C++ Operator Overloading

เทพพิทักษ์ การุญบุญญานันท์

การ Overload Operator

- ข้อมูลชนิดธรรมดา

```
int a, b, c, x;
x = a * b + c;
```

- ข้อมูลที่เป็นออบเจกต์

```
class Complex { /* ... */ };

Complex a, b, c, x;
x = a * b + c;
```

สามารถทำได้ด้วยการ *overload operator*

การ Overload Operator

```
class Complex {
public:
  Complex (double re, double im) : re (re), im (im) {}

  Complex operator+ (Complex c)
    { return Complex (re + c.re, im + c.im); }
  Complex operator* (Complex c)
    { return Complex ((re*c.re - im*c.im),
                      (re*c.im + im*c.re)); }

private:
  double re, im;
};
```

การ Overload Operator

```
Complex a (2, 3);
Complex b (4, 5);
```

```
Complex p = a + b;
```

มีความหมายเทียบเท่า:

```
Complex p = a.operator+ (b);
```

การ Overload Operator

- การกำหนดพฤติกรรมใหม่ให้กับ operator เรียกว่า “การ overload operator”
- ลำดับการกระทำของ operator เป็นไปตามข้อกำหนดของ C++

```
Complex p = a + b * c;
```

มีความหมายเทียบเท่า:

```
Complex p = a.operator+ (b.operator* (c));
```

การ Overload Operator

- operator ใน C++ สามารถโอเวอร์โหลดได้เกือบทั้งหมด *ภายใต้ syntax เดิม* (จะกล่าวในรายละเอียดต่อไป)

```
+     -     *     /     %     ^     &
|     ~     !     =     <     >     +=
-=    *=    /=    %=    ^=    &=    |=
<<    >>    >>=   <<=   ==    !=    <=
>=    &&    ||    ++    --    ->*   ,
->    []    ()    new   new[] delete delete[]
```

การ Overload Operator

- operator ที่โอเวอร์โหลดไม่ได้:

```
::      .       .*      ?:
```

- ไม่สามารถกำหนด operator ใหม่ที่ไม่มีใน C++ ได้

```
class Complex {
public:
  // ...
  Complex operator** (Complex x); // wrong!!
};
```

- โอเวอร์โหลด operator กับชนิดพื้นฐานล้วน ๆ ไม่ได้

Member and Non-Member Operator Functions

- โดยทั่วไป operator สามารถโอเวอร์โหลดด้วย member หรือ non-member function ก็ได้

```
class Complex {
public:
  Complex (double re = 0, double im = 0);
  Complex operator+ (Complex x); // member
};
```

หรือ:

```
Complex operator+ (Complex x, Complex y); // non-member
```

อย่างใดอย่างหนึ่ง (หลีกเลี่ยงความกำกวม)

Member and Non-Member Operator Functions

- operand ทางซ้ายของแบบ member function ต้องเป็นออบเจกต์เท่านั้น

```
int main()
{
  Complex a (1, 2);
  Complex b (3, 4);

  Complex x = a + 1.0; // member: OK;    non-member: OK
  Complex y = 1.0 + a; // member: error; non-member: OK
  Complex z = a + b;   // member: OK;    non-member: OK

  return 0;
}
```

Member and Non-Member Operator Functions

- member function สามารถเข้าถึง private member ได้
- non-member function เข้าถึงได้เฉพาะ public member เท่านั้น

```
Complex operator+ (Complex x, Complex y)
  { return Complex (x.re + y.re, x.im + y.im); } // error!
```

ยกเว้น ประกาศให้เป็น **friend** กับคลาส

```
class Complex {
public:
  // ...
  friend Complex operator+ (Complex x, Complex y);
};
```

Member and Non-Member Operator Functions

- operator ที่ต้องโอเวอร์โหลดด้วย member function เท่านั้น:

```
=       []      ()      ->
```

(ข้อบังคับของ C++ เพื่อประกันว่า operand แรกจะเป็น lvalue เสมอ)

- operator ที่ควรโอเวอร์โหลดด้วย member function เช่นกัน:

```
+=      -=      *=      /=      %=      ^=      &=
|=      >>=     <<=     ++      --      ->*
```

Member and Non-Member Operator Functions

- หลักทั่วไป:
  - operator ที่เปลี่ยนแปลงออบเจกต์ ควรใช้ member function
    *(assignment ทั้งหลาย, **++**, **--**)*
  - operator ที่เข้าถึงออบเจกต์แบบ lvalue ได้ ควรใช้ member function
    *(array index, function call, **->**, **->***)*
  - operator ที่ใช้ค่าออบเจกต์ในการคำนวณค่าใหม่เท่านั้น ควรใช้ non-member function
    (เครื่องหมายเลขคณิต, การคำนวณบิต, การคำนวณตรรก)

Binary and Unary Operators

- operator บางตัวเป็นได้ทั้ง binary และ unary

```
c = a - 1; // binary -
c = -a;    // unary -
b = a & b; // binary &
p = &a;    // unary &
```

แยกความแตกต่างได้ด้วยอาร์กิวเมนต์

```
Complex operator- (Complex x, Complex y); // binary -
Complex operator- (Complex x);            // unary -
Complex operator& (Complex x, Complex y); // binary &
Complex* Complex::operator& ();           // unary &
```

Prefix and Postfix Operators

- operator ++ และ -- ใช้ได้ทั้งเป็น prefix และ postfix

```
c = ++a;    // prefix increment
c = a++;    // postfix increment
c = --a;    // prefix decrement
c = a--;    // postfix decrement
```

แยกความแตกต่างได้ด้วยอาร์กิวเมนต์ **int** ซึ่งไม่มีการใช้งาน

```
Complex Complex::operator++ ();          // prefix inc.
Complex Complex::operator++ (int);       // postfix inc.
Complex operator-- (Complex& x);         // prefix dec.
Complex operator-- (Complex& x, int); // postfix dec.
```

ตัวอย่าง: Complex

Requirements:

```
Complex a (1, 2);
Complex b = 3;
Complex c = a + 3.4;
Complex d = 5 + b;
Complex e = -b - c;
b = c*2/d;
c += (d == e) ? a : b;
cout << e << endl;
```

Complex: Class & Constructors

```
class Complex {
public:
  Complex (double re = 0, double im = 0)
    : re (re), im (im) {}
  Complex (const Complex& a)
    : re (a.re), im (a.im) {}

private:
  double re;
  double im;
};
```

Complex: Class & Constructors

ข้อสังเกต: Copy Constructor ต้องรับ const ref:

```
Complex (const Complex& a);
```

จะรับ **Complex** เฉย ๆ ไม่ได้

```
Complex (Complex a); // wrong !!
```

เพราะจะเกิด infinite loop เมื่อใช้:

```
Complex x = y;
```

เพราะการ copy **y** ให้กับอาร์กิวเมนต์ **a** ก็จะต้องใช้ copy constructor!

Complex: Operator เลขคณิต

```
class Complex {
public:
   // ...
   Complex& operator+= (Complex c);
};

inline Complex& Complex::operator+= (Complex c)
{
   re += c.re;
   im += c.im;
   return *this;
}
```

Complex: Operator เลขคณิต

```
inline Complex operator+ (Complex a, Complex b)
{
  Complex res = a;
  return res += b;
}
```

หรือ:

```
inline Complex operator+ (Complex a, Complex b)
{
  return Complex (a) += b;
}
```

Complex: Operator เลขคณิตข้ามชนิด

```
class Complex {
public:
  // ...
  Complex& operator+= (Complex c);
  Complex& operator+= (double d);
};

inline Complex& Complex::operator+= (double d)
{
  re += d;   // NOTE: im is untouched
  return *this;
}
```

Complex: Operator เลขคณิตข้ามชนิด

```
inline Complex operator+ (Complex c, double d)
{
  // calls Complex::operator+= (double)
  return Complex (c) += d;
}

inline Complex operator+ (double d, Complex c)
{
  // calls Complex::operator+= (double)
  return Complex (c) += d;
}
```

Complex: Operator เลขคณิตข้ามชนิด

การคำนวณจะเกิดเท่าที่จำเป็น

- ไม่มีการสร้างออบเจกต์ชั่วคราวเพื่อเป็นอาร์กิวเมนต์
- ไม่มีการบวก **im** ด้วย 0:

```
void f (Complex a, Complex b)
{
  Complex x = a + 3.4; // calls operator+(Complex,double)
  Complex y = 5 + b;   // calls operator+(double,Complex)
  Complex z = a + b;   // calls operator+(Complex,Complex)
}
```

Complex: Operator เลขคณิตข้ามชนิด

ข้อสังเกตเทคนิคการโอเวอร์โหลด operator เลขคณิต:

- โอเวอร์โหลด composite assignment op ก่อน
  - กระทำกับ private member
  - จัดการกับอาร์กิวเมนต์ภายนอกเพียงตัวเดียว
- โอเวอร์โหลด binary op ให้เรียก composite assignment
  - ไม่ต้องเข้าถึง private member โดยตรง
  - จัดการกับอาร์กิวเมนต์ชนิดต่าง ๆ ในระดับบน

Complex: Accessing Functions

เพิ่ม accessing function สำหรับผู้เรียกภายนอก:

```
class Complex {
public:
  // ...
  double real() const { return re; }
  double imag() const { return im; }
};
```

Complex: Operator เปรียบเทียบค่า

ทำให้สามารถ implement operator == ได้:

```
inline bool operator== (Complex a, Complex b)
{
  return (a.real() == b.real()) && (a.imag() == b.imag());
}
```

Complex: Operator เปรียบเทียบค่า

operator != ก็อาจ implement เช่นนี้ได้:

```
inline bool operator!= (Complex a, Complex b)
{
  return (a.real() != b.real()) || (a.imag() != b.imag());
}
```

แต่แบบนี้จะกระชับและไม่ซ้ำซ้อนกับ operator ==

```
inline bool operator!= (Complex a, Complex b)
{
  return !(a == b);
}
```

Complex: Stream Output

```
#include <iostream>

// ...

inline ostream& operator<< (ostream& os, Complex c)
{
  return os << "(" << c.real()
            << " + " << c.imag() << "i)";
}
```

Complex: Stream Output

ทำให้เรียกใช้เช่นนี้ได้:

```
Complex a (1, 2);
cout << a << endl;
```

กรณีเหล่านี้ล่ะ?:

```
cout << Complex (1) << endl;
cout << Complex (0) << endl;
cout << Complex (0, 2) << endl;
cout << Complex (3, -4) << endl;
```

Complex: Stream Output

ปรับรายละเอียดการทำงาน:

```
ostream& operator<< (ostream& os, Complex c)
{
  if (c.imag() == 0) return os << c.real();
  if (c.real() == 0) return os << c.imag() << 'i';

  os << "(" << c.real();
  if (c.imag() > 0) {
    os << " + " << c.imag();
  } else {
    os << " - " << -c.imag();
  }
  return os << "i)";
}
```

Complex: Stream Output

ข้อสังเกต:

- การทำ I/O stream ไม่สามารถใช้ member function ได้ เพราะอาร์กิวเมนต์แรกเป็นออบเจกต์ของ **`iostream`** ซึ่งเราไม่สามารถเข้าไปแก้โค้ดได้

Complex: Stream Output

ข้อสังเกต:

- ค่ารีเทิร์นของ operator `<<` เป็น `ostream&` → ทำงานเป็นลูกโซ่ได้:

```
cout << a << endl;
```

จะถูกตีความเป็น:

```
(cout << a) << endl;
```

และเป็น:

```
operator<< (operator<< (cout, a), endl);
```

Friends

friend = การอนุญาตให้ฟังก์ชันภายนอกเข้าถึง private member ได้

- friend function: ประกาศอนุญาตให้กับฟังก์ชันภายนอก

```
class Vector {
  double x, y, z;
public:
  // ...
  friend double Dot (Vector a, Vector b);
};

double Dot (Vector a, Vector b)
{
  return a.x*b.x + a.y*b.y + a.z*b.z;
}
```

Friends

- friend class: ประกาศอนุญาตให้กับทุก member function ของคลาสหนึ่ง ๆ

```
class Vector {
public:
  // ...
  friend class VectorReader;
};
```

→ ทุก member function ของคลาส **VectorReader** จะสามารถเข้าถึง private member ของ **Vector** ได้

Complex: Stream Input

- stream input สามารถทำได้ด้วยการโอเวอร์โหลด operator >>
- ข้อเท็จจริงเกี่ยวกับ stream input:
  - จำเป็นต้องเขียน private data ของออบเจกต์
  - ไม่สามารถเป็น member function ของคลาสเป้าหมายได้ เนื่องจากอาร์กิวเมนต์แรกเป็นออบเจกต์ของ iostream ซึ่งเราไม่สามารถเข้าไปแก้ไขโค้ดได้
- ทำได้โดยใช้ friend function

Complex: Stream Input

```
class Complex {
public:
  // ...
  friend istream& operator>> (istream& is, Complex& c);
};

istream& operator>> (istream& is, Complex& c)
{
  return is >> c.re >> c.im;
}
```

Large Objects

เมื่อออบเจกต์มีขนาดใหญ่ (cost ของการ copy สูง):

$\rightarrow$ operator ต่าง ๆ ควรรับ const reference

```
class Matrix {
public:
  Matrix();
  Matrix (const double v[4][4]);
  Matrix (const Matrix& m);

  friend Matrix operator+ (const Matrix& m1,
                           const Matrix& m2);

private:
  double m[4][4];
};
```

Conversion Operator

- constructor ที่รับอาร์กิวเมนต์เดียวสามารถใช้แปลงค่า *จากชนิดที่มีอยู่แล้วมาเป็นคลาส* ได้
- แต่ constructor ดังกล่าวไม่สามารถแปลงค่า *จากคลาสเป็นชนิดที่มีอยู่แล้ว* ได้
- การแปลงค่า *จากคลาสเป็นชนิดที่มีอยู่แล้ว* ทำได้โดยการ *กำหนด conversion operator*

ตัวอย่าง: String

```
class String {
public:
   String();
   String (const char* str);
   ~String();

private:
   char* s;
};
```

String: Constructor & Destructor

```
#include <cstring>
using namespace std;
```

```
inline String::String() : s (0) {}
inline String::String (const char* str)
{
  s = new char[strlen (str) + 1];
  strcpy (s, str);
}
inline String::~String() { delete[] s; }
```

String: Conversion Operator

```
class String {
public:
  // ...
  operator const char* () const;
};

inline String::operator const char*() const
{
  return s;
}
```

String: Conversion Operator

ทำให้สามารถใช้งานในลักษณะนี้ได้:

```
// Conversion from (const char*) to String
String s = "Hello";
// ...
// Conversion from String to (const char*)
if (strcmp (s, "Hello") == 0) {
  cout << "OK" << endl;
}
```

String: Conversion Operator

ข้อสังเกต:

- conversion operator กำหนดเป็น member function ของ *คลาสต้นทาง* ที่จะแปลงค่า
- ฟังก์ชันของ conversion operator ไม่มี return type (คล้าย constructor)

```
class String {
public:
  // ...
  operator const char*() const;                // right
  const char* operator const char*() const; // wrong!!
};
```

Explicit Constructor

Constructor บางตัวที่รับอาร์กิวเมนต์เดียวอาจไม่มีจุดประสงค์สำหรับแปลงค่า

```
class String {
public:
  String();
  String (int nChars);  // pre-allocate n chars
  String (const char* str);
  ~String();

private:
  char* s;
  int   sz;
};
```

String: Constructor & Destructor

```
inline String::String() : s (0), sz (0) {}
inline String::String (int nChars) : sz (nChars)
{
  s = new char[nChars];
}
inline String::String (const char* str)
{
  sz = strlen (str) + 1;
  s = new char[sz];
  strcpy (s, str);
}
inline String::~String() { delete[] s; }
```

String: Possible Abuse

การใช้งานที่ถูกกฎแต่ไม่ต้องการ

```
String s1 = 10; // not conversion as expected
```

String: Explicit Constructor

ป้องกันได้ด้วยการประกาศ constructor เป็น **explicit**

```
class String {
public:
   String();
   explicit String (int nChars);  // pre-allocate n chars
   String (const char* str);
   ~String();

private:
   char* s;
   int   sz;
};
```

String: Explicit Constructor

ผล: การประกาศต่อไปนี้กลายเป็น error

```
String s1 = 10; // error, no valid implicit constructor
```

ต้องประกาศเช่นนี้เท่านั้น:

```
String s1 (10); // ok, explicit constructor is used
```

ซึ่งช่วยป้องกันความเข้าใจผิดขณะอ่านโค้ดได้

การ Overload Assignment Operator

การ copy ค่าของออบเจกต์เกิดได้ 2 ลักษณะ:

- ขณะสร้างออบเจกต์ โดยอาศัย copy constructor

```
String a ("Hello");
String b = a;  // using copy constructor
```

- หลังจากสร้างออบเจกต์ไปแล้ว โดยใช้ operator =

```
String a ("Hello");
String b ("Bye");
a = b;          // using assignment operator
```

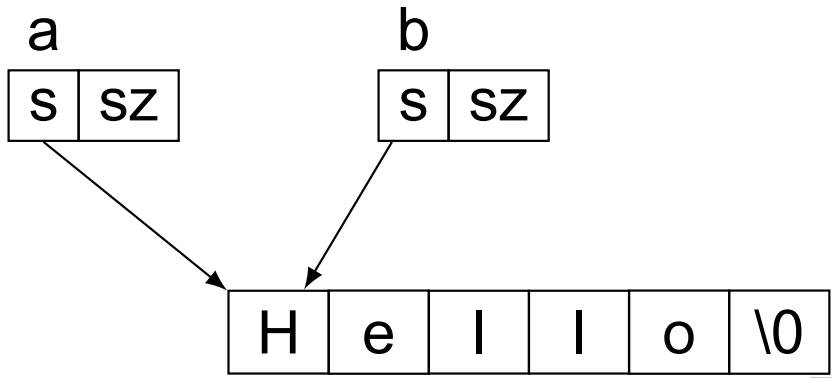
String: Copy Constructor

สำหรับคลาส **String** การ copy แบบตื้น (shallow copy) ไม่เพียงพอ

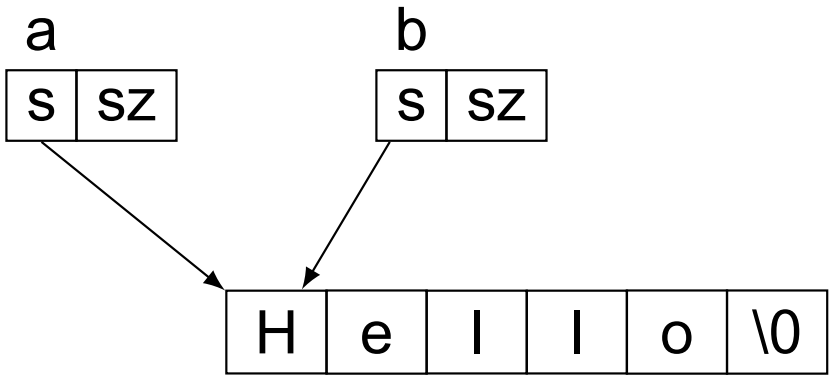
```
String::String (const String& t) : s (t.s), sz (t.sz) {}
```

```
String a ("Hello");
String b = a;  // using copy constructor
```

- **a.s** และ **b.s** จะชี้ไปยังที่เดียวกัน



String: Copy Constructor

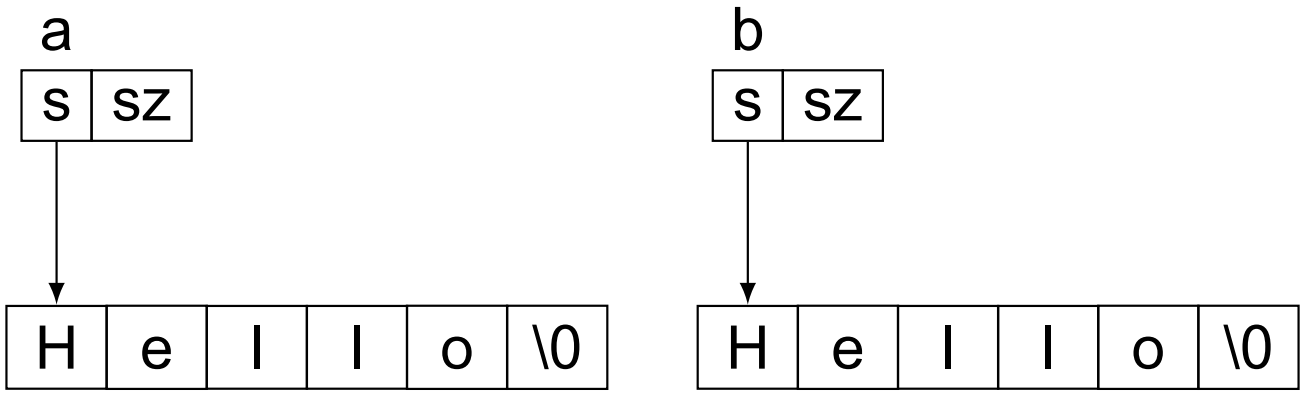


- เมื่อ **a** หมดอายุขัย → destructor จะ **delete[] a.s**
- เมื่อ **b** หมดอายุขัย → destructor ก็จะ **delete[] b.s** ซึ่งชี้ไปยังหน่วยความจำที่ **delete[]** ไปแล้วขณะ destruct **a**
- เกิด double free error!

String: Copy Constructor

∴ ต้อง copy แบบลึก (deep copy) ด้วยการจองเนื้อที่ใหม่

```
String::String (const String& t) : sz (t.sz)
{
  s = new char[sz];
  strncpy (s, t.s, t.sz);
}
```



String: Assignment Operator

การโอเวอร์โหลด `operator =`

```
class String {
public:
  // ...
  String& operator= (const String& t);
};
```

- ต้องเป็น member function เท่านั้น (ข้อกำหนดภาษา C++)
- ควร return non-const ref ไปยังออบเจกต์ที่ assign (เพื่อรองรับ chain assignment)

```
a = b = c = d;
```

String: Assignment Operator

การโอเวอร์โหลด **operator =**

```
String& String::operator= (const String& t)
{
  if (this != &t) {              // prevent self-assignment
    delete[] s;                  // deallocate old memory
    s = new char[t.sz];          // allocate new memory
    strncpy (s, t.s, t.sz); // copy data
    sz = t.sz;
  }
  return *this;
}
```

การ Overload Assignment Operator

ขั้นตอนโดยทั่วไป:

1. ตรวจสอบเพื่อป้องกันการ assign ทับตัวเอง
2. ลบข้อมูลเดิม
3. จองหน่วยความจำใหม่
4. คัดลอกข้อมูล
5. return ref ไปยังออบเจกต์ปัจจุบัน

การ Overload Subscript Operator

สมมุติว่าเราต้องการให้ผู้ใช้คลาส **String** สามารถเข้าถึงอักขระรายตัวได้

```
class String {
public:
  // ...
  char getChar (int idx) const;
  void setChar (int idx, char c);
};
```

```
  String s = "Hello";
  char c = s.getChar (1);
  s.setChar (1, 'a');
```

การ Overload Subscript Operator

อีกวิธีหนึ่งคือโอเวอร์โหลด subscript operator (`[]`)

```
class String {
public:
  // ...
  const char& operator[] (int idx) const;
  char&       operator[] (int idx);
};
```

```
  String s = "Hello";
  char c = s[1];
  s[1] = 'a';
```

String: Subscript Operator

Subscript operator พร้อม boundary check

```
inline const char& String::operator[] (int idx) const
{
  return (idx < sz) ? s[idx] : s[sz - 1];
}

inline char& String::operator[] (int idx)
{
  return (idx < sz) ? s[idx] : s[sz - 1];
}
```

ข้อสังเกต: `operator[]` ต้องเป็น member function เท่านั้น

ตัวอย่าง: Associative Array

Associative Array: แอร์เรย์ในรูป (key, value)
อาจกำหนด method ในลักษณะนี้:

```
class AssocArray {
public:
  // ...
  const String& getVal (const String& key) const;
  void          setVal (const String& key,
                        const String& val);
};
```

```
  AssocArray a;
  a.setVal ("Name", "Somsak");
  String name = a.getVal ("Name");
```

ตัวอย่าง: Associative Array

หรือโอเวอร์โหลด subscript operator ในลักษณะนี้:

```
class AssocArray {
public:
  // ...
  const String& operator[] (const String& key) const;
  String&       operator[] (const String& key);
};
```

```
AssocArray a;
a["Name"] = "Somsak";
String name = a["Name"];
```

การ Overload Function Call Operator

```
class Polynomial {
public:
  Polynomial (double c0, double c1, double c2, double c3);

  Polynomial derivative() const;
  double operator() (double x) const;
};
```

```
  Polynomial p (1, 1, 1, 1);
  cout << "p(1) = " << p(1) << endl;
  cout << "p'(1) = " << p.derivative()(1) << endl;
```

Polynomial: Data & Initialization

```
class Polynomial {
public:
  // ...
private:
  double c[4];  // c[i] is coeff. of x^i
};

Polynomial::Polynomial (double c0, double c1,
                        double c2, double c3)
{
  c[0] = c0; c[1] = c1; c[2] = c2; c[3] = c3;
}
```

Polynomial: `derivative()` *Method*

```
Polynomial Polynomial::derivative() const
{
  Polynomial d;
  // c[i+1]*x^(i+1) -> (i+1)*c[i+1]*x^i
  for (int i = 0; i < 3; i++) {
    d.c[i] = (i + 1) * c[i + 1];
  }
  d.c[3] = 0;

  return d;
}
```

Polynomial: Function Call Operator

```
double Polynomial::operator() (double x) const
{
  // p(x) = ((c3*x + c2)*x + c1)*x + c0
  double v = c[3];
  for (int i = 2; i >= 0; i--) {
    v = v*x + c[i];
  }
  return v;
}
```

ข้อสังเกต: `operator()` ต้องเป็น member function เท่านั้น

การโอเวอร์โหลด Dereference Operator

สมมุติ: คลาส **`RecordPtr`** สำหรับเข้าถึงระเบียนในดิสก์

- เป็น pointer เกียจคร้าน ไม่เรียกใช้ก็ไม่อ่านดิสก์
- อ่านเข้าสู่หน่วยความจำแล้วสามารถเข้าถึงและเปลี่ยนแปลงข้อมูลได้
- เขียนข้อมูลกลับลงดิสก์เมื่อสิ้นอายุขัย

RecordPtr: The Record

```
struct Record {
  String name;
  int    score;

  void readFile (const char* fileName);
  void writeFile (const char* fileName);
};
```

*(สังเกต: **struct** ใน C++ คือ **class** ที่ default access เป็น **public**)*

RecordPtr: Data, Constructor, Destructor

```
class RecordPtr {
public:
   RecordPtr (const char* fileName);
   ~RecordPtr();

private:
   String  fileName;
   Record* pRec;
};
```

RecordPtr: Data, Constructor, Destructor

```
inline RecordPtr::RecordPtr (const char* fileName)
  : fileName (fileName), pRec (0) {}

RecordPtr::~RecordPtr()
{
  if (pRec) {
    pRec->writeFile (fileName);
    delete pRec;
  }
}
```

RecordPtr: Dereference Operator

```
class RecordPtr {
public:
   // ...
   Record* operator->();
   Record& operator*();

private:
   void ensureRec();
};
```

RecordPtr: Reading File

```
void RecordPtr::ensureRec()
{
  if (!pRec) {
    pRec = new Rec;
    pRec->readFile (fileName);
  }
}
```

RecordPtr: Dereference Operator

```
Record* RecordPtr::operator->()
{
  ensureRec();
  return pRec;
}
Record& RecordPtr::operator*()
{
  ensureRec();
  return *pRec;
}
```

RecordPtr: การใช้งาน

```
void PrintRecord (const Record& r);

int main()
{
  RecordPtr p ("John");
  // using p.operator->()
  cout << p->name << " : " << p->score << endl;
  p->writeFile ("backup");
  p->score += 10;
  // using p.operator*()
  PrintRecord (*p);

  return 0;
}
```

การโอเวอร์โหลด Increment/Decrement Operator

increment (++) และ decrement (--)

- มีแบบ prefix และ postfix

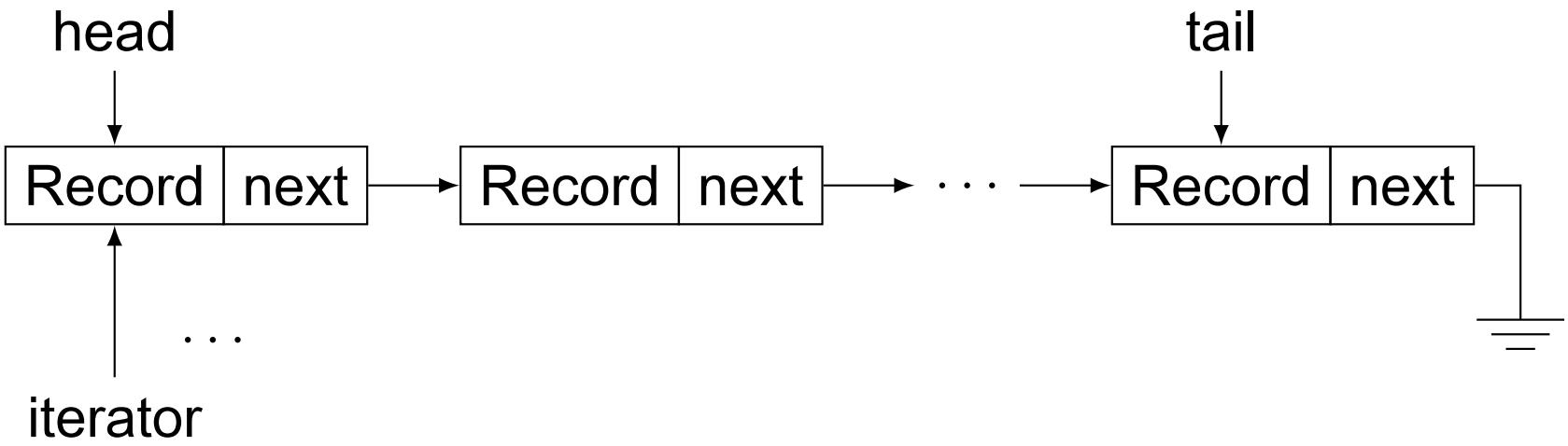
```
a = ++i;   // prefix
b = i++;   // postfix
x = --i;   // prefix
y = i--;   // postfix
```

- การโอเวอร์โหลด: แบบ postfix มีอาร์กิวเมนต์นิรนามชนิด `int` เพิ่ม

```
T T::operator++();    // prefix
T T::operator++(int); // postfix
T T::operator--();    // prefix
T T::operator--(int); // postfix
```

ตัวอย่าง: *RecQueue*

ต้องการสร้างคลาส **`RecQueue`** ซึ่งแทนคิวของระเบียนด้วย linked list



โดยมีตัวไล่แจง (iterator) สำหรับเข้าถึงแต่ละระเบียนตามลำดับ

ตัวอย่าง: RecQueue

struct Record สำหรับแทนระเบียนหนึ่ง ๆ

```
struct Record {
  char* name;
  int   score;

public:
  Record (const char* name, int score);
  ~Record();
};
```

*(สังเกต: **struct** ใน C++ คือ **class** ที่ default access เป็น **public**)*

RecQueue: Record Constructor & Destructor

```
inline Record::Record (const char* name, int score)
  : score (score)
{
  this->name = new char [strlen (name) + 1];
  strcpy (this->name, name);
}

inline Record::~Record()
{
  delete[] name;
}
```

RecQueue: RecNode

class RecNode สำหรับแทน node ใน linked list

```
class RecNode {
  friend class RecIter;
  friend class RecQueue;

private:
  RecNode (const char* name, int score)
    : rec (name, score), next (0) {}

private:
  Record   rec;
  RecNode* next;
};
```

RecQueue: RecNode

ข้อสังเกตของ **`class RecNode`**:

- เป็น class สำหรับใช้เป็นการภายในเท่านั้น
- constructor เป็น **`private`** ทำให้ผู้ใช้ทั่วไปสร้างออบเจกต์โดยตรงไม่ได้
- อนุญาตให้คลาส **`RecIter`** และ **`RecQueue`** ใช้งานได้เท่านั้น

RecQueue: RecIter

class RecIter ซึ่งจะใช้เป็นตัวไล่แจง (iterator) ในคิว

```
class RecIter {
  friend class RecQueue;
public:
  Record*       operator->();
  const Record* operator->() const;

  bool operator== (RecIter i);
  bool operator!= (RecIter i);

  RecIter& operator++();
  RecIter  operator++ (int);
};
```

RecQueue: RecIter

`class` `RecIter` ส่วน private

```
class RecIter {
  // ...
private:
  RecIter (RecNode* p);

private:
  RecNode* p;
};
```

- constructor เป็น **`private`** ทำให้ผู้ใช้ทั่วไปสร้างออบเจกต์โดยตรงไม่ได้
- อนุญาตให้คลาส **`RecQueue`** สร้างออบเจกต์ได้เท่านั้น
- อนุญาตให้ผู้ใช้ทั่วไปใช้งานได้ผ่าน operator ต่าง ๆ

`class RecIter`: constructor

```
inline RecIter::RecIter (RecNode* p) : p (p) {}
```

- ไม่ต้องมี destructor เนื่องจากพอยน์เตอร์เพียงแต่ชี้โดยไม่ได้ครอบครองออบเจกต์ (associate ไม่ใช่ compose)

RecQueue: RecIter

class RecIter: dereference operator

```
inline Record* RecIter::operator->()
{
  return &p->rec;
}
inline const Record* RecIter::operator->() const
{
  return &p->rec;
}
```

- ทะลุชั้น **RecNode** ที่ครอบอยู่ เพื่อให้ผู้ใช้เข้าถึงเฉพาะข้อมูลที่เปิดให้ใช้

class RecIter: comparison operator

```
inline bool RecIter::operator== (RecIter i) const
{
  return p == i.p;
}
inline bool RecIter::operator!= (RecIter i) const
{
  return !(*this == i);
}
```

- ใช้รูป member function ได้ เนื่องจากมีการควบคุมการสร้างออบเจกต์ (ไม่เกิดการแปลงชนิดที่อยู่นอกเหนือการใช้งานในคลาส)

RecQueue: RecIter

class RecIter: increment operator

```
inline RecIter& RecIter::operator++()
{
  if (p) p = p->next;
  return *this;
}
inline RecIter  RecIter::operator++ (int)
{
  RecIter old (p);
  if (p) p = p->next;
  return old;
}
```

ข้อสังเกตในการโอเวอร์โหลด increment operator

- รูป prefix ควรรีเทิร์น non-const ref เพื่อรองรับนิพจน์แบบนี้:

```
++++++i;
```

ซึ่งมีความหมายเป็น:

```
++(++(++i));
```

(ผลลัพธ์ของ prefix increment/decrement เป็น lvalue)

- รูป postfix ควรรีเทิร์นออบเจกต์ชั่วคราวที่แทนค่าเดิมก่อนกระทำ
(ผลลัพธ์ของ postfix increment/decrement เป็น rvalue)

RecQueue: ตัวคลาส RecQueue

```
class RecQueue {
public:
  RecQueue();
  ~RecQueue();

  void add (const char* name, int score);

  RecIter begin();
  RecIter end();

private:
  RecNode* head;
  RecNode* tail;
};
```

RecQueue: Constructor & Destructor

```
RecQueue::RecQueue() : head (0), tail (0) {}

RecQueue::~RecQueue()
{
  RecNode* p = head;
  while (p) {
    RecNode* next = p->next;
    delete p;
    p = next;
  }
}
```

RecQueue: `add()` Method

```
void RecQueue::add (const char* name, int score)
{
  RecNode* p = new RecNode (name, score);
  if (!head) {
    // first node
    head = tail = p;
  } else {
    // second node and beyond
    tail = tail->next = p;
  }
}
```

RecQueue: **`begin()`**, **`end()`** *Methods*

```
RecIter RecQueue::begin()
{
  return RecIter (head);
}

RecIter RecQueue::end()
{
  return RecIter (0);
}
```

- โหนดเริ่มต้นคือโหนดที่ต้นคิว
- สิ้นสุดการไล่แจงเมื่อผ่านโหนดสุดท้ายซึ่ง **`next`** เป็น null หรือคิวว่างเปล่า (**`head`** เป็น null)

RecQueue: การใช้งาน

```
int main()
{
  RecQueue q;
  q.add ("John", 8);
  q.add ("James", 7);
  q.add ("Jack", 5);

  for (RecIter i = q.begin(); i != q.end(); ++i) {
    cout << i->id << " : " << i->name << endl;
  }

  return 0;
}
```