C++

1. C와는 다른 C++

① C와 C++ 차이

C의 자료형이 어려운 것을 쉽게 만드는 것

② 프로젝트 생성해보기

- C++에서는 사용자 정의 헤더가 아닌, 표준헤더들은 .h 생략

  ex) iostream

- _tmain은 유니코드를 지원하기 위해 재선언 됨.

③ 인스턴스

- 그냥 변수와 같은 의미

  ex) int a → int 자료형에 대한 인스턴스 a.

④ Std::cout

- cout은 iostream 클래스의 인스턴스.
- printf와 다르게, 출력 결과에 맞춰 자료형 지정할 필요 X.

⑤ Std::cin

- cout처럼 자료형 맞출 필요 X
- String 이용시 배열처럼 길이 고려 X

⑥ 자료형

- C와 기본적으로 같음.
- auto, decltype(expr) 추가.

⑦ 변수 선언 및 정의

- C → int a = 10;
- C++ → int a(10);

  int b(a); → 복사생성자로 이어짐

  int (10); → 이름없는 변수 선언 (컴파일 → 상수처리)

⑧ auto

- 초기값 형식에 맞추어 선언하는 인스턴스 형식이 자동으로 결정
- 범위기반 for 문에서 활용

⑨ 메모리 동적 할당

- new와 malloc 차이 (메모리 크기 지정 안해도 됨.)
- new[]로 생성한건 delete[]로 삭제

• auto는 컴파일 / 런타임 시점 중 언제 자료형이 결정나는가?

- 컴파일 시점에 오른쪽 =16 이런 항을보고 변수의 타입을 결정. 그래서 초기화가 의무적.

• new[]로 생성한걸 delete로 삭제하거나, 그 반대일 경우 일어나는 문제는?

- 일반 데이터 타입은 상관 X. 객체도 파괴자가 없다면 상관 X 그게 아니면 new할때 할당되는 크기가 들어간다. 아예 소멸자가 있으면 문제. delete는 포인터만 알지 몇번 파괴자 호출해야 하는지를 모름. []를 적어야 메모리 위치 한칸 앞에서 배열 개수를 확인함.

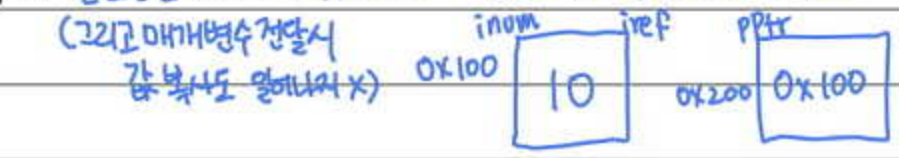
⑩ 참조자 형식
- 포인터와 구조적으로 비슷
- 선언과 동시에 초기화
- 상수에는 참조자 선언 불가.
- 참조는 '주소'를 의미

• 참조자와 포인터의 차이는?

메모리 공간 소모(주소값) ← 포인터는 메모리의 주소를 가지고 주소 값을 통한 메모리 접근 (간접 참조)

같은 공간이라 메모리 X ← 참조자는 참조하는 변수를 대신할 또하나의 이름 (직접 참조)

(그리고 매개변수 전달시 값 복사도 일어나지 X)



⑪ r-value 참조
- int && data = 3;
- 임시결과

• r-value 의 의미는?

l-value는 l-value만, r-value는 r-value만 참조가능.

표현식이 종료되면 존재하지 않는 임시 값을 참조했을 때 장점은?

l-value는 참조생성자 연산 후에도 매개인자 값이 남아 복사연산.

r-value는 " " 값이 남지 않아 이동연산.

원본의 복사가 불가능하도록 유일한 소유권 구현가능.

ex) unique_ptr, shared_ptr 에서 사용.

이동 연산의 이동은 직접 구현 해야함!

⑫ 범위 기반 for 문
- 반복 횟수는 배열 요소 개수에 맞춰 자동으로.
- for(auto n : list) → 읽기만 가능한 접근
- for(auto &n : list) → 수정도 가능.

2. C++ 함수와 네임스페이스

① 디폴트 매개변수
- 호출자만 보고 함수 원형 추측 불가
- 반드시 오른쪽 매개변수부터 기술

② 함수 다중정의
- 오버로딩 (함수 매개변수가 달라지는 등)
- 다형성을 제공한다.

③ 다중정의와 모호성
- f(int a)
- f(int a, int b=10)
- → 이 두 경우 f(1)를 호출하면 모호해진다.

④ 함수 템플릿
- 오버로딩 VS 함수템플릿
  - 오버로딩 → 유지보수 측면에서 문제. 불필요한 코드 작성시 메모리 낭비.
  - 함수템플릿 → 컴파일러가 생성. 컴파일 시간↑

⑤ 인라인 함수
- 함수 호출로 인한 오버헤드 극복
  - → 스택메모리↑, 매개변수 때문에 메모리 복사
- 최적화 옵션을 켜두면 컴파일러가 알아서 결정해준다.

⑥ 네임스페이스
- 협업중, 다른 개발자와 변수명이 겹치는 문제 해결

⑦ Using 선언
- 자주 사용하는 네임스페이스 생략

⑧ 네임 스페이스 중첩
- 네임스페이스 안에 또 다른 네임스페이스 속할 수 있다.
- 이럴땐 TEST::DEV::g_data 와 TEST::g_data 이런식으로 구별해줘야 한다.

⑨ 식별자 검색 순서
- 전역 함수 (현재블록 범위 → 현재 블록 포함된 상위블럭. → 가장 최근에 선언된 전역변수, 함수 → Using 사용된 네임스페이스)
- 클래스메서드 (현재 블록 범위 → " → 클래스 멤버 → 부모클래스의 멤버 → 가장 최근에 선언된 전역변수나 함수
  → 호출자가 속한 namespace의 상위 namespace, → Using 선언된 namespace

3. 클래스

① 객체지향 프로그래밍 개요

-인터페이스 함수란? 사용자가 함수 구조까지 알 필요X. 기능만 제공받아 쓰면 됨.

② 클래스 기본 문법

-생성자

(멤버가 참조자? → 반드시 생성자 초기화 목록)

-접근제어 지시자

클래스 : 기본 Private

구조체: 기본 Public

- 생성자 / 소멸자.

생성자는 다중정의 가능

소멸자는 불가능.

컴파일러가 알아서 디폴트 생성자 소멸자 만듦.

- 전역변수로 선언한 클래스의 생성자는 main 보다 먼저 호출.

• 생성자 초기화 목록 VS 생성자에서 초기화

↳ 단순 대입연산자

• Private 생성자를 만들게 되면?

③ 동적 객체의 생성과 소멸

-new[]로 생성한 것은 delete[]로 삭제해야한다.

-안 그러면 메모리 릭 남음

• 객체 + 소멸자가 있는 경우 delete[]로 삭제하지 않으면 몇개의 객체를 삭제해야하나 포인터만 갖고는 모른다.

④ 참조형식 멤버 초기화

-참조자는 선언과 동시에 초기화해야함. 그래서 생성자 초기화 목록을 써야한다.

Test(int& ref) : m_data(ref) { } (O) → 생성자 초기화 목록

Test(int &ref) { m_data = ref; } (X) → 대입연산자

⑤ 생성자 다중정의

-C++11부터 생성자 위임가능.

↳ 생성자 초기화 목록에서 다른 생성자 추가로 호출

같은 일을 하는 코드가 여러번 있을 필요가 없어짐

⑥ 명시적 디폴트 생성자 → 지금은 존재만 알고, 템플릿에서.

-CTEST(void) = default;

- 선언과 정의를 한번에 끝냄.

⑦ 메서드

-클래스의 멤버 함수

• this를 쓴게 빠를까? 안쓴게 빠를까?

둘다 똑같다. 내부적으로 this를 다 써서 상대위치로 이동한다.

⑧ this 포인터

-소속을 정확하게 명시

클래스 멤버 함수 호출하면 다 그 안에서 포인터→멤버 이렇게 본다.

그런데!!! 내 멤버를 접근할 일이 많으면, 지역변수에 옮겨두고 쓰는게 좋다.

멤버접근은 this 기준으로 그 멤버위치까지 한 단계 뛰어야 도달한다.

지역변수도 ebp기준으로 더하긴 하지만 레지스터에 있는 값이라 한방에 뛰어간다

⑨ 상수형 메서드

- 읽기는 O, 쓰기는 X
- 함수 원형 뒤에 const
- 특징 : 상수화 방법 → this 포인터를 상수형 포인터로 변경
  (CTest *this 를 const CTest *this 로 만듦)
- 상수화된 메서드가 아니면, 멤버 함수도 호출 불가.
- mutable 로 선언한 변수는 상수형 메서드에서 쓰기 가능
- const_cast< >로 억지로 쓸 수 있게 형변환 가능

⑩ delete 예약어

- void func(double d) = delete;
  ↳ 실수로 double형 인자가 넘어오는 경우 차단.

⑪ 정적 멤버

- 사실상 전역변수, 함수와 같다.
- 인스턴스 선언 없이 직접 호출가능
- 선언과 정의 분리

ex) int CTest :: m_nCount = 0; → CTest 클래스의 정적 멤버변수 정의.

ex) static int func() { return m_nCount; } → 정적 메서드 선언

CTest :: func() → 정적 멤버에 접근. 인스턴스 없이 가능

4. 복사 생성자와 임시객체

① 복사생성자

- 객체의 복사본을 생성할때 호출되는 생성자.
- 복사될 원본은 const로 불러주자. 원본이 손상되는 일은 없어야 함.

ex) CData(const CData &rhs)

main 복사생성자

① CTest a; ③ CTest (const CTest & rhs) : m_ndata (rhs.m_ndata)

② func(a); ④ 전역변수 func(CTest Param)

호출순서 ① ② ③ ④

func의 매개변수에 대한 복사본 생성후 전역변수 호출.

처음부터 func(CTest &param) 으로 하고

넘겨줬으면 복사생성자 호출 X.

그리고 원본 보존해야하니 func (const CTest& Param)

② 깊은복사 VS 얕은복사

복사에 의해 실제로 두개의 값 생성

대상값은 1개인데 접근 포인터만 2개로 늘어남

- 얕은 복사의 경우 이미 삭제된 메모리를 한번더 삭제하려 할 경우 오류 발생 (포인터가 존재해야 문제 됨)
- 위와 같은 경우 복사 생성자를 정의해서 깊은 복사로 만들어 줌

ex) m_data = new int; → 메모리 할당

*m_data = *rhs.m_data; → 내 포인터 위치에 값 복사

③ 대입연산자

- 단순 대입 → 얕은 복사 실행
- 대입연산자 다중정의로 깊은복사 수행하도록 작성

```
ex) CMyData& operator=(const CMyData &rhs){
    *m_pData = *rhs.m_pData;
    return *this; → 자신에 대한 참조 반환
}
```

④ 변환 생성자

- explicit 예약어로 차단 가능

ex) func(05) → func(CTest(5)) 이렇게 컴파일러가 임시객체 생성자를 부름 (형이 안 맞는 경우)

그래서 생성자에 explicit CTest(int n): m_data(n){ } 이렇게 붙임

묵시적으로 불가능해져서 이제 func(5); 라는 코드는 오류. 명확히 func(CTest(5)) 이렇게 해야 함.

⑤ 이름없는 임시 객체

- 객체를 return 하는 경우 임시객체가 복사생성자로 생겼다 사라짐
- CTest &rdata = fun(10);

→ 이렇게 만들면 리턴후 사라지면서, main 이 끝난후 사라짐 (해당 지역 끝난 후)

이름이 없는 임시 객체에 이름을 부여했기 때문.

⑥ r-value 참조

func (3+7); void func(int && rParam);

→ 연산에 따라 생성된 임시객체

→ 이게 필요한 이유는 기본 자료형 말고, 클래스에 적용될 때 필요

⑦ 이동 시맨틱

- 이동생성자 + 이동대입 연산자

ex) CTest(CTest && rhs) : m_data(rhs.m_data){ } → 이동생성자

→ 어차피 사라질 객체이므로 얕은 복사를 수행해서 성능을 높이는 것.

(return 뒤 사라질거니 데이터만 이동생성자로 복사해옴)

5. 연산자 다중정의

① 연산자 함수란?

연산자를 이용하듯 호출할 수 있는 메서드

② 산술 연산자 (+)

- operator int() → 형변환 연산자

- MyData operator +(const MyData &rhs) → 덧셈 연산자

- MyData& operator =(const MyData &rhs) → 단순 대입 연산자

→ MyData & 로 리턴하는 것이 아니므로 이동생성자가 호출된다.

왜 &를 붙이지 않는가? → 3+4 같은 기본형식이라면 연산의 중간 결과값을 CPU의 레지스터가 담았을 것.

하지만, 클래스 객체는 그럴수 없고 꼭 메모리에 담아야 함. 그래서 기본 자료형과 비교하면 성능이 더 좋을 수 없음.

③ 대입연산자 (=)

void operator =(const MyData & rhs)

{ delete m_data; → 기존 값 삭제

m_data =new int (*rhs.m_data); →새로 할당한 값 메모리에 저장

}

- 이 코드의 경우 a=a 연산을 하면 오류가 난다: 왜? 복사하기 전에 원본을 지웠기 때문. if문으로 this == &rhs 면 바로 return하게 수정.

즉, r_value가 자신이면 대입 수행 X

- 또, a=b=c도 오류. 왜? 리턴 값이 void인데, 이걸 l-value로 사용불가.

또, 없는 값을 대입하는 것 불가.

→리턴 형식을 void 말고 MyData& 로 하고 return *this로 수정

④ 복합대입 연산자 (+=)

- MyData& operator +=(const MyData& rhs) 같은 연산자

⑤ 이동대입 연산자 (=&&)

-연산에 의한 임시결과

1. a+b 연산에 의한 것

2. 함수 호출에 의한 것

MyData& operator =(MyData && rhs) → 내부에서 얕은 복사 실행 후 원본 rhs는 NULL로 초기화

a = b+c → 이동대입연산자 실행

→여기에서 리턴 되는 임시 객체를 r-value로 삼아 이동대입연산자 =를 호출

⑥ 배열 연산자 ([])

- int& operator[](int idx); → int&로 반환하는 것은 l-value로 사용되는 경우를 고려하기 때문. 일반적 ex) 대입용

- int operator[](int idx) const; → 상수형 참조를 통해서만 호출 가능. 오로지 r-value로만 사용됨. ex) 단순 출력용

⑦ 관계연산자 (==) (!=) → 보통 Strcmp로 구현되어 있음.

⑧ 단항 증감 연산자 (++) 1. 전위(++a) → 그냥 바로 return ++a. 증가한 뒤 그 값을 리턴.

2. 후위 (a++) → 값을 증가시키기 전에 백업후 증가. 리턴은 백업된 값.

6. 상속기본

① 상속이란? (is-a / has-a)

- 객체 단위 코드를 재사용 하는 방법
  → 기능적 확장/재사용

② 문법

Class 이름 : 접근제어 지시자 부모 Class 이름

- 파생클래스 객체가 생성될 때 기본 클래스 생성자도 호출됨. (부모 생성자 → 자식 생성자)
- 파생클래스는 기본클래스의 Private로 선언된 것 외에는 다 접근 가능
- 파생클래스 객체를 통해 기본클래스 메서드 호출가능

순서 ① 자식 객체 선언

② 자식 객체 생성자 호출 → 먼저 호출되지만, 실행은 나중에.

③ 상위 부모 객체 생성자 호출

④ 상위 부모 객체 생성자 실행 → 실행 후 반환하면, ⑤ 실행.

⑤ 자식 객체 생성자 실행

③ 메서드 재정의

- 오버라이드 : 기존의 메서드를 무시하고 재정의

  기존 부모의 메서드를 대체.

-특징 ① 자식이 부모 함수를 오버라이드 했으면, 쓸 때 (func 함수 안에) 소속을 명시해야 함. 안그러면 재귀호출이 된다. → 부모::func();

② 자식이 부모함수를 오버라이드 했지만 부모버전 함수를 호출하고 싶다면? 명시적으로 a.부모::func() 부르면 됨. 단, Public만. Protected 안됨.

④ 참조형식과 실형식

ex) 자식 a; → 참조형식 / 실제형식

부모 &rdata = a;

rdata.함수() → 참조형식 ≠ 실제형식 → 참조형식 메서드 실행. (∴부모 함수 실행)

ex) 부모 *p = new 자식;

P→함수();

→ 실형식이 자식임에도, 참조 형식이 부모라 부모함수 실행.

delete P;

⑤ 상속에서 생성자와 소멸자

C 객체 생성시 생성자 호출순서 : C→B→A → ∵ 호출은 먼저 되어도 실행은 상위 클래스 부터.

실행순서 : A→B→C

C 객체 파괴시 소멸자 호출/실행순서 : C→B→A → 출력 순서 (∵ 실행순서와 같다)

문제! 자식이 소멸자에서 부모의 멤버 변수를 delete 하고, 부모도 delete 하면 오류. 그러니 부모껀 직접 쓰기 하지 말자. 자기것만 초기화/수정.

⑥ 생성자 선택

ex) 자식( )

자식(int n) : 부모(n)

자식(double n) : 부모()

실행 순서
① 부모()
자식()
② 부모(int)
자식(int)
③ 부모()
자식(double)

7. 상속심화

① 가상함수 (virtual)

- 자식이 메서드 재정의시, 기존의 정의 완전히 무시

+) 가상함수 : 늦은 바인딩 (= 동적 바인딩)

(늦은 바인딩: 함수의 변수나 주소가 런타임에 결정)

(이른 바인딩: , 컴파일타임에 결정)

→ 일반적인 함수의 경우, 호출할 함수가 달라질 가능성 x

∴ 이른 바인딩은, 기계어로 번역했을시 그 주소가 명백히 드러남.

ex) Call Func (01231500h)

→ 함수가 Func 라는 상수

반면 늦은 바인딩은

ex) Call dword ptr[pfTest]

→ 피연산자가 상수가 아니라 포인터 변수.

따라서 변수에 저장된 주소의 함수가 호출될 것.

② 문법

- virtual 반환형 메서드이름

Virtual Void Printdata();

- 일반 메서드는 참조 형식을 따르고, 가상함수는 실형식을 따른다.

부모	자식
virtual void f() void testF()	virtual void f()

main
자식 a;
a.f(); → 자식의 f() 실행
부모 &b = a; (참조형식 → 실형식)
b.f(); → 실형식인 자식의 f() 실행
a.testF() → 마지막에 재정의 된 함수 호출 (여기서는 부모의 testF())

③ final → 특정 가상함수, 미래에 재정의 되는 것 막기.

virtual void f() final; → 재정의 시도시 컴파일 오류

④ 소멸자 가상화

부모 *p = new 자식

↳ 추상자료형 : 상위 클래스로 하위파생클래스를 참조할 때

- 문제점: 추상자료형을 이용해서, 동적 생성한 파생 객체를 참조할 경우, 심각한 메모리 누수
- 왜? : 파생형식의 소멸자가 호출되지 않아서. 일반 메서드는 실형식이 아닌, 참조형식의 소멸자가 호출된다.
- 해결 : 기본 클래스의 소멸자를 가상화 한다. (virtual ~부모())

그러면, 파생클래스의 소멸자까지도 제대로 호출된다. (자식 소멸자 실행 → 부모 소멸자 실행)

한번 가상함수는 영원히 가상함수. 자식 소멸자는 virtual 없어도 자동으로 가상화 된다.

⑤ 가상함수 테이블

- 함수 포인터 배열

- 가상함수를 멤버로 갖는 클래스는, 자신만의 가상함수 테이블을 갖는다.

부모	자식
virtual ~부모()	virtual ~자식()
virtual F1()	virtual F1()
virtual F2()	virtual F2()

main
부모 *p = new 자식 → 부모 생성자 실행 → 자식 생성자 실행 (__vfptr의 부모 가상함수 테이블의 [0]을 가리킴 → 자식생성자가 실행되며 __vfptr은 자식 가상함수 테이블의 [0]으로 덮어쓰임)
p→F2(); → virtual 함수이므로 실형식인 자식 함수 실행 → 그래서 여기서 p→__vfptr→F2(); 이렇게 되어서 자식 F2가 불리는 것.
delete p; → 자식 소멸자 실행 → 부모 소멸자 실행

⑥ 순수 가상 클래스

- 순수가상함수를 멤버로 가진 클래스

↳ 선언은 지금 해두지만, 정의는 미래에 하도록 놔둔 함수

Virtual int GetData() const =0;

추상자료형을 상속받은 자식클래스가 인스턴스를 생성하면,
부모인 추상 클래스의 생성자가 실행되고 자식클래스 생성자가 실행된다

- 특징 : 인스턴스를 직접 선언할 수 없다.

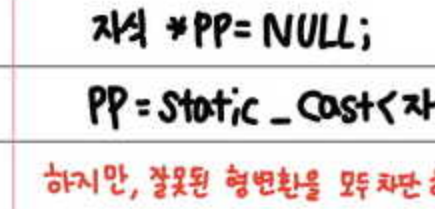
파생클래스에서 반드시 기본 클래스의 순수가상함수를 재정의 해야 함 → 안하면 컴파일 오류
추상자료형으로 파생클래스 인스턴스를 참조 하더라도 , 가상함수라 실형식을 따르므로 파생클래스 생성자가 불린다

⑦ 인터페이스 상속

- 인터페이스 : 서로 다른 두 객체가 서로 맞닿아 상호작용할 수 있는 통로나 방법

ex) 전역함수 → void f(부모 *P){ P→getID; }
(getID: 순수가상함수 / 여기로 넘어오는 자식이 정의한 getID 함수 호출)

⑧ 추상 자료형의 사용 예



→ 순수가상 클래스 (virtual fare()=0;)

→ 파생클래스

person→fare() ⇨ 가상함수이므로 사용자 입력에 따라 실제 생성된 실형식의 fare()가 호출될 것.

∴ 다중 if문/switch-case의 경우의 수가 늘어나면 성능이 떨어진다는 문제점 해결 가능.

⑨ 상속과 형변환

- <const_cast<> : 상수형 포인터에서 const 제거
- Static_Cast<> : 컴파일시 형변환 (상향 or 하향 형변환) 사용자 정의 자료형을 막 char *로 변환 한다던지 그러면 x. 이건 C 형변환 연산자라면 가능.
- dynamic_cast<> : 런타임시 형변환 (상향 or 하향 형변환)
- reinterpret_cast<> : C 형변환 연산자와 흡사 (강제 형변환)

⑩ Static_Cast

- 상향형변환

ex) 부모 *P = new 자식; → 가리키는 대상은 파생형식.

자식 *PP = NULL;

PP = Static_Cast<자식*>(P) → ∴ 파생형식에 대한 포인터로 형변환 시도.

하지만, 잘못된 형변환을 모두 차단 하지는 x

부모 *P = new 부모;
자식 *PP = NULL;
PP = Static_cast<자식*>(P);

위 경우, 에러가 안남. 왜냐면, 이 이후 코드에서 자식클래스의 멤버 데이터에 접근하지 않기 때문.

⑪ dynamic_cast (안쓰는게 좋다. 성능을 떨어트리는 주범.)

- 동적으로 생성된 객체가 어떤 객체에 대한 인스턴스인지 확인하고 싶을때.
- 형변환에 실패하면 NULL 반환. 반환 값이 NULL 인지 if 문으로 확인하며 어떤 형인지 확인

⑫ 상속과 연산자 다중정의

- 부모만 연산자를 정의하는 경우, 자식이 쓰지 못함.(∵ 넘겨줄때 매개변수인 r-value 의 형이 틀리기 때문.)

→ 위경우, 자식도 연산자를 오버라이딩 해주면 된다.

⑬ 다중 상속 (안쓰는게 좋다. 설계오류의 위험성.) → 유일한 좋은 예 : 인터페이스 다중상속

- 한 클래스가 2개 이상의 클래스 상속
- 두 부모 클래스가 같은 메서드 이름을 쓴다면 호출이 모호해진다. (명시적으로 :: 을 써서 해결 가능)

⑭ 가상상속 (파생형식 클래스 앞에 virtual.) ex) class 자식 : virtual public 부모

- 족보가 꼬이는 경우의 해결법 → 상속이 겹칠 때 하나를 무시.
  ↳ 같은 부모를 갖는 두 자식을 다중상속받은 손자는, 생성자를 부르면 부모클래스 생성자가 두번 호출됨.

8. 수평적 관계와 집합관계

① friend 존재 이유 : 자유도를 높이기 위해 (=응집성)

friend class 이름

friend 함수 원형

② friend 함수

- Private 멤버변수에 접근 가능.

③ friend 클래스

- friend 선언한 클래스 통째로 접근제어 지시자 영향 X.

-사용 예) list 클래스에서 node 클래스의 private 멤버에 접근해서 다음노드 연결

④ 집합관계

- composition 관계 : UI 클래스가 자료구조 객체를 구성요소로 포함

- aggregation 관계 : UI 클래스가 인스턴스 선언 될 때 list 클래스 인스턴스가 참조로 전달

9. 템플릿

① 클래스 템플릿

template <typename T> ← 자료형

② 템플릿 특수화

template <>

class CMyData<char*>

이렇게 char* 형식을 선언하면 형식을 특수화 가능

③ 클래스 템플릿과 상속

- template<typename T>

class CMyDataEx : public CMyData<T>

④ 스마트 포인터

- auto_ptr : 동적 할당된 인스턴스 자동으로 삭제. 가장 오래 존재했던 스마트 포인터
- shared_ptr : 포인팅 횟수를 계산해서 0이 되면 대상을 삭제한다.
- unique_ptr : shared_ptr과 다르게 한 대상을 오로지 한 포인터로만 포인팅한다. 하나의 소유자만 허용
- weak_ptr : 하나이상의 shared_ptr 인스턴스가 소유하는 개체에 접근할 수 있게 하지만, 참조 수로 계산하지 X. 특수한 경우만 사용

⑤ auto_ptr

- 가장 구형 → 사용하지 않는게 바람직
  - 배열 지원 X
  - 포인터의 고질적인 '얕은복사' 문제 해결 X
- 할당한 지역이 끝나면 자동으로 소멸.

하지만, auto_ptr<CMyData> ptr(new CMyData[3]); 이렇게 배열로 동적 할당 하면 문제 발생.

① 배열로 생성하면 배열로 삭제해야 하는데, 첫번째 개체만 소멸. 나머지는 남아 있음.

② 얕은 복사를 진행함 (포인터를 대입하면 원본 포인터엔 NULL이 들어간다)

〈단순대입연산은(어처구니 없게도 복사가 아니라 이동)

대안↓

⑥ shared_ptr

- 포인팅 횟수를 계산해서 0이 되면 대상을 삭제
- 또, 배열로 객체를 삭제할 수 있는 방법 제공. (개발자가 함수를 작성해야 함.)

```
void F(CTest* p)
{delete[] p; }
```

- auto_ptr을 완벽히 대체 가능.

⑦ unique_ptr

- shared_ptr과 유사하지만, 이건 오로지 한 대상을 한 포인터로만 포인팅 가능.
- 즉, 얕은 복사가 일어날 가능성 완전 차단

⑧ weak_ptr

- shared_ptr이 가리키는 대상의 참조 형식으로 포인팅 가능
- 참조 카운터 영향x.
- 포인팅만. 참조도 못하고 참조하려면 shared_ptr로 변환해야함

10. 예외처리

① try, throw, catch 문

-문제가 생기면 집어던지고, 받은 쪽에서 수습한다.

```
try {
  if (error)
   throw errorcode; }
catch (int exp) {
 ... }
```

② 기본 활용법

-사용이유 : 구조적으로 간결해 짐. (코드가 줄지는 않는다.)

③ catch 다형화

- try-catch 문에서 catch를 여러개로 구현하는 것을 의미.

```
try {
   F1();
   F2(); }
catch (int exp) {
}
catch (char ch) {
}
```

④ 예외 클래스

- class 클래스 { private: int error_code;
  char msg[128]; }

  클래스 exp ; throw exp;

  Catch (클래스 &exp) { exp. getmessage(); }

⑤ 스택풀기

- 예외가 발생하면 스택을 호출한 순서대로 모두 반환 되도록 코드를 작성해야 한다.

- throw를 하면 가장 밖의 catch에 걸린다.

⑥ 메모리 예외처리

- 너무 큰 메모리 할당시 오류 발생. → 메모리를 할당해주는 함수나 연산자는 NULL 리턴

-이때 try catch (bad_alloc &exp) 사용

12. 앞으로 할것

① 람다식과 함수객체

- 람다 : 이름없는 함수

ex) auto func = [](int n) → int
{
  cout << n << endl;
  return n;
}
func(5);

(int n: 매개변수, int: 반환형)

② 함수 포인터와 콜백

ex) 형태 compareData(const void *left, const void *Right) { ... return ... }

qsort(list, 5, sizeof(int), compareData)

→ 각 항을 비교하는 방법으로 함수 주소를 콜백함수로 전달한다.
함수 내부에서 두 요소를 비교할 때마다 이 주소를 호출.

③ 함수 객체(=펑터)

- 함수호출 연산자를 다중정의한 클래스 → 마치 함수 템플릿처럼 사용가능.
  → 하지만 자료하는 형식과 규모를 특정할 수 없음

ex) int operator()(int a, int b)
double operator()(double a, double b)

④ 람다 캡처 (복사캡처: call by value / 참조캡처: call by reference)

- 복사 캡처 : 람다식 선언 []의 내부에, 외부에서 사용할 변수 이름을 작성해 캡처하는 것. (ex. [nData](void) → void).
- 참조 캡처 : 〃 참조 변수(&)이름을 작성해 캡처하는 것 (ex. [&nData](void) → void).
- 디폴트 복사 캡처 : 람다식 외부의 사용할 수 있는 모든 변수 (람다 선언 구현에 이미 선언되었고 소멸되지 않은 모든 사용범위 안 변수들)를 복사로 한번에 캡처 (ex. [=](void) → void)
- 디폴트 참조 캡처 : 디폴트 복사 캡처와 같은 의미인데, 복사 대신 참조로 한번에 캡처. (ex. [&](void) → void)

⑤ Solid 원칙

- 단일 책임 원칙 : 한 클래스는 하나의 책임만.
- 개방 폐쇄 원칙 : 확장은 쉽게, 변경은 어렵게.
- 리스코프 치환 원칙 : 객체는 프로그램 정확성을 깨지 않으면서 하위타입 인스턴스로 바꿀 수 있다.
- 인터페이스 분리 원칙 : 내가 사용할 인터페이스만 정의, 인터페이스 여러개가 범용 인터페이스 하나보다 낫다.
- 의존성 역전 원칙 : 프로그래머는 구체화가 아닌, 추상화에 의존해야 한다