10-auto

June 2, 2024

1 Type inference

Because anything is strictly typed in C++, you have to specify the type of any variable when you declare it. This can be a bit cumbersome, especially when the type is long or complex. Yet, compilers have long been able to perform *type deduction* for function templates.

C++11 introduces a similar mechanism, called *type inference* based on keyword `auto`, which enables to infer the type of a variable from its initial value. The use of `auto` has progressively been enlarged to more and more situations.

1.1 Keyword `auto` as a variable type

1.1.1 Starting with C++11, the type of a variable can be deduced from its initial value

[ ]:
```
std::map<std::vector<int>,std::list<float>> m ;
auto itr = m.begin() ;
```

1.1.2 Possible `const` and `&` are dropped

[ ]:
```
int const i = 2 ; // int const
auto j = i ;      // int
int & k = j ;     // int &
auto l = k ;      // int
```

1.1.3 On the contrary, one can add `const`, `&` or `*`

[ ]:
```
int i = 2 ;
auto & j = i ;
auto const & k = j ;
auto const * l = &k ;
auto const * const m = &l ;
```

1.2 Keyword `decltype`

1.2.1 In order to reuse the type of an expression

[ ]:
```
std::map<std::vector<int>,std::list<float>> collection1 ;
decltype(collection1) collection2 ;
```

1.2.2 So to help type inference, when there is no initial value

[ ]:
```
std::map<std::vector<int>,std::list<float>> collection ;
decltype(collection)::iterator itr ;
```

1.2.3 So to avoid `const` and `&` dropping

[ ]:
```
int i = 10 ;           // int
int & j = i ;          // int &
decltype(auto) k = i ; // int
decltype(auto) l = j ; // int &
```

1.3 How to know which type has infered the compiler ?

- Your editor, if smart enough, may help.
- To be sure, you can also trigger an intentional compiler error, whose message contains the type name.

[ ]:
```
%%file tmp.inference.cpp

template <typename T> struct TypeDisplayer ;

int main()
 {
  int const i = 2 ;
  auto & j = i ;
  TypeDisplayer<decltype(j)> td ;
}
```

[ ]:
```
!rm -f tmp.inference.exe && g++ -std=c++17 tmp.inference.cpp -o tmp.inference.
 ↪exe
```

1.4 Keyword `auto` in a range-based for

1.4.1 What is a range-based for ?

For any collection which is supported by `std::begin()` and `std::end()`, one can now use the **ranged-based for** notation.

[2]:
```
#include <iostream>

int const MAX = 5 ;
```

```
double values[MAX] = { 1.1, 2.2, 3.3, 4.4, 5.5 } ;

for ( double value : values  )
 { std::cout << value << " " ; }
```

```
1.1 2.2 3.3 4.4 5.5
```

1.4.2 Combined with auto

[3]:
```
#include <vector>
#include <iostream>

int i = 0 ;
double values[MAX] = { 1.1, 2.2, 3.3, 4.4, 5.5 } ;

for ( auto value : values  )
 { std::cout << value << " " ; }
```

```
1.1 2.2 3.3 4.4 5.5
```

1.4.3 Modifications requires &

[4]:
```
#include <iostream>

int i = 0 ;
std::vector<int> values(5,0) ;

for ( auto & value : values  )
 { value = ++i ; }

for ( auto value : values  )
 { std::cout << value << " " ; }
```

```
1 2 3 4 5
```

1.5 Keyword auto as function return type

1.5.1 When used alone

When auto is used instead of the usual return type, the compiler will infer this type from the return statements in the function body.

[5]:
```
auto nb( int i )
 {
  if (i<10) return 1 ;
  if (i<100) return 2 ;
  return 99 ;
 }
```

[6]:
```
nb(15)
```

[6]:
```
2
```

1.5.2 When used with a trailing return type

The keyword `auto` can also be used in a *trailing return type* declaration, as shown below. There is no inference there. That is just a syntax to declare the type after the function parameters.

[7]:
```
%%file tmp.trailing.cpp

#include <iostream>

auto add( int p1, int p2 ) -> int
 { return (p1+p2) ; }

int main()
 {
  std::cout << add(39,3.14) << std::endl ;
 }
```

```
Writing tmp.trailing.cpp
```

[8]:
```
!rm -f tmp.trailing.exe && g++ -std=c++11 tmp.trailing.cpp -o tmp.trailing.exe␣
 ↪&& ./tmp.trailing.exe
```

```
42
```

This can help when the return type depends on the parameter types:

[9]:
```
%%file tmp.trailing.cpp

#include <iostream>

template< typename T1, typename T2 >
auto add( T1 value, T2 offset ) -> decltype(value)
 { return (value+offset) ; }

int main()
 {
  std::cout << add(39,3.14) << std::endl ;
 }
```

```
Overwriting tmp.trailing.cpp
```

[10]:
```
!rm -f tmp.trailing.exe && g++ -std=c++11 tmp.trailing.cpp -o tmp.trailing.exe␣
 ↪&& ./tmp.trailing.exe
```

```
42
```

1.6 Structured Bindings (C++17)

The functional programming style should lead you to write "pure" functions, with constant input data, and all the ouput returned as a compound data. The new C++17 structured bindings enable you to dispatch such a compound return value into several variables, almost as simply as in Python...

[11]:
```
%%file tmp.bindings.cpp

#include <iostream>
#include <utility>

int main() {
  auto [ first, second ] = std::make_pair(42, 3.14) ;
  std::cout << "first: " << first << std::endl ;
  std::cout << "second: " << second << std::endl ;
}
```

```
Overwriting tmp.bindings.cpp
```

[12]:
```
!rm -f tmp.bindings.exe && g++ -std=c++17 tmp.bindings.cpp -o tmp.bindings.exe␣
 ↪&& ./tmp.bindings.exe
```

```
first: 42
second: 3.14
```

Actually, on the right side of =, one can place many kinds of arrays, structs, tuples, and user-defined classes with tuple-like properties.

[17]:
```
%%file tmp.bindings.cpp

#include <iostream>

int main() {
  int values[] = { 1, 2 } ;
  auto [ v1, v2 ] = values ;
  std::cout << v1 << ", " << v2 << std::endl ;
}
```

```
Overwriting tmp.bindings.cpp
```

[18]:
```
!rm -f tmp.bindings.exe && g++ -std=c++17 tmp.bindings.cpp -o tmp.bindings.exe␣
 ↪&& ./tmp.bindings.exe
```

```
1, 2
```

[19]:
```
%%file tmp.bindings.cpp

#include <iostream>

struct Vector { double x, y, z ; } ;
```

```
int main() {
  Vector origin = { 0., 0., 0. } ;
  auto [ ox, oy, oz ] = origin ;
  std::cout << ox << ", " << oy << ", " << oz << std::endl ;
}
```

```
Overwriting tmp.bindings.cpp
```

[20]:
```
!rm -f tmp.bindings.exe && g++ -std=c++17 tmp.bindings.cpp -o tmp.bindings.exe␣
 ↪&& ./tmp.bindings.exe
```

```
0, 0, 0
```

[21]:
```
%%file tmp.bindings.cpp

#include <iostream>
#include <tuple>

int main() {
  std::tuple t(42, 3.14, "hello", "world") ;
  auto [ t1, t2, t3, t4 ] = t ;
  std::cout << t1 << ", " << t2 << ", " << t3 << ", " << t4 << std::endl ;
}
```

```
Overwriting tmp.bindings.cpp
```

[22]:
```
!rm -f tmp.bindings.exe && g++ -std=c++17 tmp.bindings.cpp -o tmp.bindings.exe␣
 ↪&& ./tmp.bindings.exe
```

```
42, 3.14, hello, world
```

Structured bindings also prove useful within `range-based for`:

[23]:
```
%%file tmp.bindings.cpp

#include <iostream>
#include <map>
#include <string>

int main() {

  std::map<std::string, int> grades ;

  grades["Francoise"] = 12 ;
  grades["Antoine"] = 18 ;
  grades["David"] = 3 ;

  for ( auto [ key, value ] : grades )
   { std::cout << key << " " << value << std::endl ; }
```

```
}
```

```
Overwriting tmp.bindings.cpp
```

[24]:
```
!rm -f tmp.bindings.exe && g++ -std=c++17 tmp.bindings.cpp -o tmp.bindings.exe
```

[25]:
```
!./tmp.bindings.exe
```

```
Antoine 18
David 3
Francoise 12
```

1.7 Keyword `auto` as function parameter type (C++20)

When used as parameter type, `auto` is a shortcut for the template syntax. Yet, each parameter must be declared with `auto` individually, and those parameters are independants one from each other.

```
bool compare( auto param1, auto param2 )
 { return param1==param2 ; }
```

means:

```
template <typename T1, typename T2>
bool compare( T1 param1, T2 param2 )
 { return param1==param2 ; }
```

not:

```
template <typename T>
bool compare( T param1, T param2 )
 { return param1==param2 ; }
```

2 Quizz : what is the return type ?

```
auto join( std::vector<std::string> const & values, std::string separator )
 {
  if (std::empty(values))
   { return "" ; }
  auto result = values[0] ;
  for ( std::size_t i = 1 ; i < std::size(values) ; ++i )
   { result += separator ; result += values[i] ; }
  return result ;
 }
```

3 Take away

- The keyword `auto` basically avoid typing redundant types.
- The keyword `decltype` helps to keep the exact original type.
- Since C++20, `auto` is also a simplified syntax for simple templates.

- Type inference and template instanciation “mostly” follow the same rules…
- **Overuse of `auto` obfuscates the code!**

4 Questions ?

5 Exercise

1. In the code below, simplify the friend operator `<<`.
2. Provide the class with methods `begin()` and `end()`.
3. Move the operator `<<` outside the class.

[ ]:
```
%%file tmp.inference.cpp

#include <iostream>
#include <string>
#include <vector>

class Sentence
 {
  public :
    void add( char const * word )
     { m_words.push_back(static_cast<std::string>(word)) ; }
    friend std::ostream & operator<<( std::ostream & os, Sentence const & s )
     {
      typedef typename std::vector<std::string>::const_iterator Iterator ;
      for ( Iterator word = s.m_words.begin() ; word != s.m_words.end() ;␣
 ↪++word )
       { os<<(*word)<<" " ; }
      return os ;
     }
  private :
    std::vector<std::string> m_words ;
 } ;

int main()
 {
  Sentence s ;
  s.add("Hello") ;
  s.add("world") ;
  s.add("!") ;
  std::cout<<s<<std::endl ;
 }
```

[ ]:
```
!rm -f tmp.inference.exe && g++ -std=c++17 tmp.inference.cpp -o tmp.inference.
 ↪exe
```

[ ]:
```
!./tmp.inference.exe
```