# 1부 전문가를 향한 C++ 첫걸음

2023년 12월 29일 금요일 오후 3:33

1장 C++와 표준 라이브러리 초단기 속성 코스

※ 헤더 파일 안에서는 절대로 using 문을 작성하면 안된다.
헤더 파일을 인클루드하는 모든 파일에서 using 문으로 지정한 방식으로 호출해야 하기 때문이다.

**중첩 네임스페이스**
Namespace MyLibraries::Networking::FTP{

}

**네임스페이스 앨리어스**
namespace MyFTP = MyLibraries::Networking::FTP;

## 1.1.4 변수

1. **균일 초기화 (uniform initializtation)**
   -> int initializedInt { 7 }; // int initializedInt = 7;
   기존 대인 문법 대신 균일 초기화를 사용하는 것이 바람직하다.
2. **숫자 경곗값**
   정수의 양의 최댓값을 알아내는 표준 방법
   <limits> std::numeric_limits 클래스 템플릿을 사용하는 것이 바람직하다.
   min()과 lowest()가 다름.
   정수에서는 같지만, 부동소수점수에서는 최솟값은 표현 가능한 가장 작은 양의 값
   최젓값은 표현 가능한 가장 작은 음수(-max())
3. **캐스트**
   1) 강제 현변환
      - long someLong { someShort }; // 명시적으로 캐스트할 필요가 없다.
        short 타입이 long 타입으로 자동 변환된다.
        long 타입의 정밀도(표현 범위)가 short 타입보다 넓기 때문에
4. **부동소수점수**
   1) 주어진 부동소수점수가 NaN인지 확인하려면 std::isnan()을 사용
      무한인지 검사하려면 std::isinf()를 사용
      둘 다 <cmath>에 정의되어 있다.
      numeric_limits<double>::infinity

## 1.1.6 열거타입

숫자를 나열하는 방식과 범위를 마음대로 정의해서 변수를 선언하는 데 활용할 수 있음

1. **강타입 열거 타입 strongly typed enumeration type**
   1) 변수에 지정할 수 있는 값의 범위를 엄격하게 제한
   2) ※ using enum 또는 using 선언문의 스코프를 최소화 하는 것이 좋음
      스코프가 너무 크면 이름이 충돌하는 현상이 발생할 수 있음.
2. **예전 방식의 열거 타입**
   1) 새로운 버전에 맞게 코드를 작성할 때는 강타입 열거를 사용
      but, 예전에 작성된 레거시 코드에서는 enum class가 아닌 enum으로 선언된 것을 알 수 있음
   2) 예전 방식 열거 타입의 값은 스코프가 자동으로 확장됨
      ex) PieceType myPiece { PieceTypeQueen };
   3) 상위 스코프에 같은 이름이 있으면 컴파일 에러가 발생
      - bool ok { false };
      - enum Status { error, ok };
      - 예전 방식으로 열거 타입 정의 시에는 멤버 이름을 고유한 이름으로 지정해야함
      - 강타입이 아니기 때문에 타입에 안전하지 않음

### 1.1.7 구조체

구조체를 사용하면 기존에 정의된 타입을 한 개 이상 묶어서 새로운 타입으로 정의할 수 있다.

### 1.18 조건문

조건문을 사용하면 어떤 값이 참 또는 거짓인지에 따라 주어진 코드를 실행할 지 결정할 수 있다.

1. **if/else 문**
   1) if의 초기자
      - if 문 안에 초기자를 넣을 수 있다.
      - if (<초기자>; <조건문>) {
        <if의 본문>
        }...
      - if 문의 <초기자>에서 정의한 변수는 <조건문>과 <if의 본문>, <else if의 조건문>과
        <else if의 본문>, <else의 본문> 안에서만 사용할 수 있음. if문 밖에서는 사용할 수 없음
      - ex : if (Employee employee { getEmployee() }; emplyee.salray > 1000) { … }
2. **switch 문**
   1) switch 문은 모두 if/else 문으로 변환할 수 있다.
   2) switch 문은 표현식의 만족 여부가 아닌 표현식의 다양한 결과값마다 수행할 동작을 결정하는데 주로 사용한다.
      if/else 문을 연달아 쓰기 보다는 switch 문으로 작성하는 것이 훨씬 더 깔끔하다.
   3) 폴스루 fallthrough (흘려보내기)

break 문이 나타날 때까지 실행
[[fallthrough]]; // 어트리뷰트 지정해서 의도적으로 폴스루 방식으로 작성했다고 컴파일러에 알려준다.

4) switch문의 초기자
   - switch (<초기자>; <표현식>) { <본문> }
   - <초기자>에서 선언한 변수는 <표현식>과 <본문> 안에서만 사용할 수 있고, switch 문 밖에서는 사용할 수 없다.

### 1.19 조건 연산자

1. **삼항연산자**
   1) 문장이 아니라 표현식이기 때문에 코드 안에서 원하는 곳에 간편히 추가할 수 있다는 것이 장점

### 1.10 논리 연산자

C++20 에서는 3방향 비교 연산자. (일명 우주선 연산자)가 추가됐음.
C++는 논리 표현식을 평가할 때 단락 논리 short-circuit logic(축약 논리)를 사용한다.
표현식을 평가하는 도중에 최종 결과가 나오면 나머지 부분은 평가하지 않는다. => 불필요한 작업 생략
단락 기능은 프로그램 성능을 높이는 데 도움이 된다.
※ 단락되는 논리식을 작성할 때는 가볍게 검사할 수 있는 부분을 앞에 적고, 시간이 좀 걸리는 부분은 뒤에 둔다. / 또는 포인터값이 올바르지 않으면 그 포인터를 사용하는 표현식이 실행되지 않게 할 때도 단락을 활용하면 좋다.

### 1.1.11 3방향 비교 연산자 three-way comparison operator

두 값의 순서를 결정하는 데 사용된다. 우주선 연산자라고도 부른다.

1. 연산자 기호 <=>
2. 주어진 표현식의 평가 결과가 비교 대상이 되는 값과 같은지 아니면 그보다 크거나 작은지 알려준다.
3. true나 false가 아닌 세 가지 결과 중 하나를 알려줘야 하기 때문에 부울 타입을 리턴할 수 없다.
   1) <compare>에 정의되고 std 네임스페이스에 속한 열거 타입으로 리턴한다.
   2) 피연산자가 정수 타입이면 **강한 순서 strong ordering** 라고 부르며, 다음 세 가지 중 하나가 된다.
      - strong_ordering::less : 첫 번째 피연산자가 두 번째 피연산자보다 작다.
      - strong_ordering::greater : 첫 번째 피연산자가 두 번째 피연산자보다 크다.
      - strong_ordering::equal : 두 피연산자가 같다.
   3) 피연산자가 부동소수점 타입이라면 결과는 부분 순서 partial ordering 다.
      - partial_ordering::less : 첫 번째 피연산자가 두 번째 피연산자보다 작다.
      - partial_ordering::greater : 첫 번째 피연산자가 두 번째 피연산자보다 크다.
      - partial_ordering::equivalent : 두 피연산자가 같다.

- partial_ordering::unordered : 두 피연산자 중 하나는 숫자가 아니다.

4) **약한 순서 weak ordering**
   - 자신이 직접 정의한 타입에 대해 3방향 비교 연산을 구현할 때 활용
   - weak_ordering::less
   - weak_ordering::greater
   - weak_ordering::equivalent

4. 기본 타입에 대해서는 3방향 비교 연산자를 사용해도 기존 비교 연산자를 사용하는 것보다 좋은 점은 없다.
   1) 비교 작업이 복잡한 객체에서는 상당히 유용하다.
   2) 객체의 순서를 비교할 때 다소 무거운 기존 비교 연산자를 두 번 호출할 필요 없이 3방향 비교 연산자 하나만으로 결정할 수 있기 때문이다.
5. <compare>에서는 순서의 결과를 해석해주는 **이름 있는 비교 함수 named comparison function** 이 있다. 비교한 결과를 true나 false로 리턴한다.
   1) std::is_eq() : ==
   2) is_neq() : !=
   3) is_lt() : <
   4) is_lteq() : <=
   5) is_gt() : >
   6) is_gteq() : >=

### 1.1.12 함수

프로그램의 가독성을 높이려면 코드를 간결하고 명확한 함수 단위로 나눠야 한다.

1. C++에서 함수를 사용하려면 먼저 선언해야 한다.
   특정한 파일 안에서만 사용할 함수는 선언과 구현(정의)을 모두 소스 파일 안에 작성한다.
   반면 함수를 다른 모듈이나 파일에서도 사용한다면 선언은 모듈 인터페이스 파일로부터 익스포트하고, 정의는 모듈 인터페이스 파일이나 모듈 구현 파일에 작성한다.
2. 함수 리턴 타입 추론
   1) 함수의 리턴 타입을 컴파일러가 알아서 지정할 수 있다. 리턴 타입 자리에 auto 키워드만 적으면 된다.
3. 현재 함수 이름
   1) 모든 함수는 내부적으로 _func_라는 로컬 변수가 정의되어 있다. 이 변수의 값은 현재 함수의 이름이며, 주로 로그를 남기는 데 활용한다.
4. 함수 오버로딩 overloading
   1) 이름은 같지만 매개변수 구성은 다른 함수를 여러 개 제공한다는 뜻
   2) **리턴 타입만 달라서는 안되고 매개변수의 타입이나 개수도 달라야 한다.**
   3) 컴파일러는 주어진 인수를 기반으로 두 가지 오버로딩된 함수 중에서 적합한 버전을 선택한다.

### 1.1.13 어트리뷰트 attribute

어트리뷰트는 소스 코드에 벤더에서 제공하는 정보나 옵션을 추가하는 메커니즘
C++11부터 [[어트리뷰트]] 와 같이 대괄호를 이용한 형식을 사용하도록 표준화

1. [[nodiscard]] : 어떤 값을 리턴하는 함수에 대해 지정할 수 있다. 이 함수가 호출될 때 리턴값에 아무런 작업을 하지 않으면 경고 메시지를 출력한다.
   1) 에러 코드를 리턴하는 함수 등에 활용할 수 있다.
   2) 클래스, 함수, 열거형에 적용할 수 있다.
   3) C++20 부터 [[nodiscard]] 어트리뷰트에 이유를 설명하는 스트링을 추가할 수 있다.
      - [[nodiscard("Some explanation")]] int func();
2. [[maybe_unused]] : 뭔가 사용하지 않았을 때 컴파일러가 경고 메시지를 출력하지 않도록 설정하는 데 사용된다.

```
int func(int param1, int param2)
{
    return 42;
}
```

   컴파일러 경고 수준이 높게 설정된 경우 waring C4100: 'param2': unreferenced formal parameter... 등과 같이 출력됨.
   이때 maybe_unused 어트리뷰트를 지정할 수 있음
   1) 클래스, 구조체, 비 static(static이 아닌) 데이터 멤버, 유니온, typedef, 타입 앨리어스, 변수, 함수, 열거형, 열거값 등에 대해 지정할 수 있다.
3. [[noreturn]] : 함수에 지정하면 호출 지점으로 다시 돌아가지 않는다. 주로 프로세스나 스레드 종료와 같이 뭔가가 끝나게 만들거나, 익셉션을 던지는 함수가 여기에 해당한다.
   이 어트리뷰트를 이용하면 컴파일러가 특정한 경고나 에러 메시지를 출력하지 않게 만들 수 있다.
4. [[deprecated]] : 지원 중단된 대상임을 지정하는 데 사용.
   현재 사용할 수는 있지만 권장하지 않는 대상임을 표시한다.
   지원 중단되는 이유를 표현하는 인수를 옵션으로 지정할 수 있다.
5. [[likely]] 와 [[unlikely]] : 컴파일러가 최적화 작업을 수행하는 데 도움을 줄 수 있음.
   이 어트리뷰트를 이용하여 if와 switch 문에서 수행될 가능성이 높은 브랜치를 표시할 수 있다.
   하지만, 이 어트리뷰트가 필요한 경우는 드물다.
   최신 컴파일러와 하드웨어는 브랜치 예측 능력이 상당히 뛰어나기 때문에 이런 어트리뷰트를 지정하지 않아도 알아서 잘 처리하지만, 성능에 민감한 부분 등과 같이 특정한 경우에는 컴파일러에 도움을 줄 수 있다.

### 1.1.14 C 스타일 배열

C++에서 배열을 선언할 때는 반드시 배열의 크기를 지정해야 하는데, 변수로 지정할 수는 없고 반드시 상수 또는 상수 표현식 constant expression(constexpr)으로 지정해야 한다.

1. int myArray[] { 1, 2, 3, 4 }; // 컴파일러는 원소 네 개로 구성된 배열을 생성한다.
2. 초기화 리스트에 나온 원소의 개수가 배열의 크기로 지정한 수보다 적으면 나머지 원소는 0으로 초기화된다.
   1) int myArray[3] { 2 }; 나머지 원소는 모두 0으로 초기화

3. 스택 기반의 C 스타일 배열의 크기는 std::size() 함수로 구할 수 있다. <array> 헤더를 인클루드해야 한다. 이 함수는 <cstddef>에 정의된 부호 없는 정수 타입ㅇ니 size_t 값을 리턴한다.
    1) size_t arraySize { std::size(myArray) };
4. 스택 기반 C 스타일의 배열의 크기를 구할 때 sizeof 연산자를 활용할 수도 있다.
    1) size_t arraySize { sizeof(myArray} / sizeof(myArray[0]) };

**1.1.15 std::array**

C++는 std::array라는 고정 크기 컨테이너를 제공한다. 이 타입은 <array> 헤더 파일에 정의되어 있다.

1. std::array는 C 스타일 배열에 비해 장점이 많다.
    1) 항상 크기를 정확히 알 수 있음
    2) 자동으로 포인터를 캐스트(동적 형변환)하지 않아서 특정한 종류의 버그를 방지할 수 있음
    3) 반복자(이터레이터)로 배열에 대한 반복문을 쉽게 작성할 수 있다

C++는 CTAD class template argument deduction (클래스 템플릿 인수 추론) 이라는 기능을 제공한다.

꺾쇠괄호 사이에 템플릿 타입을 지정하지 않아도 된다.

CTAD는 초기자를 사용할 때만 작동한다. 컴파일러가 템플릿 타입을 자동으로 추론하는 데 이 초기자를 사용하기 때문이다.

- array arr{ 9, 8, 7 };
- C스타일 배열과 std:array는 둘 다 크기가 컴파일 시간에 결정되어야 하며, 실행 시간(run time)에 늘어나거나 줄어들 수는 없다.

**1.1.16 std::vector**

C++ 표준 라이브러리는 크기가 고정되지 않은 컨테이너를 다양하게 제공한다.

vector는 C 스타일의 배열 대신 사용할 수 있고 훨씬 유연하고 안전하다.

프로그래머는 메모리 관리를 신경 쓸 필요가 없다. 메모리를 확보하는 작업은 vector가 알아서 처리

1. vector는 동적이다. 실행시간에 원소를 추가하거나 삭제할 수 있다.
2. std::array와 마찬가지로 꺾쇠괄호 안에 템플릿 매개변수를 지정해야 한다.
3. vector는 제너릭 컨테이너다. 거의 모든 종류의 객체를 담을 수 있다.
4. vector를 사용할 때에는 반드시 꺾쇠괄호 안에 원하는 객체 타입을 명시해야 한다.

std::array와 마찬가지로 CTAD를 지원한다.

**1.1.17 std::pair**

<utility> 헤더에 정의된 std::pair 클래스 템플릿은 두 값을 하나로 묶는다. 각 값은 public 데이터 멤버인 frst와 second로 접근할 수 있다.

pair도 CTAD를 지원한다.

**1.1.18 std::optional**

<optional>에 정의된 std::optional은 특정한 타입의 값을 가질 수도 있고, 아무 값도 갖지

않을 수도 있다.

1. optional은 기본적으로 함수 매개변수에 전달된 값이 없을 수도 있는 상황에 사용된다.
2. 값을 리턴할 수도 있고, 그렇지 않을 수도 있는 함수의 리턴 타입으로 사용하기도 한다.
    1) 기존에 리턴 값이 없는 경우를 표현하기 위한 nullptr, end(), -1, EOF와 같은 특수한 값을 사용하지 않아도 됨.
3. **함수의 리턴값은 수행 결과의 성공 여부를 나타내는 부울 타입으로 표현하고, 실제 결과는 출력용 매개변수를 이용하여 함수의 인수에 전달하는 식으로 작성하지 않아도 된다.**
4. optional 타입은 클래스 템플릿이므로 optional<int>와 같이 실제 타입을 꺾쇠괄호 안에 반드시 지정해야 한다.
5. optional에 값이 있는지 확인하려면 has_value() 메서드를 사용하거나, if문을 사용한다.
    1) optional에 값이 있을 때는 value()나 역참조 연산자로 그 값을 가져올 수 있음
    2) 값이 없는 optional에 대해 value()를 호출하면 std::bad_optional_access 익셉션이 발생한다.
6. value_or()을 사용하면 optional에 값이 있을 때는 그 값을 리턴하고, 값이 없을 때는 다른 값을 리턴한다.
7. 레퍼런스는 optional에 담을 수 없다. 따라서 optional<T&>와 같이 작성할 수 없다. 대신 optional에 포인터를 저장할 수는 있다.

### 1.1.19 구조적 바인딩 structured binding

구조적 바인딩을 이용하면 여러 변수를 선언할 때 array, struct, pair 등에 담긴 원소들을 이용하여 변수값을 한꺼번에 초기화할 수 있다.

1. 구조적 바인딩을 적용하려면 반드시 auto 키워드를 붙여야 한다.
2. 구조적 바인딩에서 왼쪽에 나온 선언할 변수 개수와 오른쪽에 나온 표현식에 담긴 값의 개수는 반드시 일치해야 한다.
    1) array values { 1, 2, 3 };
    auto [x, y, z] { values };
3. 구조적 바인딩은 배열뿐만 아니라 비 static 멤버가 모두 public으로 선언된 구조체에도 적용할 수 있다.
    1) struct Point { double m_x, m_y, m_z; };
    Point point;
    point.m_x = 1.0; point.m_y = 2.0; point.m_z = 3.0;
    auto [x, y, z] { point };
4. auto& / const auto&를 이용하여 구조적 바인딩으로 비 const에 대한 레퍼런스나 const에 대한 레퍼런스를 생성할 수도 있다.

### 1.1.20 반복문

C++는 while, do/while, for, 범위 기반 range-based for 등 네 가지 반복 메커니즘을 제공한다.

1. while 문 : 주어진 표현식이 true인 동안 주어진 코드 블록을 계속해서 반복한다.
    1) break 키워드를 사용하면 반복문을 즉시 빠져나와 프로그램을 계속 진행한다.

2) continue 키워드를 사용하면 즉시 반복문의 첫 문장으로 돌아가서 while 문에 지정한 표현식을 다시 평가한다.
※ continue를 자주 사용하는 것은 바람직하지 않다. 프로그램의 실행 흐름이 갑작스레 건너뛰기 때문.

2. do/while 문 : 동작은 while 문과 비슷하지만, 먼저 코드 블록부터 실행한 뒤 조건을 검사하고, 그 결과에 따라 루프를 계속 진행할지 결정함
   1) 이 구문을 활용하면 코드 블록을 최소 한 번 실행하고, 그 뒤에 더 실행할 지는 주어진 조건에 따라 결정할 수 있다.
3. for 문 : for 문으로 작성한 코드는 모두 while 문으로 변환할 수 있고, 그 반대도 가능하다. for의 문법이 좀 더 편할 때가 많다.
   1) 초기 표현식, 종료 조건, 매번 반복이 끝날 때마다 실행할 문장으로 반복문을 구성
4. 범위 기반 for 문 : 컨테이너에 담긴 원소에 대해 반복문을 실행하는 데 편함
   1) C 스타일의 루프, 초기자 리스트, std::array, std::vector, 표준 라이브러리에서 제공하는 모든 컨테이너 반복자를 리턴하는 begin()과 end() 메서드가 정의된 모든 타입과 표준 라이브러리에서 제공하는 모든 컨테이너에 적용할 수 있음
   2) C++20부터는 범위 기반 for 문에서도 if 문이나 switch 문처럼 초기자를 사용할 수 있음
      - for (**<초기자>**; <for-범위-선언> : <for-범위-초기자>) { <본문> }
      - <초기자>에 지정한 변수는 모두 <for-범위-초기자>와 <본문>에서 사용할 수 있지만, 범위 기반 for 문 밖에서는 사용할 수 없음

### 1.1.21 초기자 리스트

초기자 리스트는 <initializer_list> 헤더 파일에 정의되어 있으며, 이를 활용하면 여러 인수를 받는 함수를 쉽게 작성할 수 있다.

1. std::initializer_list 타입은 클래스 템플릿이다. 원소 타입에 대한 리스트를 꺾쇠 괄호로 묶어서 지정해야 한다.
2. 초기자 리스트는 타입에 안전하다. 그러므로 초기자 리스트를 정의할 때 지정한 타입만 허용한다.
   1) 다른 타입이 들어갈 경우 컴파일 에러 또는 경고메시지가 출력된다.

### 1.1.22 C++의 스트링

C++에서 스트링을 다루는 방법은 두 가지다.

1. C 스타일 : 스트링을 문자 배열로 표현
2. C++ 스타일 : C 스타일로 표현된 스트링을 쉽고 안전하게 사용할 수 있도록 스트링 타입으로 감싼 방식

C++의 std::string 타입은 <string> 헤더 파일에 정의되어 있고, 기본 타입처럼 사용할 수 있다.

### 1.1.23 C++의 객체지향 언어 특성

C와 달리 C++는 객체지향 언어이다.
객체지향 프로그래밍 object-oriented progamming(OOP)에서는 코드 작성 방식이 기존과 달리 훨씬 직관적이다.

1. 클래스 정의
    1) 클래스는 객체의 특성을 정의한 것이다. C++에서 클래스를 정의하는 코드는 주로 모듈 인터페이스 파일에 작성하고, 이를 구현하는 코드는 모듈 인터페이스 파일에 함께 적거나 소스 파일에 작성한다.
    2) 클래스를 구성하는 데이터 멤버와 메서드를 선언한다.
        - 각각의 데이터 멤버와 메서드마다 public, protected, private 등으로 접근 수준을 지정한다.
        - public으로 지정한 멤버는 클래스 밖에서 접근할 수 있다.
        - private으로 지정한 멤버는 클래스 외부에서 접근할 수 없다.
        - 대체로 데이터 멤버는 모두 private으로 지정하고, 이에 대한 getter나 setter를 public으로 지정한다.
        - protected는 상속과 관련이 있다.
    3) 모듈 인터페이스 파일을 작성할 때는 작성하려는 모듈을 반드시 export module 선언문으로 시작해야 한다. 또한 그 모듈을 사용하는 이들에게 제공할 타입을 명시적으로 익스포트해야 한다.
    4) 클래스와 이름이 같고 리턴 타입이 없는 메서드를 **생성자 constructor**라 부른다. 이 메서드는 해당 클래스의 객체를 생성할 때 자동으로 호출된다.
    5) 생성자와 형태는 같지만 앞에 틸드(~)를 붙인 메서드를 소멸자 destructor라 부른다. 이 메서드는 객체가 제거될 때 자동으로 호출된다.
    6) 클래스를 구현하는 코드는 소스파일에 작성한다. 소스 파일은 첫 머리에 모듈 선언문이 나온다.
    7) 생성자로 데이터 멤버를 초기화하는 방법은 여러 가지가 있다.
        - 생성자 초기자 constructor initializer : 생성자 이름 뒤에 콜론( : )을 붙여서 표현
        - 생성자의 본문에서 초기화하는 방법
        - 생성자에서 다른 일은 하지 않고 데이터 멤버를 초기화하는 일만 한다면 굳이 생성자를 따로 정의할 필요가 없다. 클래스 정의 코드 안에서 곧바로 초기화할 수 있기 대문이다. (**클래스 내부 초기자**)
    8) 클래스에서 파일을 열거나 메모리를 할당하는 것처럼 다른 타입에 대한 초기화 작업은 생성자에서 처리해야 한다.
    9) 소멸자에서 할 일이 없을 때는 클래스에서 소멸자 코드를 생략해도 된다.
        - 파일을 닫거나 메모리를 해제하는 등의 정리 작업이 필요하다면 소멸자를 작성해야 한다.
2. 클래스 사용 : 클래스를 사용하려면 먼저 모듈을 임포트해야 한다.

### 1.1.24 스코프 지정

어떤 이름이 디폴트 스코프 지정 범위에 적용되지 않게 하려면 그 이름 앞에 스코프 지정 연

산자인 ::를 이용하여 원하는 스코프를 지정하면 된다.

1. 글로벌 스코프에는 스코프 이름이 없지만 (접두어 없이) 스코프 지정 연산자만 작성해서 글로벌 스코프에 직접 접근할 수 있다.
2. 네임스페이스를 정의할 때 별도로 이름을 지정하지 않은 상태로 글로벌 스코프에 있는 함수와 동일한 함수가 있으면 컴파일 에러가 발생한다.

### 1.1.25 균일 초기화 Uniform initialization

1. 균일 초기화
   1) C++11 이전에는 타입의 초기화 방식이 일정하지 않았다. C++11 이전에는 Struct 타입 변수와 Class 타입 변수를 초기화하는 방법이 서로 달랐다.
      구조체는 { ... } 문법을 적용한 반면 클래스에 대해서는 함수 표기법인 (...)로 생성자를 호출했다.
      C++11부터 타입을 초기화할 때 {...} 문법을 사용하는 균일 초기화 uniform initialization(중괄호 초기화, 유니폼 초기화)로 통일되었다.
   2) C++ 11 이전
      CircleStruct myCircle1 = { 10, 10, 2.5 };
      CircleClass myCircle2(10, 10, 2.5);
   3) C++ 11 부터 ~
      CircleStrucy myCircle3 = { 10,10,2.5 };
      CircleClass myCircle4 = { 10,10,2.5 }; => CircleClass의 생성자가 자동으로 호출
      - 또한 등호를 생략해도 된다.
        CircleStrucy myCircle5 { 10,10,2.5 };
        CircleClass myCircle6 { 10,10,2.5 };
   4) 균일 초기화 구문은 구조체나 클래스뿐만 아니라 C++에 있는 모든 대상을 초기화 하는데 적용됨
      - int a = 3;
      - int b(3);
      - int c = { 3 }; // 균일 초기화
      - int d{ 3 }; // 균일 초기화
   5) 균일 초기화는 변수를 **영 초기화zero initialization(제로 초기화)** 할 때도 적용할 수 있음
      - int e { };
      - *※ 영 초기화란 주어진 객체를 디폴트 생성자로 초기화하는 것으로, 기본 정수타입은 0으로, 기본 부동소수점 타입은 0.0으로, 포인터 타입은 nullptr로 초기화
   6) 균일 초기화를 사용하면 축소 변환 narrowing(좁히기)을 방지할 수 있음
      - C++에서는 암묵적으로 축소 변환될 때가 있음. 축소 변환으로 인한 버그가 발생할 수 있으므로 무시해서는 안된다.
      - C++11 에서는 균일 초기화를 사용할 때 에러 메시지가 생성된다.
      - 축소 변환 캐스트를 하려면 GSL Guideline Support Library에서 제공하는 gsl::narrow_cast() 함수를 사용
   7) 균일 초기화는 동적으로 할당하는 배열을 초기화할 때도 적용할 수 있음

- int * pArray = new Int[4] {0,1,2,3};
- C++20부터는 배열의 크기를 생략할 수 있음
  - int * pArray = new Int[] {0,1,2,3};

8) 생성자의 초기자에서 클래스 멤버로 정의한 배열을 초기화할 때도 사용할 수 있음
9) 균일 초기화는 vector와 같은 표준 라이브러리 컨테이너에도 적용할 수 있음.
10) 변수 초기화에 대입 구문을 이용하기보다는 균일 초기화를 사용하는 것이 바람직하다.

2. 지정 초기자

C++20부터 지정 초기자 designated initializer가 도입되었음. 이 초기자는 묶음 타입의 데이터 멤버를 초기화하는 데 사용된다.

1) 묶음 타입 aggregate type이란
   - public 데이터 멤버만 갖고,
   - 사용자 정의 생성자나 상속된 생성자가 없고,
   - virtual 함수도 없으며,
   - virtual, private, protected 베이스 클래스도 없는

   배열 타입 객체나 구조체 객체, 클래스 객체를 말한다.
2) 지정 초기자는 점 뒤에 데이터 멤버의 이름을 적는 방식으로 표기한다.
   - 지정 초기자에 나오는 데이터 멤버는 반드시 데이터 멤버가 선언된 순서를 따라야함
   - 지정 초기자와 비지정 초기자를 섞어 쓸 수는 없음
   - 지정 초기자로 초기화되지 않은 데이터 멤버는 모두 디폴트값으로 초기화된다.
     - 클래스 내부 초기자를 가진 데이터 멤버는 거기서 지정된 값을 갖게 된다.
     - 클래스 내부 초기자가 없는 데이터 멤버는 0으로 초기화된다.
3) 지정 초기자를 사용하면 균일 초기자를 사용할 때보다 초기화할 대상을 훨씬 쉽게 파악할 수 있다.
   - Employee anEmployee{
     .firstInitial = 'J',
     .lastInitial = 'D',
     .employeeNumber = 42,
     .salary = 80'000
     };
   - 지정초기자를 사용할 때 주어진 디폴트값을 사용하고 싶은 멤버에 대해 초기화를 생략할 수 있음.

     employeeNumber에 대한 초기화를 생략 시 (클래스 내부 초기자가 없으면 0으로 초기화)

     Employee anEmployee{
     .firstInitial = 'J',
     .lastInitial = 'D',
     .salary = 80'000
     };

- 균일 초기자 구문을 사용할 때는 이렇게 생략할 수 없고, employeeNumber 자리에 0을 지정해야 한다.
  Employee anEmployee { 'J', 'D', 0, 80'000 };

4) 구조체에 멤버를 추가하더라도 지정 초기자를 이용한 기존 코드는 그대로 작동한다. 새로 추가된 데이터 멤버는 디폴트값으로 초기화됨.

### 1.1.26 포인터와 동적 메모리

동적 메모리를 이용하면 컴팡리 시간에 크기를 확정할 수 없는 데이터를 다룰 수 있다.

1. 스택과 프리스토어
   1) C++ 애플리케이션에서 사용하는 메모리는 크게 스택 Stack 과 프리스토어 free store로 나뉜다.
   2) 최상단 스택 프레임에는 주로 현재 실행 중인 함수를 가리킨다. 현재 실행 중인 함수에 선언된 변수도 최상단 스택 프레임 stack frame의 메모리 공간에 담긴다.
   3) 스택 프레임은 각 함수마다 독립적인 메모리 공간을 제공한다.
      - 함수의 실행이 끝나면 해당 스택 프레임이 삭제되고, 그 함수 안에 선언된 변수는 더 이상 메모리 공간을 차지하지 않음.
      - 스택에 할당된 변수는 프로그래머가 직접 할당 해제 deallocate(삭제) 할 필요 없이 자동으로 처리 됨.
   4) 프리스토어 free store는 현재 함수 또는 스택 프레임과는 완전히 독립적인 메모리 공간
      - 함수가 끝난 후에도 그 안에서 사용하던 변수를 유지하고 싶으면 프리스토어에 저장한다.
      - 프리스토어는 스택보다 구조가 간결함.
      - 프리스토어에 할당된 메모리 공간은 직접 할당 해제(삭제) 해야함.
      - 스마트 포인터를 사용하지 않는 한 자동으로 할당 해제되지 않음.
2. 포인터 사용법
   1) 메모리 공간을 적당히 할당하기만 하면 어떠한 값이라도 프리스토어에 저장할 수 있음.
   2) 정수값을 프리스토어에 저장하려면 정수 타입에 맞는 메모리 공간을 할당해야하고, 포인터를 선언해야 함
   3) 포인터를 초기화하지 않고 사용하면 crash가 일어나므로 포인터 변수는 항상 선언하자마자 초기화한다.
   4) 널 포인터란 정상적인 포인터라면 절대로 가지지 않을 특수한 값이며, 부울 표현식에서는 false로 취급한다.
   5) 포인터 변수에 메모리를 동적으로 할당할 때는 new 연산자를 사용한다.
   6) 포인터가 가리키는 값에 접근하려면 포인터를 역참조 dereference 해야 한다. 역참조란 포인터가 프리스토어에 있는 실제 값을 가리키는 화살표를 따라간다는 뜻이다.
   7) 동적으로 할당한 메모리를 다 쓰고 나면 delete 연산자로 그 공간을 해제해야 한다. 해제한 포인터를 다시 사용하지 않도록 포인터 변수의 값을 nullptr로 초기화

하는 것이 좋다.

8) 포인터는 스택과 같은 다른 종류의 메모리를 가리킬 수도 있다. 원하는 변수의 포인터값을 알고 싶다면 레퍼런스 연산자인 &를 사용한다.
9) 구조체의 포인터는 * 연산자로 참조해서 구조체 자체에 접근한 뒤 필드에 접근할 때는 . 연산자로 표기한다.
10) 단락 논리를 적용하면 잘못된 포인터에 접근하지 않게 할 수 있다.

3. 동적으로 배열 할당하기
   1) 배열을 동적으로 할당할 때도 프리스토어를 활용한다. 이때 new[] 연산자를 사용한다.
   2) C에서 사용하던 malloc ( )이나 free ( )는 사용하지 말고, new와 delete 또는 new[ ] 와 delete[ ]를 사용한다.
   3) 메모리 누수가 발생하지 않도록 new를 호출할 때마다 delete도 쌍을 이루도록 호출한다. new[ ]와 delete[ ]도 마찬가지다.
4. 널 포인터 상수
   1) C++11 이전에는 NULL이란 상수로 널 포인터를 표현했다. NULL은 실제로 상수 0과 같아서 문제가 발생할 여지가 있다.
   2) NULL은 포인터가 아니라 정수 0에 해당하기 때문에 매개변수로 전달 시에 포인터를 인수로 받는 함수를 호출하는 것이 아닌 정수를 인수로 받는 함수가 호출되는 경우과 같이 의도와 달리 동작할 수 있다. 이럴 때는 실제 널 포인터 상수인 nullptr을 사용하면 해결할 수 있다.

### 1.1.27 const의 다양한 용도

C++에서 const 키워드는 다양하게 사용된다.

- const 키워드는 상수를 의미하는 'constant'의 줄임말로서 변경되면 안 될 대상을 선언할 때 사용한다.
- 컴파일러는 const로 지정한 대상을 변경하는 코드를 발견하면 에러를 발생시킨다.
- const로 지정한 대상을 최적화할 때 효율을 더욱 높일 수 있다.

1. const 상수
   1) C 언어에서는 버전 번호처럼 프로그램을 실행하는 동안 변경되지 않을 값에 이름을 붙일 때 전처리 구문인 #define을 주로 사용했음
   2) C++에서는 상수를 #define 대신 const로 정의하는 것이 바람직함
   3) const는 글로벌 변수나 클래스 데이터 멤버를 비롯한 거의 모든 변수에 붙일 수 있음
   4) const 포인터
      - 포인터로 가리키는 값이 수정되지 않게 하려면 const 키워드를 포인터 타입 변수(* ip)의 선언문에 붙인다.
        const int* ip or int const* ip;
        ip가 가리키는 값을 변경할 수 없게 된다.
      - 변경하지 않게 하려는 대상이 ip 자체라면 const를 ip 변수에 바로 붙인다.
        int* const ip { nullptr };

ip 자체를 변경할 수 없게 되기 때문에 이 변수를 선언과 동시에 초기화해야 한다.

- 포인터와 포인터가 가리키는 값을 모두 const로 지정할 수 있다.
  int const* const ip{ nullptr} / const int* const ip{ nullptr }
- const 키워드는 항상 바로 왼쪽에 나온 대상에 적용된다.

5) const 매개변수

- C++에서는 비 const 변수를 const 변수로 캐스트할 수 있다.
  다른 코드에서 변수를 변경하지 못하게 보호할 수 있다.
- const는 기본 타입 매개변수에도 붙일 수 있다.
  함수 본문에서 매개변수를 변경하지 못하게 할 수 있음.

2. const 메서드
   1) const는 클래스 메서드에도 지정할 수 있다. 해당 클래스의 데이터 멤버를 수정할 수 없게 만든다.
   2) 읽기 전용 메서드를 const로 지정할 경우, 데이터 멤버를 수정하려고 하면 컴파일 에러가 발생하게 된다.

### 1.1.28 constexpr 키워드

C++는 상수 표현식 constant expression 을 제공한다. 상수 표현식이란 컴파일 시간에 평가되는 표현식이다.

1. 배열의 크기를 정의할 때는 크기를 상수 표현식으로 지정해야 한다.
2. 함수에 constexpr을 적용하면 그 함수에 상당히 많은 제약사항이 적용된다. (컴파일 시간에 평가해야하기 때문에..)
   1) constexpr 함수는 다른 constexpr 함수를 호출할 수 있지만, constexpr이 아닌 함수는 호출할 수 없음
      이런 함수는 부작용 side effect이 발생해서도 안 되고 익셉션을 던질 수도 없기 때문에
   2) constexpr 함수는 C++의 고급 기능에 해당한다.
   3) constexpr 생성자를 정의하면 사용자 정의 타입에 대한 상수 표현식 변수를 만들 수 있다.
   4) constexpr 클래스를 적용하면 상당히 많은 제약사항이 적용된다.

#### 1.1.29 consteval 키워드

constexpr 키워드는 함수가 컴팡리 시간에 실행될 수도 있다고 지정할 뿐 반드시 컴파일 시간에 실행되도록 보장하는 것은 아니다.

1. 함수가 항상 컴파일 시간에 평가되도록 보장하고 싶다면 C++20부터 제공하는 consteval 키워드로 해당 함수를 즉시 실행 함수 immediate function로 만든다.

### 1.1.30 레퍼런스

- C++에서 말하는 레퍼런스란 변수에 대한 앨리어스 alias다.
- 레퍼런스에 대해 수정한 내용은 그 레퍼런스가 가리키는 변수의 값에 그대로 반영된다.

- 레퍼런스는 변수의 주소를 가져오거나 변수에 대한 역참조 연산을 수행하는 작업을 자동으로 처리해주는 특수한 포인터라고 볼 수 있음.
- 또는 변수에 대한 다른 이름(별칭)이라고 생각해도 된다.

1. 레퍼런스 변수
    1) 레퍼런스 변수는 반드시 생성하자마자 초기화해야 함.
    2) 변수의 타입 뒤에 &를 붙이면 그 변수는 레퍼런스가 됨.
    3) 레퍼런스 변수를 클래스 밖에서 선언만 하고 초기화하지 않으면 컴파일 에러가 발생함.
    4) 레퍼런스는 처음 초기화할 때 지정한 변수만 가리킨다. 한번 생성되고 나면 가리키는 대상을 바꿀 수 없음.
        - 한 번 선언된 레퍼런스에 다른 변수를 대입하면 레퍼런스가 가리키는 대상이 바뀌는 것이 아니라 레퍼런스가 원래 가리키던 변수의 값이 새로 대입한 변수의 값으로 바뀌게 됨
    5) const 레퍼런스
        - 레퍼런스는 가리키는 대상을 변경할 수 없기 때문에 기본적으로 const 속성을 갖는다. 그러므로 명시적으로 const로 지정할 필요가 없음.
        - 레퍼런스에 대한 레퍼런스를 만들 수 없기 때문에 참조가 한 단계뿐이다. 여러 단계로 참조하려면 포인터에 대한 레퍼런스를 만들 수 밖에 없음.
        - C++에서 const 레퍼런스
          ```
          int z;
          const int& zRef { z };
          zRef = 4;    // 컴파일 에러 발생
          ```
          zRef를 const라고 지정한 것은 z에 영향을 미치지 않는다. 그러므로 zRef를 거치지 않고 z에 접근하면 값을 변경할 수 있다.
        - 정수 리터럴처럼 이름 없는 값에 대해서는 레퍼런스를 생성할 수 없음. 단, const 값에 대해서는 레퍼런스를 생성할 수 있음.
        - 임시 객체에 대한 비 const 레퍼런스는 만들 수 없지만 const 레퍼런스는 얼마든지 만들 수 있음.
    6) 포인터에 대한 레퍼런스와 레퍼런스에 대한 포인터
        - 포인터에 대한 레퍼런스도 만들 수 있음
        - 레퍼런스가 가져온 주소는 그 레퍼런스가 가리키는 변수의 주소와 같다.
        - 레퍼런스에 대한 레퍼런스를 선언할 수 없음.
    7) 구조적 바인딩과 레퍼런스
        - 구조적 바인딩에 레퍼런스를 적용하여 분해할 수 있음.
2. 레퍼런스 데이터 멤버
    1) 클래스의 데이터 멤버도 레퍼런스 타입으로 정의할 수 있음.
    2) 레퍼런스 데이터 멤버는 반드시 **생성자 초기자**에서 초기화해야 함.
3. 레퍼런스 매개변수
    1) 레퍼런스 변수나 레퍼런스 데이터 멤버를 별도로 선언해서 사용하는 일은 많지 않음. 레퍼런스는 주로 함수나 메서드의 매개변수로 많이 사용한다.

2) 매개변수는 값 전달 방식 pass-by-value을 따르기 때문에 함수는 인수의 복사본을 받는다. C에서처럼 포인터를 사용하여 역참조하여 메모리를 변경하면 포인터 연산이 많아져서 간단한 작업이라도 코드가 복잡해진다.
3) C++에서는 레퍼런스 전달 방식 pass-by-reference을 제공한다. 매개변수가 포인터값이 아닌 레퍼런스로 전달된다.
4) 레퍼런스로 전달하는 매개변수에 리터럴을 지정하면 컴파일 에러가 발생한다. 이럴 때는 const 레퍼런스로 전달한다.
5) 매개변수가 레퍼런스인 함수나 메서드에 포인터를 전달할 때에는 역참조해서 레퍼런스로 변환할 수 있다.
6) const 레퍼런스 전달 방식
    - const 레퍼런스 매개변수의 가장 큰 장점은 성능이다. const 레퍼런스로 전달하면 복제되지도 않고 원본 변수가 변경되지도 않는 장점을 모두 취할 수 있다.
      const 레퍼런스는 특히 객체를 다룰 때 유용하다. 객체는 대체로 크고 복제하는 동안 의도하지 않은 부작용이 발생할 수 있기 때문이다.
7) 레퍼런스 전달 방식과 값 전달 방식
    - 효율 : 큰 객체는 복제하는 데 시간이 오래 걸릴 수 있음. 하지만 레퍼런스 전달 방식은 객체에 대한 레퍼런스만 함수에 전달한다.
    - 지원 : 값 전달 방식을 허용하지 않는 클래스가 있음.
    - 이런 장점을 활용하면서 원본 객체를 수정할 수 없게도 만들고 싶다면 매개변수를 const 레퍼런스로 선언한다.
    - 값 전달 방식은 인수가 int나 double과 같이 함수 안에서 따로 수정할 일이 없는 기본 타입일때만 사용하는 것이 좋다.
    - 함수에 객체를 전달할 때는 값으로 전달하기보다는 const 레퍼런스로 전달하는 것이 좋다. 전달할 객체를 함수 안에서 수정해야 할 때는 비 const 레퍼런스로 전달한다.

4. 레퍼런스 리턴값
    1) 함수나 메서드의 리턴값도 레퍼런스 타입으로 지정할 수 있다. 함수 종료 후에도 계속 남아 있는 객체에 대해서만 레퍼런스로 리턴할 수 있다. (스코프가 함수를 벗어나지 않는 로컬 변수는 레퍼런스로 리턴하면 안된다)
    2) 레퍼런스를 리턴해야 하는 주된 경우는 리턴값을 대입문의 왼쪽에 나오는 lvalue(좌측ㄱ값)에 직접 대입 할 때다. =, +=등을 오버로딩했을 때는 레퍼런스를 리턴하는 경우가 많다.
5. 레퍼런스와 포인터의 선택 기준
    1) 레퍼런스를 사용하면 코드를 깔끔하고 읽기 쉽게 작성할 수 있음.
    2) 포인터보다 훨씬 안전하다.
    3) 레퍼런스의 값은 널이 될 수 없고, 레퍼런스를 명시적으로 역참조할 수도 없다.
    4) 포인터처럼 역참조 과정에서 에러가 발생할 가능성이 없다.
    5) 단, 포인터가 하나도 없을 때만 레퍼런스가 더 안전하다고 말할 수 있다.
    6) 포인터를 사용한 코드는 거의 대부분 레퍼런스로 표현할 수 있다.

7) 반드시 포인터를 써야 하는 경우가 있다. 대표적인 예로 가리키는 위치를 변경해야 하는 경우.
8) 동적 할당 메모리의 주소는 레퍼런스가 아닌 포인터에 저장해야 한다.
9) 주소값이 nullptr이 될 수도 있는 optional타입은 반드시 포인터를 사용해야 한다.
10) 컨테이너에 다형성 타입을 저장할 때도 포인터를 사용해야 한다.
11) 소유권을 이전해야 할  때 일반 포인터보다 스마트 포인터 smart pointer를 사용하는 것이 바람직하다.
12) 포인터보다 레퍼런스를 사용하는 것이 좋다. 레퍼런스로 충분하지 않을 때만 포인터를 사용한다.
13) 배열을 동적으로 할당하는 식으로 작성하지 않는 것이 좋다.
14) 결과를 매개변수로 전달하는 방식은 가급적 사용하지 않는 것이 좋다. **출력 매개변수가 아닌 리턴문을 사용**한다.
15) return object; 와 같이 작성하면 object가 로컬 변수거나, 함수에 대한 매개변수거나, 임시값일 때 **리턴값 최적화 return value optimization(RVO)**가 적용된다. object가 로컬 변수라면 **이름 있는 리턴값 최적화 named return value optimization(NRVO)**도 적용된다.
둘 다 복제 생략 copty elision의 한 종류로서, 함수에서 객체를 리턴하는 과정을 효율적으로 처리해준다.
복제 생략 시 컴파일러는 함수에서 리턴하는 객체를 복제 하지 않는다. 복제 없는 값 전달 방식 zero-copty pass-by-value을 구현한다.

### 1.1.31 const_cast()

C++에서는 모든 변수는 항상 특정한 타입을 따른다. 경우에 따라서 다른 타입으로 캐스트할 때가 있다.

C++는 const_cast(), static_cast(), reinterpret_cast(), dynamic_cast() 그리고 C++20부터 추가된 std::bit_cast()라는 캐스트 방법을 제공한다.

1. const_cast()는 변수에 const 속성을 추가하거나 제거하는 다섯 가지 캐스트 방법 중에서 가장 이해하기 쉬우며, 유일하게 const 속성을 제거하는 기능을 제공한다.
2. 변수를 const로 지정했다는 말은 const 상태를 유지하겠다는 뜻이기 때문에 const_cast()를 사용할 일이 없어야 한다.
    1) const 매개변수를 받도록 정의했는데, 그 값을 다시 비 const 매개변수를 받는 함수에 전달해야 하는 경우가 생길 수 있으며 그 함수는 비 const 인수를 수정하지 않는다는 것을 보장할 수 있다고 하자.
    2) 정석대로 처리하려면 프로그램 전체에서 일관되게 const로 지정해야 하지만 서드파티 라이브러리처럼 수정할 수 없는 경우에는 부득이 const 속성을 일시적으로 제거할 수 밖에 없다.
    3) 호출할 함수가 객체를 변경하지 않는다고 보장될 때만 이렇게 처리해야 한다. 그렇지 않으면 프로그램을 수정하는 수밖에 없다.
3. 표준 라이브러리는 std::as_const()란 헬퍼 메서드를 제공한다. 이 메서든느 <utility> 헤더에 정의되어 있으며, 레퍼런스 매개변수를 const 레퍼런스 버전으로 변환해준다.

1) 기본적으로 as_const(obj)는 const_cast<const T&> (obj)와 같다.
2) 비 const를 const로 캐스트할 때는 const_cast()보다 as_const()를 사용하는 것이 훨씬 간결하다.

### 1.1.32 익셉션

C++는 굉장히 유연한 반면 안전한 편은 아니다. 메모리 공간을 무작위로 접근하거나 0으로 나누는 연산을 수행하는 코드를 작성해도 컴파일러는 내버려둔다. 그러므로 C++의 안전성을 좀 더 높이기 위해 익셉션 exception(예외) 이라는 기능을 제공한다.

1. 익셉션이란 예상하지 못한 상황을 말한다. 예상하지 못한 상황을 적절히 처리하기 위해 그 함수에서 nullptr이나 에러 코드와 같은 특수한 값을 리턴할 수도 있지만 익셉션을 활용하면 문제가 발생했을 때 좀 더 융통성 있게 대처할 수 있다.

### 1.1.33 타입 앨리어스

타입 앨리어스 type alias란 기존에 선언된 타입에 다른 이름을 붙이는 것이다. 타입을 새로 정의하지 않고 기존 타입 선언에 대한 동의어를 선언하는 문법이라 생각할 수 있다.

- uisng IntPtr = int*;
  - int* p1;
  - IntPtr p2;
- 새로 정의한 타입 이름으로 생성한 변수는 기존 타입 표현으로 생성한 변수와 완벽히 호환됨
  - p1 = p2;
  - p2 = p1;

1. 타입 앨리어스는 복잡하게 선언된 타입 표현을 좀 더 간편하게 만들기 위한 용도로 많이 사용함
   템플릿을 이용할 때 이런 경우가 많다.
2. 표준 라이브러리에서 제공하는 스트링은 std::basic_string<T>와 같이 작성해야 한다. 이 표현은 템플릿 클래스로서 T는 스트링을 구성하는 각 문자의 타입(예:char)을 가리킨다.
   이런 타입을 언급할 때마다 반드시 템플릿 매개변수도 함께 지정해야 한다.
3. 변수를 선언할 때뿐만 아니라 함수 매개변수나 리턴 타입을 지정할 때도 basic_string<char>와 같이 적어줘야 함.
   1) basic_string<char>를 사용할 일이 많기 때문에 표준 라이브러리는
   using string = basic_string<char>
   와 같이 간결하고 의미가 분명히 드러나는 타입 앨리어스를 제공한다.

### 1.1.34 typedef

타입 앨리어스는 C++11부터 도입되었다. 그전에는 타입 앨리어스로 하는 일을 typedef로 구현해서 다소 복잡했음. 오래된 방식이지만 레거시 코드에서 종종 볼 수 있음.

- using IntPtr = int*
- typedef int* IntPtr;
- 선언하는 순서가 반대라서 가독성이 훨씬 떨어진다.
- typedef는 기본적으로 타입 앨리어스와 같지만 완전히 똑같은 것은 아니다.

- 템플릿에 활용할 때는 typedef보다 타입 앨리어스를 사용하는 것이 훨씬 유리하다.

### 1.1.35 타입 추론 type inference

타입 추론은 표현식의 타입을 컴파일러가 스스로 알아내는 기능이다. 타입 추론과 관련된 키워드로

- auto
- decltype

이 있다.

1. auto 키워드
   1) auto 키워드는 다양한 상황에서 사용함
      - 함수의 리턴 타입을 추론
      - 구조적 바인딩에 사용
      - 표현식의 타입을 추론하는 데 사용
      - 비타입(not-type, 타입이 아닌) 템플릿 매개변수의 타입을 추론하는 데 사용
      - 축약 함수 템플릿(abbreviated function template) 구문
      - decltype (auto)
      - 함수에 대한 또 다른 문법으로 사용
      - 제네릭 람다 표현식에서 사용
   2) auto&

      auto를 표현식 타입을 추론하는 데 사용하면 레퍼런스와 const가 제거된다.
      - auto를 지정하면 레퍼런스와 const 한정자가 사라지고 값이 **복제**된다. const 레퍼런스 타입으로 지정하려면 auto 키워드 앞뒤에 레퍼런스 타입과 const 타입을 붙인다.
      - const auto& f2 { foo() };
      - as_const() : 매개변수에 대해 const 레퍼런스를 리턴
        - as_const()와 auto를 조합할 때 주의할 필요가 있음
        - auto는 레퍼런스와 const를 제거하기 때문에 값이 복제되어버릴 수 있다.
   3) auto*

      포인터를 다룰 때는 auto* 구문을 사용하여 대상이 포인터임을 명시적으로 드러내는 것이 바람직하다.
      - auto*를 사용하면 auto, const, 포인터를 함께 쓸 때 발생하는 이상한 동작도 방지할 수 있다.
      - const auto p1{ &i } / auto const p2 { &i }
        - int* const다. (비 const 정수에 대한 const 포인터)
      - auto* 와 const를 함께 쓰면 의도한 대로 작동한다.
      - const auto* const p5 { &i } *를 생략하면 이 작업을 수행할 수 없음.
   4) 복제 리스트 초기화와 직접 리스트 초기화

      초기화 구문은 두 가지가 있으며, 초기자 리스트를 중괄호로 묶어서 표현한다.
      - 복제 리스트 초기화(copy list initialization): T obj = { arg1, arg2, ... }
      - 직접 리스트 초기화(direct list initialization): T obj { arg1, arg2, ... };
      - C++17 부터는 auto 타입 추론 기능과 관련하여 복제 리스트 초기화와 직

접 리스트 초기화의 차이가 커짐

- □ 복제 리스트 초기화에서 중괄호 안에 나온 원소는 반드시 타입이 모두 같아야 한다.
- □ 이전 버전(C++11/14)에서는 복제 리스트 초기화와 직접 리스트 초기화 둘 다 initializer_list<>로 처리했다.

2. decltype 키워드

decltype 키워드는 인수로 전달한 표현식의 타입을 알아낸다.

1) decltype은 레퍼런스나 const 지정자를 삭제하지 않는다는 점에서 auto와 다르다
2) decltype(foo()) f2 { foo() }
   - decltype으로 정의하면 const string& 타입이 되어 복제 방식으로 처리하지 않음
3) decltype은 템플릿을 사용할 때 상당히 강력한 효과를 발휘한다.

### 1.1.36 표준 라이브러리

C++는 표준 라이브러리를 제공한다. 유용한 클래스가 다양하게 정의되어 있다.
C++를 사용할 때 표준 라이브러리를 활용하면 C 방식보다 훨씬 쉽고 안전하게 구현할 수 있다.
std::string, std::array, std::vector, std::pair, std::optional 등