Chapter1) Deducing type

Introducing

`C++98`에서는 type deduction을 할 수 있는 방법은 오직 함수 템플릿이었다.
그러나 c++가 발전함에 따라, 다양한 타입 추론 방법이 생겼다. `C++11`은 `auto`키워드와 `decltype`을 사용하게 되었으며, 현재 `C++14`에 들어서서는 `decltype`과 `auto`를 더 많이 사용하게 되었다.

타입추론의 장단점
장점
타입추론을 사용하게 되면 타이핑할 양이 줄어든다.
단점
compiler가 개발자의 의도와 맞지 않는 타입을 추론하는 경우가 있다.

Chapter1에서는 어떻게 auto가 만들어지게 되었으며, decltype에 대해 알아본 다음, 컴파일러가 어떻게 타입을 추론하는지 알아보도록 하자.

Item 1 : Understanding template type deduction

들어가기에 앞서

`auto`의 타입 추론 방법은 `template`의 타입 추론 방식을 기반으로 한다. 그러나 `완벽하게 동일한 방식으로 진행하지는 않는다.` 이에 많은 개발자들이 auto타입 추론을 어려워한다.

auto의 타입추론을 이해하기 위해서, template타입 추론 방식을 한 번 알아보도록 하자.

parameter와 argument의 차이점

```
template<typename T>
void f(paramType param);

f(expr);
```

1. 템플릿 인자와 매개변수 타입은 다르다
2. 매개변수 타입은 인자의 타입 외에도 정의된 수식어에 영향을 받는다

컴파일이 실행이 되면, 컴파일러는 `expr`을 사용해서 `T`와 `paramType`을 추론하게 된다. 당연하게도 `argument와 parameter type`은 다르다. `parameter type`은 수식어(const, reference)에 영향을 받기 때문이다.

```
template<typename T>
void f(const T& param); // paramType is const T&

int x = 0;

f(x); // call f with an int
```

위의 예시를 보면, x는 `int`가 된다. 이에 반해 f의 paramter는 `const int&`가 된다.

case는 크게 3가지로 나누어진다.

- **1. paramType이 pointer혹은 reference이지만 universal reference가 아닌 경우**
- **2. paramType이 universal reference인 경우**
- **3. paramType이 pointer와 reference가 아닌 경우**

Case 1 : ParamType is a Reference or Pointer, but not a Universal Reference

만약 ParamType이 Universal Reference가 아닌 reference거나 pointer인 경우 아래와 같은 절차를 따른다

```
1. 만약 `expr`타입이 reference인 경우, reference 부분을 무시한다.
2. T를 구하기 위해서는 expr타입을 paramType으로 패턴매칭을 진행한다.
```

```
template<typename T>
void f(T& param); // param is a reference

int x = 27; // x is an int
const int cx = x; // cx is const int
const int& rx = x; // rx is a reference to x as a const int

f(x); // T is int and param's type is int&
```

```
f(cx); // T is const int, param's type is const int&
f(rx); // T is const int, param's type is const int&
```

```
template<typename T>
void f(const T& param); // param is now a ref-to-const

int x = 27; // x is an int
const int cx = x; // cx is const int
const int& rx = x; // rx is a reference to x as a const int

f(x); // T is int, param's type is const int&
f(cx); // T is int, param's type is const int&
f(rx); // T is int, param's type is const int&
```

Case 2: paramType is a Universal Reference(&&)

`Universal Reference란??` : &&

위의 경우에서는 rvalue를 가르키는 것 같지만, 함수의 argument에 lvalue가 전달되었는지, rvalue가 전달되었는지에 따라서 다르게 작동한다.

```
1. 만약 expr이 lvalue라면 T와 ParamType은 lvalue reference로 추론된다. 이는 특별한 경우이며, 일반적인 템플릿 타입 추론에서 `T`가 참조형으로 추론되는 유일한 경우다
2. 만약 expr이 rvalue라면 일반적인(1번 case) 법칙이 사용된다.
```

```
template<typename T>
void f(T&& param);

int x = 27;
const int cx = x;
const int& rx = x;

f(x); // x is lvalue, so T is int&, param's type is also int&
f(cx); // cx is lvalue, so T is const int&, param's type is also const int&
f(rx); // rx is lvalue, so T is const int&, param's type is also const int&
f(27); // 27 is rvalue, so T is int, param's type is therefore int&&
```

Case3: paramType is Neither a Pointer nor a Reference

만약 paramType이 pointer와 reference가 아닌 경우, `pass-by-value` 이다.

이 뜻은 param는 어떤객체가 전달되더라도, `복사본` 이 전달될 것이며, 완전하게 `새로운 객체` 를 가지게 된다. param이 새로운 object라는 것은 T가 expr에의해 추론되는 법칙을 알려준다

```
1. 만약 expr의 type이 reference라면, reference부분을 무시한다
2. 만약 expr의 reference를 지웠는데, expr이 const라면 그것 또한 무시한다. volatile도 마찬가지이다.
```

📌 `volatile` 객체는 많이 사용하는 객체는 아니다, device driver에 많이 사용된다.

```
int x = 27;
const int cx = x;
cons tint& rx = x;

f(x); // T and param's type are both int
f(cx) // T and param's types are again both int
f(rx); // T's and param's types are still both int
```

정리

1. 템플릿의 타입 추론을 하는 경우, reference성을 가지고 있는 argument들은 non-reference로 취급한다.
2. universal reference parameter들을 추론하는 경우, lavalue argument들은 특별한 취급을 당한다.
3. value-parameter를 추론하는 경우 `const` 와 `volatile` argument들은 `non-const` , `non-volatile` 로 취급된다.

Item 2: Understanding `auto` type deduction

기본적으로 template과 auto의 타입 추론은 비슷하다.

```
template<typename T>
void f(paramType param);

f(expr); // call f with some expression
```

📌template `T`는 expr을 통해 타입 추론을 진행한다.

auto는 어떻게 타입추론을 하나요??
기본적으로 변수가 `auto`로 선언이 된다면 `auto`는 template에서 `T`역활을 맡는다. 이에 반해 변수의 type specifier는 `paramType`과 같은 역활을 한다.

```
auto x = 27;
const auto cx = x;
const auto& rx = x;


template<typename T> // conceptual template for
void func_for_x(T param); // deducing x's type
func_for_x(27); // conceptual call: param's
                // deduced type is x's type

template<typename T> // conceptual template for
void func_for_cx(const T param); // deducing cx's type
func_for_cx(x); // conceptual call: param's
                // deduced type is cx's type

template<typename T> // conceptual template for
void func_for_rx(const T& param); // deducing rx's type
func_for_rx(x); // conceptual call: param's
                // deduced type is rx's type
```

결과

1. x = `int`
2. cx = `const int`
3. rx = `const int&`

auto또한 template의 타입 추론과 비슷한 과정을 거치기 때문에 3가지 Case가 있다.

```
1. type specifier가 pointer와 reference이고, universal reference가 아닌 경우
2. type specifier가 universal reference인 경우
3. type specifier가 pointer와 reference모두 아닌 경우
```

auto와 template의 다른 점이 뭔가요??

auto와 template의 차이점을 알아보기 전에 변수의 선언의 4가지 방법을 알아볼 필요가 있다.

```
int x1 = 1;
int x2(1);

C++11
int x3 = {1};
int x4{1};
```

아래의 2가지 방법은 `initializer_list`를 사용해서 객체를 선언하는 것이다.
이를 `auto`를 사용하면 아래와 같이 변경할 수 있다.

```
auto x1 = 1;
auto x2(2);
auto x3 = {1};
auto x4{1};
```

컴파일이 진행되면 x3과 x4는 완전하게 다른 의미를 가진다. 앞서 살펴본 바와 같이, `{}`을 사용하게 되면 `initializer_list`를 사용하는 것이다. 즉 `auto`를 사용하게 되면, x3과 x4는 `int`가 아닌 `initializer_list<int>` 객체 추론이 진행된다.

따라서 아래와 같은 코드는 컴파일이 안된다

```
auto x5 = {1,2,3.0}; // error can't deduce T for std::initializer_list<T>
```

⭐위의 statement에서는 2번의 type deduction이 일어난다.

먼저 `x5`의 타입추론이다. `x5`는 먼저 `std::initializer_list`로 타입 추론이 일어난다.

그러나 `std::initializer_list`는 template이다. 인스턴스화를 하면 특정 타입 `T`에 관해서 `std::initializer_list<T>`를 의미한다. 이는 `T`또한 타입 추론이 진행되어야한다는 의미이다.

위의 과정에서 타입 추론이 실패한다. 왜냐하면 정수 2개와 3.0이라는 double형이 있기 때문이다.

주의점
최신 C++에서는 함수의 return type에 `auto`를 하용하게 한다. 그리고 람다 함수에도 parameter에 `auto`를 선언하게 해준다. 그러나 이런 `auto`들은 `template type deduction`을 진행한다. 그래서 `auto`를 리턴타입으로 주는 경우 `{}초기화`를 주면 컴파일이 되지 않는다.

```
auto createInitList()
{
        return {1,2,3}; // error : can't decuce type
}
```

```
auto resetV = [&v](const auto& newValue) {v = newValue;};

resetV({1,2,3}); // error
```

⭐C++14에서는 `auto`에 정확한 타입을 요구한다. `braced initializer`는 이러한 타입 추론을 지원하지 않는다.

정리

1. `auto type deduction`은 `template type deduction`과 비슷한 부분이 많다. 그러나 `auto deduction`은 `{}`마크를 `std::initializer_list`로 타입 추론을 한다. 그리고 그 이후에 진행되는 template type deduction을 진행하지 않는다
2. 함수의 return type과 lambda parameter에 `auto`를 사용할 수 있게 되었다. 그러나 이 경우 template type duduction을 사용하게 된다. `auto type deduction`을 진행하지 않는다. 이에 컴파일 오류가 발생하게 된다.

Understand `decltype`

들어가기에 앞서서

`decltype`이란 c++에서 특정 표현식(expression)의 타입을 컴파일러가 추론하도록 하는 키워드이다. 이랄 통해 코드의 가독성과 유지보수성을 높일 수 있다

`decltype`의 규칙

1. 단순 변수의 타입 추론 : 단순히 변수명을 전달하면 그 변수의 타입이 추론된다. 이는 *참조 타입*이나 *const*들의 특성도 포함된다.

```
int a = 5;
decltype(a) b; // b의 타입은 int
```

2. 참조 반환 : 만약 표현식이 참조 타입이라면, `decltype`은 참조 타입을 유지한다.

```
int x = 42;
int& ref = x;
decltype(ref) y = x; // y는 int& 타입
```

3. 괄호에 따라 타입이 달라질 수 있음 : 변수명을 괄호로 감싸면, 참조가 아닌 값 타입이 추론된다.

```
int a = 5;
decltype(a) b = a; // b는 int
decltype((a)) c = a; //c는 int&, 괄호로 인해 참조 타입으로 추론됨
```

4. 함수와 멤버 함수 : `decltype`을 함수의 반환 타입 추론에도 사용할 수 있다. 특히, 복잡한 반환 타입을 다루는 경우 유용하다.

후행 반환 타입(trailing return type)

정의 : 함수의 반환 타입을 함수 매개변수 리스트 뒤에 선언하는 방식. 기존의 함수 선언 방식과 달리, `매개변수를 먼저 정의한 후에` 그 매개변수를 사용하여 반환 타입을 지정할 수 있다.

기본 문법

```
auto 함수이름(parameter list) -> 반환타입{
        // function body
}
```

- `auto`는 반환 타입이 후행(return type 뒤에) 선언될 것임을 나타내며, 반환 타입은 `->`뒤에 명시된다.
- 후행 반환 타입의 주된 목적은 **함수 매개변수를 사용해 반환 타입을 지정**하는 것이다.

⭐후행 반환 타입은 반환타압이 `auto`일 때만 사용하는 것은 아니다. 후행 반환 타입은 `모든 반환 타입에` 적용이 가능하다. 그러나 주로 `복잡한 반환 타입`이나 `탬플릿 함수`에서 유용하게 사용된다.

후행 반환 타입의 필요성

1. 템플릿 함수에서 타입 추론

- 템플릿을 사용할 때 "매개변수 타입에 따라 반환타입이 변경되는 경우가 많다". 후행 반환 타입을 사용하면 매개변수로부터 반환 타입을 자연스럽게 추론할 수 있다.

```
template<typename T1, typename T2>
auot multiply(T1 a, T2 b) -> decltype(a * b)
{
        return a * b;
}
```

**2. 복잡합 타입의 반환(iterator, 함수 객체)

- 반복자(iterator)나 함수 객체 등 복잡합 타입을 반환해야 할 때, 후행 반환 타입을 사용하면 반환 타입을 더 간단하고 명확하게 정의할 수 있다.

```
template<typename Container>
auto begin(Container& c) -> decltype(c.begin()){
        return c.begin();
}
```

3. 람다함수의 return type선언

- 후행 반환 타입은 람다 함수의 반환 타입을 명시해야 할 때도 유용하다

```
auto lambda = [](int a, int b) -> int{
        return a + b;
};
```

C++11 vs C++14

- C++11에서는 반환 타입을 `auto`로 지정하고, 후행 반환 타입에서 `정확한 타입`을 명시해야 했다. `decltype(a+b)` 그러나 C++14에서는 함수의 반환 타입을 `auto`로 지정해서, 컴파일러가 본문에서 반환하려는 포현식의 타입을 자동으로 추론한다. 따라서 후행 반환 타입에 굳이 `decltype`을 사용할 필요는 없다.

```
auto lambda = [](int a, int b) -> auto{
        return a + b;
};
```

본 내용 시작

```
const int i = 0; // decltype(i) is const int

bool f(const Widget& w) // decltype(w) is const Widget&
                                                // decltype(f) is bool(const Widget&)

struct Point{
        int x, y;
        //decltype(Point::x) is int
        //decltype(Point::y) is int
}

Widget w; // decltype(w) is Widget

if (f(w)) ... // decltype(f(w)) is bool

template<typename T> // simplified version of std::vector
class vector{
public:
        T& operator[](std::size_t index);

};

vector<int> v; // decltype(v) is vector<int>

if(v[0]==0) // decltype(v[0]) is int&
```

C++11에서는 `decltype`을 사용하는 주된 이유는 function의 return type이 "parameter type"에 따라 달라지는 경우에 많이 사용된다.
예를 들어서, `[]`을 사용해서 contaier indexing을 구현하고 싶은 경우를 예시로 들어보자
operator []는 return typedl T&이다. (대부분의 container `deque`, `vector`)그러나 `std::vector<bool>`은 []가 bool&을 return 하지 않는다. 대신 새로운 object를 return한다.

Code Example

```
template<typename Container, typename Index>
auto autoAndAccess(Container& c, Index i) -> decltype(c[i])
{
        authenticateUser();
```

```
        return c[i]
}
```

- 함수의 이름 앞에 있는 `auto`는 함수의 반환 타입이 `후행 반환 타입`과 관련이 있다는 뜻이다.
- 이 뜻은 함수의 parameter를 바탕으로 함수의 return type을 정하겠다는 의미를 가진다.

```
template<typename Container, typename Index>
auto autoAndAccess(Container& c, Index i)
{
        authenticateUser();
        return c[i]
}
```

- C++14는 trailing return type을 생략할 수 있다.
- `auto`의 뜻은 함수의 return type을 compiler가 함수 본문에서 `반환되는 값` `c[i]`를 보고 자동으로 추론한다.

그러나 문제점이 있다. 아래의 코드를 한 번 살펴보자

```
std::deque<int> d;
autoAndAccess(d,5) = 10; // authenticate user, return d[5], then assign 10 to it,
                                        // this won't compile
```

위의 `autoAndAccess`함수는 d[5]를 반환해야한다(`int&`)
그러나 일반적으로 `auto`를 반환 값으로 진행을 하게 된다면 reference 부분이 없어지기 때문에 반환 타입은 `int`가 된다. (반환 표현식의 타입을 값으로서 추론하기 때문)
그리고 `autoAndAccess`함수의 return값은 rvalue가 되기 때문에
`autoAndAccess(d,5) = 10` 이라는 선언문은 rvalue에 값을 할당하려는 시도라서 컴파일 에러가 발생한다.

위의 `autoAndAccess`가 `int&`를 반환하게 하기 위해서는 `decltype`을 함수의 return value에 적용은 해야한다. 이는 C++14부터 가능해졌다.

예시코드

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i)
{
        authenticateUser();
        return c[i];
}
```

위의 방식같이 code를 작성하게 된다면 c[i]는 T&를 반환하게 된다.

그렇다면 함수에서만 사용되는 건가요??

`decltype(auto)`는 함수의 return type에만 국한되지는 않는다.
변수를 초기화 하는 경우에도 많이 사용된다. 아래의 예시를 보면

```
Widget w;
const Wiedge& cw = w;
auto myWidget1 = cw;
decltype(auto) myWidget2 = cw;
```

`myWidget1`은 객체의 값을 기반으로 타입 추론을 진행하여 "Widget"타입을 가지게 된다.
반면에 `decltype(auto)`는 "표현식의 정확한 타입"을 추론하기 때문에 표현식의 "원래 타입(값 타입 또는 참조 타입)"인 `const Widget&`를 추론하게 된다.

`decltype`에 대해 더 알아보자

```
int x = 0;
```

`x`는 변수의 이름이다. 그래서 `decltype(x)`는 `int`이다.
그러나 `x`는 `(x)`로 변경하게 된다면 `reference`가 된다.
왜냐하면 C++에서 `(x)`는 lvalue로 정의했기 때문이다. 그래서 `decltype((x))`는 `int&`로 정의가 된다.

```
decltype(auto) f1()
{
        int x = 0;
        return x; // decltype(x) is int, so f1 return int
}
```

```
decltype(auto) f2()
{
        int x = 0;
        return (x); // decltype((x)) is int&, so f2 returns int&
```

```
}
```

- f1과 f2의 return type은 다르다. f2는 reference를 return하기 때문에 지역 변수의 reference를 가져올 수 있다.

Things to Remember

1. `decltype`은 거의 항상 변수나 표현식의 타입을 수정 없이 그래도 제공한다.
2. `이름이 아닌 lvalue 표현식`인 경우, `decltype`은 항상 타입을 `T&`로 보고한다.
3. C++14는 `decltype(auto)`를 지원한다. 이는 `auto`와 유사하게 초기화값에서 타입을 추론하지만 `decltype`규칙을 사용해서 타입을 추론한다.

PS) C++11과 C++14의 trailing return type의차이 요약

특징	C++11	C++14
후행 반환 타입	필요함: 반환 타입을 명시적으로 `decltype` 사용	불필요: 반환 타입을 생략하고 `auto`로 자동 추론
자동 반환 타입 추론	제한적: 단일 구문의 람다만 자동 추론 가능	모든 함수와 람다에서 자동 추론 가능
반환 타입 지정 위치	후행 반환 타입(`-> decltype(c[i])`) 필요	반환 타입을 생략 가능, `auto`로 자동 추론
템플릿 함수에서 반환 타입	명시적인 타입 추론 필요	반환 타입을 자동 추론

Know how to view deduced types