

# 제11장

## 분리컴파일과 네임 스페이스

# 학습목표

- 분리컴파일
  - 캡슐화 리뷰
  - 헤더파일과 실행파일
- 네임스페이스
  - 지시자 사용하기
  - 이름 지정하기
  - 명명되지 않은 네임스페이스
  - 도움을 주는 함수 감추기
  - 중첩된 네임스페이스

# 분리 컴파일

- 프로그램 기능들
  - 분리된 별개의 파일에 저장하여 각기 컴파일하기
  - 프로그램이 실행되기 전에 서로 링크하기
- 클래스 정의
  - 프로그램을 사용하는 것으로부터 분리시키기
  - 클래스 라이브러리 만들기
    - 다른 여러 프로그램에서 재사용 될 수 있음
    - 이미 정의된 라이브러리를 사용한 것과 같음

# 클래스 분리

- 클래스의 독립
  - 클래스 정의와 명세부분을 분리: 인터페이스(interface)
  - 클래스 구현 부분을 분리
  - 정의+명세와 구현을 각각 두 개의 파일에 저장
- 만약 구현부분 변경하고자 하면 → 오직 그 파일만 변경!!
- 캡슐화 원리를 물리적인 파일 분리로 구현
  - 클래스 구현의 세부사항을  
프로그래머에 의해 어떻게 사용되는 지에서 분리시키는 것
  - 구현의 수정 → 다른 프로그램들에 어떠한 영향도 끼치지 않아야 함
- OOP 원리에 근거함

# 캡슐화 규칙

- 분리를 보장하기 위한 규칙들:
  1. 모든 멤버 변수들은 비공개변수로
  2. 클래스의 기본 함수는 ( 아래목록 중 하나 ):
    - 공개 멤버 함수 / Friend 함수 or 일반 함수 / 오버로드 함수
    - 클래스 정의, 함수 및 연산자 선언들을 모아 그룹화
      - 클래스의 인터페이스(interface) 라고 한다.
  3. 클래스 사용 프로그래머에게는 클래스의 구현사용을 할 수 없게 해야 함
- 클래스 분리 방식
  - 인터페이스 파일: 클래스 정의, 함수 및 연산자 선언들을 포함
    - 프로그래머들이 '보는' 부분. 컴파일 단위로 분리.
- 구현 파일: 멤버 함수 정의를 포함
  - 컴파일 단위로 분리

# 클래스의 헤더 파일

- 인터페이스 파일은 항상 헤더파일 내에
  - .h 라는 이름 지시자 사용
- 클래스 사용하는 프로그램에 포함시켜야 할 것
  - #include “사용할 클래스이름.h”
  - 따옴표는 이미 정의된 헤더파일을 의미
    - 현재 디렉토리(Directory)에서 검색가능
  - 예를 들어, <iostream>와 같은 라이브러리를 불러냄
    - <>괄호는 이미 정의된 헤더파일을 의미
    - 라이브러리 디렉토리에서 검색가능

# 클래스 구현 파일

- C++파일 내에서 클래스 구현
  - 일반적으로 인터페이스 파일과 구현파일은 동일한 이름을 가짐
    - myclass.h and myclass.cpp
  - 모든 클래스 멤버 함수를 .cpp에서 구현
  - 구현 파일은 클래스의 헤더파일을 반드시 #include 해야 함
- .cpp 파일에서 일반적으로 실행 가능한 코드를 포함한다.
  - e.g., Function definitions, including main()

# 다수의 헤더파일 컴파일

- 헤더 파일은 일반적으로 여러 번 포함됨
  - 예를 들어, 클래스 인터페이스는 클래스의 구현과 프로그램의 파일에 의해 포함됨
  - 컴파일은 오직 한 번만 가능!
- 전처리기 사용하기
  - 컴파일러에게 헤더파일은 한 번만 포함하라고 지시함
  - `#ifdef` `#ifndef` 사용하기
    - `FNAME` 은 일반적으로 일관성과 접근성이 있는 파일의 이름
  - 헤더파일의 중복 정의를 막음
  - `#pragma once` 사용하기

```
#ifndef FNAME_H
#define FNAME_H
... //Contents of header file
...
#endif
```



# 다른 라이브러리 파일

- 단지 클래스만을 위한 라이브러리가 아님
- 관련된 기능
  - prototypes → 헤더파일
  - 정의 → 구현 파일
- 다른 유형의 정의
  - 구조체, 간단한 typedefs → 헤더파일
  - 변함없는 선언들 → 헤더파일

# 네임 스페이스와 using 지시자

- 네임스페이스: 이름 정의들의 집합
  - 클래스 정의, 변수들의 선언
- 프로그램이 사용하는 여러 클래스와 함수가 가지는 같은 이름!
  - 네임스페이스는 중복 이름을 처리
    - 만일 이름 충돌이 일어나지 않으면 → 꺼짐
- using 지시자
  - using namespace std;
    - std 네임스페이스의 모든 정의들을 이용 가능하게 만든다.
  - 왜 꼭 필요하지 않게 될까?
    - 표준적인 의미를 갖고 있지 않은 cout, cin을 만들 수 있음
      - cout, cin을 재정의 해야할 필요가 있을 수 있음
    - 다른 것들을 재정의할 수 있음

# std namespace v.s. global namespace

- std namespace
  - 지금까지 계속 std 네임스페이스를 사용해 옴
  - 많은 표준 라이브러리 파일에 정의된 모든 이름들을 포함함
  - 예를 들어: `#include <iostream>`
    - 모든 이름정의 (`cin`, `cout` 등)을 std 네임스페이스에 위치시킴
    - 프로그램이 네임스페이스의 정의에 접근하도록 하기 위해 반드시 구체화해야
- global namespace
  - 모든 코드는 어떤 네임스페이스에든 속함
  - 네임스페이스가 명시되지 않았다면 → 전역 네임스페이스
    - `⌞` `using` 지시자가 필요 없음
    - 전역 네임스페이스는 항상 사용가능
    - 묵시적인 `using` 지시자를 사용

# 여러 가지 이름과 네임스페이스의 구체화

- 다수의 네임스페이스

- 예를 들어, 일반적으로 사용되는 전역 네임스페이스, std 네임스페이스 등
- 만약 이름이 std와 전역 두 군데에 모두 정의된다면, 에러
- 각기 다른 네임스페이스 모두에서는 사용가능
- 어느 네임스페이스가 언제 사용되는지 반드시 명시해야 함

- 네임스페이스 NS1, NS2가 있다고 가정

- 두 네임스페이스 모두에 각각 void 함수 myFunction()가 정의 되어 있을 때

```
{  
    using namespace NS1;  
    myFunction();  
}
```

```
{  
    using namespace NS2;  
    myFunction();  
}
```

- using 지시자는 영역제한(block-scope)이 있음

# 네임스페이스의 생성

- 네임스페이스의 그룹화를 이용 :

```
namespace Name_Space_Name
{
    Some_Code
}
```

- Some\_Code에 정의된 모든 이름들을 네임스페이스 Name\_Space\_Name로

```
namespace Space1
{
    void greeting();
}
```

```
namespace Space1
{
    void greeting()
    {
        cout << "Hello from namespace Space1.\n";
    }
}
```

- 이후에 사용할 때는 `using namespace Name_Space_Name`
- ex. linux에서 `/usr/include/c++/7`

# using 지시자와 선언

- 개인적인 이름들을 네임스페이스로부터 구체화할 수 있음
- 고려해 볼 것:
  - Namespaces NS1, NS2 가 존재하고 각각 함수 fun1(), fun2()를 갖는다 가정
  - syntax 선언: `using Name_Space::One_Name;`
    - 어느 네임스페이스인지와 번호를 지시자로 구체적으로 명시
  - `using NS1::fun1;`  
`using NS2::fun2;`
- using 지시자 v.s. using 선언
  - using 선언
    - 하나의 이름만을 네임스페이스에서 사용 가능하게 함
    - 이름의 다른 사용들이 허용되지 못하도록 이름들을 소개함
  - using 지시자
    - 사용 가능한 네임스페이스의 모든 이름을 만듦
    - 오직 잠재적으로만 이름들을 소개함

# 이름 제한

- 이름이 어디서 왔는지 구체화할 수 있음
  - 제한자(qualifier)와 범위해결 연산자를 이용
  - 한 번 혹은 적은 횟수의 이용을 목적으로 사용되었을 때
- NS1::fun1();
  - fun()함수가 네임스페이스 NS1에서 왔음을 명시함
- 특히, 인자형식에서 유용하게 쓰임

`int getInput(std::istream inputStream);`

  - istream의 std 네임스페이스에서 찾을 수 있는 인자형식
  - Using 지시자나 using선언을 사용할 필요를 없애줌

# 네임스페이스를 위한 이름 선택

- 독특한 문자열을 포함시킨다.
  - 예를 들면, 성(last name)
  - 다른 네임스페이스와 동일한 이름을 사용할 확률을 감소
- 종종 같은 프로그램을 수행하는 데에 있어서 여러 프로그래머가 네임스페이스를 사용한다.
  - 반드시 독특한 이름을 가져야 함
  - 그렇지 않다면, 동일 영역에서 동일한 이름들이 여러 가지 정의를 가질 수
    - 에러 발생



# 디스플레이 11.6 네임스페이스 안에서 클래스 정의하기 (헤더 파일)

## 디스플레이 11.6 네임스페이스 안에서 클래스 정의하기(헤더 파일)

---

```
1  //이 파일은 헤더 파일 dtime.h이다.
2  #ifndef DTIME_H
3  #define DTIME_H
4
5  #include <iostream>
6  using std::istream;
7  using std::ostream;
8
9
10 namespace DTimeSavitch
11 {
12
13     class DigitalTime
14     {
15
16         <DigitalTime 클래스의 정의는 디스플레이 11.1과 동일하다.>
17     };
18
19 } //DTimeSavitch
20
21 #endif //DTIME_H
```

디스플레이 11.8과 11.9에 이 클래스 정의의 개정 버전이 존재한다.

네임스페이스 DTimeSavitch가 2개의 파일에 걸쳐 정의되었다. 나머지 부분은 디스플레이 11.7에 나타나 있다.

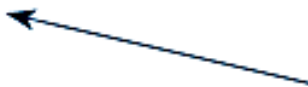
# 디스플레이 11.7 네임스페이스 안에서 클래스 정의하기(구현 파일)

## 디스플레이 11.7 네임스페이스 안에서 클래스 정의하기(구현 파일)

---

```
1 //이 파일은 구현 파일 dtime.cpp이다.
2 #include <iostream>
3 #include <cctype>
4 #include <cstdlib>
5 using std::istream;
6 using std::ostream;
7 using std::cout;
8 using std::cin;
9 #include "dtime.h"
```

using 지시자 using namespace std;를  
사용하면 4개의 using 선언문을 대체할 수 있다.  
그러나 4개의 using 선언문을 사용하는 것이 보다  
바람직한 스타일이다.



```
10 namespace DTimeSavitch
11 {
12
13     <디스플레이 11.2의 모든 함수 정의가 여기에 위치한다.>
14
15 } // DTimeSavitch
```

# 명명되지 않은 네임스페이스

- 컴파일의 단위는 “파일 ”
  - 클래스 구현 파일, 클래스를 위한 인터페이스 헤더 파일 등의 파일
- 모든 컴파일 단위는 명명되지 않은 네임스페이스를 가짐.
  - 동일한 방식으로 쓰여지나, 이름이 주어지지 않음
  - 모든 함수들은 컴파일 단위 안에서만 사용될 수 있는 지역함수
    - 즉, 명명되지 않은 네임스페이스는 지역적으로 사용

```
namespace
{
    int digitToInt(char c)
    {
        return ( int(c) - int('0') );
    }
}
```

## • 전역 네임스페이스

vs. 명명되지 않은 네임스페이스

- 같지 않다
- 전역 네임스페이스: 그룹화하지 않고, 전역에서 사용가능
- 명명되지 않은 네임스페이스: 그룹화 내에 정의되고 로컬 범위에서 사용

# 중첩된 네임스페이스

- 합법적인 중첩된 네임스페이스
- Sample을 호출하는 방식:
  - S1::S2::sample();

```
namespace S1
{
    namespace S2
    {
        void sample()
        {
            ...
        }
    }
}
```

# 보조 함수 감추기

- 보조 함수 감추기:
  - 활용도 낮음
  - 일반적으로 사용되지 않음
- 보조 함수를 감추는 두 가지 방법:
  - 비공개 멤버로 만든다.
    - 만일 함수가 호출 객체를 받아들이는다면
  - 명명되지 않은 네임스페이스 안에 둔다.
    - 만일 호출 객체가 필요가 없다면
    - 코딩을 보다 명확하게 만드는 역할

# 요약

- 클래스 정의와 구현 구분 짓기 가능 → 별도의 파일에 저장 가능
  - 분리 컴파일 단위
- 네임스페이스는 이름 정의의 집합
- 네임스페이스로부터 이름을 사용하는 세가지 방법 :
  - Using 지시어 이용 / Using 선언 사용 / 지정 연산자 사용
- 네임스페이스는 네임스페이스의 그룹화 내에 정의 되어야
  - 명명되지 않은 네임스페이스: 지역 이름 정의로 쓰여  
컴파일 단위 안에서만 사용가능
  - 전역 네임스페이스 : 네임스페이스 그룹화에 전혀 포함되지 않아  
전역에서 사용가능