【C++】Day seven

Class	C++
Date	@November 25, 2021
Material	
Series Number	
Summary	Type Alias&auto

【Ch2】Deal with Types

2.5.1 Type Aliases

A type alias is a name that is a synonym for another type. Type aliases let us simplify complicated type definitions, making those types easier to use.

We use a typedef to define a type alias:

```
typedef double wages; //wages is a synonym for double
typedef wages base, *p; //base is a synonym for double, p for double*
```

We can also use the keyword using to define a type alias:

```
using SI = Sales_Item;
```

Pointers, Const, and Type Aliases

Declarations that use type aliases that represent compound types and const can yield surprising results. See the following example:

```
typedef char *pstring; //pstring is a pointer to char
const pstring cstr = 0; //cstr is a constant ponter to char
const pstring *ps; //ps is a pointer to a constant pointer to char
```

The base type in these ceclarations is `const pstring`. A `const` that appears in the base type modifies the given type. The type of `pstring` is "pointer to char." So, const pstring is a contant pointer to char—not a pointer to const char.

Note: It is incorrect to interpret a declaration that uses a type alias by conceptually replacing the alias with its corresponding type.

2.5.2 Tye auto Type specifier

It is common to want to store the value of an expression in a variable. But to declare the variable, we have to know the type of that expression.

Under the new standard, we can let the compiler figure out the type for us by using the `auto` type specifier. `auto` tells the compiler to deduce the type from the initializer. By implication, a variable that uses auto as its type specifier must have an initializer:

```
//The type of item is deduced from the type of the result of adding va1 and val2
auto item = val1 + val2;
```

As with other type specifier, we can define multiple variables using auto. Because a declaration can involve only a single base type, the initializers for all the varialbes in the declaration must have types that are concsistent with each other:

```
auto i = 0, *p = &i; //ok: i is intt and p is a pointer to int
auto sz = 0, pi = 3.14; //error: inconsistent types for sz and pi
```

`auto` ordinarily ignores top-level consts. Low-level consts, such as when an initializer is a pointer to const, are kept:

```
const int ci = i, &cr = ci;
auto b = ci; //b is an int(top-level const in ci is dropped)
auto c = cr; //c is an int(cr is an alias for ci whose const is top-level)
auto d = &i; //d is an int*
auto e = &ci; //e is a pointer to const int(& of a const object is low-level const)
```

If we want the deduced type to have a top-level const, we must say so explicityly:

```
const auto f = ci; //deduced type of ci is int, f has type const int
```

We can also specify that we want a reference to the auto-deduced type. Normal intialization rules still apply:

```
auto &g = ci; //g is a const int& that is bound to ci
auto &h = 42; //error: we can't bind a plain reference to a literal
const auto &j = 42; //ok: we can bind a const reference to a literal
```

The initializers must provide consistent auto-deduced types:

```
auto &n = i, *p = &ci; //error: type deduced from i is int; type deduced from &ci is const int
```