Object-Oriented Programming and C++ 단기코스

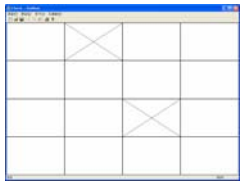
321190
2008년 가을학기
9/9/2008
박경신

Overview

- C++ Class 선언, 정의, 사용
- Data Encapsulation
- Class constructor, destructor
- Function, Operator Overloading
- Inheritance
- Function Overriding
- Virtual Function
- new, delete
- Default parameter
- Reference

Object-Oriented Programming

- 객체지향 프로그래밍 (Object-Oriented Programming)
  - 프로그램 기본단위를 객체(object)로 해서 프로그램을 개발
  - 프로그램 기본단위: C는 함수이고, C++는 클래스
- 구조적 프로그래밍 vs. 객체지향 프로그래밍
  - 예제: 클릭한 곳에 X표하는 프로그램
  - 구조적 프로그래밍
    - 마우스가 클릭되면, 클릭된 점의 좌표를 계산
    - 클릭된 점의 좌표가 몇 번째 격자인지 계산
    - 그 격자의 모서리 점을 계산하여 대각선을 그리기



  - 객체지향 프로그래밍: 각각의 격자를 하나의 오브젝트로 처리
    - 마우스가 윈도우에 클릭되면, 자기자신 윈도우 전체에 대각선을 그리기

Class

- 클래스는 객체를 정의한 데이터 타입
- class 키워드를 사용하여 객체를 구성하는 데이터와 함수들을 정의
- 클래스의 인스턴스(객체)를 생성하여 사용

```
//Point class 선언
class Point {
//데이터(멤버변수)
        int _x;
        int _y;
public:
//메소드(멤버함수)
        void setX(int x){ _x = x; }
        void setY(int y){ _y = y; }
        void move(int x, int y){…}
}
```

```
//Point 객체 사용
void main() {
//Point 클래스의 인스턴스 생성
    Point p;
//Point 객체의 함수 접근(호출)
    p.setX(100);
    p.setY(40);
    p.move(20, 50);
}
```

Class 선언, 구현, 사용

- 클래스 선언

```
class Where
{
public:
    int data;
    void PrintPointer( );
};
```

- 클래스 구현

```
void Where::PrintPointer( )
{
    cout << "오브젝트의 주소는 " << this << " 번지입니다.₩n";
}
```

- 클래스 사용

```
void main( )
{
    Where a, b, c;

    a.PrintPointer( );
    b.PrintPointer( );
    c.PrintPointer( );
}
```

  - 인스턴스 (Instance): 메모리에 생성된 클래스의 실체
  - 멤버변수는 각 인스턴스마다 독립적으로 생성
  - 멤버함수는 메모리에 한번만 로딩되고 모든 인스턴스가 이를 공유

Data Abstraction

- 자료 추상화 (Data Abstraction)
  - 캡슐화 (Encapsulation), 정보은닉 (Information Hiding)
- 캡슐화 (Encapsulation)
  - 캡슐화의 필요성
    - 사용자는 오디오의 사용법만 파악
    - 사용자가 오디오의 반도체 동작원리나 내부회로까지 파악하여 내부부품을 떼었다 붙였다 하고 배선을 끊었다 이었다 하면 고장
  - C 구조체 (structure)
    - 변수만 캡슐화, 외부함수에 의해 수동적으로 제어
  - C++ 클래스
    - 변수,함수를 캡슐화, 내부함수를 통해 능동적으로 동작
    - public: 외부에서 보이는 변수
    - protected, private: 내부에만 보이는 변수

C 언어 Structure

- 인터페이스 파일과 구현파일의 분리
  - 헤더 파일(.h): 인터페이스 파일 & 소스 파일(.c): 구현 파일
- 헤더 파일
  - 함수 프로토타입만 보여 줌
  - 블랙 박스(정보의 은닉, 구현을 볼 수 없음)
  - 계약서 역할(작업의 정의를 자세하고 정확하게 기술)

```
/* 헤더 파일의 예 */
typedef struct  {
        int _x;
        int _y;
} Point;

void setX(int x);
void setY(int y);
void move(int x, int y);
```


Class 내부와 외부

- 클래스 내부에서는 클래스의 멤버변수, 멤버 함수를 직접 접근 가능
- 클래스 외부에서는 클래스의 인스턴스를 통하여 공개된 (public) 멤버 변수, 멤버 함수를 접근
- 클래스 명을 선언하고 포함된 멤버변수와 멤버함수를 선언

```
class Date {
        private:
                int year;
                int month;
                int day;
                bool valid;
                bool IsValidDate(int y, int m, int d);
        public:
                bool SetDate(int y, int m, int d);
                void PrintDate();
};
```

Class 내부와 외부

- 클래스명::멤버함수(..) 형식으로 멤버 함수 정의

```
BOOL Date::IsValidDate(int y, int m, int d) {
        static int lenMonth[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
        if (y<1900||y>2006) return false;
        if (m<1 || m >12) return false;
        if (d<1 || d>lenMonth[m]) return false;
        return true;
}
```

- 클래스의 멤버함수에서의 멤버함수, 멤버변수 접근

```
BOOL Date::SetDate(int y, int m, int d) {
        if (IsValidDate(y, m, d) == true) { // 멤버함수 IsValidDate() 접근
                year = y;           // 멤버함수에서 멤버변수 year 접근
                month = m;          // 멤버함수에서 멤버변수 month 접근
                day = d;            // 멤버함수에서 멤버변수 day 접근
                valid = true;       // 멤버함수에서 멤버변수 valid 접근
                return true;
        } else {  valid = false;
                return false;
}
```


Class 내부와 외부

- 클래스의 외부함수에서의 멤버함수, 멤버 변수 접근

```
void main() {
        Date MyBirthday; // Date 클래스의 인스턴스 생성
        MyBirthday.SetDate(1987, 11, 11); // 인스턴스의 공개된 멤버함수 접근
        MyBirthday.PrintDate();           // 인스턴스의 공개된 멤버함수 접근
        if (d<1 || d>lenMonth[m]) return false;
        return true;
}
```


Default Parameter

- Default Parameter (기본 인자)의 사용법

```
class Date {
public:
        bool SetDate(int y=2000, int m=1, int d=1);
…
}

void main()
{
        Data date;
        date.SetDate();     // 2000, 1, 1을 지정
        date.SetDate(2001);         // 2000, 1, 1을 지정
        date.SetDate(2001, 5);      // 2000, 5, 1을 지정
        date.SetDate(2001, 8, 28); // 2000, 8, 28을 지정
}
```


Class Constructor & Destructor

- 생성자 (Constructor) 함수
  - 클래스를 초기화하며 인스턴스가 생성될 때 호출
  - 클래스이름::클래스이름(매개변수 가능)
- 소멸자 (Destructor) 함수
  - 클래스를 정리하며 인스턴스가 소멸될 때 호출
  - 클래스이름::~클래스이름(매개변수 없음)
- 주의사항
  - 생성자와 소멸자는 반환값 없음

Class Constructor & Destructor

```
#include "Point.h"

void main()
{
    // 인스턴스 생성
    Point myPosition, yourPosition;

    // 변수 값 초기화
    myPosition.SetPosition(10, 30);
    yourPosition.SetPosition(50, 30);

    // 좌표 변경 (점의 위치 이동)
    myPosition.Move(20, 50);
    yourPosition.Move(30, 40);

    // 현재 좌표 출력
    myPosition.Show( );
    yourPosition.Show( );
}

class Point
{
public:
    Point();
    virtual ~Point();

    // 멤버 함수
    void SetPosition(int nX, int nY);
    void Move(int nX, int nY);
    virtual void Show();

    // 멤버 변수
    int m_nX, m_nY;
};
```

```
#include <iostream.h>
#include "Point.h"

Point::Point()
{
    m_nX = 0;
    m_nY = 0;

    cout << "생성자 호출됨₩n";
}

Point::~Point()
{
    cout << "소멸자 호출됨₩n";
}

void Point::SetPosition(int nX, int nY)
{
    m_nX = nX;
    m_nY = nY;
}

void Point::Move(int nX, int nY)
{
    m_nX += nX;
    m_nY += nY;
}


void Point::Show()
{
    cout << "X=" << m_nX << ", Y=" << m_nY << "₩n";
}
```

Overloading Constructor

- 생성자는 같은 이름으로 다른 유형, 또는 다른 매개변수 (parameter)를 가지는 여러 개의 함수들로 오버로딩될 수 있다.

```
class Rectangle {
public:
    Rectangle(): width(1.0), height(1.0) { };          // default
    Rectangle(float w, float h): width(w), height(h) { };      // conversion
    Rectangle(const Rectangle& );                      // copy
    void Set(float w, float h) { width = w; height = h; }
    float Area() { return width*height; }
private:
    float width;    float height;
};
Rectangle rect(3.0, 4.0);
Rectangle * ptrRect = new Rectangle;
ptrRect->Set(7.0., 8.0);
cout << “ptrRect area=“ << ptrRect->Area() << endl;
```


Copy Constructor

- 복사 생성자(copy constructor)
  - 선언되는 객체와 같은 자료형의 객체를 인수로 전달하는 생성자
  - 매개변수(parameter)는 대개 const & 형으로 전달
- 디폴트 복사 생성자 (default copy constructor)
  - 복사 생성자 정의 생략 시 자동으로 삽입되는 복사 생성자
  - 매개변수로 전달되는 객체의 멤버변수를 선언되는 객체의 멤버변수로 복사

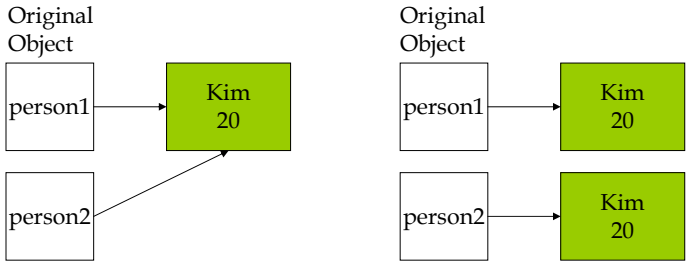
```
// 정의한 copy constructor는 default copy constructor와 같은 형태
Rectangle::Rectangle(const Rectangle& r) {
    width = r.width;
    height = r.height;
}
Rectangle rect(3.0, 4.0);
Rectangle rect2(rect); // call copy constructor
```


Copy Constructor

```
class Person {
public:
    Person();
    Person(char *, int );
    ~Person();
private:
    char * name;
    int age;
};
Person::Person() {
    name = new char[10];
    strcpy(name, “”);
    age = 0;
}
Person::Person(char * n, int a) {
    name = new char[strlen(n)+1];
    strcpy(name, n);
    age = a;
}
```

```
Person::~Person() {
    delete [] name;
}
int main()
{
    Person person1(“Kim”, 20);
    // default copy constructor 사용 시
    // shallow copy에 인한 메모리 참조 에러 발생
    // DEEP COPY copy constructor가 필요함
    Person person2(person1);
    Person person3 = person1;
    Person person4;
    // default assignment operator 사용 시
    // shallow copy에 인한 메모리 참조 에러 발생
    // DEEP COPY operator=가 필요함
    person4 = person1;
    return 0;
}
```

Shallow Copy vs. Deep Copy



Shallow Copy **Deep Copy**

- person2, person3 선언 시 복사 생성자에서 메모리 할당없이 객체 person1의 name의 포인터만 복사 (shallow copy)
- person2, person3 소멸자 호출하여 name 포인터가 가리키는 메모리 해제 (소멸자 호출순서는 생성자와 반대순서) 후 person1 소멸자 호출하여 name 포인터가 가리키는 메모리 해제 시 실행에러
- 메모리를 할당하는 깊은 복사(deep copy)를 하는 복사 생성자를 작성해야 함
- person4 = person1의 경우 default는 얕은 복사를 함. 따라서 깊은 복사를 하는 operator=를 추가 작성해야 함

Copy Constructor & Assignment Operator

```
class Person {
public:
    Person(const Person&);                      // DEEP COPY copy constructor
    Person& operator=(const Person&);           // DEEP COPY operator=
…
};
Person::Person(const Person& p) {
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name);
    age = p.age;
}
Person& Person::operator=(const Person& p) {
    if (this == &p) return *this;               // if lhs is the same as rhs, return *this
    delete [ ] name;                            // delete storage for lhs
    name = new char[strlen(p.name)+1];          // create new storage for rhs
    strcpy(name, p.name);                       // copy rhs "stuff" to lhs
    age = p.age;
    return *this;                               // return *this
}
```


Member Variables

- 변수 (Variables)
  - 전역변수: 프로그램이 시작될 때 생성되고 끝날 때 소멸
  - 정적변수: 프로그램이 시작될 때 생성되고 끝날 때 소멸
  - 지역변수: 함수가 시작될 때 생성되고 끝날 때 소멸
  - 자동변수: new할 때 생성되고 delete할 때 소멸
- 예제
  - 클래스 CTest의 선언 (Test.h)
    - 멤버 변수: m_name
    - 멤버 함수: 생성자(char *msg), 소멸자
  - 클래스 CTest의 구현 (Test.cpp)
    - 생성자(char *msg): msg("정적 호출")를 m_name에 저장 후 출력
    - 소멸자(): m_name을 출력
  - 클래스 CTest의 사용: (main.cpp)
    - 지역적, 정적, 동적으로 CTest을 3번 호출

출력결과

```
생성자: 지역적 호출
생성자: 정적 호출
생성자: 동적 호출
파괴자: 동적 호출
파괴자: 지역적 호출
파괴자: 정적 호출
```


Static Member Variables

- 정적 멤버 변수 (Static Member Variables)
  - 한 클래스에서 하나만 생성되어 모든 인스턴스가 공유하는 변수

```
class Point {
        int x; int y;
public:
        static int count;    // 인스턴스의 개수를 세기 위한 변수
        Point() { count++; }        // 생성자
        ~Point() { count--; }       // 소멸자
};

// static(정적) 변수는 클래스 선언부 밖에서 별도로 선언필요, 초기화 작업
int Point::count = 100;
```

Const Member Variables/Functions

- const 변수
  - 초기화는 할 수 있지만 변경할 수 없는 상수

```
const double pi = 3.141592;
pi=10;  // 에러
```

- const 함수
  - 변수의 값을 변경할 수 없는 읽기 전용 함수

```
class Count {
public:
    int GetCount() const;
    void SetCount( int nCount );
private:
    int m_nCount;
};

int Count::GetCount() const
{
    return m_nCount;    // 읽기 전용 함수
}

void Count::SetCount( int nCount )
{
    m_nCount = nCount;  // 멤버 변수의 값 변경
}
```


This Pointer

- 클래스의 this 포인터란
  - 현재 클래스의 인스턴스가 위치한 메모리의 주소
  - 다른 클래스에 자기자신을 매개변수로 넘겨줄 때 사용

```
#include <iostream.h>

class Where
{
public:
    int data;
    void PrintPointer( );
};

void Where::PrintPointer( )
{
    cout << "오브젝트의 주소는 " << this << " 번지입니다.₩n";
}

void main( )
{
    Where a, b, c;

    a.PrintPointer( );
    b.PrintPointer( );
    c.PrintPointer( );
}
```

출력결과

```
오브젝트의 주소는 0x0012FEE0 번지입니다.
오브젝트의 주소는 0x0012FEDC 번지입니다.
오브젝트의 주소는 0x0012FED8 번지입니다.
```

Polymorphism

- 다형성(Polymorphism)
  - "One interface, multiple implementation"
  - Compile time 다형성
    - 함수 Overloading
    - 연산자 Overloading
  - Run time 다형성
    - 가상함수(virtual)를 이용한 함수의 Overriding
- NOTE:
  - 다형성을 가진 기본 클래스(base class)의 소멸자는 항상 virtual로 선언한다 - 파생 클래스(derived class)의 소멸자도 호출하도록 하여 memory leak이나 이상한 현상을 막을 수 있도록 한다.

Function Overloading

- 함수 오버로딩 (Function Overloading)
  - 함수의 이름은 같지만 함수의 매개변수나 반환 값이 다르게 정의
  - 호출 시 함수의 매개변수 개수나 데이터 타입에 따라 구별되어 처리

```
class Overload {
public:
        int Max(int a, int b) {
                if (a > b) return a;
                else  return b;
        }
        double Max(double a, double b) {
                if (a > b) return a;
                else  return b;
        }
};
void main() {
        Overload o;
        int x;      double y;
        x = o.Max(10, 50);
        y = o.Max(10.6, 50.3);
}
```

Operator Overloading

- 연산자 오버로딩 (Operator Overloading)
  - 연산자를 함수처럼 재정의하여 사용
- 연산자 오버로딩 연산자 종류
  - 단항 연산자 : ++, --
  - 이항 연산자 : +, -, *, / , %, ^, &, |, ~, !, , , =, <, >, <=, >=, <<, >>, ==, !=, ->, += 등
- 연산자 오버로딩
  - Operator 키워드 뒤에 연산자를 넣은 함수이름으로 구현

Operator Overloading

- 연산자 오버로딩 (Operator Overloading)

```
class Point
{
public:
    void Increase();
    ...
};

void Point::Increase()
{
    m_nX++;
    m_nY++;
}

void main()
{
    Point p;
    p.SetPosition(10, 10);
    p.Increase();
    p.Show();
}
```

```
class Point
{
public:
    void operator++();
    ...
};

void Point::operator++()
{
    m_nX++;
    m_nY++;
}

void main()
{
    Point p;
    p.SetPosition(10, 10);
    ++p;
    p.Show();
}
```

Point q=++p; // 오류
반환값이 없으므로 대입 불가

```
class Point
{
public:
    Point(int nX, int nY);
    Point operator++();
    ...
};

Point::Point(int nX, int nY)
{
    m_nX = nX;
    m_nY = nY;
}

Point Point::operator++()
{
    return Point(++m_nX, ++m_nY);
}

void main()
{
    Point p;
    p.SetPosition(10, 10);
    ++p;
    p.Show();
}
```


Operator Overloading

```
class Vector {
    int x, y;
public:
    Vector(): x(0), y(0) { }                // default constructor
    Vector(int a, int b): x(a), y(b) { };
    ~Vector();

     // operator overloading
    Vector& operator= (const Vector&);          // assignment operator
    Vector& operator+= (const Vector&);
    Vector& operator-= (const Vector&);
    Vector operator+ (const Vector&);           // + operator
    Vector operator- (const Vector&);
    Vector operator++ ();                       // increment
    Vector operator-- ();
    bool operator==(const Vector&) const;       // equality
    // operator<<
    friend ostream& operator<<(ostream&, const Vector&);
};
```


Operator Overloading

```
Vector& Vector::operator+=(const Vector& v) {
    x += v.x;
    y += v.y;
    return *this;
}
Vector Vector::operator+(const Vector& v) {
    Vector t;
    t.x = x + v.x;
    t.y = y + v.y;
    return t;
}
Vector& Vector::operator++() {
    x++;
    y++;
    return *this;
}
bool Vector::operator==(const Vector& v) {
    return (x == v.x && y == v.y);
}
```

Operator Overloading

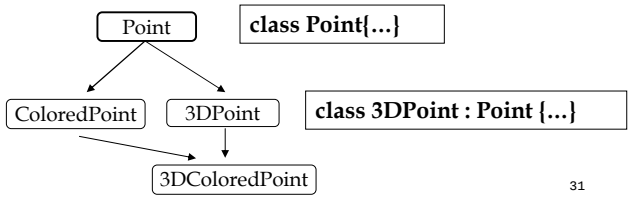
```
ostream& operator << (ostream& out, const Vector& v) {
    out << "(" << v.x << ", " << v.y << ")" << endl;
    return out;
}
int main()
{
Vector a(3, 1);
Vector b(1, 2);
Vector c = a + b;  // can use default copy constructor or assignment operator
Vector d;
d += a;
d++;
cout << "Vector c=" << c;
cout << "Vector d=" << d;
if (c==d)
    cout << "c and d is the same" << endl;
}
```


Friends

- Overloading operator<< 함수
  - 클래스의 멤버정보를 쉽게 프린트하기 위해 사용
  - operator<<는 반드시 friend function이어야 한다 - operator<<는 ostream 클래스의 멤버이므로 개별 클래스의 멤버함수가 될 수 없다.
- Friend
  - 클래스의 private 데이터와 함수에 접근할 수 있다.
  - Friend class나 function일 수 있다.
  - NOTE: friend는 상속(inherit)할 수 없으며, 가상함수 (virtual)로 지정할 수 없다.
    - Q: 파생클래스 (Derived class)에서 operator<< 함수에 기반 클래스 (Base class)의 private 멤버를 사용하려면?
    - A: 기반클래스의 operator<< 함수를 부른 후, 파생클래스의 추가적인 멤버 데이터를 부른다.

Inheritance

- 상속 (Inheritance)
  - 이미 만들어진 기반 클래스에 구현된 모든 특징을 그대로 계승받아 새로운 파생클래스를 생성
    - 상위/기반 클래스 (Super class, Base class)
    - 하위/파생 클래스 (Subclass, Derived class, Extended class)
- 상속을 통한 오브젝트의 계층구조적 구현
  - 비슷한 속성을 여러 번 중복 구현하는 일이 없어짐
  - 새로운 클래스를 추가하기가 쉬워짐



Inheritance

- 기반 클래스 (Base Class)와 파생 클래스 (Derived Class)

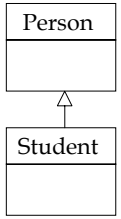
```
class BaseClass{
public:
    // 멤버변수
    int BaseVariable1;
    int BaseVariable2;

    //멤버 함수
    BaseClass();            // 생성자 함수
    virtual ~BaseClass();   // 소멸자 함수
    void BaseFunction1();
    void BaseFunction2();
};

class DerivedClass : public BaseClass
{
public:
    // 멤버변수
    int  DerivedVariable;
    void DerivedFunction();
};
```

  - Derived Class에서 사용 가능한 멤버변수는 3개 – 즉, BaseVariable1, BaseVariable2, DerivedVariable
  - Derived Class에서 사용 가능한 멤버함수는 3개 – 즉, BaseFunction1, BaseFunction2, DerivedFunction

Example

- 클래스 Person
  - 멤버변수: name(이름), age(나이)
  - 멤버함수
    - Person(char *n, int a): 이름, 나이 저장
    - Show(): "이름, 나이"를 출력
- 클래스 Student는 Person을 상속
  - 멤버변수: kor(국어), eng(영어), math(수학), avg(평균)
  - 멤버함수
    - Student(char *n, int a, int k, int e, int m): 평균을 계산하고, 이름, 나이, 국어성적, 영어성적, 수학성적, 평균성적을 저장
    - Show(): 기반 클래스 Person의 Show()를 호출하고, 국어성적, 영어성적, 수학성적, 평균성적을 출력

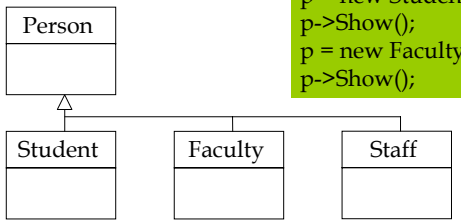
```
출력결과
A의 이름은? 이정진
A의 나이는? 29
B의 이름은? 수애
B의 나이는? 27
B의 국어성적은? 89
B의 영어성적은? 75
B의 수학성적은? 67
-----------------------
   학생     국어 영어 수학 평균
이정진 29세
수애 27세  89  75  67  77
```



Example

- 클래스 Faculty는 Person을 상속
  - 멤버변수: course(과목)
  - 멤버함수
    - Faculty(char *n, int a, char *c): 이름, 나이, 과목을 저장
    - Show(): 기반 클래스 Person의 Show()를 호출하고, 과목을 출력

```
Person * p;
p = new Student("Lee", 20, 3, 4, 5);
p->Show();
p = new Faculty("Park", 40, "HCI2");
p->Show();
```



Overriding

- 재정의 (Overriding)
  - 기반클래스의 마음에 안드는 함수만 고쳐서 파생클래스에서 재정의해서 사용

```
class Point
{
public:
    int m_nX, m_nY; // XY 좌표
    void Show();
    ...
};
```

```
void Point::Show()
{
    cout << "X=" << m_nX << ", Y=" << m_nY << "₩n";
}
```

```
class Point3D
{
public:
    int m_nZ;   // Z 좌표
    void Show();// Show 함수의 재정의
    ...
};
```

```
void Point3D::Show()
{
    cout << ", Z=" << m_nZ << "X=" << m_nX << ", Y=" << m_nY << "₩n";
}
```

```
void Point3D::Show()
{
    cout << ", Z=" << m_nZ; // 새로운 기능 추가
    Point::Show();          // 기반 클래스의 Show()함수를 호출
}
```


Overriding

- 오버로딩 (Overloading)과 재정의 (Overriding)
  - Overloading: 동일한 함수명에 매개변수가 다른 함수를 둘 이상 정의
    - int Max(int a, int b);
    - double Max(double a, double b);
  - Overriding: 동일한 함수명에 동일한 매개변수를 둘 이상 정의
    - Point Class: void Show();
    - Point3D Class: void Show();
      - 단, Point3D 클래스는 Point 클래스를 상속받은 파생 클래스

Virtual Function

- 바인딩 (Binding)
  - 함수를 호출하는 부분에 함수가 위치한 메모리번지를 연결시켜주는 작업
  - 정적 바인딩 (Static Binding)
    - 컴파일할 때 호출될 함수가 결정
  - 동적 바인딩 (Dynamic Binding)
    - 실행할 때 호출될 함수가 결정
- 가상함수 (Virtual Function)
  - 함수를 선언할 때 virtual 키워드를 붙여주면 가상함수는 동적 바인딩을 사용
  - 가상함수 이외의 모든 함수는 정적 바인딩으로 처리
- 동적 바인딩의 효과
  - 파생된 클래스의 오버라이딩된 함수가 제대로 호출되려면 반드시 가상함수를 오버라이딩하여 동적 바인딩이 가능하도록 처리

Virtual Function

- 가상함수의 필요성

```
class Point
{
public:
    void Show();
}

void Point::Show()
{
    cout << "Point 클래스의 Show 함수 호출";
}
```

```
class Point3D : public Point
{
public:
    void Show();    // 함수 재정의
}

void Point3D::Show()
{
    cout << "Point3D 클래스의 Show 함수 호출";
}
```

```
void main()
{
    Point point;
    Point3D point3d;

    point.Show();
    point3d.Show();
}
```

출력결과

```
Point 클래스의 Show 함수 호출
Point3D 클래스의 Show 함수 호출
```

```
void main()
{
    Point *p;
    Point point;
    Point3D point3d;

    p=&point;
    p->Show();
    p=&point3d;
    p->Show();
}
```

출력결과

```
Point 클래스의 Show 함수 호출
Point 클래스의 Show 함수 호출
```


Virtual Function

- 가상함수의 필요성

```
class Point
{
public:
    virtual void Show();
}

void Point::Show()
{
    cout << "Point 클래스의 Show 함수 호출";
}
```

```
class Point3D : public Point
{
public:
    void Show();    // 함수 재정의
}

void Point3D::Show()
{
    cout << "Point3D 클래스의 Show 함수 호출";
}
```

```
void main()
{
    Point point;
    Point3D point3d;

    point.Show();
    point3d.Show();
}
```

출력결과

```
Point 클래스의 Show 함수 호출
Point3D 클래스의 Show 함수 호출
```

```
void main()
{
    Point *p;
    Point point;
    Point3D point3d;

    p=&point;
    p->Show();
    p=&point3d;
    p->Show();
}
```

출력결과

```
Point 클래스의 Show 함수 호출
Point3D 클래스의 Show 함수 호출
```


Pure Virtual Function

- Pure virtual function
  - 파생 클래스는 interface만 상속
  - 모든 파생 클래스는 반드시 pure virtual function (아래 예제의 draw 함수)을 가져야 함

```
class Shape {
public:
    virtual void draw() const = 0;      // purely virtual function
    virtual void error(const string&) const;
    int objectID() const;
};
class Rectangle : public Shape {};
class Circle : public Shape {};
Shape * s = new Shape;          // error: Shape is abstract base class (ABC)
Shape * r = new Rectangle;      // ok
r->draw();                      // Rectangle의 draw()를 호출
```

Polymorphic List

```
// DrawAllShapes는 각 Shape(즉, Rectangle 또는 Circle)의 draw()를 호출
void DrawAllShapes(const vector<Shape*> s)
{
    vector<Shape*> itr = s.begin();
    while (itr != s.end()) {
        (*itr)->draw();
    }
}

vector<Shape*> aList;
aList.push_back(new Rectangle(…));
aList.push_back(new Circle(…));
aList.push_back(new Rectangle(…));
aList.push_back(new Triangle(…));    // 새로운 Shape을 쉽게 추가할 수 있음
DrawAllShapes(aList);
```


Inline Function

- inline 함수란
  - 함수를 호출하는 위치마다 코드가 통째로 복사
  - 실행파일의 크기는 증가하지만 수행속도는 빨라짐
  - 빈번히 **호출되며** 크기가 매우 작은 함수에 적합
- inline 함수 만드는 방법

묵시적인 방법	명시적인 방법

`class Point`
`{`
`public:`
`  int m_nX, m_nY;`
`  void SetPosition(int x,int y ) { m_nX=x; m_nY=y; };`
`};`
 	
함수의 선언과 구현을 같이 해주면 자동으로 inline 함수가 생성
	`class Point`
	`{`
	`public:`
	`  int m_nX, m_nY;`
	`  void SetPosition(int x,int y );`
	`};`
	
	`inline void Point::SetPosition(int x,int y )`
	`{`
	`  m_nX=x;`
	`  m_nY=y;`
	`}`

메모리 관리

- C++ 프로그램에서 다룰 수 있는 메모리의 종류
  - 정적 (static)
    - 프로그램의 시작과 동시에 할당되고 프로그램의 종료와 함께 해제됨
    - Static을 사용하거나 파일 범위에서 선언된 변수는 정적이 됨
  - 자동 (automatic)
    - 인스턴스가 선언된 지역에 있는 블록 내에서 유효한 것으로 선언된 지점에서 생성되고 블록을 벗어나는 순간 해제됨. Auto를 사용하거나 블록 범위에서 선언된 변수는 자동이 됨
    - 함수에 매개변수를 전달할 때 복사본을 전달
  - 동적 (dynamic)
    - 블록이나 프로그램과 관계없이 사용자가 지정하는 순간 할당되고 해제됨
    - New, delete 연산자를 이용해서 할당 또는 해제됨
    - 동적할당은 효과적으로 메모리를 사용할 수 있으나 메모리에 대한 반환에 유의해야 함 (메모리 누수)

Class Template

- 템플릿이란
  - 데이터형만 다르고 형태가 같은 클래스를 여러 번 찍어낼 때 사용

```
#include <iostream.h>

template <class type> class Point
{
public:
    // 멤버 함수
    void SetPosition(type nX, type nY);
    void Move(type nX, type nY);
    void Show();

protected:
    // 멤버 변수
    type m_nX, m_nY;
};
```

```
template <class type>
void Point<type>::SetPosition(type nX, typ
{
    m_nX = nX;
    m_nY = nY;
}

template <class type>
void Point<type>::Move(type nX, type nY)
{
    m_nX += nX;
    m_nY += nY;
}

template <class type>
void Point<type>::Show()
{
    cout << "X=" << m_nX << ", Y=" << m_nY << "₩n";
}
```

```
#include "Point.h"

void main()
{
    // 인스턴스 생성
    Point <double> dPosition;
    Point <int> nPosition;

    // 변수 값 초기화
    dPosition.SetPosition(10.45, 30.52);
    nPosition.SetPosition(50, 30);

    // 현재 좌표 출력
    dPosition.Show();
    nPosition.Show();
}
```

Exceptions

- C++의 예외 (Exception)
  - 예외란 프로그램이 비정상적으로 종료될 수 있는 상황을 의미
    - 배열을 사용할 때 유효범위 밖의 인덱스에 접근하는 경우
    - 동적 메모리를 할당하려고 했는데 메모리 할당이 실패하는 경우
    - NULL 포인터에 데이터를 저장하는 경우
    - 0로 나누기를 수행하는 경우
    - 사용자가 잘못된 값을 입력해서 프로그램이 올바르게 수행할 수 없는 경우
- C++의 예외처리 구문인 try, catch, throw 이용
  - try는 예외가 발생하는지 감시하는 범위를 지정
  - catch 블록은 예외를 처리
  - throw는 예외를 던지는 (raise) 문장으로 예외 발생 조건일 때 throw 값의 형식으로 사용

Exception Handling

- try 블록 내에서 발생하는 예외는 모두 catch 블록에서 한꺼번에 처리될 수 있다.
- 생성자처럼 리턴 값을 갖지 않는 함수의 경우나 [] 연산자 함수처럼 연산자 오버로딩에 의해서 리턴 값으로 함수의 성공 실패 여부를 리턴할 수 없는 경우에서 C++의 try, catch, throw 를 이용하면 예외처리가 가능하다.
- 예외처리의 규칙

```
try {
    …
    if (예외조건)
        throw 예외 객체;
}
catch (예외객체) {
    예외처리 …
}
```


Exception Handling

```
void fun(int number) {
    if (number%2 == 0) throw "error";
    cout << "This is fun" << endl;
}
int main() {
    try {
        fun(2);  // 예외경우 발생
        fun(3);
    }
    catch (char * msg) {
        cout << msg << endl;
        return 1;
    }
    return 0;
}
```


Exception Handling

```
int main() {
    int a, b;
    try {
        cout << "Enter two positive number:";
        cin >> a;
        if (a<0) throw a;
        cin >> b;
        if (b==0) throw "cannot divided by 0";
        cout << " the result is " << (float) a/b << endl;
    }
    catch (int a) {
        cout << a  << " is a negative number" << endl;
        return 1;
    }
    catch (char * msg) {
        cout << msg << endl;
        return 1;
    }
    return 0;
}
```