2단원

- 포인터의 참조자를 선언할 경우 int *(&ref)=ptr; 과 같은 형식으로 선언 (여기서 int* ptr= #)

5단원

- 복사 생성자: Classname(const Classname &copy) : ~~~~~{ }

- Classname obj2= obj1 or Classname obj2(obj1) (생성자) 를 입력하면 멤버 대 멤버로 복사가 진행됨.
- 복사 생성자는 자동으로 삽입이 됨.
- 이때 explicit Classname(const Classname &copy): num1(copy.num1), num2(copy.num2) { } 등
explicit 키워드를 넣으면 Classname obj2= obj1이 복사기능을 못하게 만듦.
- 전달인자가 하나인경우 Classname one= 1 or Classname one(1)도 가능
- 복사할 경우 멤버 변수의 주소까지 복사해버림 -> 변수가 두 개 이므로 소멸자가 두번 작동해야하지만 처음 소멸자가 두 변수의 멤버변수를 한번에 소멸해버림 -> Person(const Person& copy) : age(copy.age) { name = new char[strlen(copy.name)+1];
strcpy(name, copy.name)식으로

- 복사 생성자 호출 타이밍

- 기존 객체를 이용해 새로운 객체를 초기화할 때
- call-by-value 방식의 함수를 호출할 때, 객체를 인자로 전달하는 경우
- 객체를 반환하되, 참조형으로 반환하지 않는 경우
-> 예를 들어 classname func(classname ob){ return ob;}
와 func(obj)의 반환 값은 obj가 아니라 obj를 복사한 임시 객체이므로 기존에 생성된 obj와는 별개이다. 이때 임시 객체는 참조되거나 새로운 이름을 할당 받지 않을 경우, 생성 후 다음 행에서 소멸된다

6단원

7단원 <상속>

- class A가 class B를 상속한다-> A에는 B의 멤버 변수, 멤버함수를 포함한다. A를 초기화 할 때, B의 멤버변수도 초기화 해줘야함.

-> class A: public B
{~~
A(인자들) : B(인자들)
{ A에 대한 내용} 같은 방법으로 B의 멤버를 초기화하면서 A의 멤버를 초기화 할 수 있음
?? 284쪽: 초기화랑 접근은 다른거?

- 유도 클래스의 생성자에서 기초 클래스에 대한 생성자 호출이 없으면 기초 클래스의 void 생성자가 호출된다.

*생성자에서 동적 할당한 메모리 공간은 소멸자에서 해제한다.

- 클래스의 멤버선언: private은 해당 클래스 내에서만 접근 가능, protected는 해당 클래스와 상속하는 클래스에서 접근 가능, public은 외부에서 접근가능

- 상속의 형태: protected 상속- protected보다 접근 범위가 넓은 멤버는 protected로 상속함. private도 이와 비슷. public은 private 이외 나머지 그대로

8단원

- 기초 클래스의 포인터로 유도 클래스의 객체를 가리킬 수 있음
- Base * ptr = new Derived(); 에서 ptr-> Derivedfunc(); 등 포인터 ptr에 대해 Derived 클래스의 함수를 호출하면 에러가 뜸

-> 포인터의 자료형을 기준으로 판단

- 유도 클래스의 포인터 변수에 기초 클래스로 정의된 포인터 변수를 넣는 것은 불가능
- 함수 오버라이딩: 객체 포인터에 함수를 호출할 경우 이름이 같은 기초, 유도클래스의 함수 중 해당 포인터의 클래스의 함수 호출
- 가상 함수: 키워드 virtual (예: virtual void func())

-> 가상함수는 함수 호출 시 포인터 변수가 실제로 가리키는 객체를 참조하여 그 객체의 함수를 호출 virtual 선언을 하면 해당함수를 오버라이딩 하는 함수도 차례로 가상함수가 됨

- 순수 가상함수: virtual func() const = 0; 으로 몸체를 정의하지 않은 불완전한 함수를 멤버함수로 하여 해당 클래스를 객체 생성목적으로 사용하지 않을 수 있음
- 가상 소멸자 virtual ~Classname(){..}: 원래의 소멸자는 포인터의 자료형이 갖는 소멸자만 소멸시키지만 가상소멸자는 포인터의 자료형을 상속하는 최하위 유도클래스 부터 순차적으로 소멸됨
- 위는 상속의 관계 뿐 아니라 참조의 관계에서도 성립 -> 유도 클래스의 객체를 기초 클래스의 참조자로 참조할 수 있음

9단원

- 가상 함수는 클래스마다 부여되는 가상함수 테이블에 저장되며 오버라이딩 된 가상함수의 주소정보(주로 기초 클래스의 함수)는 유도 클래스의 테이블에 저장되지 않음

10 단원

- 연산자 오버로딩: 함수 이름을 operator+ 로 정의하면 해당 객체사이에 +연산자를 넣으면 자동으로 함수가 호출됨. pos1+pos2 가 pos1.operator+(pos2)로 인식됨. 전역함수 또한 같이 취급
- 오버로딩이 불가능한 연산자: { ./ .* /:: /?: / sizeof / typeid/ static_cast/ dyanamic_cast/ const_cast/ reinterpret_cast}
- 멤버함수 기반으로 접근이 가능한 연산자: = () [] ->
- 연산자의 기본 기능을 변경하는 형태의 연산자 오버로딩은 불가
- 단항 연산자-> 함수와 같이 쓰이면 객체 앞에 연산자가 쓰이면 연산한 후 이를 대입, 뒤에 쓰이면 객체 대입 후 연산
- operator 함수 반환형의 const 선언을 하면 반환하는 객체는 값의 변경을 허용하지 않게됨
- 교환 법칙이 성립하려면 Point operator*(int times, Point& ref){} 처럼 참조형 인자를 같이 넣으면 됨.

11단원

* classname& operator+(const int& abc): 인자를 참조형으로 받는 이유는 처리속도 증가 반환형을 참조형으로 받는 이유는 참조값 반환하여 함수 내에서 이용한 주소를 그대로쓰지 않기위해(?)

- 대입연산자: 예) Classname& operator= (const Classname& ref) { member= ref.member; return *this;}
- 디폴트 대입연산자의 문제점 두 객체 중 하나의 객체의 멤버에 대해서 메모리 누수가 일어난다. 소멸될 때, 한쪽 객체를 소멸하면서 다른쪽 객체의 멤버가 소멸되며, 재소멸의 문제가 발생
- 유도 클래스의 대입연산자를 명시하지 않고 사용해도 기초 클래스의 대입연산자가 호출 안됨 (유도클래스 대입연산자 정의 안해도 디폴트 대입연산자 실행은 됨)
- 유도 클래스의 대입연산자 정의에서, 기초 클래스의 대입 연산자 호출문이 없으면 기초 클래스의 멤버변수는 복사 대상에서 제외됨.

461쪽 궁금한 점 : 매열을 멤버로 선언하는 경우에 `~~

- 객체의 저장을 위한 배열 클래스: int& operator[] (int idx){ retur arr[idx]} 와 int operator[] (int idx) const { return arr[idx]} 두 가지 클래스가 있을 때, arr[2] = 23등의 문장에서는 전자의 연산자가 호출됨.

12단원

- 헤더 파일 <string>에는 +, - 등의 연산이 string 클래스의 객체에 대해 오버로딩 되어있음.

13단원

- 함수 템플릿:

template <typename T>
T funcname(T num1, ~){ } 등을 함수 템플릿이라고 하며 funcname<int>(1,2) 등의 문장으로 호출할 수 있음. 템플릿이 만들어내는 함수를 템플릿 함수라고 함. <int> 생략하면 인자의 자료형을 인식하여 typename을 자동으로 정함

- 인자의 자료형이 두 가지인 경우, template <class T1, class T2> 등

으로 선언한 후 템플릿 정의

? return strlen(a) > strlen(b) ? a: b; 무슨 뜻?

- 함수 템플릿의 특수화: 특정 클래스에 대해서는 다른 함수를 호출 하고 싶으면 template <>// char* func(char* a, char* b){} 등으로 특수화 할 수 있음
- 클래스 템플릿: template <typename T> // class Point // ~// private: // T xpos, ypos; 등으로 정의 -> 객체를 선언할 Point<int> pos1(1,3); 처럼 선언하면 됨
- 클래스 템플릿에서 멤버함수를 외부에서 정의 하고 싶으면 각각의 함수에 template <typename T>// 반환형 Classname<T>:: funcname() 처럼 쓰기

14단원

- 클래스 템플릿의 특수화: 비슷하게 template <>// class Classname <char*> 등의 형식으로 사용
- 템플릿 매개변수에는 변수 선언 가능: template <typename T, int len> -> 호출할 때는 Classname< int, 2> obj; 등으로 호출가능 -> 디폴트 값도 지정 가능