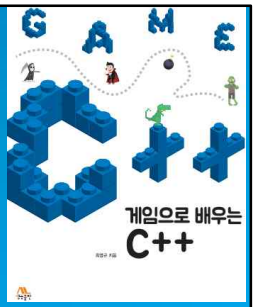
게임으로 배우는 C++



CHAPTER 11

프렌드와 연산자 중복

몬스터월드6(여유있는 삶)

- Monster World 6
  - 처녀귀신이 그라운드에서 가장 활동량이 많음
  → 대부분의 경우 처녀귀신이 가장 많은 아이템을 먹음
  - 이동거리를 계산→거리가 기준 이상이면 강제 휴식 처리
  - 신인류는 제외. 왜? 인간이니까.

2

11장 학습 목표

- friend의 의미와 친구 클래스, 친구 함수를 이해한다.
- 연산자 중복의 개념과 잠재적인 문제점을 이해한다.
- 연산자 중복에서 friend의 필요성을 이해한다.
- 다양한 연산자 중복 함수를 구현하는 능력을 기른다.

3

11.1 프렌드 선언

- 프렌드 클래스
- friend 클래스 선언과 활용 예
- 프렌드 함수

4

프렌드 클래스

- 외부 클래스에 자신의 내부를 사용할 수 있도록 허용

프렌드 클래스

- 형식

```
class 허용해주는_클래스명 {
        friend class 허용받는_클래스명;
        ...
};
```

- 객체지향의 정보 은닉의 개념에 반하는 방법
  - friend를 무절제하게 사용하는 것은 좋지 않음

friend 클래스 선언과 활용 예

```
class Point {
  int x, y;
  friend class Shape;
  friend class Line;
  ...
};
```

```
void main() {
  Line l(3, 4, 5, 6);
  l.draw();
}
```

```
C:₩WINDOWS₩system32₩cmd.exe
[ 선분 ] P1=(3,4) P2=(5,6)
계속하려면 아무 키나 누르십시오 . . .
```

```
class Shape {
protected:
  Point  p;
public:
  Shape(int x=0, int y= 0) : p(x, y) { }
  void draw() { printf("[ 형태 ] 위치=(%d,%d)\n", p.x, p.y); }
};
```

```
class Line : public Shape {
  Point  q;
public:
  Line(int x1=0, int y1=0, int x2=0, int y2=0)
    : Shape(x1,y1), q(x2,y2) { }
  void draw() {
     printf("[ 선분 ] P1=(%d,%d) P2=(%d,%d)\n", p.x, p.y, q.x, q.y);
  }
};
```

7

프렌드 함수

- 일반 함수에게 클래스 내부를 사용할 수 있도록 허용

```
class Point {
    ...
    void print(char *s = "점") { printf("%s(%d,%d)", s, x, y); }

    friend Point addPoint(Point& p, Point& q) {     // 일반 함수. inline
        ...
    }
    friend Point subPoint(Point& p, Point& q) {     // 일반 함수. inline
        return Point(p.x - q.x, p.y - q.y);
    }
    friend double avgPointX(Point* list, int n);    // 일반 함수
};

// double Point::avgPointX(...)가 아님
double avgPointX(Point* list, int n)                // 일반 함수. 외부 구현
{
    double sum=0;
    for (int i = 0; i < n; i++)
        sum += list[i].x;
    return sum / n;
}
```

8

11.2 연산자 중복

- 연산자 중복
- 복소수 덧셈의 다양한 구현 방법
- 연산자 중복의 잠재적인 문제점

연산자 중복

- 이미 사용해 본 연산자 중복

```
int i=1, j=2, k;          // int형 변수 선언
double a=1.0, b=2.0, c;   // double형 변수 선언
k = i + j;                // 정수의 덧셈
c = a + b;                // 실수의 덧셈
```

- 복소수를 더하는 다양한 함수 → 방법4: 연산자 중복

복소수의 다양한 연산들

$(a+bi)+(c+di)=(a+c)+(b+d)i$

$(a+bi)-(c+di)=(a-c)+(b-d)i$

$(a+bi)(c+di)=(ac-bd)+(bc+ad)i$

$(a+bi)/(c+di)=(\frac{ac+bd}{c^2+d^2})+(\frac{ac+bd}{c^2+d^2})\ i$

$\overline{(a+bi)}=(a-bi)$

방법1: `c3.add(c1,c2);`

방법2: `c3 = add(c1,c2);`

방법3: `c3 = c1.add(c2);`

방법4: `c3 = c1 + c2;`

복소수 클래스에서 하나의 연산을 구현하는 방법에도 여러 가지가 있다.

복소수 덧셈의 다양한 구현 방법

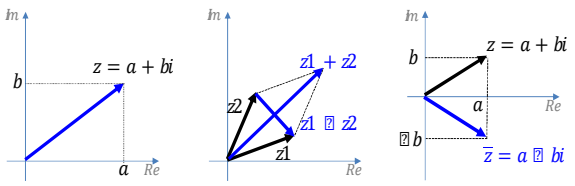
```
class Complex
{
    double      real, imag;
    ...
    void add(Complex a, Complex b) {              // 방법 1: 멤버 함수
       real = a.real + b.real;
       imag = a.imag + b.imag;
    }
    friend Complex add(Complex a, Complex b) {    // 방법 2: 일반 함수
       return Complex(a.real + b.real, a.imag + b.imag);
    }
    Complex add(Complex b) {                      // 방법 3: 멤버 함수
       return Complex(real + b.real, imag + b.imag);
    }
    // 방법 4: 연산자 중복 함수 (멤버 함수로 구현)
    Complex operator+(Complex b) {
       return Complex(real + b.real, imag + b.imag);
    }
};
```

```
s1.add(c1, c2);
s2 = add(c1, c2);
s3 = c1.add(c2);
s4 = c1 + c2;
```

```
C:₩WINDOWS₩system32₩cmd.exe
s1.add(c1,c2) :  4.00 + 6.00i
s2=add(c1,c2) :  4.00 + 6.00i
s3=c1.add(c2) :  4.00 + 6.00i
s4= c1 + c2   :  4.00 + 6.00i
계속하려면 아무 키나 누르십시오 . . .
```

11

연산자 중복의 잠재적인 문제점



복소수 공간 | 복소수의 덧셈과 뺄셈 | 켤레 복소수(conjugate)

12

11.3 연산자 중복의 종류

- 연산자 중복의 특징
- 단항 연산자와 이항 연산자
- 멤버 함수와 일반함수
- Lab: 복소수 클래스

연산자 중복의 특징

- 정해진 피연산자의 수를 변경할 수 없음
  - 단항/이항
- 멤버 함수와 일반 함수로 구현이 가능함
- 멤버 함수로 구현
  - 왼쪽 피연산자가 속한 클래스의 멤버 함수로만 구현 가능
- 우선순위와 결합 방향은 정확히 동일하게 유지
- 특징적인 연산자
  - 멤버 함수로만 중복: "=", "( )", "[ ]", "->" 등
  - 중복 불가: "::", ".", "? :" 등
- 객체가 포함된 연산에 대해서만 가능
  - 3 + 4 는 중복 불가
  - a와 b가 Complex의 객체라면 a+b, a+4, 3+b 등은 가능

단항연산자와 이항연산자

- 단항 연산자 –를 멤버 함수로 구현

```
Complex operator -() {
        return Complex(-real, -imag);
}
```

- 이항 연산자 – 를 멤버 함수로 구현

```
Complex operator -( Complex a ) {
        return Complex(real - a.real, imag - a.imag);
}
```


멤버 함수와 일반함수

- 예: 복소수와 실수의 곱셈

```
w = z * 0.5; // 복소수 객체 * double
```

– 멤버 함수로 구현

```
Complex operator* (double s) {
        return Complex(s*real, s*imag);
}
```

– 일반 함수로 구현

```
friend Complex operator *( Complex z, double s) {
        return Complex(z.real * s, z.imag * s);
}
```

- 예: 실수와 복소수의 곱셈

```
w = 0.5 * z; // 복소수 객체 * double
```

- 멤버 함수로 구현 불가. Why?

Lab: 복소수 클래스

```
class Complex
{
  double real, imag;
public:
  Complex(double r=0.0, double i=0.0): real(r), imag(i) {  }
  void print(char* msg = "복소수 = ") { ... }

  // 산술 단항 - : -c(음수)
  Complex operator-() { return Complex(-real, -imag); }
  // 산술 단항 ~ : ~c 켤레 복소수 (의미가 애매하므로 중복하지 않는 것이 더 좋음)
  Complex operator~() { return Complex(real, -imag); }
  // 산술 이항 - : a - b
  Complex operator-(Complex b) {return Complex(real-b.real, imag-b.imag); }

  // 산술 * : 복소수와 스칼라의 곱. c*s와 s*c 연산이 각각 다른 연산자에 의해 처리됨
  Complex operator*(double s) { return Complex(s*real, s*imag); }
  friend Complex operator*(double s, Complex c) {
    return Complex(s*c.real, s*c.imag);
  }
```


```
 // 비교 : c1 == c2, c1 != c2
  bool operator==(Complex a) { return real==a.real && imag==a.imag; }
  bool operator!=(Complex a) { return real != a.real || imag != a.imag; }

  // 다른 비교 연산자 >, <, >=, <=, 등은 개념이 애매함. 중복하지 않는 것이 바람직함.
  // 대입 : c1=c2, c1+=c2, c1-=c2
  Complex& operator=(Complex a) {real =a.real; imag=a.imag; return *this;}
  Complex& operator+=(Complex a){real+=a.real;imag+=a.imag; return *this;}
  Complex& operator-=(Complex a){real-=a.real;imag-=a.imag; return *this;}

  // 다른 대입 연산자 <<=, >>=, 등도 개념이 애매함. 중복하지 않는 것이 바람직함.
};
```

```
void main() {
  Complex c1(1, 2), c2(3, 4), c3, c4, c5, c6, c7, c8;
  c1.print("c1(1, 2)      : ");
  c2.print("c2(3, 4)      : ");
  c3 = -c1;       c3.print("c3 = -c1      : ");
  c4 = ~c1;       c4.print("c4 = ~c1      : ");
  c5 = c1 - c2;   c5.print("c5 = c1 - c2  : ");
  c5 = c1 * 2.0;  c5.print("c5 = c1 * 2.0 : ");
  c6 = 3.0 * c1;  c6.print("c6 = 3.0 * c1 : ");
  c7 = c8 = c1;   c7.print("c7 = c8 = c1  : ");
  c7 += c1;       c7.print("c7 += c1      : ");
  c8 -= c1;       c8.print("c8 -= c1      : ");
}
```

```
C:₩WINDOWS₩system32₩cmd.exe
c1(1, 2)       :  1.00 + 2.00i
c2(3, 4)       :  3.00 + 4.00i
c3 = -c1       :  -1.00 + -2.00i
c4 = ~c1       :  1.00 + -2.00i
c5 = c1 - c2   :  -2.00 + -2.00i
c5 = c1 * 2.0  :  2.00 + 4.00i
c6 = 3.0 * c1  :  3.00 + 6.00i
c7 = c8 = c1   :  1.00 + 2.00i
c7 += c1       :  2.00 + 4.00i
c8 -= c1       :  0.00 + 0.00i
계속하려면 아무 키나 누르십시오 . . .
```


11.4 특별한 연산자 중복

- 증감 연산자
- 비트이동 연산자
- 형 변환 연산자
- 인덱스, 함수 호출 연산자
- Lab: Rational 클래스 사용 예

특별한 연산자 중복

- 증감 연산자

```
Complex& operator++()     { real += 1; return *this; }
Complex& operator++(int)  { imag += 1; return *this; }
```

- 비트이동 연산자

```
Complex c;
cout << "복소수의 실수부와 허수부를 입력하세요 = " ;
cin >> c ;
cout << x << endl;
```

```
friend ostream& operator << (ostream& os, const Complex& c) {
        os << "(" << c.real << "," << c.imag << ")";
        return os;
}
friend istream& operator >> (istream& is, Complex& c) {
        is >> c.real >> c.imag;
        return is;
}
```


특별한 연산자 중복

- 예: 유리수 클래스

```
Class Rational
{
        int top;          // 유리수의 분자
        int bottom;       // 유리수의 분모(0이 아니어야 함)
Public:
        Rational(int t = 0, int b = 1) : top(t), bottom(b) { }
        ...
};
```

- 증감 연산자

```
        Rational& operator++() { top += bottom; return *this; }
        Rational operator++(int) {
                top += bottom;
                return Rational(top - bottom, bottom);
        }
```

특별한 연산자 중복

- 비트이동 연산자

```
friend ostream& operator<<(ostream& os, const Rational& f) {
        os << f.top << "/" << f.bottom;
        return os;
}
friend istream& operator>>(istream& is, Rational& f) {
        is >> f.top >> f.bottom;
        return is;
}
```

- 형 변환 연산자

```
operator double() { return (double)top/bottom; }
```

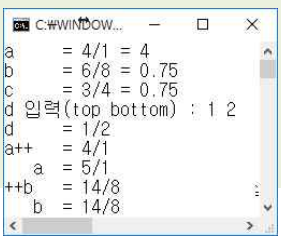
- 인덱스, 함수 호출 연산자

```
int& operator[](int id) {
        if (id == 0) return top;
        else if (id == 1) return bottom;
        else exit(0);
}
int& operator()(int id) { return (*this)[id]; }
```

23

Lab: Rational 클래스 사용 예

```
void main()
{
        Rational a(4), b(6, 8), c, d;
        c[0] = 3;
        c(1) = 4;
        cout << "a = " << a << " = " << (double)a << endl;
        cout << "b = " << b << " = " << (double)b << endl;
        cout << "c = " << c << " = " << (double)c << endl;
        cout << "d 입력(top bottom) : ";
        cin >> d;
        cout << "d = " << d << endl;
        cout << "a++ = " << a++ << endl;
        cout << " a = " << a << endl;
        cout << "++b = " << ++b << endl;
        cout << " b = " << b << endl;
}
```



24

11.5 깊은 복사와 연산자 중복

- 깊은 복사와 연산자 중복
- Vector의 대입연산?
- Vector에서 깊은 복사의 구현
- Matrix 클래스와 ()연산자

깊은 복사와 연산자 중복

- Vector 클래스의 add()와 operator+() 함수

```
class Vector {
    double* arr;
    int  dim;
Public:
    Vector(int d=0): dim(d),arr(NULL) { arr = new double[dim]; }
    ~Vector() { delete[] arr; }
    double& operator[] (int id){ return arr[id]; }
    friend ostream& operator << (ostream& os, Vector& v) {...}

    void add(Vector& u, Vector& v) {
         for (int i = 0; i < dim; i++)
                  arr[i] = u[i] + v[i];
    }

    Vector operator + (Vector& v) {
         Vector sum(v.dim);
         for (int i = 0; i < dim; i++)
             sum[i] = arr[i] + v[i];
         return sum;
    }
};
```

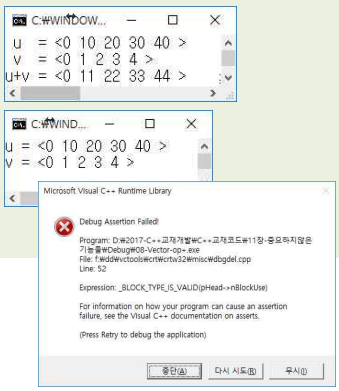
Vector의 대입연산?

- 실행 결과

```
void main()
{
        Vector  u(4), v(4), w(4);
        for (int i = 0; i < 4; i++) {
                u[i] = i*10.0;
                v[i] = i;
        }
        cout << " u = " << u << endl;
        cout << " v = " << v << endl;

        w.add(u, v);
        cout << "u+v = " << w << endl;

        w = u + v;
        cout << "u+v = " << w << endl;
}
```



- 대입 연산자 중복 정의 필요!!!

Vector에서 깊은 복사의 구현

```
...
Vector& operator = (Vector& v) {            // 대입연산자 중복
        clone(v);
        return *this;                       // v1 = v2 = v3 = v4; 를 위해
}
void clone(Vector& a) {                     // 깊은 복사를 위한 clone 함수
        if (dim > 0) delete[] arr;
        dim = a.dim;
        arr = new double[dim];
        for (int i = 0; i < dim; i++)
                arr[i] = a.arr[i];
}
Vector(Vector& v) : arr(NULL) {clone(v);}   // 복사 생성자
...
```

Matrix 클래스와 ()연산자

```
class Matrix {
        int     rows, cols;
        int**   mat;
public:
        ...
        int&    operator()(int x, int y)  { return mat[y][x]; }
        ...
};
void main()
{
        Matrix m(3, 2);
        m(0, 0) = 0;    m(1, 0) = 1;    m(2, 0) = 2;
        m(0, 1) = 3;    m(1, 1) = 4;    m(2, 1) = 5;
        m.print();
}
```

```
matrix 2x3
  0   1   2
  3   4   5
```

29

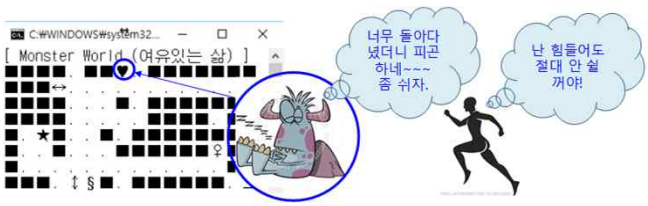
11.6 응용: MonsterWorld 6

- Monster World 6: 여유 있는 삶
- Point 클래스 + 연산자 중복
- Monster와 자식 클래스
- 고찰

30

Monster World 6: 여유 있는 삶

- Monster와 자식 클래스
  - 몬스터에 움직인 거리 계산과 쉬는 시간을 관리
  - 멤버 변수를 추가 및 관련 함수
- Point 클래스 사용
  - 좌표를 표현
  - 여러 가지 연산자 중복
  - Canvas 클래스에 출력할 좌표로 Point 객체를 전달받음



31

Point 클래스

- friend + 연산자 중복 사용

```
class Point {
        int x, y;
        friend class Monster;
        friend class Canvas;
public:
        Point(int xx = 0, int yy = 0) : x(xx), y(yy) { }
        int& operator[] (int id) {        // 인덱스 연산자
                if (id == 0) return x;
                else if (id == 1) return y;
                else exit(0);
        }
        operator double() { return sqrt((double)x*x + y*y); }
        void operator() (int maxx, int maxy) {...}
        Point operator- ()        { return Point(-x, -y); }
        bool operator== (Point &p){ return x == p.x && y == p.y; }
        bool operator!= (Point &p){ return x != p.x || y != p.y; }
        Point operator- (Point &p){ return Point(x - p.x, y - p.y); }
        Point operator+ (Point &p){ return Point(x + p.x, y + p.y); }
        void operator+= (Point &p){ x += p.x, y += p.y; }
        void operator-= (Point &p){ x -= p.x, y -= p.y; }
};
```

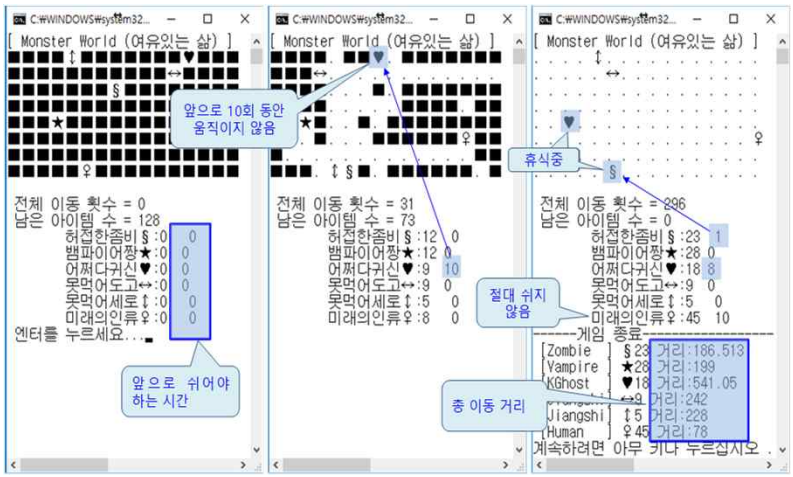
32

Monster와 자식 클래스

- 수정사항
  - 현재 위치로 Point 객체 p 사용.
  - 거리 계산 및 휴식 처리를 위한 데이터 멤버 추가
  - Point의 연산자 중복 함수를 사용하도록 멤버 함수 수정
  - move() 함수에서 isSleep()을 검사하도록 수정
  - Point 의 x와 y 값 접근 방법 수정:p[0], p[1] ➔ 더 불편해 짐

```
class Monster{
    ...
    Point q, p;                    // 추가: 이전 위치(q)와 현재 위치(p)
    int nSleep;                    // 추가: 쉬어야 하는 횟수
    double dist;                   // 추가: 쉬고 난 다음 움직인 거리
    double total;                  // 추가: 총 움직인 거리
    void eat(int** map) {...}      // 수정
    bool isSleep() {...}           // 추가
public:
    virtual void move(int** map, int maxx, int maxy) {...}  // 수정
        ...
};
```


실행 결과



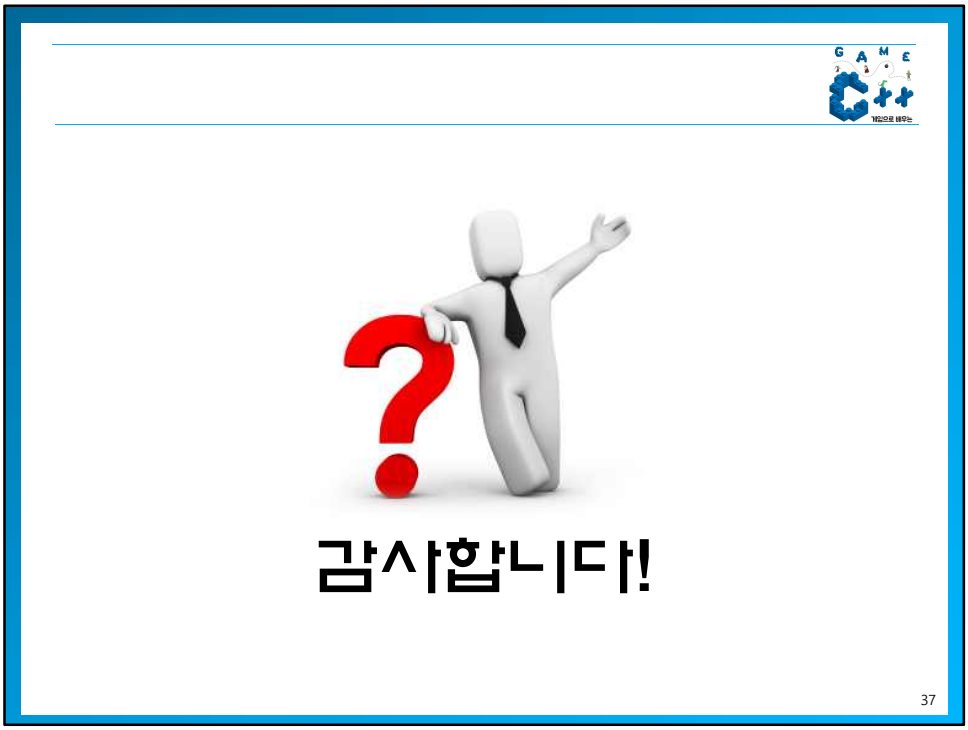
고찰

- 자동으로 움직이는 몬스터들은 일정 거리 이상 움직이면 10회 동안 쉰다. 인간은 스스로 판단하므로 쉬지 않는다.
- 처녀귀신이 보통 가장 멀리 움직이므로 쉬는 상황이 가장 자주 발생한다.
- 총 이동거리에 비해 인간이 가장 효율적으로 아이템을 먹을 수 있다.
- Point 클래스를 사용하고 다양한 연산자 중복과 friend 선언을 사용해보았다.
- 연산자 중복은 제한적으로 사용하는 것이 좋다.
  - 예를 들어, x와 y좌표를 참조하기 위해 [0], [1]과 같이 인덱스 연산자를 사용한 부분이다, 함수 호출 연산자를 clip() 용으로 사용한 것은 약간은 억지스러워 보인다. 선택은 개발자의 몫이다.

35

11장 요약문제, 연습문제, 실습문제

36


감사합니다!
37