C++ 객체지향 프로그래밍 II

(3주차)

학습개요

- 학습 목표
  - 예외가 발생 시 제어할 수 있는 예외처리 기법을 활용할 수 있다.
  - C++의 연산자 오버로딩 기법을 익힐 수 있다.
  - C++의 템플릿 활용법을 익힐 수 있다.

- 학습 내용
  - 예외처리
  - 연산자 오버로딩
  - 템플릿
  - 실습

예외와 예외처리

- 예외(exception)
  - 프로그램이 실행되는 동안 발생하는 예상하지 못한 오류

- 예외 처리(exception handling)
  - 예상하지 못한 오류인 예외를 프로그래머가 적절히 대처해서 정상적인 제어를 통해 종료 같은 처리를 수행하는 것
  - C++는 예외 처리를 위해 try, catch, throw 문을 제공

try {...} catch {...}

- try { ... }
  - 예외가 발생할 수 있는 부분을 try 블록으로 묶음
  - try 블록에서 throw 문으로 예외를 발생시켜 던지면, catch 블록에서 예외를 받아 처리함

- throw
  - 예외를 발생시키는 throw 문
  - 프로그램의 제어를 throw된 자료형에 대응하는 catch 블록으로 옮겨서 해당 블록을 실행하게 함
  - throw 문에서 사용하는 자료형은 정수, 문자 등 어떤 것이든 가능하며, 예외 발생을 알리기 위해 예외객체의 생성해 사용할 수 있으며, 이를 위해 예외 클래스를 정의할 필요가 있음
  - 예외 객체를 사용하기 위해 C++ 표준라이브러리 exception 클래스를 활용할 수 있으며, 사용자 정의 예외 클래스를 정의해 사용할 수도 있음

- catch (...) { ... }
  - 괄호 안에 예외 처리할 자료형에 대한 인수를 선언하여 throw된 예외를 처리함
  - 동일한 자료형에 대한 catch는 하나만 있어야 함
  - catch 블록에서 예외를 처리하고 나면 프로그램의 제어는 catch 블록 다음으로 이동하며, 여러 catch 블록이 있을 경우에는 마지막 catch 블록 다음으로 이동함
  - throw된 예외에 대응하는 catch 블록이 없으면 프로그램이 종료됨

연산자 오버로딩 (1)

- 대부분의 연산자 기능을 연산자 함수(operator function)로 정의하여 연산자 오버로딩이 가능함
  - 연산자 함수는 전역 함수 또는 클래스의 멤버 함수일 수 있음
  - 키워드 operator로 연산자 함수와 다른 함수를 구별함
  - 예) + 연산자를 연산자 함수로 정의하는 형식

```
returnType operator+(parameters)                // 전역 연산자 함수
returnType className::operator+(parameters)     // 클래스 멤버 연산자 함수
```

  - returnType : 연산자 +의 연산 결과 자료형
  - className :연산자 + 왼쪽에 있는 객체의 클래스 이름
  - parameters : 연산자 +의 오른쪽을 인수로 받음

연산자 오버로딩 (2)

- 연산자 함수 사용 예

```
MyString s1("abcd");
MyString s2("efgh");
s3 = s1 + s2; // s3 = s1.operator+(s2);
```

  - 덧셈 연산자 +는 s1.operator+(s2)와 같은 방식으로 호출됨. 즉, + 연산자 함수는 s1 객체에 의해 호출되고, s2 객체는 + 연산자 함수의 인수가 됨

- 연산자 함수 오버로딩에서 지켜야 할 규칙
  - 피연산자(operand)의 수(unary, binary)는 변경할 수 없음
  - 본래의 연산자 우선순위(priority)와 결합 순서(associativity)가 유지됨
  - 이미 정의되어 있는 자료형에서 연산자의 의미는 변경할 수 없음

코드 재활용과 템플릿

- 코드의 재활용
  - 프로그램 개발의 생산성 향상
  - 코드 재활용을 높이는 한 가지 방법으로 서로 다른 자료형에 대해 동일하게 동작하는 코드를 사용하는 것으로, 이러한 프로그래밍 메커니즘을 제너릭 프로그래밍(generic programming) 또는 파라메트릭 다형성(parametric polymorphism)이라고 함
  - C++는 템플릿을 통해 파라메트릭 다형성 메커니즘을 지원함

- 템플릿(template)
  - 인수 자료형에 기반하여 함수 또는 클래스를 생성하는 기능
  - 템플릿을 사용하면 서로 다른 자료형에 해당하는 각각의 함수 또는 클래스를 생성하지 않고, 한 템플릿 함수 또는 클래스만 정의하고, 자료형을 인수로 사용하여 각 자료형에 해당하는 함수 또는 클래스의 인스턴스를 생성할 수 있음
  - 장점
    - 함수 또는 클래스의 제너릭 버전(generic version)인 템플릿만 정의하므로 프로그램 작성이 쉬움
    - 간단하게 자료형 정보를 추상화하기 때문에 이해하기도 쉬움
    - 자료형을 컴파일 시간에 알 수 있기 때문에 자료형 검사(type checking)를 할 수 있어 안전함

함수 템플릿

- 함수 템플릿(function template)을 사용하면 동일한 코드를 사용하여 서로 다른 자료형에서 사용할 수 있는 함수를 작성할 수 있음
- 함수 템플릿 형식

```
template <class Type>
returnType functionName(argumentList)
{
    // 함수 구현
}
```

  - 일반 함수의 앞부분에 template <class Type>을 추가한 형태
  - 템플릿 정의에서 사용하는 키워드 class는 Type 위치로 전달할 자료형으로, 모든 형이 가능함을 의미함
  - Type은 식별자로 사용자가 의미에 맞게 사용할 수 있음

- 함수 템플릿 인스턴스 생성
  - 함수 템플릿은 특정 자료형으로 처음 호출될 때 해당 자료형에 대한 구체화된 버전인 인스턴스를 생성함. 즉, 특정 자료형에 해당하는 인스턴스는 하나만 존재하고, 템플릿으로 정의된 함수가 해당 자료형으로 사용될 때마다 인스턴스가 호출됨

클래스 템플릿 (1)

- 일반 클래스의 앞부분에 template<class Type>을 추가하여 정의함
- 클래스 템플릿 형식

```
template <class Type>
class className
{
    // 클래스 멤버
};
```

- 템플릿 클래스의 멤버 함수
  - 템플릿 클래스의 멤버 함수를 클래스 정의 안에서 구현할 때는 일반 클래스와 동일함
  - 클래스의 정의 외부에서 구현할 때는 템플릿 클래스임을 나타내는 부분이 추가됨

```
template<class Type, int i>
CStack<Type, i>::CStack()
{
    top = i;
    pBuffer = new Type[i*sizeof(Type)];
}
```

클래스 템플릿 (2)

- 클래스 템플릿의 인스턴스 생성
  - 클래스 템플릿을 사용할 때는 해당 클래스 템플릿에 인수를 전달하여 명시적으로 클래스의 인스턴스를 생성함

```
CStack<int, 5> test1; // 크기 5인 정수 스택
CStack<char, 10> test2; // 크기 10인 문자 스택
```

학습정리

- 예외가 발생할 수 있는 부분을 try 블록으로 묶고, try 블록에서 throw 문으로 예외를 발생시켜 던지면, catch 블록에서 예외를 받아 처리할 수 있다
- 연산자 기능을 연산자(operator) 함수로 정의하여 연산자 오버로딩을 할 수 있다.
- C++는 템플릿을 통해 제너릭 프로그래밍(파라메트릭 다형성 메커니즘)을 지원한다.
- 템플릿을 사용하면 서로 다른 자료형에 해당하는 각각의 함수 또는 클래스를 생성하지 않고, 한 템플릿 함수 또는 클래스만 정의하고, 자료형을 인수로 사용하여 각 자료형에 해당하는 함수 또는 클래스의 인스턴스를 생성할 수 있다.