

6장 복사 생성자

- ▣ 객체의 생성과 대입
- ▣ 객체의 값에 의한 전달
- ▣ 복사 생성자
- ▣ 디폴트 복사 생성자
- ▣ 복사 생성자의 재정의
- ▣ 객체의 값에 의한 반환
- ▣ 임시 객체

1. 객체의 생성과 대입

✦ int형 변수 : 선언과 동시에 초기화하는 방법 (C++)

- `int a = 3;`
- `int a(3);` // 기본 타입 역시 클래스와 같이 처리 가능

✦ 객체의 생성 (복습)

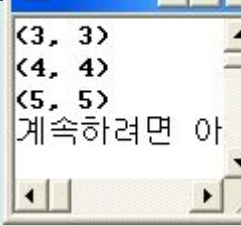
```
class CPoint {
private :
    int x, y;

public :
    CPoint(int a) : x(a), y(a) { }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

int main(void)
{
    CPoint P1(3);
    CPoint P2 = CPoint(4);
    CPoint P3 = 5;

    P1.Print();
    P2.Print();
    P3.Print();

    return 0;
}
```



일반적 방법

객체 배열 생성 시 주로 사용

CPoint(5)로 형변환 후 초기화

1. 객체의 생성과 대입

✦ 복사 생성과 대입

```
class CPoint {
private :
    int x, y;

public :
    CPoint(int a) : x(a), y(a) { }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

int main(void)
{
    CPoint P1(3);        // 객체 생성, P1 : (3, 3)
    CPoint P2(4);        // 객체 생성, P2 : (4, 4)
    CPoint P3 = P2;      // 복사 생성, P3 : (4, 4)
    CPoint P4(P2);       // 복사 생성, P4 : (4, 4)

    P1 = P2;             // 객체 대입, P1 : (4, 4)

    P1.Print();
    P2.Print();
    P3.Print();
    P4.Print();

    return 0;
}
```

객체 생성과 객체 대입을
구별하고
객체 생성 중에서도
일반 생성과 복사 생성을
구별하라.

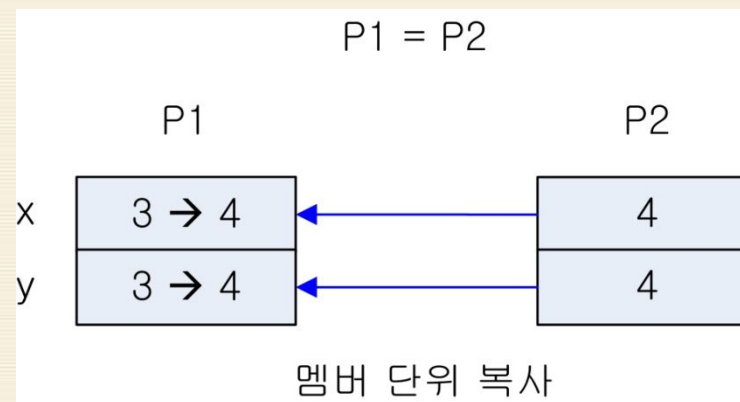
1. 객체의 생성과 대입

▣ 객체의 복사 생성과 대입

- 복사 생성 : 생성자(그 중에서 복사 생성자)가 동작함
- 대입 : 대입 연산자가 동작함

▣ 복사 생성과 대입 연산의 디폴트 동작

- 멤버 단위 복사!
- 디폴트 동작 방식은 동일



- 멤버 단위 복사만으로 충분한가? → 다음 페이지 예제

1. 객체의 생성과 대입

✦ 예 : 문자열을 다루기 위한 CString 클래스 구현

```
#include <iostream>
#include <cstring> // strlen, strcpy 함수
using namespace std;

class CString {
private :
    int len;          // 문자열의 길이
    char *str;        // 문자열 포인터

public :
    CString(char *s = "Unknown") {
        len = strlen(s);
        str = new char[len + 1];
        strcpy(str, s);
    }
    ~CString() { delete [] str; }
    void Print() { cout << str << endl; }
};
```

str3 = str1 + str2; // 가능하려면?
→ 연산자 오버로딩 (7장)



언제, 어디서 에러가 발생하는 것일까?

```
int main(void)
{
    CString str1 = "C++ Programming";
    CString str2 = str1; // 복사 생성
    CString str3;

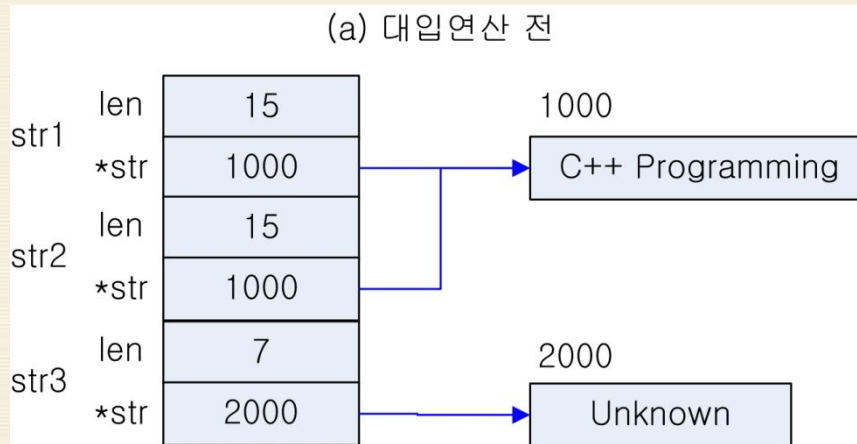
    str3 = str1;          // 대입 연산

    str1.Print();
    str2.Print();
    str3.Print();

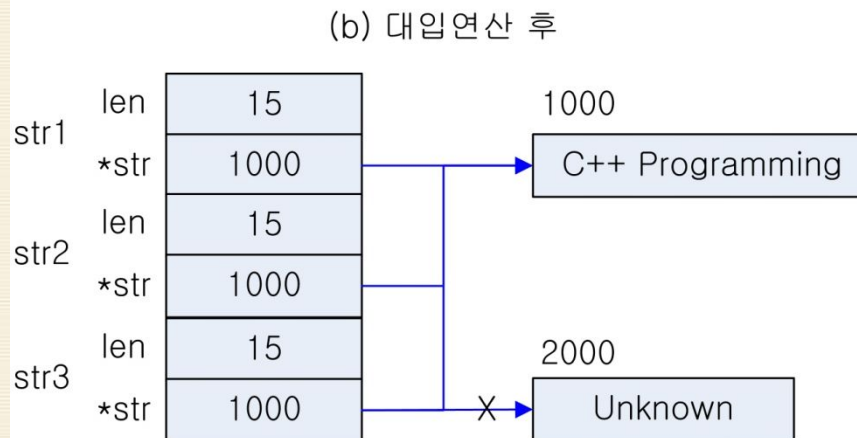
    return 0;
}
```

1. 객체의 생성과 대입

⌘ CString 클래스의 문제점 분석



← **CString str2 = str1;**의 수행 결과



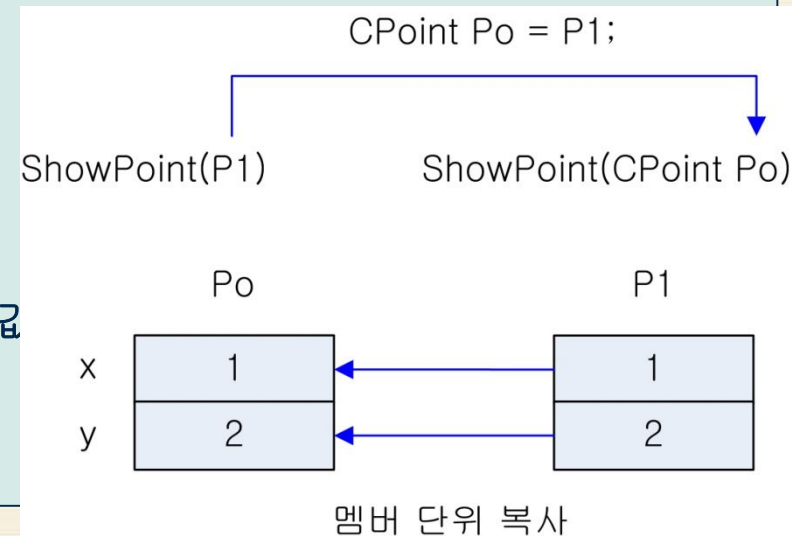
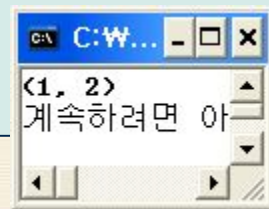
← **str3 = str1;**의 수행 결과

모든 객체의 *str이 동일한 메모리 주소를 가리킴
→ 함수 종료 시 각 객체에 대한 소멸자가 수행된다면!
동일한 주소에 대한 delete 시도
→ 에러 발생!
→ 복사 생성과 대입 연산 모두 문제!

2. 객체의 값에 의한 전달

■ 객체의 값에 의한 전달 : 복사 생성 발생

```
class CPoint {  
private :  
    int x, y;  
  
public :  
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }  
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }  
};  
  
void ShowPoint(CPoint Po)    // 값에 의한 객체 전달  
{  
    Po.Print();  
}  
  
int main(void)  
{  
    CPoint P1(1, 2);  
    ShowPoint(P1);           // 값  
  
    return 0;  
}
```



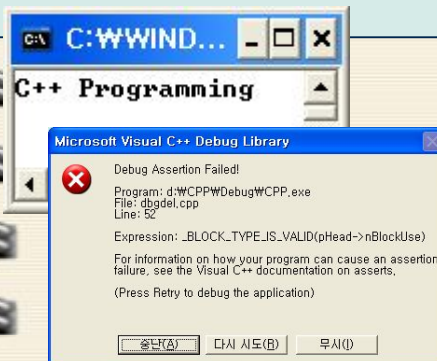
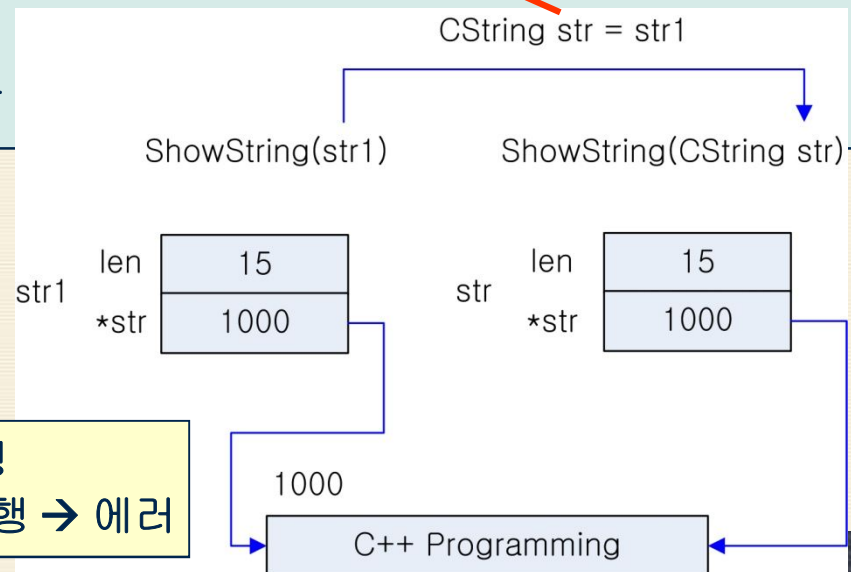
2. 객체의 값에 의한 전달

예 : CString 클래스 객체의 값에 의한 전달 : 문제점은?

```
class CString {  
private :  
    int len;           // 문자열의 길이  
    char *str;         // 문자열 포인터  
  
public :  
    CString(char *s = "Unknown") {  
        len = strlen(s);  
        str = new char[len + 1];  
        strcpy(str, s);  
    }  
    ~CString() { delete [] str; }  
    void Print() { cout << str << endl; }  
};
```

```
void ShowString(CString str) // 값에 의한 전달  
{  
    str.Print();  
}
```

```
int main(void)  
{  
    CString str1 = "C++ Programming";  
    ShowString(str1);  
}
```

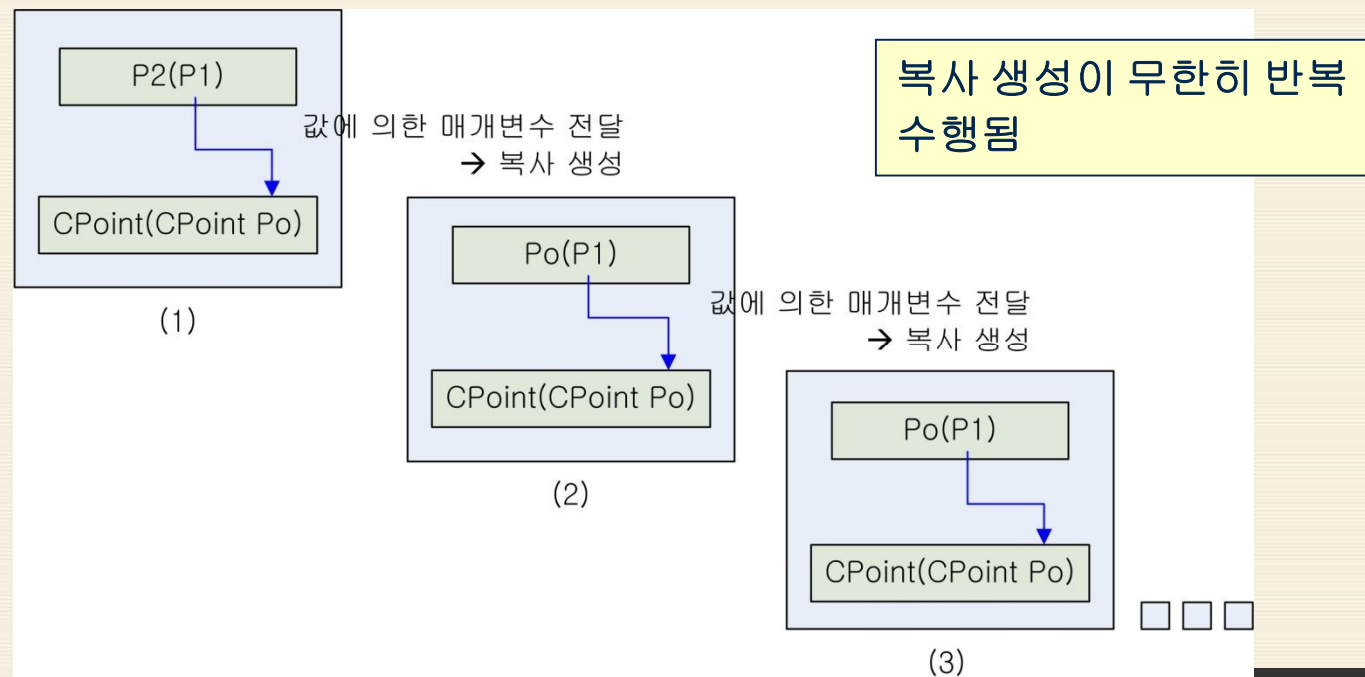


1. str의 소멸자 수행
2. str1의 소멸자 수행 → 에러

3. 복사 생성자

■ 복사 생성자

- 복사 생성 시 호출되는 특수한 생성자
 - 복사 생성자의 모양 유추
 - 일반 생성자 : `CPoint P1(3, 4);` → `CPoint(int a, int b);`
 - 복사 생성자 : `CPoint P2(P1);` → `CPoint(CPoint Po);` // ok???
- ✓ 문제점 : 복사 생성을 위해 P1을 매개변수로 전달 시 또 다시 복사 생성 발생



3. 복사 생성자

복사 생성자의 모양

- `CPoint(CPoint &Po);` // 참조에 의한 전달!
- `CPoint(const CPoint &Po);` // 실매개변수에 대한 변경 방지

```
class CPoint {  
private :  
    int x, y;  
  
public :  
    CPoint(const CPoint &Po) { x = Po.x; y = Po.y; } // 복사 생성자  
    // 일반 생성자  
    CPoint(const CPoint &Po, int a) { x = Po.x * a; y = Po.y * a; }  
    CPoint(int a = 0, int b = 0) : x(a), y(b) { } // 일반 생성자  
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }  
};
```

```
int main(void)  
{
```

```
    CPoint P1(1, 2);
```

```
    CPoint P2(P1);
```

```
    CPoint P3(P1, 3);
```

```
    P1.Print();
```

```
    P2.Print();
```

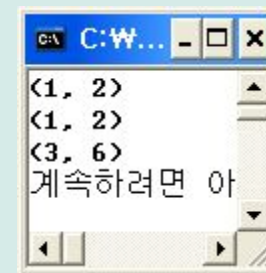
```
    P3.Print();
```

```
    return 0;
```

```
}
```

복사 생성

일반 생성



4. 디폴트 복사 생성자

자동으로 생성되는 멤버 함수

- 디폴트 생성자 : 4.7절
- 디폴트 소멸자 : 4.7절
- 디폴트 복사 생성자 : 본 절
- 디폴트 대입 연산자 : 7.10절

디폴트 복사 생성자

- 멤버 단위 복사
- 예 : CPoint

```
CPoint(const CPoint &Po)
{
    x = Po.x;
    y = Po.y;
}
```

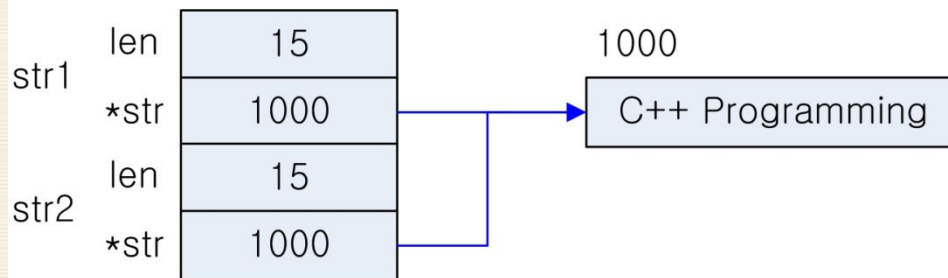
- 복사 생성자를 명시적으로 만드는 경우
 - 디폴트 복사 생성자 사라짐
 - 디폴트 생성자 역시 사라짐

5. 복사 생성자의 재정의

다음 코드에 대한 올바른 동작?

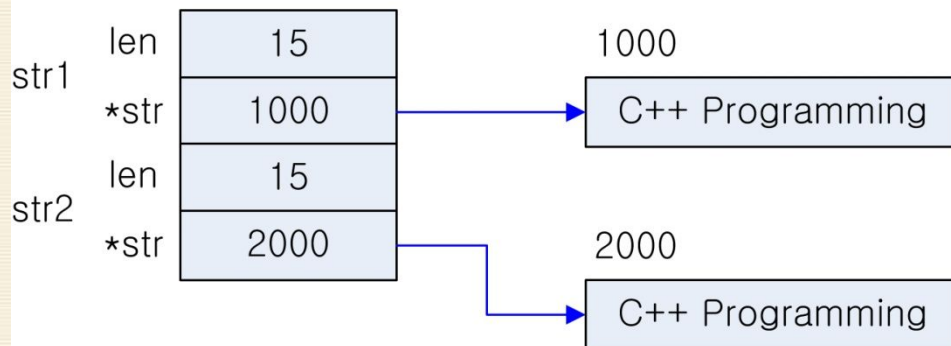
```
CString str1 = "C++ Programming";  
CString str2 = str1;
```

(1) 디폴트 복사 생성자



디폴트 복사 생성자를 사용하는 경우
소멸자 호출 시
에러 발생

(2) 새로운 복사 생성자



원하는 동작
→ 이에 맞게 복사 생성자
재정의!

5. 복사 생성자의 재정의

CString 클래스에 대한 복사 생성자 재정의

```
class CString {
private :
    int len;
    char *str;

public :
    CString(const CString &string) {
        len = string.len;
        str = new char[len + 1];
        strcpy(str, string.str);
    }
    CString(char *s = "Unknown") {
        len = strlen(s);
        str = new char[len + 1];
        strcpy(str, s);
    }
    ~CString() { delete [] str; }
    void Print() { cout << str << endl; }
};
```

복사 생성자

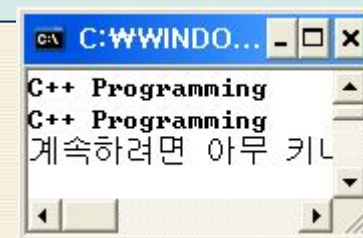
일반 생성자

```
// 값에 의한 전달, 복사 생성
void ShowString(CString str)
{
    str.Print();
}

int main(void)
{
    CString str1 = "C++ Programming";
    CString str2 = str1;    // 복사 생성

    str1.Print();
    ShowString(str2);    // 값에 의한 전달

    return 0;
}
```



6. 객체의 값에 의한 반환

■ 복사 생성자가 호출되는 경우

- 객체의 선언 및 초기화 : CPoint P2(P1);
- 객체의 값에 의한 전달 : void ShowString(CString str) { ... }
- 객체의 값에 의한 반환 : CString GetPoint() { ... return str; }

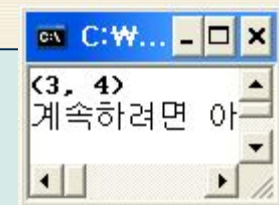
■ CPoint 객체의 값에 의한 반환 예

```
class CPoint {
private :
    int x, y;

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

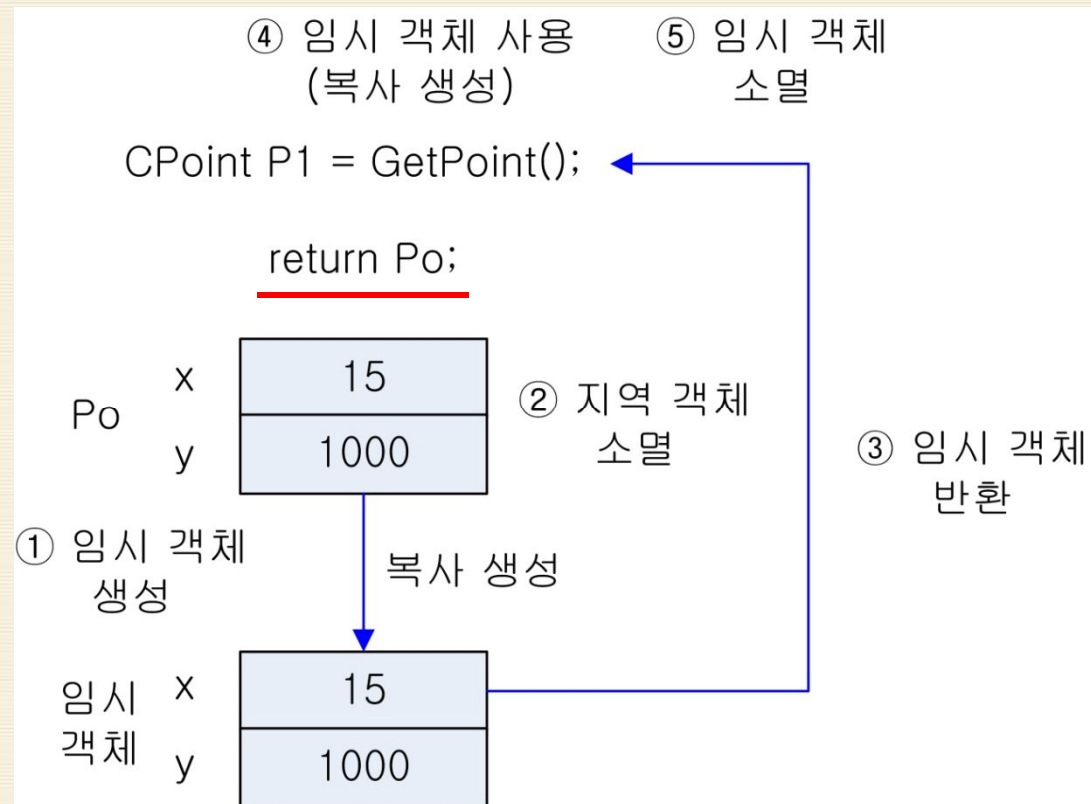
CPoint GetPoint(void)
{
    CPoint Po(3, 4);           // 지역 객체 Po 생성
    return Po;                 // 지역 객체 값 반환
}

int main(void)
{
    CPoint P1 = GetPoint();    // GetPoint 함수 호출
    P1.Print();
    return 0;
}
```



6. 객체의 값에 의한 반환

CPoint 객체의 반환 예의 동작 원리



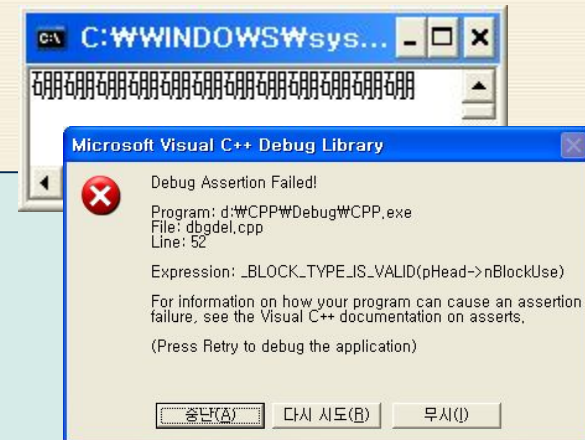
6. 객체의 값에 의한 반환

CString 객체의 반환 예

```
class CString {
private :
    int len;
    char *str;

public :
    CString(char *s = "Unknown") {
        len = strlen(s);
        str = new char[len + 1];
        strcpy(str, s);
    }
    ~CString() { delete [] str; }
    void Print() { cout << str << endl; }
};

CString GetString(void)
{
    CString str("Current String"); // 객체 생성
    return str;                    // 객체값 반환, 임시객체 생성
}
```



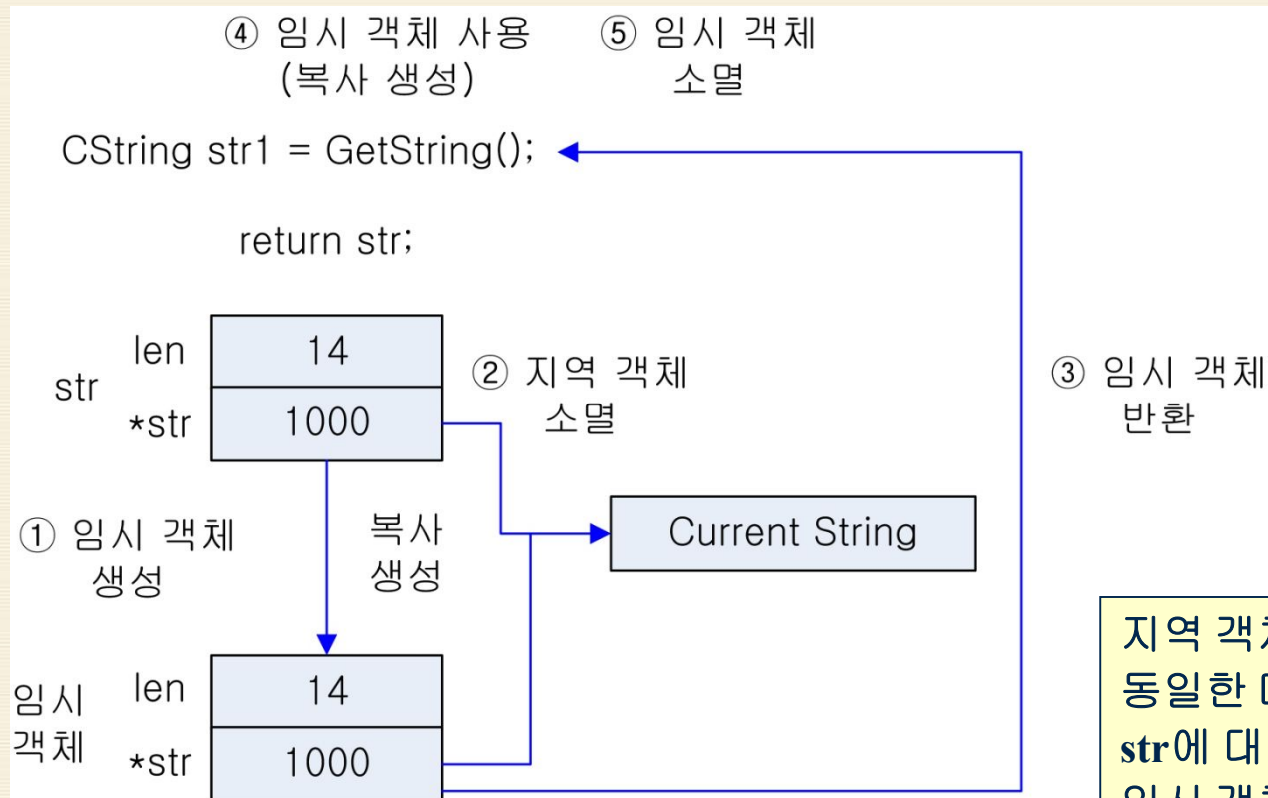
왜 에러가 발생하는 것일까?
→ 객체 반환에 따른 복사 생성

```
int main(void)
{
    CString str1 = GetString();
    str1.Print();

    return 0;
}
```

6. 객체의 값에 의한 반환

CString 객체 반환 시 에러 발생 원인



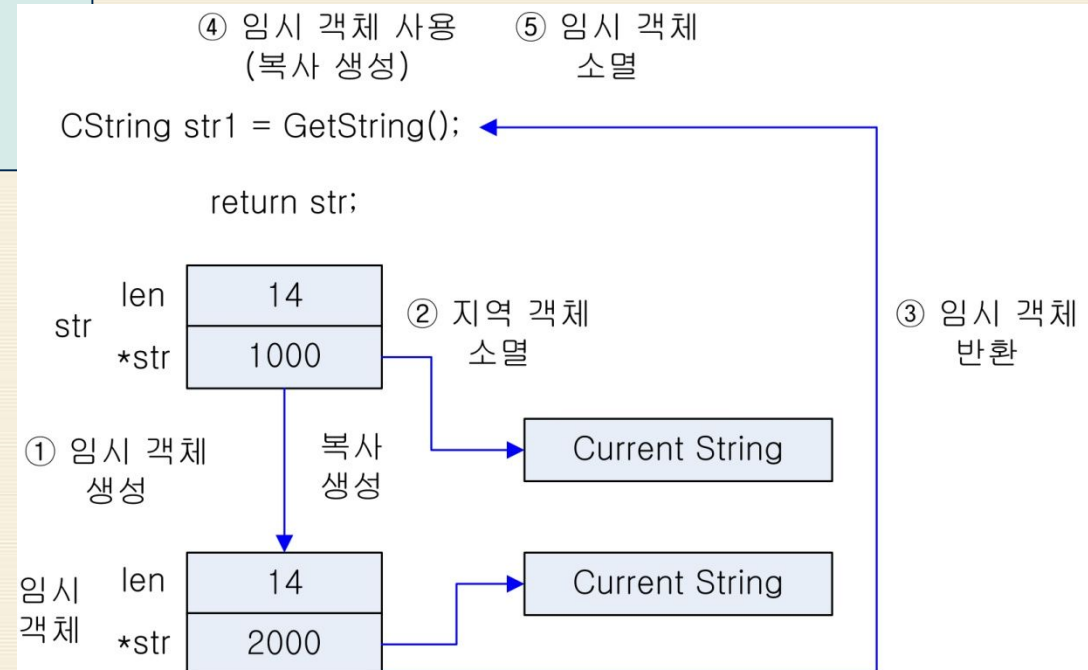
지역 객체인 `str`과 임시 객체가 동일한 메모리를 가리킴
`str`에 대한 소멸자 호출 후
임시 객체에 대한 소멸자 호출 시 에러 발생!

6. 객체의 값에 의한 반환

CString 객체 반환 예의 문제점 해결

- 복사 생성자만으로 ok!

```
CString(const CString &string) {  
    len = string.len;  
    str = new char[len + 1];  
    strcpy(str, string.str);  
}
```



7. 임시 객체

임시 객체(temporary object)의 사용 예

```
class CPoint {
private :
    int x;
    int y;

public :
    CPoint(const CPoint &Po) : x(Po.x), y(Po.y) {
        cout << "복사 생성자 : " << x << ", " << y << endl; }
    CPoint(int a = 0, int b = 0) : x(a), y(b) {
        cout << "생성자1 : " << x << ", " << y << endl; }
    CPoint(const CPoint &Po, int a, int b) {
        x = Po.x + a; y = Po.y + b;
        cout << "생성자2 : " << x << ", " << y << endl; }
    ~CPoint() { cout << "소멸자 : " << x << ", " << y << endl; }
    void Print() { cout << "(" << x << ", " << y << ")" << endl; }
};

CPoint GetPoint(CPoint Po)
{
    return CPoint(Po, 2, 2);           // 임시객체 생성 및 반환
}
```

7. 임시 객체

※ 코드 계속

```
int main(void)
{
    CPoint P1 = GetPoint(CPoint(1, 1)); // GetPoint 함수 호출
    CPoint P2 = CPoint(100, 100);      // 임시객체 생성, P2 초기화
    CPoint &P3 = CPoint(200, 200);      // 임시객체 생성, P3이 참조
    CPoint P4;                          // 일반 생성
    P4 = CPoint(300, 300);              // 임시객체 생성 및 대입

    P1.Print();
    P2.Print();
    P3.Print();
    P4.Print();
    CPoint(300, 300).Print();           // 임시객체 생성 & 멤버 함수 호출

    cout << "프로그램 종료" << endl;

    return 0;
}
```

명시적으로 임시 객체를 만드는 방법

→ **CPoint(1, 1)**

임시 객체는 왜 사라지지 않는 것일까? → 컴파일러 의존적

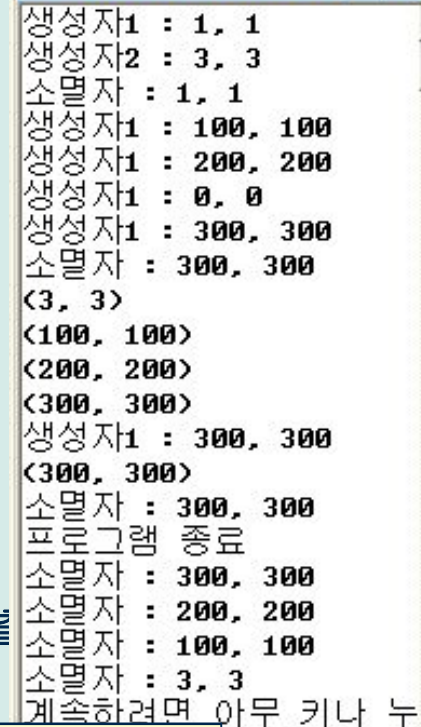
: 형식매개변수인 **Po**가 임시 객체를 그대로 사용하면 됨

: **GetPoint** 함수에서 **CPoint** 객체가 반환되는 과정에서도
똑 같은 원리가 적용됨 → **P1**은 임시객체를 그대로 사용!

사실은 앞서 **CString** 객체의 반환 예에서도
CString str1 = GetString();의 결과로
임시 객체는 소멸되지 않고 그대로
str1으로 사용됨

임시 객체의 사용 원리

- 필요한 곳에서 임시 객체 생성 가능
 - : 명시적 생성 또는 묵시적 생성
- 임시 객체의 생성 주기는 임시 객체가
필요한 기간과 일치



```
생성자1 : 1, 1
생성자2 : 3, 3
소멸자 : 1, 1
생성자1 : 100, 100
생성자1 : 200, 200
생성자1 : 0, 0
생성자1 : 300, 300
소멸자 : 300, 300
<3, 3>
<100, 100>
<200, 200>
<300, 300>
생성자1 : 300, 300
<300, 300>
소멸자 : 300, 300
프로그램 종료
소멸자 : 300, 300
소멸자 : 200, 200
소멸자 : 100, 100
소멸자 : 3, 3
계속하려면 아무 키나 누
```