    unsigned int count; // count
public:
    Counter() : count(0) {}              // Constructor
    Counter(int count) : count(count) {} // Constructor
    unsigned int getCount() {             // return count
        return count;
    }
    Counter operator++() { // increment (prefix)
        ++count;            // increment count
        Counter temp;       // make a temporary Counter
        temp.count = count; // give it same values as this obj
        return temp;        // return the copy
    }
};

class CountDn : public Counter // derived class
{
public:
    Counter operator--() { // decrement count (prefix)
        return Counter(--count);
    }
};
```

```
int main()
{
    CountDn c1;                              // c1 of class CountDn
    std::cout << "c1 = " << c1.getCount(); // display c1
    ++c1;
    ++c1;
    ++c1;                                     // increment c1, 3 times
    std::cout << "\nc1 = " << c1.getCount(); // display it
    --c1;
    --c1;                                    // decrement c1, twice
    std::cout << "\nc1 =" << c1.getCount(); // display it
    std::cout << std::endl;
    return 0;
}
```

Mari kita bedah code di atas

Membuat Derived Class

Untuk membuat class baru bisa menggunakan syntax berikut

```
class CountDn : public Counter // derived class
{ };
```

Pertama adalah keyword class, diikuti dengan nama derived class-nya. Kemudian menggunakan simbol titik dua (:) lalu diberikan access modifier dan Anda cantumkan base class-nya apa, pada contoh di atas, yaitu dari class Counter.

Access Specifier Untuk Derived Class

Terdapat tiga jenis access specifier

- Public
- Protected
- Private

Public inheritance adalah access specifier yang paling umum digunakan, specifier lainnya biasanya jarang digunakan. Ketika Anda menggunakan access specifier public inheritance maka akan terjadi hal berikut. Member base class yang bersifat public akan tetap public, kemudian member base class yang bersifat protected akan tetap bersifat protected pada derived class, dan member yang bersifat private pada base class tidak bisa diakses dalam derived class.

Access specifier in base class	Access specifier when inherited publicly
Public	Public
Protected	Protected
Private	Inaccessible

Protected inheritance biasanya jarang digunakan dan hanya digunakan pada kasus-kasus tertentu saja. Dengan protected inheritance, member base class yang bersifat public dan protected akan menjadi bersifat protected dalam derived class. Kemudian, member base class yang bersifat private tidak akan bisa diakses dalam derived class

Access specifier in base class	Access specifier when inherited protectedly
Public	Protected
Protected	Protected
Private	Inaccessible

Terakhir, private inheritance akan membuat semua member base class yang bersifat public dan protected akan menjadi private dalam derived class. Untuk member base class yang private, tetap tidak dapat diakses dalam derived class. Access specifier private dapat digunakan jika Anda memiliki derived class yang tidak terlalu berhubungan dengan base class, tetapi dalam implementasi derived class memerlukan member atau function dari base class.

Access specifier in base class	Access specifier when inherited privately
Public	Private
Protected	Private
Private	Inaccessible

Derived Class Constructor

Perhatikan bahwa pada contoh code class Counter dan turunannya, yaitu CountDn. Saat Anda membuat objek dari class CountDn, Anda tidak bisa menginisialisasikannya dengan suatu value. Hal tersebut terjadi karena kita belum specify constructor pada class CountDn. Pembuatan constructor pada class turunan implementasinya sedikit berbeda karena akan memanggil constructor dari base class-nya. Berikut adalah contoh modifikasi code sebelumnya

```cpp
#include <iostream>

class Counter
{
protected:
    unsigned int count; // count

public:
    Counter() : count(0) {}              // Constructor
    Counter(int count) : count(count) {} // Constructor
    unsigned int getCount()              // return count
    {
        return count;
    }
    Counter operator++() // increment (prefix)
    {
        ++count;             // increment count
        Counter temp;        // make a temporary Counter
        temp.count = count; // give it same values as this obj
        return temp;         // return the copy
    }
};

class CountDn : public Counter // derived class
{
public:
    CountDn() : Counter() // constructor, no argument
        { }
    CountDn(int c) : Counter(c) // Constructor, 1 argument
        { }
    CountDn operator--() {// decrement count (prefix)
        return CountDn(--count);
    }
};

int main()
{
    CountDn c1; // c1 of class CountDn
    CountDn c2(100);

    std::cout << "c1 = " << c1.getCount() << '\n';     // display c1
    std::cout << "c2 = " << c2.getCount() << '\n';     // display c2
```

```
    ++c1; ++c1; ++c1;                                   // increment c1, 3 times
    std::cout << "c1 = " << c1.getCount();              // display it

    --c2; --c2;                                         // decrement c2, twice
    std::cout << "\nc2 = " << c1.getCount();            // display it

    CountDn c3 = --c2;
    std::cout << "\nc3 = " << c3.getCount() << std::endl;
    return 0;
}
```

Syntax constructor pada derived class seperti berikut

```
CountDn() : Counter() // constructor, no argument
    { }
CountDn(int c) : Counter(c) // Constructor, 1 argument
    { }
```

Ketika compiler membuat objek CountDn, constructor CountDn akan dipanggil yang mana hal tersebut akan memanggil pula constructor dari class Counter. Pada contoh di atas, function body, yaitu statement di antara tanda kurung kurawal ({}) adalah kosong. Akan tetapi, Anda bisa isi dengan statement, sesuai dengan kebutuhan Anda.

Function Overriding

Salah satu fitur inheritance di C++ adalah function overriding. Function overriding adalah suatu function di dalam derived class yang memiliki nama yang sama dengan member function pada base class, tetapi memiliki definisi implementasi yang berbeda. Berikut adalah contoh function overriding

```
#include <iostream>

class Base
{
public:
    void print() {
        std::cout << "Base Function printing\n";
    }
};
```

```cpp
class Derived : public Base
{
public:
    void print() {
        std::cout << "Derived Function printing\n";
    }
};

int main()
{
    Base baseClass;
    Derived derivedClass;

    std::cout << "baseClass.print() :\n";
    baseClass.print();

    std::cout << "\nderivedClass.print() :\n";
    derivedClass.print();

    // scope resolution
    std::cout << "\nderivedClass.Base::print() :\n";
    derivedClass.Base::print();
    std::cout << '\n';

    return 0;
}
```

Perhatikan pada contoh di atas, terdapat dua fungsi memiliki nama yang sama dengan implementasi definisi yang berbeda. Perhatikan bahwa pada base class, fungsi print() bersifat public. Artinya, pada class turunannya, fungsi print dari base class dapat dipanggil kembali di derived class. Jika Anda membutuhkan bahwa pada derived class, implementasi dari fungsi print() berbeda dengan base class maka Anda dapat melakukannya dengan langsung memberikan definisi yang berbeda pada fungsi di derived class. Hal tersebut merupakan legal dan dapat dilakukan di inheritance pada bahasa C++. Perhatikan pula, karena fungsi print pada base class adalah public dan kita pun memberikan implementasi yang berbeda pada derived class. Saat menggunakan function overriding. untuk memberitahu C++ mana fungsi yang ingin dipanggil kita bisa menggunakan scope resolutionnya. Perhatikan code ini

```cpp
    std::cout << "\nderivedClass.print() :\n";
    derivedClass.print();
```

akan menjalankan fungsi dari si class Derived. Akan tetapi, jika Anda menggunakan scope resolution seperti berikut

```
    // scope resolution
    std::cout << "\nderivedClass.Base::print() :\n";
    derivedClass.Base::print();
    std::cout << '\n';
```

dengan scope resolution, Anda dapat memanggil fungsi print dari class Base melalui class Derived.

Virtual Functions

Dalam C++, virtual function adalah mekanisme yang memungkinkan untuk sebuah function pada base class menjadi overridden oleh derived class-nya. Dengan kata lain, ketika kita mendeklarasikan sebuah function sebagai virtual function dalam base class, kita sedang meminta compiler C++ untuk memperhatikan bahwa ada fungsi dari base class yang namanya sama di derived class dan kita memberikan implementasi yang berbeda pada fungsi di derived class tersebut.

Umumnya virtual function digunakan jika kita memiliki sebuah pointer ke base class yang menunjuk pada salah satu objek dari derived class-nya. Misal seperti berikut

```
int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Perhatikan bahwa dengan virtual function, function yang akan dipanggil adalah implementasi function dari objek yang ditunjuk pointer, bukan function dari type pointer-nya

Syntax untuk Virtual Function

```
    virtual void namaFunction() { }
```

syntax untuk virtual function sangat mudah, Anda hanya perlu menambahkan keyword *virtual* didepan statement definisi function-nya. Peletakan virtual function hanya pada function di base class. Berikut merupakan contoh dari class yang menggunakan virtual function, perhatikan detail kecilnya!

```cpp
#include <iostream>

class Base
{
public:
    virtual void print() {
        std::cout << "Base Function printing\n";
    }
};

class Derived : public Base
{
public:
    void print() {
        std::cout << "Derived Function printing\n";
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* base1 = &derived1;

    // calls member function of Derived class
    base1->print();

    return 0;
}
```

Pada function print() di base class, ditambahkan keyword *virtual* maka function print tersebut merupakan virtual function dan hasil keseluruhan code di atas ouputnya adalah

```
Derived Function printing
```

Sedangkan jika Anda tidak menggunakan keyword virtual pada function print() pada class base maka program di atas akan mengeluarkan output

```
Base Function printing
```