Type related

10 September 2021 05:56

1. Type alias (used for datatypes only)

using <newtypename> = <oldtypename> instead of typedef <oldtypename> <newtypename> .. Typedef and using help create aliases.. Scoping rules are same as variables..
Favor type aliases over typedefs.

2. Type Inference in C++

auto, decltype -> compiletime
typeid -> runtime

Use **auto&** to make type as reference
Const/volatile auto needs to be specified

3. decltype

Inspects type of a entity (or) **type + value category** of an entity or expression

```
decltype(entity)
decltype(expression)
```

a) if the value category of expression is xvalue, then decltype yields T&&;
b) if the value category of expression is lvalue, then decltype yields T&;
c) if the value category of expression is prvalue, then decltype yields T.

decltype is useful when declaring types that are difficult or impossible to declare using standard notation, like

1. lambda-related types
2. types that depend on template parameters

https://en.cppreference.com/w/cpp/language/decltype

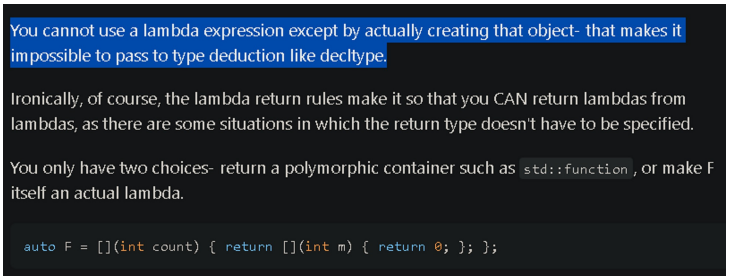
Note that if the name of an object is parenthesized, it is treated as an ordinary lvalue expression, thus decltype(x) and decltype((x)) are often different types.

```
struct A { double x; };
const A* a;

decltype(a->x) y;       // type of y is double (declared type)
decltype((a->x)) z = y; // type of z is const double& (lvalue expression)
```

Decltype need created object?

4. ??



Move initialization.. std::function<int(int)>(lambda expression) creates a r-value function object of lambda

```
// function::operator= example
#include <iostream>     // std::cout
#include <functional>   // std::function, std::negate

int main () {
  std::function<int(int)> foo,bar;
  foo = std::negate<int>();                          // target
  bar = foo;                                         // copy
  foo = std::function<int(int)>([](int x){return x+1;}); // move
  bar = nullptr;                                     // clear
```

5. std::declval<type>(..) - return rvalue reference to may be incomplete type

```
template <class T>
typename add_rvalue_reference<T>::type declval() noexcept;
```

This is a helper function used **on** members of a class in unevaluated operands, especially when either the constructor signature is unknown or when no objects of that type can be constructed (such as for abstract base classes).

```cpp
// declval example
#include <utility>      // std::declval
#include <iostream>     // std::cout

struct A {              // abstract class
  virtual int value() = 0;
};

class B : public A {    // class with specific constructor
  int val_;
public:
  B(int i,int j):val_(i*j){}
  int value() {return val_;}
};

int main() {
  decltype(std::declval<A>().value()) a;  // int a
  decltype(std::declval<B>().value()) b;  // int b
  decltype(B(0,0).value()) c;   // same as above (known constructor)
  a = b = B(10,2).value();
  std::cout << a << '\n';
  return 0;
}
```

6. auto keyword - compiler type-inference or type-deduction - done by compiler

   Avoid using type inference for function return types.. Use against complex datatypes or var definition which is explicit eg auto a {4.5}
   func(auto x) wont work until C++20 .. Use templates to make input args generic (doesn't apply to lambda)

7. Auto type deduction uses same rules as reference/template type deduction. Wont deduce constness!!

```
auto val = arg;      // arg can be l-value, r-value. References converted to l-values.
auto& val = arg;     // arg can be references/l-values/r-values (based on constness)
auto&& val = arg;    // arg can only be r-values. L-values means val becomes l-value-reference
```

8. typename

   Used to indicate that the identifier that follows is a type whenever referring to a nested name that is a *dependent name*, i.e. nested inside a template instance with unknown parameter

   https://stackoverflow.com/questions/7923369/when-is-the-typename-keyword-necessary

```cpp
template <class T>
typename add_rvalue_reference<T>::type declval() noexcept;
```

9. #include <typeinfo> / std::type_info
   - Stores information about a type in a class std::type_info
   - typeid() operator returns object of this class
   - Can be directly used to compare two types using relational operators !=, == in run-time
   - Has name method which returns type in run-time.. Null-terminated char sequence
   - If applied to a reference type (lvalue), the type_info returns identifies the referenced type only.
   - Any const or volatile qualified type is identified as its unqualified equivalent.
   - typeid is applied to a **reference** or **dereferenced pointer** to an object of a polymorphic class type), it considers its dynamic type (i.e., the type of the most derived object). This requires the RTTI (Run-time type information) to be available.

```cpp
AClass* aptr = &bobj;
BClass* bptr = &bobj;
AClass& aref = aobj;

std::cout << typeid(*aptr).name() << std::endl; // BClass
std::cout << typeid(aref).name() << std::endl;  // AClass
```

   From <https://www.cplusplus.com/reference/typeinfo/type_info/>

   typeid(<var>).name()
   typeid(<var1>) == typeid(<var2>)
   typeid(<var1>) != typeid(<var2>)

10. #include <type_traits> - COMPILE-TIME playaround with types (checks, transformations on type)

    Series of classes to obtain type information on compile-time.
    The header contains:
    - **Helper classes**: Standard classes to assist in creating compile-time constants.
      - std::integral_const<type, type value> - creates integral constant at compile time..

```cpp
template <class T, T v>
struct integral_constant {
  static constexpr T value = v;
  typedef T value_type;
  typedef integral_constant<T,v> type;
  constexpr operator T() { return v; }
};
```

    - **Type traits**: Classes to obtain characteristics of types in the form of compile-time constant values.
      - Check variable types (is array, is class, is enum etc)
      - Check type qualification properties, features, relationships (is const, is volatile, is polymorphic, is move constructable etc)

- is_rvalue_reference<T>::value checks if T's value category is r-value
- https://en.cppreference.com/w/cpp/types/is_same

- **Type transformations**: Classes to obtain new types by applying specific transformations to existing types.
  - Type transformations - add_rvalue_reference, remove_reference, enable_if, remove_const (Obtains the type `T` without top-level `const` qualification)
  - std::conditional<bool cond, Type if true, Type if false>::type
  - std::enable_if<bool cond, Type if true>::type …
    useful to hide signatures on compile time when a particular condition is not met, since in this case, the member `enable_if::type` will not be defined and attempting to compile using it should fail.

//The type `bool` is enabled as member type `enable_if::type` if `Cond` is true.

```
template <class T>
typename std::enable_if<std::is_integral<T>::value, int>::type is_odd (T i) {return bool(i%2);}
```

```
 std::remove_const<const char>::type a;          // char a
 std::remove_const<const char*>::type b;   // const char* b
 std::remove_const<char* const>::type c;   // char* c
```

https://www.cplusplus.com/reference/type_traits/