**[Chapter 01 C언어 기반의 C++ 1]**

1-1. printf와 scanf를 대신하는 입출력방식

1) C++에서는 입출력을 std::cout, std::cin을 통해 수행하며 이를 위해서는 #include<iostream>을 선언해야 한다.

2) std::cout<<출력대상 형태로 출력을 할 수 있으며 <<를 계속 사용하여 여러 대상을 출력할 수 있다. 출력대상 위치에 endl를 넣으면 줄넘김이 된다.

3) std::cin>>변수 형태로 입력을 할 수 있다.

1-2. 함수 오버로딩 (Function Overloading)

1) 함수 오버로딩 : C언어와 다르게 매개 변수의 선언 형태가 다른 경우 동일한 이름의 함수 정의도 허용한다.

2) 매개변수 자료형이 다른 경우 : int myFunc(char c) / int myFunc(int n)는 다른 함수로 허용한다.

3) 매개변수의 개수가 다른 경우 : int myFunc(int n) / int myFunc(int n1, int n2)는 다른 함수로 허용한다.

1-3. 매개변수의 디폴트 값(Default Value)

1) C++에서는 함수의 매개변수에 디폴트 값을 설정할 수 있다. 매개변수에 디폴트 값을 설정한 경우 인자가 전달되지 않으면 매개변수의 값이 디폴트 값이 된다.

2) 디폴트 값이 설정되어 있다면 함수를 호출할 때 일부만 전달하거나 전부 전달하지 않아도 된다.

3) 디폴트 값은 반드시 맨 오른쪽에 있는 매개변수부터 채워 넣어야 한다.

ex) int cal(int num1, int num2 = 7) (o) / int cal(int num1 = 5, int num2) (x)

1-4. 인라인(inline) 함수

1) 인라인 함수는 컴파일 과정에서 함수명 대신 함수 몸체가 대신 들어가는 형태의 함수이다. 매크로 함수를 떠올리면 된다.

2) 장점 : 매크로 함수에 비해 정의가 훨씬 쉬우며 최적화도 잘되어 있다.

3) 단점 : 매크로 함수와 다르게 자료형을 사전에 정의해야 해서 자료형 의존도가 높다.

>> 추후에 나오는 템플릿을 이용해 개선이 가능하다.

1-5. 이름공간(namespace)

1) 이름공간(namespace)는 라이브러리 사용 혹은 협업간에 이름이 충돌하는 것을 방지하기 위해 공간을 구분할 수 있게 해주는 문법이다.

2) namespace안에 있는 함수나 변수는 namespace이름 :: 함수나변수 와 같은 형태로 접근이 가능하다.

3) 이름 공간의 명시가 필요없다고 생각하는 경우 using 키워드를 통해 namespace이름:: 이 과정을 생략할 수 있다.

ex) using namespace std : std이름의 namespace에 접근할 때는 :: 접근 과정을 생략한다.

ex) using std::cout : std안에 있는 cout에 접근할 때는 :: 접근 과정을 생략한다.

**[Chapter02 C언어 기반의 C++ 2]**

2.1. Chapter 02의 시작에 앞서

1) const int num : num을 상수화

2) const int \*ptr = &val : 포인터 ptr으로 val값 수정 불가능

3) int const \*ptr = &val : 포인터 변수 ptr이 상수화

2.2. 새로운 자료형 bool

1) bool은 true / false를 저장하는 자료형이다.

2) true : 참 / false : 거짓을 의미한다.

3) true / false는 각각 1/0이 아닌 참과 거짓을 나타내는 1바이트 크기의 자료형일 뿐이다.

2.3. 참조자(Reference)의 이해

1) 참조자 : 변수를 선언할 때 변수 명 앞에 &를 붙여 선언한다. 이미 할당된 메모리 영역에 또 다른 이름을 선언하는 것이다.

2) 참조자는 선언과 동시에 누군가를 참조해야하며 참조의 대상을 바꾸는 것도 불가능하다.

3) 참조자는 상수를 선언하거나 NULL로 초기화할 수 없다.

2.4. 참조자(Reference)와 함수

1) 참조자는 C에서 Call-by-reference를 수행할 때 사용하던 포인터를 대체할 수 있다.

2) const로 참조자를 선언하면 해당 참조자로는 값을 변경할 수 없다.

3) 매개변수를 const참조자로 선언하면 함수 선언부만 보더라도 해당 함수에서 전달한 인자의 값을 변경하지 않는다는 것을 알릴 수 있으며 상수도 입력받을 수 있다.

2.5. new & delete

1) new와 delete는 기존 C언어의 malloc, free 역할을 수행한다. new, malloc이 다른 C++ 기능과 상호작용하기 때문에 malloc, free는 사용하지 않는다.

2) new : 메모리 공간을 할당한다. int \*ptr = new int 와 같이 사용한다. 배열은 int \*ptr = new int[3]와 같이 사용한다.

3) delete : 할당된 메모리 공간을 반환한다. delete ptr와 같이 사용한다. 배열은 delete []ptr와 같이 사용한다.

2.6. C++에서 C언어의 표준함수 호출하기

1) 기존 C언어에서 사용하던 라이브러리는 C언어 라이브러리에서 .h를 빼고 앞에 c를 붙인다.

2) 예시 : include<stdio.h> -> include<cstdio>

3) C라이브러리와의 호환성을 위해 이렇게 바꾸었으며 C++ 문법에 맞추어 라이브러리를 변경했기에 위와같은 형태로 라이브러리를 사용해야 한다.

**[Chapter03 클래스의 기본]**

3.1. C++에서의 구조체

1) 구조체 안에 함수를 삽입하면 데이터를 전달하지 않아도 구조체 안에있는 데이터에 접근이 가능하다.

2) 특정 구조체만을 위한 enum상수를 선언하기 위해서는 두가지 방법이 있다. 구조체 안에 직접 선언하는 방법과 이름공간을 이용해 상수가 사용되는 영역을 명시하는 방법이 있다.

3) 구조체는 클래스의 일종이다.

3.2. 클래스(Class)와 객체(Object)

1) 클래스는 기본적으로 외부에서 접근을 허용하지 않는 변수와 함수의 집합체이다.

- 멤버변수 : 클래스를 구성하는 변수

- 멤버함수 : 클래스를 구성하는 함수

2) 클래스는 선언과 동시에 초기화가 불가능하며 변수와 함수의 접근 허용 범위를 별도로 선언해야한다.

- private: 클래스 내에서만 접근 허용

- public: 어디서나 접근 가능

- protected: 상속관계 혹은 유도클래스에서만 접근 허용

3) 클래스로 선언된 변수를 객체라고 한다.

3.3. 객체지향 프로그래밍의 이해

1) 객체지향 프로그래밍은 현실에 존재하는 것을 객체로 등장시키고 객체의 행위를 실체화 시키는 프로그래밍이다.

2) 객체는 상태(state)와 행동(behavior)으로 구성되어있다.

- 상태(state) : 객체의 상태정보, 데이터 등을 의미한다. 주로 멤버변수로 표현된다.

- 행동(behavior) : 객체의 행동, 기능 등을 의미한다. 주로 멤버함수로 표현된다.

3) C++에서 정의하는 객체 생성 방법은 두가지이다.

- 일반적인 변수의 선언 방식 : ClassName objname;

- 동적 할당 방식 : ClassName \*ptrObj = new ClassName;

**[Chapter 04 클래스의 완성]**

<4-1. 정보은닉(Information Hiding)>

1. 정보 은닉은 멤버 변수를 private로 선언한 뒤 해당 변수에 접근하는 별도의 함수를 선언하여 안전하고 제한된 형태로 멤버 변수에 접근하도록 유도하는 방식이다.

2. 엑세스 함수는 클래스 외부에서의 멤버변수 접근을 목적으로 정의된 함수들을 의미한다. 당장 호출하지 않아도 정의를 해두는 것이 좋다.

3. 함수 뒤에 const가 붙는 경우 멤버 변수의 변경을 허용하지 않는다. 또한, const 함수내에서는 const함수만 호출이 가능하다.

> 값의 변화가 없는 경우 const를 붙여 코드의 안정성을 높이고 함수의 의도를 직관적으로 보이게 하자.

<4..2 캡슐화(Encapsulation)>

1. 캡슐화는 하나의 목적을 이루기 위한 기능들을 한 곳에 모아 코드를 간결하게 하고 객체들의 상호작용을 쉽게 파악할 수 있게 하는 것이다.

2. 캡슐화를 함으로써 프로그램의 복잡도를 낮추고 객체의 활용을 쉽게 하며 진행순서가 중요한 경우 구조가 무너짐을 방지할 수 있다.

3. 캡슐화의 범위를 결정하는 것은 정답이 없고 프로그램의 성격과 특성에 따라 달라지기에 많은 경험이 필요하다.

<4-3. 생성자(Constructor)와 소멸자(Destructor)>

1. 생성자 : 객체 생성시 딱 한 번 호출되는 함수

- 형태 : 클래스 이름과 함수 이름이 같으며 반환형을 선언하지 않는다.

- 용도 : 객체의 생성과 동시에 멤버 초기화

- 특징

> 생성자가 매개변수를 가지고 있으면 객체를 정의할 때 생성자의 인자를 넣어야 한다.

> 오버로딩, 디폴트 값 설정이 가능하다.

> 생성자가 없어도 자동으로 디폴트 생성자가 삽입된다.

- 멤버 이니셜 라이저 : 생성자에서 멤버 선언과 동시에 초기화 할 수 있는 기능

> 형태 : 생성자 함수명 뒤에 : 대상(초기화값), 대상(초기화값), ...

ex) SimpleClass(int num) : val(30), ...

> 멤버를 초기화 할 때 기본적으로 사용해야 한다.

2. 소멸자 : 객체 소멸 시 딱 한 번 호출되는 함수

- 형태 : 클래스 이름 앞에 ~가 붙고 반환형과 매개변수 모두 없는 형태

ex) ~SimpleClass()

- 용도 : 주로 할당한 메모리를 반환할 때 사용한다.

- 특징

> 소멸자가 없어도 자동으로 디폴트 소멸자가 삽입된다.

> 함수 오버로딩, 디폴트 값은 불가능하다.

3. 객체의 생성 과정

1) 메모리 공간 할당

2) 이니셜라이저를 이용한 멤버의 초기화

3) 생성자 몸체 부분 실행

> 이 과정이 필요하므로 디폴트 생성자라도 넣는다.

<4-4. 클래스와 배열 그리고 this 포인터>

1. 객체 배열의 선언 방식은 크게 두 가지 이다.

- 객체 배열 선언

> 형태 : ClassName arr[10];

- new 연산자를 통한 동적 할당

> 형태 : ClassName \*ptrArr = new ClassName[10];

> 후에 delete로 반환을 해야 한다.

2. 객체 배열 선언 시 주의할 점

- 생성자를 호출하지만 인자를 전달할 수 없으므로 인자를 받지 않는 생성자가 존재해야 배열을 선언할 수 있다.

- 배열로 선언한 객체를 초기화하고 싶다면 하나씩 별도로 초기화 해야 한다.

- 배열이 소멸할 때 각 객체의 소멸자가 동시에 호출된다.

3. this 포인터 : 객체 자신의 주소값을 의미한다. 즉, 자기 자신을 가리킨다.

**[Chapter05 복사 생성자(Copy Constructor)]**

<5-1. '복사 생성자'와의 첫 만남>

1. 묵시적 변환 : 객체 생성 및 초기화를 할 때 '='를 사용한 경우 인자가 하나인 생성자가 존재할 때 '()'로 알아서 변환되는 것

ex) SimpleClass obj = 3 -> SimpleClass obj(3) 묵시적 변환

2. 복사 생성자 : 같은 객체를 인자로 받아 대응하는 멤버 변수끼리 값을 복사하는 생성자

ex) SimpleClass(const SImpleClass &copy)와 같은 생성자

- 복사 생성자가 있을 경우 SimpleClass obj = copy와 같은 문장이 성립된다.

- 복사 생성자에서 인자를 받을 때는 반드시 &를 사용하며, const를 사용하는 것이 좋다.

3. 디폴트 복사 생성자 : 복사 생성자가 따로 정의하지 않은 경우 알아서 맴버 대 맴버 복사를 진행하도록 생기는 생성자

- 뒤에 설명할 '깊은 복사'가 필요하지 않으면 단순 맴버 대 맴버 복사에서는 문제가 발생하지 않는다.

<5-2. '깊은 복사'와 '얕은 복사'>

1. 얕은 복사 : 맴버 대 맴버로 값만 복사하는 경우

2. 얕은 복사의 문제점 : 포인터 혹은 참조자가 있을 경우 같은 대상을 참조하도록 복사하게 된다. 이는 예상치 못한 결과를 발생시킨다.

ex) ptr1을 복사한 ptr2에서 값을 변경했더니 ptr1까지 값이 변한다.

3. 따라서, 이러한 문제를 해결하기 위해 참조의 대상까지 고려한 복사가 '깊은 복사'이다. (참조의 대상도 추가하는 복사)

<5-3. 복사 생성자의 호출 시점>

1. Call-by-value(값 전달)방식의 함수 호출 과정에서 객체를 인자로 전달하는 경우 해당 함수가 호출되는 순간 매개변수에 객체를 복사하면서 복사 생성자가 호출된다.

2. 함수의 반환형이 객체인데 참조형이 아닌 경우 함수를 호출한 곳에 반환하는 객체의 복사본을 임시 객체로 전달한다.

3. 임시 객체 : 객체의 정보를 임시로 저장하는 공간으로 다음 행에서 바로 소멸한다.

- 클래스의 이름을 클래스 외부에서 사용할 경우 임시 객체이다.

ex) SimpleClass(30);

- 단, 참조자로 임시객체를 참조하는 경우에는 바로 소멸하지 않는다.

**[Chapter06 friend와 static 그리고 const]**

<6-1. const와 관련해서 아직 못다한 이야기>

1. const로 객체를 선언하면 상수화되며 선언 이후로 데이터 변경을 허용하지 않는다는 의미이다.

2. 상수화된 객체는 const 선언이 된 멤버 함수만 호출이 가능하다.

3. const 선언 유무만으로 함수 오버로딩이 되며 const객체는 const함수를, 일반 객체는 일반 함수를 호출하게 된다.

<6-2. 클래스와 함수에 대한 friend 선언>

1. friend은 friend선언을 한 클래스의 private 멤버를 friend로 선언한 대상(객체 혹은 함수)가 접근할 수 있도록 허용한다는 의미이다.

2. friend선언은 정보은닉을 무너뜨리기에 가급적 사용하지 않는 것이 좋다.

3. 함수명만 미리 선언하듯이 클래스도 이름만 미리 선언하면 다른 곳에서도 해당 클래스가 있다는 사실을 인지시킬 수 있다.

<6-3. C++에서의 static>

1. 클래스의 멤버에 static선언이 가능하다. static선언시 특징은 다음과 같다.

- 객체의 생성과 소멸에 영향을 받지 않고 처음부터 존재해있고 초기화되지 않는다.

- 클래스 안에서 있지만 객체 안에 존재하는 것이 아닌 객체에게 접근할 수 있는 권한을 준 것이다.

> 객체에게 허용된 전역 변수/함수라고 생각하면 편하다.

- 객체안에 존재하는 것이 아니기에 클래스 이름으로 접근이 가능하다.

2. static 멤버변수는 다음의 특징을 가진다.

- 생성자로 초기화하면 안된다. 객체가 생성될 때마다 초기화되므로 static의 의미가 퇴색된다.

- 클래스 외부에서 초기화 하는 것이 일반적이다.

ex) int SimpleClass::staticVar = 2;

- public인 경우 클래스 이름으로 접근이 가능하다.

ex) SimpleClass::staticVar

- const 선언이 된 경우 클래스 내에서 일반 변수처럼 초기화가 가능하다.

3. static 멤버함수는 다음의 특징을 가진다.

- 객체의 멤버가 아니기 때문에 객체의 멤버 변수나 함수에 접근할 수 없다.

- 따라서, static 멤버변수와 멤버함수만 호출이 가능하다.

- public 선언시 클래스 이름으로 접근이 가능하다.

**[Chapter07 상속(Inheritance)의 이해]**

<7.1. 상속에 들어가기에 앞서>

1. 상속은 문법적 요소보다 상속이 가져다 주는 이점, 적용해야 할 때를 아는 것이 중요하다.

2. 좋은 소프트웨어가 되기 위해 중요하게 생각해야 하는 요소는 다음과 같다.

- 요구사항 변경에 대응하는 프로그램의 유연성

- 기능의 추가에 따른 프로그램의 확장성

3. 즉, 좋은 프로그램은 추가 요구에 쉽게 대응할 수 있어야 하며 이를 위해 필요하게 된 것이 상속이다.

<7.2. 상속의 문법적인 이해>

1. 상속은 다른 클래스의 멤버를 물려 받아 멤버 변수와 멤버 함수를 포함하게 되는 것이다.

- 정의 방법은 'class 클래스이름 : public(protected/private) 상속을주는클래스이름'처럼 한다.

- 상속을 해주는 클래스는 기초 클래스, 받는 클래스는 유도클래스라고한다.

2. 기초클래스의 멤버를 초기화 할 때는 유도클래스 생성자의 이니셜라이저에서 기초클래스의 생성자를 호출하는 방식으로 한다.

ex) (생성자)SimpleClass(int n1, int n2) : baseClass(n1), num(n2) < 여기서 baseClass가 기초클래스 이름 즉, 기초클래스의 생성자

- 클래스의 멤버는 해당 클래스의 생성자를 통해서 초기화해야 한다는 기본 원칙이 적용된 것이며 이는 소멸자도 마찬가지이다.

3. private선언된 멤버는 유도클래스에서도 접근할 수 없다.

- 즉, 정보의 은닉은 상속관계에서도 이루어진다.

<7.3. protected 선언과 세 가지 형태의 상속>

1. protected : 유도클래스에서만 접근이 가능하게 하는 지시자로 private, public 중간이라고 보면 된다.

2. 세 가지 형태의 상속

- public 상속 : private를 제외한 모두를 상속한다. 가장 일반적이다.

- protected 상속 : public 멤버변수를 protected 멤버 변수로 상속한다. 나머지는 public과 같다.

- private 상속 : public, protected 멤버변수를 private로 상속받는다.

> private 상속이 이루어진 클래스를 상속하면 이미 모두 private이기에 상속이 의미가 없다.

<7.4. 상속을 위한 조건>

1. 상속의 기본 조건은 IS-A관계가 되어야 한다.

ex) 스마트폰은 전화기이다. 여기서 스마트폰이 유도클래스, 전화기가 기초클래스이다.

2. 이러한 관계는 모든 단계의 관계에서 이루어져야 한다.

ex) 갤럭시폰은 스마트폰이다 / 스마트폰은 전화기이다 / 갤럭시폰은 전화기이다. (옳은 상속)

3. HAS - A 관계는 상속이 가능하긴 하지만 유연성과 확장성이 떨어지기에 하지 않는 것이 좋다.

ex) 경찰은 총을 가진다. 여기서 경찰이 유도클래스, 총이 기초클래스가 될 수 있다.

> 하지만, 총을 들지 않는 경찰을 표현하려면 큰 수정이 필요하기에 유연성이 떨어진다.

**[Chapter08 상속과 다형성]**

<8.1. 객체 포인터의 참조관계>

1. 클래스를 기반으로도 포인터 변수를 선언할 수 있다.

2. 클래스를 기반으로 선언한 포인터 변수는 해당 클래스의 유도 클래스 객체도 가리킬 수 있다.

ex) Person를 상속하는 Student, Student를 상속하는 HighSchoolStudent가 있다고 하자

Person \*ptr로 선언한 경우 ptr은 Student, HighSchoolStudent 객체 모두를 가리킬 수 있다.

> 이를 통해 유도 클래스의 포인터를 따로 선언할 필요가 없어 유연성과 확장성을 확보한다.

3. 함수 오버라이딩 : 기초클래스에서 상속받는 함수를 유도 클래스에서 동일한 이름과 형태로 재정의 하는 것이다.

> 유도클래스에서 해당 함수를 호출할 경우 기초클래스의 함수가 아닌 유도클래스에서 재정의된 함수가 호출되게 된다.

<8.2. 가상함수>

1. 클래스형 포인터 변수는 유도클래스도 가리킬 수 있으나, 포인터 변수 연산의 근거는 선언한 자료형을 기준으로 한다.

- 멤버를 호출할 때 유도클래스를 가리키는 포인터여도 포인터 변수를 선언한 기초 클래스의 멤버만 가져올 수 있다.

- 대입 연산을 진행할 때 유도클래스를 카리키는 포인터여도 유도클래스로 선언한 포인터에 대입 연산은 불가능하다.

ex) Base \*bptr = new Derived(); / Derived \*dptr = bptr (x)

- 오버라이딩된 함수가 존재해도 처음 선언한 클래스의 함수를 가져온다. 즉, 유도클래스의 함수가 아닌 기초클래스의 함수를 가져온다.

2. 가상함수(virtual) : 클래스형 포인터 변수가 오버라이딩된 함수를 호출할 때 자료형이 아닌 가리키는 대상의 함수를 가져오게 하는 개념이다.

- 함수 앞에 virtual을 붙이면 해당 함수를 오버라이딩한 함수 모두 가상함수가 된다.

- 확장성과 유연성을 확보하기 위해 등장하였다.

- 가상함수 끝에 = 0을 붙이면 순수 가상함수가 되며 해당 클래스는 기초 클래스로서의 의미만 가지는 추상 클래스가 된다.

3. 다형성(Polymorphism) : 모습은 같은데 형태는 다른, 즉 같은 문장이어도 다른 결과를 도출해내는 가상함수의 특성을 의미한다.

<8.3. 가상 소멸자와 참조자의 참조 가능성>

1. 클래스형 포인터 변수는 자료형이 따라가므로 소멸자도 의도한 대로 실행시키기 위해 앞에 virtual을 붙인다. (가상 소멸자)

2. 참조자도 포인터와 같이 접근할 수 있으며 선언한 자료형을 따른다, virtual을 따른다 등 포인터와 성질이 동일하다.

**[Chapter09 가상(virtual)의 원리와 다중상속]**

<9.1. 멤버함수와 가상함수의 동작원리>

1. 멤버함수는 객체 안에 존재하는 것이 아닌 별도의 메모리 주소에 존재한다.

> 객체는 멤버함수가 존재하는 메모리 주소만 저장하는 방식이다.

2. 객체에 가상함수가 존재할 경우 유도클래스에서는 오버라이딩된 가상함수의 주소값만 저장하고 기초클래스의 주소값은 저장하지 않는다.

> 이러한 원리로 마지막에 오버라이딩한 유도 클래스의 멤버함수를 불러온다.

3. 이렇게 한 번 주소를 거쳐야 하기 때문에 C++가 C보다는 느리다.

> 그러나 이렇게 하는 것이 단점에 비해 장점이 크기에 유용하게 활용된다.

<9.2. 다중상속(Multiple Inheritance)에 대한 이해>

1. 다중상속은 둘 이상의 클래스를 동시에 상속하는 것이다.

ex) class AAA : public BBB, protected CCC { ... };

2. 득보다 실이 되는 경우가 많기에 사용을 자제해야 한다.

3. 다중상속의 문제점과 해결점은 다음과 같다.

1) 기초 클래스들에서 동일한 이름의 멤버가 존재하는 경우 문제 발생

- 어느 기초클래스에서 멤버를 가져올 것인지 정의하여 해결

ex) CCC::SimpleFunc();

2) 기초 클래스들이 더 상위의 같은 기초클래스 상속할 경우 그 상위 기초클래스의 멤버를 두 번 이상 상속받음

- 가상 상속을 사용해 해결

**[Chapter10 연산자 오버로딩1]**

<10.1. 연산자 오버로딩의 이해와 유형>

1. 연산자 오버로딩 : 객체에서도 연산자를 통한 함수호출이 가능하게 하는 것

> 방법 : '리턴형 operator연산자(연산대상)' 형태로 함수를 선언한다.

ex) 멤버함수의 경우 : Point oprator+(const Point &ref)

전역함수의 경우 : Point oprator+(const Point &pos1, const Point &pos2)

- 멤버함수 혹은 전역함수로 가능하나 전역함수의 경우 객체 접근을 위해 friend 선언을 해두어야 한다.

2. 컴파일러는 객체의 연산을 볼 때 연산자를 .operator연산자(연산대상) 형식으로 치환해서 해석한다.

ex) pos1 + pos2는 pos1.oprator+(pos2)로 바뀌는 것이다.

3. 연산자 오버로딩의 주의사항

1) const로 선언하여 피연산자 변경 방지

2) 연산자의 우선순위와 결합성은 바뀌지 않음 ('+'보다'\*'가 우선)

3) 매개변수에 디폴트 값 설정 불가능

4) 기본 자료형의 연산자는 간섭 불가능

<10.2. 단항 연산자의 오버로딩>

1. 단항 연산자는 피연산자가 하나이기에 매개변수를 하나씩 줄이면 된다.

ex) 멤버함수 : Point operator++()

ex) 전역함수 : Point operator++(Point &ref)

2. 전위증가/감소 형태의 단항 연산자

- 증가하고자 하는 멤버를 증가시킨 후 return \*this를 한다. 자기 자신의 참조값을 반환한다는 의미이다.

- Point.cpp의 Point Point::oprator++()함수 참고

3. 후위증가/감소 형태의 단항 연산자

- 전위연산자와 구별을 위해 매개변수 int를 추가한다. 또한, 리턴형은 const 객체명& 으로 해둔다.

- 먼저 자기자신의 객체를 저장한 후 멤버를 증가시키고 저장해둔 객체를 반환한다.

> 후위 연산은 다른 연산이 끝난 후 자기 자신을 증가시키기에 변화하기 전 객체를 리턴한다.

- Point.cpp에 있는 const Point& Point::operator(int)함수 참고

<10.3. 교환법칙 문제의 해결>

1. 연산자 오버로딩을 이용하면 자료형이 다르더라도 연산을 진행할 수 있다.

2. 교환법칙 문제 해결을 위해서는 객체가 아닌 자료형도 연산자 오버로딩을 시켜야 한다.

- 전역함수 형태로 매개변수를 여러 개 받아 해결한다.

- Point.cpp의 Point oprator+(int numm, const Point &pos)를 참고

**[Chapter11 연산자 오버로딩2]**

<11.1 반드시 해야 하는 대입 연산자의 오버로딩>

1. 대입 연산자는 복사 생성자와 매우 유사하다.

- 디폴트 대입 연산자의 존재 (디폴트 복사 생성자)

- 디폴트 대입 연산자는 맴버 대 맴버 (얕은 복사)

- 깊은 복사가 필요하면 따로 정의해야 한다.

- 대입연산자는 복사 생성자와 다르게 이미 초기화가 끝난 두 객체를 대상으로 한다는 것이 큰 차이이다.

2. 복사 생성자와 마찬가지로 참조형 변수가 있을 때 얕은 복사는 문제를 일으킨다.

- 같은 대상을 가리키는 문제가 발생한다.

- 심지어, 대입 연산자는 이미 초기화된 객체를 대상으로 하므로 원래 가리키던 주소값을 잃게 되어 소멸이 불가능해진다.

> 이는 메모리 누수로 이어진다.

- 따라서, 깊은 복사를 따로 정의해야 하는 경우가 생긴다.

3. 대입 연산자는 복사 생성자와 다르게 유도 클래스에서 대입 연산자가 호출되어도 기초 클래스의 대입 연산자는 호출되지 않는다.

- 기초 클래스의 멤버에 대한 대입을 유도 클래스 멤버 함수에 따로 명시해야 한다는 의미이다.

<11.2. 배열의 인덱스 연산자 오버로딩>

1. []연산자도 연산자 오버로딩이 된다. 이를 통해 경계검사를 실시하지 않는 []연산자의 단점을 보완할 수 있다.

- 형태 : operator[]()

2. 연산자 오버로딩 함수 또한 const함수에 대응할 수 있어야 하므로 const 오버로딩을 해두는 것이 좋다.

3. 객체의 주소값들을 저장하는 주소값 저장 배열 기반 클래스는 복사 생성자, 대입 연산자의 얕은 복사를 신경 쓸 필요가 없어 용이하다.

<11.3. 그 이외의 연산자 오버로딩>

1. new, delete 연산자 오버로딩

- new

> 기본 형태

void \* operator new(size\_t size)

{

void \* adr = new char[size];

return adr;

}

> 기본 형태 안에서 오버로딩이 이루어져야 하며 특히 정의부분은 바뀌어서는 안된다.

- delete

> 기본 형태

void operator delete(void\*adr)

{

delete []adr;

}

> 기본 형태 안에서 오버로딩이 이루어져야 하며 특히 정의부분은 바뀌어선 안된다.

> 배열형인 경우 정의부분 delete뒤에[]만 추가하면 된다.

2. \*, -> 오버로딩

- \* 오버로딩 : 자료형& operator\*() { ...; return \*this; }

- -> 오버로딩 : 자료형\* operator->() { ...; return this; }

3. (), 형변환 오버로딩

- () 오버로딩 : 반환형 operator()(매개변수)

> 객체를 함수처럼 사용 가능

- 형변환 오버로딩 : operator 변환형 () { ... };

**[Chapter12 String 클래스의 디자인]**

1. 이번 챕터의 목적은 string 클래스의 직접 구현이다.

2. string 클래스는 C++ 표준 라이브러리에 정의되어 있는 문자열의 처리를 목적으로 정의된 클래스이다.

3. 문자의 연산, 입출력 등을 지원한다.

**[Chapter13 템플릿(Template) 1]**

<13.1. 템플릿(Template)에 대한 이해와 함수 템플릿>

1. 함수 템플릿은 자료형이 결정되어 있지 않은 함수이다. 따라서, 호출시 자료형을 결정한다.

> 기본 형태(T의 이름은 상관없으며 typename -> class로 교체 가능)

template <typename T>

T add(T num1, T num2)

{

return num1 + num2;

}

> 호출 방식 : 함수명<자료형>(매개변수);

ex) add<double>(2.5, 9.5)

2. 함수 템플릿 선언의 특징

> 매개변수에 템플릿 타입을 사용하지 않아도 된다.

ex) template <typename T>

void ShowData(double num)

{

cout<<(T1)num<<endl;

}

> 템플릿 타입이 명시할 때 쉼표를 이용해 둘 이상도 가능하다.

ex) template <typename T1, typename T2>

void print(T1 num1, T2 num2)

{

cout<<num1<<" "<<num2<<endl;

}

3. 특정 자료형인 경우 다른 기능을 수행하고 싶다면 template<>를 선언하고 함수명 뒤에 <자료형>을 붙여 정의한다.

ex) template <>

char\* add<char\*>(char\* a, char\* b)

{

strcat(a, b)

return a;

}

<13.2. 클래스 템플릿(Class Template)>

1. 함수 템플릿과 유사하게 클래스도 템플릿으로 정의가 가능하다.

> 기본 형태

template <typename T>

class Point

{

private:

T xpos, ypos;

public:

Point(T x=0, T y=0) : xpos(x), ypos(y) { }

}

> 호출 방식

Point<int>(2, 3)

2. 기존 클래스처럼 선언은 클래스 안에서, 정의는 클래스 밖에서 할 수 있다. 단, T에 대한 정의와 클래스 명 뒤에 <T>가 붙어야 한다.

ex) template <typename T>

T SimpleTemplate<T>::SimpleFunc(const T& ref)

{ ... }

3. 헤더파일과 소스파일을 나눌 경우 컴파일 과정에서 헤더파일에 함수 몸통 부분이 와야하는데 (매크로처럼) 컴파일은 파일단위로 하기 때문에 (다른 파일을 참조하지 않기 때문에) 소스파일에 있는 템플릿 클래스가 만들어 질 수 없다.

> 헤더파일에 선언, 정의를 모두 넣는 방법이 있다.

> 정의에 대한 내용이 있는 소스파일을 include 한다.

**[Chapter 14 템플릿(Template) 2]**

<14.1. Chapter 13에서 공부한 내용의 확장>

1. 템플릿 클래스는 결국 이름이 독특한 객체일 뿐이다.

2. 따라서, 템플릿 클래스를 자료형으로 하는 또 다른 템플릿 클래스도 가능하다.

ex) BoundCheckArray<Point<int>> ..

3. 그 외에 객체에 적용할 수 있는 것들을 모두 적용할 수 있다.

> typedef로 이름 간편히 바꾸기

ex) typedef Point<int> Point\_int;

> 일반함수의 인자로 받기

ex) Point<int> exampleFunction(const Point<int>&pos1)

<14.2. 클래스 템플릿의 특수화(Class Template Specialization)>

1. 함수 템플릿과 마찬가지로 특수화를 할 수 있다.

2. template<>를 명시한 후 클래스명 뒤에 <자료형>을 붙여 특수화한다.

ex) template<>

class exampleClass<int>

{

...

};

3. 템플릿 매개변수가 많을 때 부분 특수화가 가능하다.

- 이때는 특수화가 아닌 자료형은 기존처럼 템플릿형으로 선언한다.

ex) template<typename T1, typename T2>인 클래스 템플릿이 있으면

template<typename T1>

class exampleClass<T1, int>

{

...

}

<14.3. 템플릿 인자>

1. template<typename T>에서 결정되지 않은 자료형 T를 템플릿 매개변수라고 한다.

2. exampleFunction<int>(2, 3)에서 템플릿 매개변수로 전달되는 자료형 정보 int를 템플릿 인자라고 한다.

3. 템플릿 매개변수에는 변수가 올 수 있다.

ex) template<typename T, int len>

- 단, 변수로 선언한 템플릿 매개변수는 템플릿 인자로 다른 값이 오면 다른 자료형으로 인식한다.

ex) <int, 5>와 <int, 7>은 다른 함수이다. 따라서, <int, 5>로 대입연산자 오버로딩할 경우 <int, 7>로 대입연산은 불가능하다.

- 변수로 선언한 템플릿 매개변수에는 디폴트 값을 지정할 수 있다.

<14.4. 템플릿과 static>

1. 함수 템플릿에 static 지역 변수를 선언한 경우 템플릿 인자가 같은(자료형이 같은) 템플릿 함수만 자료형을 공유한다.

- <int>함수의 static 변수와 <double>함수의 static변수는 다르다.

2. 클래스 템플릿도 static 지역 변수가 있을 경우 템플릿 인자가 같은 클래스끼리만 static 변수를 공유한다.

- 이러한 이유는 자료형이 바뀜에 따라 이름이 독특한 함수를 여러 선언하는 것과 유사하기 때문이다.

3. template<>는 템플릿의 일부 혹은 전부를 정의한다고 컴파일러에게 알리는 역할이다.

- <> 안에 템플릿 매개변수 T에 대한 정보가 필요하면 넣는 것이다.

**[Chapter15 예외처리(Exception Handling)]**

<15.1. 예외상황과 예외처리의 이해>

1. 예외 : 문법적인 에러가 아닌 프로그램 논리에 맞지 않는 상황을 말한다.

2. if문을 통한 예외처리를 하면 if문이 조건문인지, 예외처리를 위한 코드인지 한 눈에 파악이 불가능하여 유지보수성이 떨어진다.

3. C++에서 제공하는 예외처리 메커니즘 try-catch-throw를 사용하면 if문과 다르게 예외처리를 위한 코드라는 것을 바로 파악이 가능하여 유지보수성을 향상시킬 수 있다.

<15.2. C++의 예외처리 메커니즘>

1. 예외처리의 핵심 키워드

- try : {}블록 안에서 예외가 발생하면 try 블록안에 문장을 중단시키고 catch문을 실행시킨다.

- catch : 예외발생시 처리하는 코드를 담는다. 예외와 관련된 데이터를 함수의 매개변수처럼 받는다.

- throw : 예외가 발생한 것을 알려 catch문을 실행하도록 한다. catch문에 예외와 관련된 데이터를 보낸다.

ex)

try

{

...

throw exceptionData;

...

}

catch(int data)

{

...

}

2. 예외처리 과정의 흐름

try문 실행 > throw문을 통해 예외발생을 인지>try문 중단 > catch문 실행 > catch문 이후 코드 실행

3. 예외처리 주의점

- try{ ... } 바로 뒤에 catch(){ ... }가 와야한다.

- 예외발생시 남은 try문 뒤를 실행하지 않는다.

<15.3. Stack Unwinding(스택 풀기)>

1. Stack Unwinding : 함수에서 예외가 발생했으나 함수안에 예외처리가 없을 경우 함수를 즉시 중단하고 함수를 호출한 곳으로 예외를 전달한다.

- 예외를 전달하면서 함수 스택이 반환되기 때문에 이러한 이름이 붙었다.

- main함수에서도 예외처리가 안된다면 프로그램을 종료시키는 terminate 함수가 호출된다.

- throw의 자료형과 catch의 자료형이 다른다면 예외를 처리할 수 없는 것으로 판단한다.

2. 하나의 try 블록 뒤에 다수의 catch문을 짤 수 있다.

- 여러 자료형에 대응할 수 있게 된다.

3. 함수 정의 뒤에 throw(자료형1, 자료형2, ...)을 작성하여 예외를 명시할 수 있다.

- 명시한 자료형 외의 자료형이 예외로 전달되면 프로그램을 종료시킨다.

- 프로그램의 실수를 방지하고 어떤 예외가 발생하는지 한 눈에 알아볼 수 있어 유지보수성도 늘어난다.

<15.4. 예외상황을 표현하는 예외 클래스의 설계>

1. 예외 객체 : 예외 발생을 알리는데 사용하는 객체이다.

- 보통은 참조자 형태로, 클래스 명을 직접 사용하여 임시 객체의 형태로 사용한다.

2. 예외 클래스 : 예외 객체로 사용하기 위해 정의된 클래스이다.

- 예외에 대한 정보를 담을 수 있다.

- 상속을 통해 예외의 처리를 단순화 할 수 있다.

ex) catch는 기초클래스로 받게한 뒤 상황에 따른 유도클래스를 throw하면 유연성이 늘어난다.

3. 상속을 이용한 예외 클래스를 이용할 떄는 가장 위에 있는 catch문을 먼저 검사한다는 점을 유의하여 유도클래스를 사용한 catch문을 위에, 기초클래스를 사용한 catch문을 밑에 두어야 한다.

<15.5. 예외처리와 관련된 또 다른 특성들>

1. new연산자가 메모리 공간 할당을 실패할 경우 bad\_alloc이라는 예외를 전달한다.

- 이를 catch하여 예외를 처리할 수 있다.

2. catch(...)로 선언하면 try블록 내에 모든 예외를 받을 수 있다.

- 보통 catch문 마지막에 사용한다.

3. catch문 안에서 throw;를 사용할 경우 함수를 호출한 영역으로 예외를 전달한다.

**[Chapter 16 C++의 형 변환 연산자]**

<16.1. C++에서의 형 변환 연산>

1. C 스타일의 형 변환 연산자는 프로그래머의 의도 파악이 힘들고 형 변환의 범위가 너무 크기에 좋은 형 변환이 아니다.

- C++에서는 새로운 형 변환 연산자와 규칙을 제공한다.

2. C++의 형 변환 (T : 변환하고자 하는 자료형의 포인터 혹은 참조형 / exper : 변환의 대상)

- dynamic\_cast<T>(expr)

>용도 : 유도클래스 >> 기초클래스 변환 / 추상클래스의 경우만 기초클래스 >> 유도클래스 변환

>특징 : 변환 시도 중 문제가 발생하면 특정 정보 반환 / 안정적인 형 변환

- static\_cast<T>(expr)

>용도 : 기초클래스 >> 유도클래스 변환 / 기본 자료형 형 변환

>특징 : dynamic\_cast보다 더 많은 형 변환 허용 / 안정성이 떨어진다.

- const\_cast<T>(expr)

>용도 : const의 성향 제거

>특징 : const함수가 아니어도 const 변수를 사용할 수 있게 해줌 / const 선언의 가치를 떨어뜨리기에 제한적으로 사용

- reinterpret\_cast<T>(expr)

>용도 : 포인터와 관련된 모든 유형 형 변환

>특징 : 포인터 주소를 정수로 바꾸는 등 용도는 다양하나 사용하려면 좀 더 숙달할 필요가 있다.