Day-41

Functions
Function Signature and Overloading, Overloading and =delete

Function Signature : 함수 시그니처

- 함수 시그니처는 함수에 대한 입력 유형과 템플릿 함수에 대한 입력, 출력 유형을 정의
- 함수 오버로딩의 핵심
- 함수의 원형에 명시되는 매개변수 리스트를 가리킴
- 예시
  - 두 함수가 매개변의 개수와 그 타입이 모두 같다면, 두 함수의 시그니처는 같다고 할 수 있음
- C++ 표준은 반환 타입만 다른 함수 선언을 금지함
- 시그니처가 다른 함수 선언은 서로 다른 반환 유형을 가질 수 있음

```
#include<iostream>
using namespace std;

void f(int a, char* b); // 시그니처 : (int, char*)

// char f(int a, char* b); // 컴파일 에러
                           // 시그니처는 동일하지만 반환 타입이 다

void f(const int a, char* b); // 시그니처 (int , char*)
                              // const int == int 이기 때문에

int main()
{
    int a = 1;
```

```
    char b[] = "abs";

    f(a, b);

    return 0;
}

void f(int a, char* b) {
    cout << a << endl;
    cout << b << endl;
}

// 함수이름, 시그니처가 위 함수와 같아
// C2084(함수 중복) 오류 발생
void f(const int a, char* b) {
    cout << a << endl;
    cout << b << endl;
}
```

Function Overloading : 함수 오버로딩

- 함수 오버로딩을 사용하면 이름은 같지만 시그니처가 다른 별개의 함수를 만들 수 있음

```
#include<iostream>
using namespace std;

void f(int a, char* b); // 시그니처 : (int, char*)

void f(int a, const char* b);

int f(float);


int main()
{
    int a = 1;
```

```
    char b[] = "abs";
    const char* b2 = "as";

    f(a, b);
    f(a, b2);
    cout << f(1.0) << endl;

    return 0;
}

void f(int a, char* b) {
    cout << a << endl;
    cout << b << endl;
}

void f(int a, const char* b) {       // 이름은 f로 동일
    cout << a << endl;               // 시그니처는 int, const
    cout << "const : " << b << endl; // 오버로딩
}

int f(float) {                       // 이름은 f로 동일
    return 10;                       // 시그니처는 float
}                                    // return 타입 다름 -> 오

// 출력
// 1
// abs
// 1
// const : as
// 10
```

Overloading Resolution Rules

- 정확히 일치
- Promotion (ex : char to int)
- 표준 타입 변환(ex : float and int)

- 생성자 또는 사용자 정의 타입 변환

```
#include<iostream>
using namespace std;

void f(int a);
void f(float b);
void f(float b, char c);


int main()
{
    f(0);           // 정확히 일치
    f('a');         // promotion
    //f(3LL);       // 컴파일 에러
    f(2.3f);        // 정확히 일치
    //f(2.3);       // 컴파일 에러
    f(2.3, 'a');    // 표준 타입 변환

    return 0;
}

void f(int a) {
    cout << a << endl;
}

void f(float b) {
    cout << b << endl;
}

void f(float b, char c) {
    cout << b << " ";
    cout << c << endl;
}

// 출력
// 0
// 97
```

```
// 2.3
// 2.3 a
```

Overloading and =delete

- =delete는 잘못된 과부화 호출을 방지하기 위해 사용할 수 있음

```
#include<iostream>
using namespace std;

void g(int) {}
void g(double) = delete;


int main()
{
    g(3);
    g(3.0); // 컴파일 에러, 삭제된 함수를 참조하려 함
    return 0;
}
```

```
#include <iostream>
#include <cstddef> // std::nullptr_t

void f(int*) {}
void f(std::nullptr_t) = delete;

int main()
{
    f(nullptr); // 컴파일 에러, 삭제된 함수를 참조하려고 함
    return 0;
}
```