제11장 별로 중요하지 않은 기능들

이번 장에서 학습할 내용

- 프렌드 함수
- 연산자 중복
- 형 변환

C++의 고급 기능인 프렌드와 연산자 중복을 살펴봅니다.

별로 중요하지 않습니다. ㅋㅋㅋ

프렌드 (friend)

- *friend*: 클래스의 private한 내부 데이터에 접근할 수 있도록 허용해 주는 키워드
  - *friend 함수*: 외부 함수에게 접근을 허용해 줌
  - *friend 클래스*: 외부 클래스에게 접근을 허용해 줌

프렌드 함수, 프렌드 클래스

- 함수의 원형은 비록 클래스 안에 포함하지만 절대 멤버 함수 아님
- 프렌드 함수는 클래스 내부의 모든 멤버 변수를 사용 가능

```
class Company {
   int sales;
   int profit;
public:
   Company(): sales(0), profit(0) { }

   // Company의 전용부분에 접근가능
   friend void sub(Company& c) {
        cout << c.profit << endl;
   }
};
```

```
void main()
{
    Company c1;
    sub(c1);
}
```

```
0
```

절대 멤버 함수가 아님!!! 일반함수임!!!

- 클래스도 프렌드로 선언할 수 있음
  - 예) MatrixI 에서 RandomWalk 클래스를 friend로 처리했음.

프렌드 함수의 용도

- 두 개의 객체를 비교할 때 많이 사용된다.

```
void main() {
  Date d1(1960, 5, 23)
  Date d2(2002, 7, 23);
  cout << d1.equals(d2) << endl;
}
```

```
void main() {
  Date d1(1960, 5, 23)
  Date d2(2002, 7, 23);
  cout << equals(d1,d2) << endl;
}
```

이해하기가 쉽다?

```
class Date {
  int year, month, day;
public:
  Date(int y, int m, int d) :year(y),month(m),day(d) {}
  bool equals(Date d2) {
    return year==d2.year && month==d2.month && day==d2.day;
  }
  friend bool equals(Date d1, Date d2) {
     return d1.year==d2.year && d1.month==d2.month && d1.day==d2.day;
  }
}
```

멤버 함수 구현

프렌드 함수 구현

복소수 클래스

- **add() 함수를 구현하는 다양한 방법**
  - c.add(a,b); ➔ 멤버 함수
  - c = a.add(b); ➔ 멤버 함수
  - c = add(a,b); ➔ 일반 함수(friend처리)

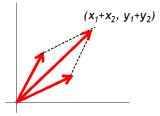
```
class Complex {
    double real, imag;
public:
    …
    void add (Complex a, Complex b);          // c.add(a,b);
    Complex add (Complex b);                  // c = a.add(b);
    friend Complex add (Complex, Complex);    // c = add(a,b);
};
```


중간 점검 문제

1. 프렌드 함수란 무엇인가?
2. 어떤 경우에 프렌드 함수가 유용한가?
3. **두 개의 Vector 객체를 더하는 프렌드 함수를 정의하라.**

연산자 중복

- **연산자 기호를 사용하는 편이** 함수를 사용하는 것보다 **이해하기 쉬운 경우**가 있다.
  - 복소수 클래스: +, -, *
  - 벡터 클래스: +, -, [ ]
    1. sum = x + y + z;
    2. sum = add(x, add(y, z));



- **잘못하면 더 헷갈리는 경우가 있다.**
  - 복소수 클래스: ++, -- (어떻게 증가? 감소?), --, /(뭘 나누지?)
  - 벡터 클래스: * (내적? 외적?), ++, --, /, ???

- 무분별하게 사용하지 말자.
  - Java등 다른 객체지향언어에서 잘 지원하지 않는다.
  - 사용하는 것만이 더 좋은 코드는 아니다.

연산자 중복

- operator overloading: 여러 연산자를 객체에 대해 적용하는 것
- C++에서 연산자는 함수로 정의

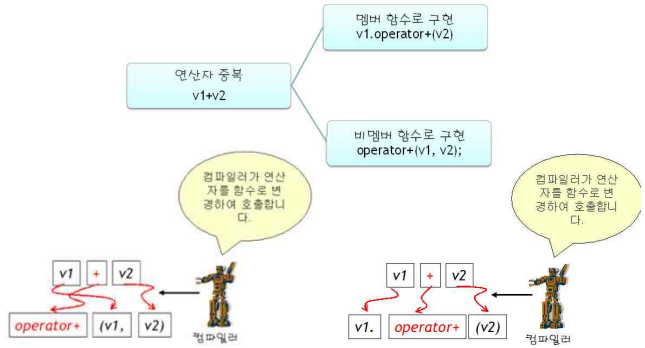
```
반환형 operator연산자(매개 변수 목록) {
        ....// 연산 수행
}
```

  - (예) `Vector operator+ (const Vector&, const Vector&);`

- 중복 함수 이름은 operator에 연산자를 붙이고 함수 기호

연산자	중복 함수 이름
+	operator+()
-	operator-()
*	operator*()
/	operator/()

연산자 중복 구현의 방법



멤버함수구현 ←→ 일반함수구현

- 두 방법 모두 가능함(많은 경우)

```
void main() {
  Complex a(1.0, 2.0);
  Complex b(3.0, 4.0);
  Complex c;
  c = a + b;
}
```

```
class Complex {
  double real, imag;
public:
  …
  Complex operator+ (Complex b) {
    Complex c(real+b.real, imag+b.imag);
    return c;
  }
};
```

멤버 함수 구현

일반함수 구현

```
class Complex {
  …
  friend Complex operator+ (Complex a, Complex b) {
    Complex c(a.real+b.real, a.imag+b.imag);
    return c;
  }
};
```


특별한 조건

- 항상 멤버 함수 형태로만 중복 정의가 가능한 연산자

연산자	설명
=	대입 연산자
()	함수 호출 연산자
[]	배열 원소 참조 연산자
->	멤버 참조 연산자

- 연산자 오버로딩이 불가능한 연산자

연산자	설명
::	범위 지정 연산자
.	멤버 선택 연산자
.*	멤버 포인터 연산자
?:	조건 연산자

중간 점검 문제

1. 벡터 사이의 뺄셈 연산자 -을 중복하여 보자.
2. 두개의 벡터가 같은지를 검사하는 == 연산자를 중복하라.
3. 문자열을 나타내는 String 클래스를 작성하고 + 연산자를 중복하라.

곱셈 연산자 중복

- 다음은 모두 각기 다른 연산자 오버로딩 함수가 필요하다.

```
Vector a, b, c;
c = a * b;          // 멤버함수, 일반함수 모두 가능
c = a * 2.0;        // 멤버함수, 일반함수 모두 가능
c = 2.0 * a;        // 일반함수로만 가능!!! Why???
   Vector operator*(Vector& u, Vector& v);
   Vector operator*(Vector& v, double alpha);
   Vector operator*(double alpha, Vector& v);
```

- **프로그래밍이 능숙하지 않으면 다음과 같이 구현하는 것이 좋다.**
  - `c.mult(a,b);` // c = a * b 의 의미
  - `c.mult(b);` // c *= b 의 의미
  - `c.mult(2.0);` // c *= 2.0 의 의미

== 연산자 중복

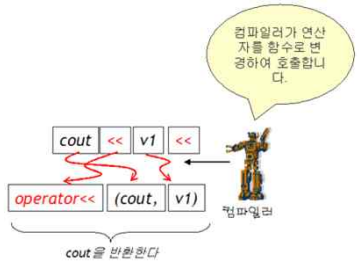
- 두개의 객체가 동일한 데이터를 가지고 있는지를 체크하는데 사용

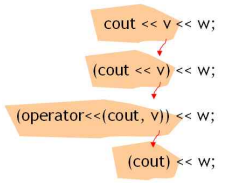
연산자	중복 함수 이름
==	operator==()
!=	operator!=()

```
class Complex {
  double real, imag;
public:
  …
  friend bool operator==(Complex a, Complex b) {
    return a.real==b.real && a.imag==b.imag;
  }
  friend bool operator!=(Complex a, Complex b) { return !(a==b); }
};
```


<<, >> 연산자와 iostream

- 연산을 수행한 후에 다시 스트림 객체를 반환하여야 함
- 일반함수 형태만 가능: 우리가 ostream 클래스를 바꿀 수 없다.
- 반드시 ostream&를 반환





<< >> 연산자의 중복

```
class Complex {
  double real, imag;
public:
  …
  friend ostream& operator<< (ostream& os, Complex c) {
    os << c.real << " + " << c.imag << "i\n";
    return os;
  }

 friend istream& operator>> (istream& in, Complex c) {
    in >> c.real >> c.imag;
    if( !in )
      c = Complex(0.0,0.0);
    return in;
  }

};
```

cout << a << b;
기능 추가

cin >> a >> b;
기능 추가

예외 처리를 해
주는 것이 좋음

= 연산자 중복

```
class Complex {
  double real, imag;
public:
  …
  Complex& operator=(Complex& a) {
    real = a.real;
    imag = a.imag;
    return *this;
  }
};
```

- 동적 할당 공간이 있으면 반드시 = 연산자를 중복 정의하여야 함
  - Vector 클래스
  - 깊은 복사/얕은 복사 문제

증가/감소 연산자의 중복

- ++와 -- 연산자의 중복

연산자	중복 함수 이름
++v	v.operator++()
--v	v.operator--()

- 전위와 후위 연산자를 구별하기 위하여 ++가 피연산자 뒤에 오는 경우에는 int형 매개 변수를 추가한다.

연산자	중복 함수 이름
++v	v.operator++()
v++	v.operator++(int)

[ ] 연산자의 중복

- 인덱스 연산자의 중복

연산자	중복 함수 이름
v[]	v.operator[]()

- 벡터 클래스

```
class Vector {
 int    dim;
 double* v;
public:
  …
  double& operator[](int id){
    return v[id];
  }
};
```

```
void main() {
  Vector vec;
  …
  vec[0] = 0.0;
  vec[1] = 1.0;
  …
}
```

포인터 연산자의 중복

- 간접 참조 연산자 *와 멤버 연산자 ->의 중복 정의

연산자	중복 함수 이름
*	operator*()
->	operator->()

```
class Pointer {
  int *pi;
public:
  Pointer(int *p): pi(p)
  {
  }
  ~Pointer()
  {
      delete pi;
  }
  int* operator->() const
  {
      return pi;
  }
  int& operator*() const
  {
      return *pi;
  }
};
```

```
int main()
{
  Pointer p(new int);
  *p = 100;
  cout << *p << endl;
  return 0;
}
```

실행결과

```
100
계속하려면 아무 키나 누르십시오 . . .
```


스마트 포인터

- 포인터 연산 정의를 이용하여서 만들어진 향상된 포인터를 스마트 포인터(smart pointer)라고 한다.
- 주로 동적 할당된 공간을 반납할 때 사용된다.

```
class Pointer {
  Car *pc;
public:
  Pointer(Car *p): pc(p){ }
  ~Pointer(){ delete pc; }
  Car* operator->() const { return pc; }
  Car& operator*() const { return *pc; }
};
```

```
int main()
{
  Pointer p(new Car(0,1,"red"));
  p->speed = 100;
  cout << *p;
  (*p).speed = 200;
  cout << *p;
  p->setSpeed(300);
  cout << *p;
  return 0;
}
```


함수 호출 연산자 ()의 중복

- 함수 호출때 사용하는 () 도 중복이 가능하다.

연산자	중복 함수 이름
f(...)	f.operator()(...)

- MatrixI 클래스에서 항목 접근
  - MatrixI m;
  - m(2,3) = 10;

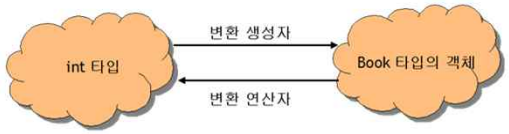
```
class MatrixI {
 int    rows, cols;
 int**  mat;
public:
  …
  double& operator()(int r, int c){
    return mat[r][c];
  }
};
```


중간 점검 문제

1. 벡터를 나타내는 Vector 클래스에 - 연산자를 중복하라.
2. 벡터를 나타내는 Vector 클래스에 += 연산자를 중복하라.
3. 문자열을 나타내는 String 클래스를 작성하고 << 연산자를 중복하라.

타입 변환

- 클래스의 객체들도 하나의 타입에서 다른 타입으로 자동적인 변환이 가능하다.
- 이것은 변환 생성자(conversion constructor)와 변환 연산자 (conversion operator)에 의하여 가능하다.



변환 생성자와 변환 연산자

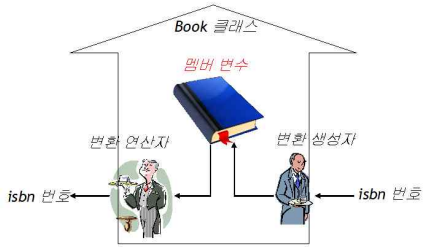


그림 12.7 변환 생성자와 변환 연산자

변환 생성자

```
class Book {
  int isbn;
  string title;
public:
  Book():isbn(0),title("unknown"){}
  Book(int nbr) {
    isbn = nbr;
    title = "unknown";
  }
  void display() {
    cout << isbn << ":"
     << title << endl;
  }
};
```

변환 생성자 (int->Book)

```
int main()
{
   // int 타입을 Book 타입에 대입
   Book b1 = 9782001;
   b1.display();
   b1 = 9783001; // 가능
   b1.display();
   return 0;
}
```

```
9782001:unknown
9783001:unknown
계속하려면 아무 키나 누르십시오 . . .
```

컴파일러는 변환 생성자를 이용하여서 정수 9782001을 Book 객체로 변환하는 것이다.

버그의 원인

- 변환 생성자는 버그의 원인이 될 수도 있다.
- Book b2 = 3.141592;
- b2.display();
- (설명) 실수->정수-> 객체
- (해결책)만약 생성자 앞에 explicit를 붙이면 컴파일러가 자동적으로 타입 변환을 하지 못한다.

변환 연산자의 중복 정의

```
class Book {
  int isbn;      // 책의ISBN
  string title; // 책의제목

public:
  Book():isbn(0),title("unknown"){}

  Book(int nbr) {
    isbn = nbr;
    title = "unknown";
  }

  operator int() const {
    return isbn;
  }

  void display() {
    cout << isbn << ":"
     << title << endl;
  }
};
```

변환 생성자 (int->Book)

변환 연산자 (Book->int)

```
void main()
{
    // 변환생성자실행!
    Book b1 = 9782001;
    b1.display();

    // 변환연산자실행!
    int isbn = b1;
    cout << isbn << endl;
}
```


연산자 중복시 주의할 점

- 새로운 연산자를 만드는 것은 허용되지 않음.
- :: 연산자, .* 연산자, . 연산자, ?: 연산자는 중복이 불가능.
- 내장된 int형이나 double형에 대한 연산자의 의미 변경 불가능.
- 연산자들의 우선 순위나 결합 법칙은 변경되지 않는다.

- 만약 + 연산자를 오버로딩하였다면 일관성을 위하여 +=, -= 연산자도 오버로딩하는 것이 좋다.
- 일반적으로 산술 연산자와 관계 연산자는 비멤버 함수로 정의한다. 반면에 할당 연산자는 멤버 함수로 정의한다.

중간 점검 문제

1. 클래스 Car를 string으로 변환하는 변환 연산자를 작성하시오.
2. 변환 연산자의 위험성은 무엇인가?

형변환

```
double f = 3.141592;
int   i = (int) f;
```

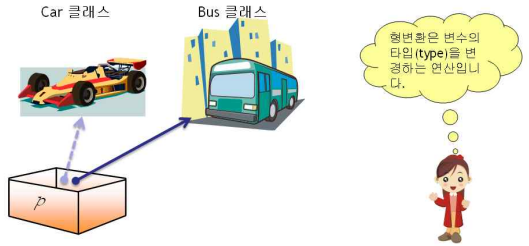


그림 11.6 형변환이란?

C언어와 C++의 형변환

- C 에서는...
- 너무 관대함!

```
double f = 3.141592;
int   i = (int) f;
```

```
int main()
{
   double f = 3.141592;
   double *pf = &f;

   int  *pi = (int *) pf;

   cout << *pi << endl;
   return 0;
}
```

실행 결과

```
-57999238
```

- C++ 에서는... 다양화

형변환 연산자	의미
static_cast	기본 타입의 변환이나 상속 관계에 있는 클래스 포인터를 변환할 때 사용
dynamic_cast	상속 관계에 있는 클래스 포인터를 안전하게 변환할 때 사용, 실행 시간에 식별 가능
const_cast	상수 속성을 변경
reinterpret_cast	관련없는 포인터 사이의 무조건 변환, 정수형과 포인터 사이의 변환

static_cast

- 컴파일 시간에 논리적으로 타당한 변환만을 수행한다
- 예: double 변수 f를 int 형으로 변환하려면
  - `i = static_cast<int>(f);`
  - 타당하지 않으면 ➔ 오류 발생

```
int main()
{
   int i = 9;
   double f = 3.141592;
   int *pi ;
   double *pf = &f;

   i = static_cast<int>(f);          // OK
   pi = static_cast<int*>(pf);       // 오류!
   return 0;
}
```

실행 결과

```
1>c:\users\chun\documents\visual studio 2008\projects\test\test\test.cpp(12) : error C2440:
'static_cast' : 'double *'에서'int *'(으로 변환할 수 없습니다
1>          겨리킨 형식이 관련이 없습니다. 변환하려면 reinterpret_cast, C 스타일캐스트 또는 함
수스타일캐스트가 필요합니다
```


예제

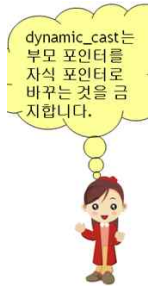
```
class Car
{
   //...;
};
class Bus : public Car
{
   //...;
};

int main()
{
   Bus *pBus1 = new Bus;
   Car *pCar2 = static_cast<Car *>(pBus1);          // OK

   Car *pCar1 = new Car;
   Bus *pBus2 = static_cast<Bus *>(pCar1);          // OK 그러나 잠재적으로 위험
   return 0;
}
```


dynamic_cast

- 기본적으로 상향 형변환(upcast)은 허용하지만 하향 형변환(downcast)은 허용하지 않는다.




```
int main()
{
   Bus *pBus1 = new Bus;
   Car *pCar1 = dynamic_cast<Car *>(pBus1);          // OK!

   Car *pCar2 = new Car;
   Bus *pBus2 = dynamic_cast<Bus *>(pCar2);          // ① 컴파일 오류!

   Car *pCar3 = new Bus;
   Bus *pBus3 = dynamic_cast<Bus *>(pCar3);          // ② 컴파일 오류!

   return 0;
}
```

dynamic_cast

- dynamic_cast 연산자는 실행 시간에 포인터가 가리키는 객체의 타입을 보고 판단하여서 변환이 올바르면 변환된 타입이 반환된다. 그렇지 않으면 NULL이 반환된다.
- 이 기능을 사용하려면 부모 클래스가 가상 함수를 사용하고 있어야 한다.

```
Car *pCar3 = new Bus;
Bus *pBus3 = dynamic_cast<Bus *>(pCar3);        // OK!
if( pBus3 == 0 )
    cout << "변환3 불가능" << endl;
```


const_cast

- const_cast는 타입에서 const 속성을 제거하거나 추가한다

```
void display(char *s)
{
  cout << s << endl;
}

int main()
{
  const char *saying = "A bad workman (always) blames his tools";
  display(const_cast<char *>(saying));
  return 0;
}
```

saying의 타입에서 const 를 제거한다.

reinterpret_cast

- 어떤 포인터 타입이라도 다른 포인터 타입으로 변환

```
class Car
{
};
class Box
{
};

int main()
{
  char *pc;
  pc = reinterpret_cast<char *>(0x10000ef);

  Car *pCar1 = new Car;
  Box *pBox1 = reinterpret_cast<Box *>(pCar1);
  return 0;
}
```

정수를 char*로 변환

Car 클래스 포인터를 Box 클래스 포인터로 변환

타입 정보

- typeid 연산자는 실행 시간에 객체의 타입을 식별할 수 있다.
- typeid가 반환하는 값은 type_info에 대한 참조자이다.

실행 결과

```
class Car *
class SportsCar
class SportsCar *
class SportsCar
```

```
class Car {
public:
  virtual void display() {}
};

class SportsCar : public Car {
};

int main() {
  SportsCar* pd = new SportsCar;
  Car* pb = pd;
  cout << typeid( pb ).name() << endl;
  cout << typeid( *pb ).name() << endl;
  cout << typeid( pd ).name() << endl;
  cout << typeid( *pd ).name() << endl;
  delete pd;
  return 0;
}
```