SWING 32기 스터디

# [ C++ ]

## 6주차: 복사 생성자, 템플릿

# 복사 생성자 정의



복사 생성자

**객체를 복사**할 때 호출되는 생성자, 한 클래스에 **한 개**만 선언 가능

얕은 복사

객체의 내용을 복사
→ 객체의 내용을 담은 메모리를 **공유**함

깊은 복사

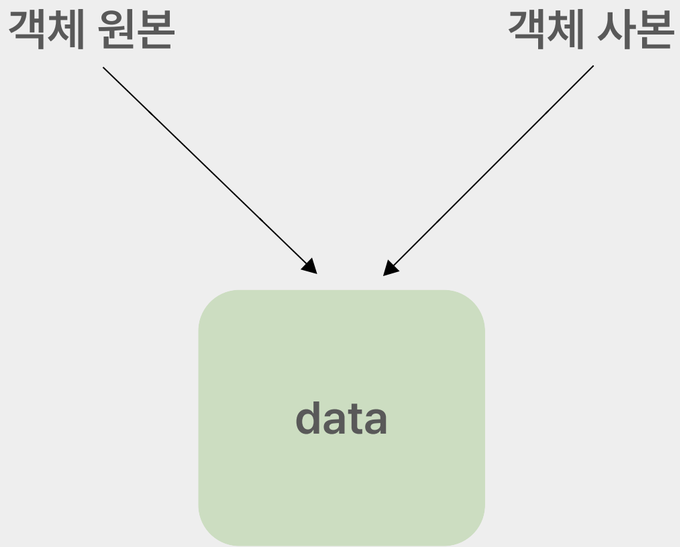
객체와 같은 또 다른 객체 공간을 마련
→ 메모리 할당을 **새로** 함

# 얕은 복사 vs. 깊은 복사



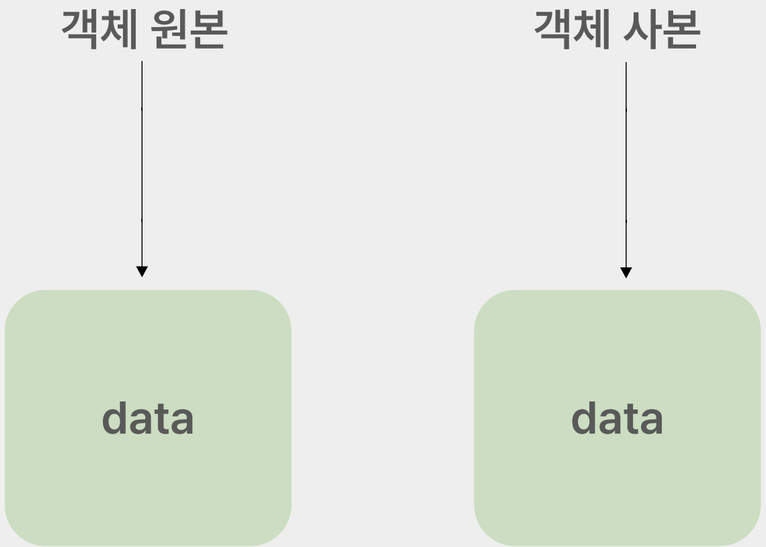
## 얕은 복사

객체의 내용을 복사
→ 객체의 내용을 담은 메모리를 공유함




## 깊은 복사

객체와 같은 또 다른 객체 공간을 마련
→ 메모리 할당을 새로 함



# 복사 생성자 선언과 실행



복사 생성자

객체를 복사할 때 호출되는 생성자, 한 클래스에 한 개만 선언 가능

```
Study(const Study& s);
```

이렇게 선언하고

생성자 함수에 **같은 타입의 객체를 참조 호출**하는 꼴
현재 코드에서 class명은 Study

```
Study CPP();
Study CPP2(CPP);
```

이렇게 실행

# 디폴트 복사 생성자



복사 생성자

객체를 복사할 때 호출되는 생성자, 한 클래스에 한 개만 선언 가능

```
Study(const Study& s);
```

이렇게 선언하지 않아도

```
Study CPP();
Study CPP2(CPP);
```

이렇게 실행하면 디폴트 복사 생성자 실행!

생성자 함수에 같은 타입의 객체를 참조 호출하는 꼴
현재 코드에서 class명은 Study

굳이 선언해주지 않아도 객체 복사 시도하면
컴파일러가 복사 생성자 삽입
→ **디폴트 복사 생성자**라고 함

# 얕은 복사의 문제점



```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class Study {
private:
    int member;
    char* subject;
public:
    Study(int member, const char*subject) {
        this->member = member;
        int len = strlen(subject);
        this->subject = new char[len + 1];
        strcpy(this->subject, subject);
    }

    Study(const Study& s); //복사 생성자

    char* showSubject() { return subject; }
};
```

배운대로 복사 생성자를 만들었다~!

```
int main() {
    Study CPP(3,"C++");
    Study CPP2(CPP);

    cout << "CPP의 과목 " << CPP.showSubject() << endl;
    cout << "CPP2의 과목 " << CPP2.showSubject() << endl;

    return 0;
}
```

# 얕은 복사의 문제점

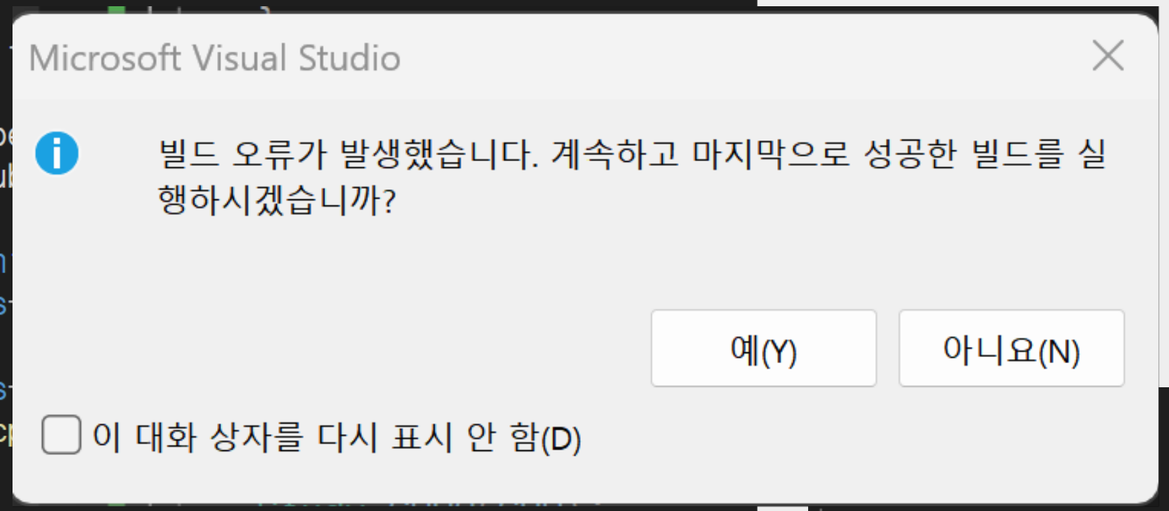


```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class Study
private:
    int memb
    char* su
public:
    Study(in
        this
        int
        this
        strc
    }

    Study(const Study& s); //복사 생성자

    char* showSubject() { return subject; }
};
```




```
    cout << "CPP의 과목 " << CPP.showSubject() << endl;
    cout << "CPP2의 과목 " << CPP2.showSubject() << endl;

    return 0;
}
```

근데 왜 오류가 나지;;;;;

저 창 그만 보고 싶다..

# 얕은 복사의 문제점



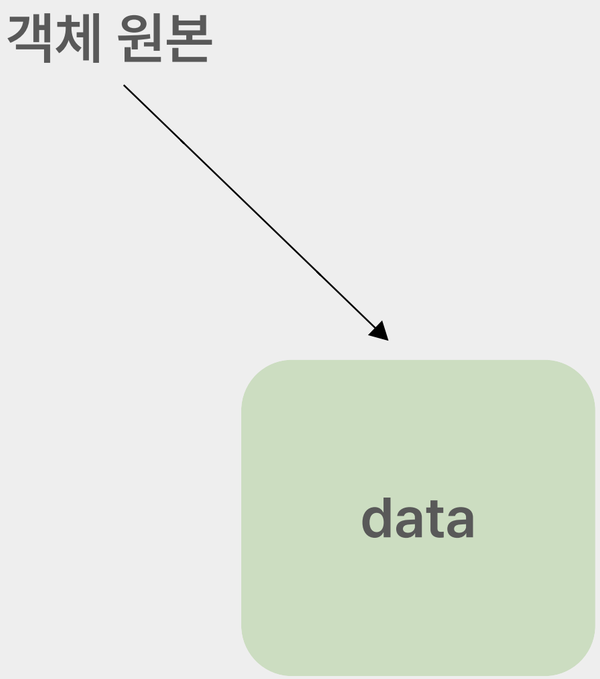
얕은 복사는 **포인터가 존재**하면 문제가 발생한다!

# 얕은 복사의 문제점



얕은 복사는 **포인터가 존재**하면 문제가 발생한다!

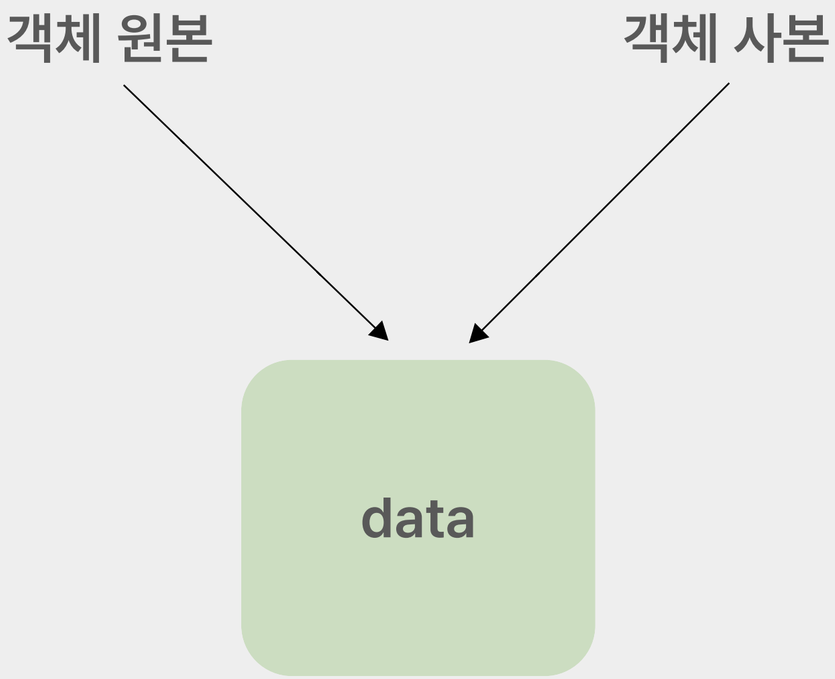
1. 객체 원본의 생성자 실행



# 얕은 복사의 문제점



얕은 복사는 **포인터가 존재**하면 문제가 발생한다!

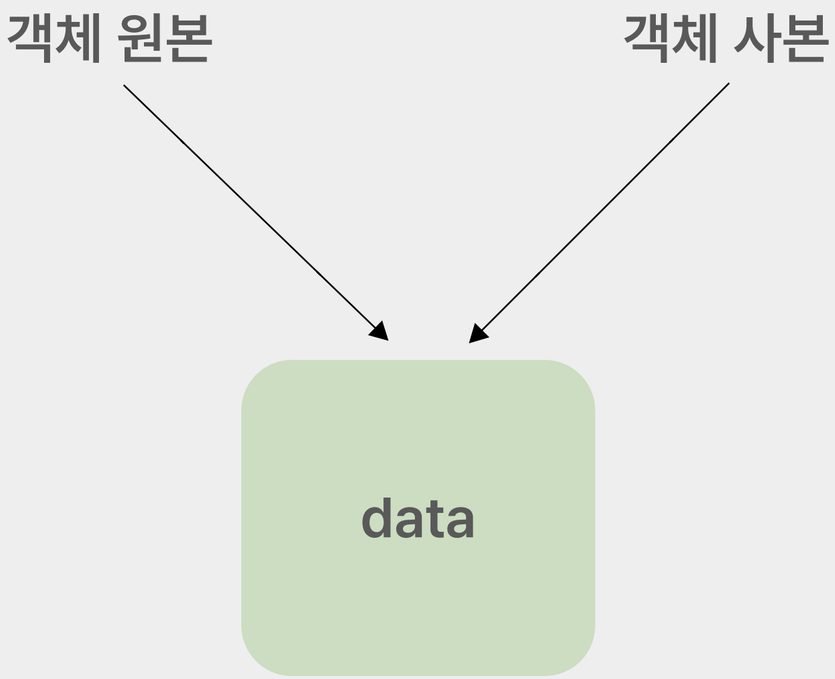



1. 객체 원본의 생성자 실행
2. 복사 생성자에 의해 객체 사본 생성

# 얕은 복사의 문제점



얕은 복사는 **포인터가 존재**하면 문제가 발생한다!




1. 객체 원본의 생성자 실행
2. 복사 생성자에 의해 객체 사본 생성
3. 원본과 사본의 포인터 모두 data를 가리킴
→ 같은 주소를 가리킨다

# 얕은 복사의 문제점



얕은 복사는 **포인터가 존재**하면 문제가 발생한다!


1. 객체 원본의 생성자 실행
2. 복사 생성자에 의해 객체 사본 생성
3. 원본과 사본의 포인터 모두 data를 가리킴
→ 같은 주소를 가리킨다
4. 사본의 소멸자 실행
→ data가 있는 **메모리 해제**

# 얕은 복사의 문제점



얕은 복사는 **포인터가 존재**하면 문제가 발생한다!


1. 객체 원본의 생성자 실행
2. 복사 생성자에 의해 객체 사본 생성
3. 원본과 사본의 포인터 모두 data를 가리킴
→ 같은 주소를 가리킨다
4. 사본의 소멸자 실행
→ data가 있는 **메모리 해제**
5. 원본의 소멸자 실행
→ **존재하는 메모리가 없는데 메모리 해제?**

# 얕은 복사의 문제점



```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class Study {
private:
    int member;
    char* subject;
public:
    Study(int member, const char*subject) {
        this->member = member;
        int len = strlen(subject);
        this->subject = new char[len + 1];
        strcpy(this->subject, subject);
    }

    Study(const Study& s); //복사 생성자

    char* showSubject() { return subject; }
};
```

오류난 코드에는 **char*** 존재

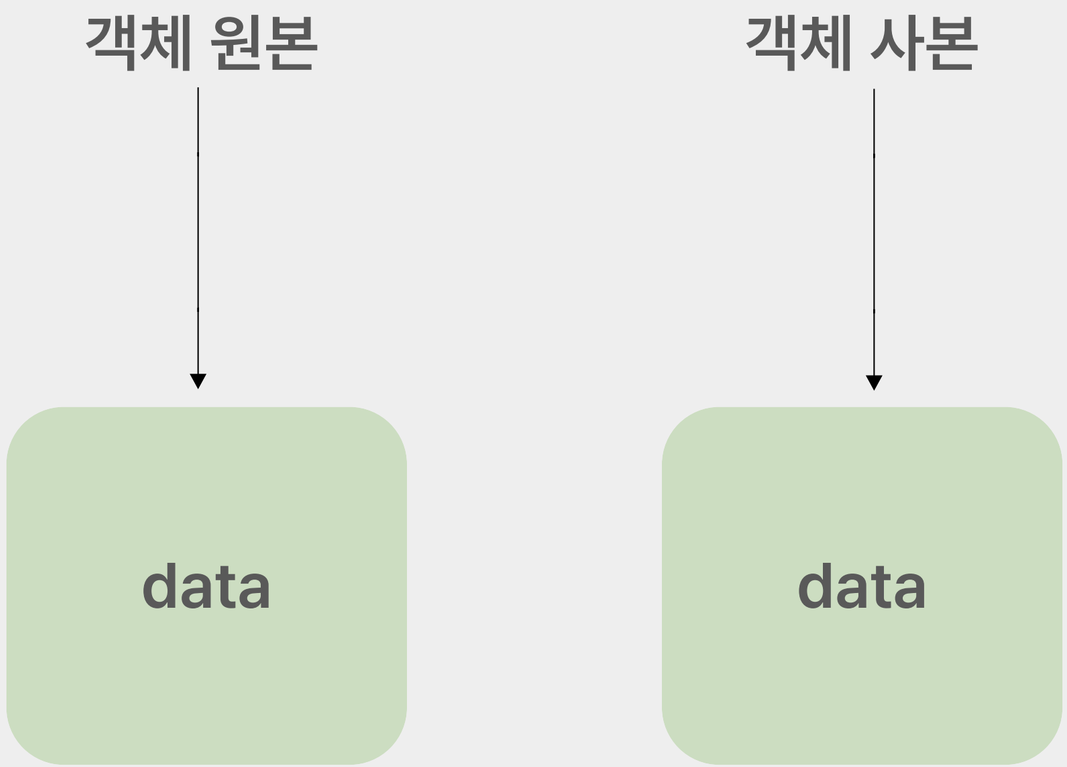
→ 얕은 복사 시 오류 발생

디폴트 복사 생성자도 얕은 복사를 따르기 때문에

깊은 복사는 직접 구현해줘야 한다!

# 깊은 복사 개념






포인터가 선언되면
**각각의 메모리를 할당**해주는 것이
깊은 복사의 핵심!

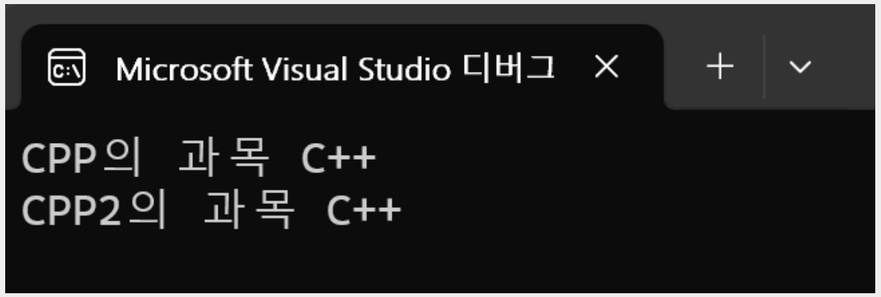
# 깊은 복사 구현



```cpp
class Study {
private:
    int member;
    char* subject;
public:
    Study(int member, const char*subject) {
        this->member = member;
        int len = strlen(subject);
        this->subject = new char[len + 1];
        strcpy(this->subject, subject);
    }
    Study(const Study& s);

    char* showSubject() { return subject; }
};

Study::Study(const Study& s) {
    this->member = s.member;
    int len = strlen(s.subject);
    this->subject = new char[len + 1];
    strcpy(this->subject, s.subject);
}
```




→ 복사 생성자 구현

# 깊은 복사 구현



```
Study::Study(const Study& s) {
    this->member = s.member;
    int len = strlen(s.subject);
    this->subject = new char[len + 1];
    strcpy(this->subject, s.subject);
}
```

포인터 변수는 다음과 같이 처리

1. 변수의 크기 len
2. len+1(문자열의 경우 널바이트를 위해 +1)만큼 new 동적 할당
3. strcpy를 이용하여 내용 복사

# 템플릿



복사 생성자 내용 끝~
이제부터 다른 내용이 나옵니다..
당황하지 않기 위한 파티션 슬라이드입니다..

# 중복 함수



계산기를 만들건데
int형 함수 따로..
float형 함수 따로..
double형 함수 따로..

반환 자료형과 매개변수 자료형만 다르고 코드 내용은 같을 텐데..

```
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}
```

# 중복 함수



계산기를 만들건데
int형 함수 따로..
float형 함수 따로..
double형 함수 따로..

반환 자료형과 매개변수 자료형만 다르고 코드 내용은 같을 텐데..

```
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}
```

함수에서 반환값이나 인자값의 자료형만 바꾸어
다양한 타입의 처리를 진행 하는 것: 오버로딩

# 템플릿



오버로딩은 자료형 각각마다 새로 함수를 작성해야 하므로
비효율적! → 템플릿으로 일반화하기

템플릿: 함수나 클래스를 일반화하기 위한 C++ 도구

# 함수 템플릿



오버로딩은 자료형 각각마다 새로 함수를 작성해야 하므로
비효율적! → 템플릿으로 일반화하기

템플릿: 함수나 클래스를 일반화하기 위한 C++ 도구

<typename 타입이름>
또는
<class 타입이름>
으로 선언..(typename 권장)

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

# 클래스 템플릿



오버로딩은 자료형 각각마다 새로 함수를 작성해야 하므로
비효율적! → 템플릿으로 일반화하기

템플릿: 함수나 클래스를 일반화하기 위한 C++ 도구

<typename 타입이름>
또는
<class 타입이름>
으로 선언..(typename 권장)

```cpp
#include <iostream>
using namespace std;

template <typename T>
class Exec { //클래스 템플릿
    T a;
    T b;
public:
    Exec(T a, T b);
    T add() { return (a + b); }
    void change(T c) { a = c; }
};

template <typename T>
Exec<T>::Exec(T a, T b) { //클래스 멤버 함수 구현마다 typename
    this->a = a;
    this->b = b;
}

int main() {
    Exec<int>swing(10, 5); //클래스 사용할 때 자료형 지정
    cout << swing.add()<<endl;
    swing.change(5);
    cout << swing.add() << endl;
    return 0;
}
```

# 과제



01

문서화 과제

02

STL 구현

# 과제



01

## 문서화 과제

들어가야 하는 내용

- 6주차 수업 내용 정리
- STL이란?
- 컨테이너, 알고리즘, 반복자 각각의 정의와 종류
- Linked List란?
- 단일 링크드 리스트의 특징
- 과제2의 알고리즘 문서화 (원소를 추가하는 원리 등)

# 과제



02

**STL 구현**

단일 링크드 리스트 구현하기

1. Node의 클래스 템플릿 구현하기
2. singleLinkedList 클래스의 멤버 함수 구현
3. main함수 작성하고 실행 화면 캡처(다음 페이지)

```
template <class T>
class singleLinkedList { //리스트의 구성
public:
    singleLinkedList();
    ~singleLinkedList();
    void push(T element); //element를 리스트의 마지막 원소로 넣는 함수
    void show(); //리스트의 모든 원소값을 출력하는 함수

private:
    Node<T>* head; //리스트의 첫 번째 원소
    Node<T>* tail; //리스트의 마지막 원소
};
```

위 캡처 화면은 singleLinkedList 클래스의 구성

# 과제



02

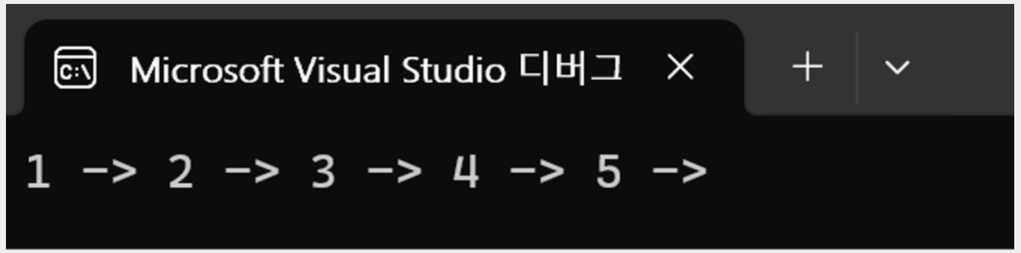
**STL 구현**

단일 링크드 리스트 구현하기

1. Node의 클래스 템플릿 구현하기
2. singleLinkedList 클래스의 멤버 함수 구현
3. main함수 작성하고 실행 화면 캡처(다음 페이지)

```
int main() {
    singleLinkedList<int>*idx=new singleLinkedList<int>;
    idx->push(1);
    idx->push(2);
    idx->push(3);
    idx->push(4);
    idx->push(5);
    idx->show();
}
```

main() 함수 코드




실행 화면

참고 자료: 31기 C++ 스터디, 최주영 교수님 C++ 수업 자료, TCP school

# THANK YOU

31기 육은서, 이시온, 황선영
질문은 카톡으로 보내주세요~