Auto - a necessary evil?

Roger Orr
OR/2 Limited
ACCU 2013
**auto** is new in C++11

- It has been under discussion for a while, as we shall see
- Some compilers added support for it early in C++0x so it has had 'field testing'
auto is re-purposed in C++11

• auto was a C++98 keyword
  – Local objects explicitly declared auto or register or not explicitly declared static or extern have automatic storage duration.
  – The storage for these objects lasts until the block in which they are created exits.

```cpp
auto int i; // explicit
int i;     // implicit
```
History

- What was the initial use case?
- David Abrahams 26 Oct 2001 (ext-4278)
  “the expression results in a very complicated nested template type which is difficult for a user to write down”. He suggested:

```cpp
typeof(<expression>) x = <expression>;
```

*(typeof became decltype in C++11)*

- could be replace with something like:

```cpp
template <class T> T x = <expression>;
```
In the subsequent discussion Andy Koenig wrote:

“I would also like to see something like
auto x = <expression>;
I know we can't use auto, but you get the idea.”
In the subsequent discussion Andy Koenig wrote:

“I would also like to see something like
auto x = <expression>;

I know we can't use auto, but you get the idea.”

But we *did* eventually use auto!

- “Google Code Search finds less than 50 uses of auto in C++ code.”
History

- First formal paper was N1478 (Apr 2003)
- Emphasis on generic programming – the draft proposal (ext-5364) begins:
  - “Proposal for "auto" and "typeof" to simplify the writing of templates”
- Contained another new keyword, `fun`, which was replaced by overloading `auto`
  - (is `auto` the new `static`?)
- and `typeof` turned into `decltype`
History

- What did we end up with?
- **auto** is repurposed and can be used as:
  - auto x = 5;
  - auto lambda1 = [](int i){ return i; };
  - new auto(1);
  - auto f()->int(*)[4];
  - template <class T, class U>
    auto add(T t, U u) -> decltype(t + u);
History

- **auto** is a compile time construct – the type is baked in by the compiler
  - This is worth highlighting, especially for those used to languages with dynamic types
- Reluctance to add special cases for **auto**
  - The general principle was to try and make use of **auto** orthogonal to other choices: so for example **auto** for function return types is not restricted to templates
History

- Once formally adopted into the working paper `auto` became available for use -
  - www.aristeia.com/C++11/C++11FeatureAvailability.htm
    - Gcc 4.4 (formal release Apr '09)
    - MSVC 10 (formal release Apr '10)

- (I've confirmed the earlier examples with gcc 4.5 & 4.7 and MSVC 10 & 11)
Interactions with other items

- R-value references
- Lambda
- NSDMI (non-static data member initialisers)
- Range-based for
- And also related to decltype
Interactions with other items

- R-value references
  
  ```
  auto var1 = <expression>;
  auto & var2 = <expression>;
  auto && var3 = <expression>;
  ```

  These are all valid (subject to constraints on the actual expression)

- The last example may not do *quite* what you expect … more on this later
Interactions with other items

- Lambda

- This was one of the motivating cases for `auto` - passing to a template is OK:

  ```cpp
template <typename T> void invoke(T t);
  invoke([](int i){ return i; });
```

  But what if you want a variable?

  ```cpp
  <type> lambda1 = [](int i){ return i; };
  int j = lambda1(7);
  ```

- What should replace `<type>`?
Interactions with other items

- **NSDMI (non-static data member initialisers)**

```cpp
class x {
    int i = 128;
    double d = 2.71828;
};
```

- Could you use `auto` instead?

```cpp
auto i = 128;
auto d = 2.71828;
```

- Short answer: no
Interactions with other items

- Range-based `for` – can use this:
  
  ```cpp
  for (std::string x : container) {
      ...
  }
  
  or this:
  ```
  ```cpp
  for (auto x : container) {
      ...
  }
  ```
Interactions with other items

- Range-based `for` can also be
  
  ```
  for (auto & x : container) {
      ...
  }
  ```

- Or
  
  ```
  for (auto const & x : container) {
      ...
  }
  ```
  
  - Note `x` is already `const` if the container is `const`
Interactions with other items

- You may or may not care that range-based for is actually specified in terms of `auto`:

```cpp
{
  auto && __range = range-init;
  for ( auto __begin = begin-expr, 
        __end = end-expr;
       __begin != __end; 
      ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```
Interactions with other items

- The keyword `decltype` obtains the type of an expression:
  - This is useful when you require the type in places where `auto` does not work – for example declaring a variable `without` an initial value:
    ```cpp
    std::vector<int> vec;
    decltype(vec.cbegin()) iter;
    ```

- There are subtle differences between the two, which I will touch on later
Where **must** you use it

- The basic principle behind `auto` is that the compiler knows the type ... but you either *can't describe it* or *don't want to*

- Lambdas
  - “The type of the *lambda-expression* is a **unique, unnamed** nonunion class type — called the closure type”

- In this case you *can't use* `decltype(expression)` as the types of identical lambdas **differ**
Where **must** you use it

- **Side note:**
- A small number of types in the standard are specified as **unspecified** so you cannot name them portably.
- **auto** gives you a way to create variables of those types
- This is almost never a genuine problem
Lambda example

- Lambdas are most often used as arguments to other functions. However, if you want one as a local variable:

```cpp
int main()
{
    auto sum = [] (int x, int y)
    {
        return x + y;
    };

    int i(1);
    int j(2);
    // ...
    std::cout << i << " + " << j << " = " << sum(i, j) << std::endl;
}
```
Lambda example

- What is the type of the variable holding the lambda?
- We may get some information by using typeid: `typeid(sum).name()`
  - MSVC: `class <lambda_8f4bf0680d354484748e55d11883b00a>`
  - gcc: `Z4mainEULiiIE_`

(demangles to `main::{lambda(int, int)#1}`)
There is no choice here, we have to use the compiler to name the type of the lambda expression

However most people recommend you use `auto` in (at least some of) the cases where giving the name of type yourself is a valid option.
Where *may* you use it

- What are the benefits and dangers of using `auto` to replace a named type?
- On the plus side:
  - Simplifies or removes complex declarations
  - Complies with the DRY principle
  - Code is (or may be...) easier to read
  - and easier to change
  - and smaller (apart from `int`)
Where may you use it

- What are the benefits and dangers of using `auto` to replace a named type?
- On the plus side:
  - Simplifies or removes complex declarations
  - Complies with the DRY principle
  - Code is (or may be...) easier to read
  - and easier to change
  - and smaller (apart from `int`)
  - and, of course, so much cooler looking
Where may you use it

- So why not use it everywhere?
- On the minus side:
  - It may not express **intent** as clearly
  - Higher cognitive overhead
  - Conflicts with “program to interfaces”
  - Subtleties, especially over
    - **const**
    - & and &&
- We'll look at some examples in a minute
Where can't you use it

- You cannot use `auto`:
  - As the type of lambda arguments
  - To declare function arguments
  - To declare function return types **without** a trailing-return-type declaration
  - To declare member data

- (at least, not at present ... I'll mention some future directions at the end of the talk)
Complex type example

- As I covered earlier declarations of complex type were one of the motivations:
  ```cpp
  std::vector<std::set<int>> setcoll;
  auto it = setcoll.cbegin();
  ```

- This is shorter than the full type:
  ```cpp
  std::vector<std::set<int>>::const_iterator
  ```

- But is it **better**?

  (* cbegin is another C++11 addition*)
Complex type example

- Many programmers were put off using the STL because of the verbosity of the variable declarations.
- With C++03 one recommendation was to use a typedef:
  ```cpp
typedef std::vector<std::set<int>> collType;
collType::const_iterator it ...
```
- This is still valid in C++11, but having to pick a type name adds to the cognitive overhead
Complex type example

- Use of `auto` removes the scaffolding of the type declaration but still leaves the type safety as the variable is still strongly typed
- So:
  - the code is quicker and easier to write
  - the purpose is not lost in the syntax
  - code generated is identical to explicit type
  - the variable automatically changes type if the collection type changes
Complex type example

- However the last point can be reworded
  - the variable automatically silently changes type if the collection type changes
- In order to know the actual type of the auto you need to keep in mind the type of the collection (and its const-ness)
- However, you probably need to keep this in mind anyway to correctly process the data
Complex type example

- Also note that the code uses `cbegin()`:
  ```
  auto it = setcoll.cbegin();
  ```
- If we'd used `begin()` we might have got a modifiable iterator. The C++03 code makes it explicit by using the actual type name:
  ```
  std::vector<std::set<int>>::const_iterator it
  ```
- The stress is slightly different and may mean changing to your interface, as with the addition of `cbegin()`
auto allows you to specify the type name once

```cpp
std::shared_ptr<std::string> str =
    std::make_shared<std::string>("Test");
```

- (1) We've repeated the `std::string`
- (2) `make_shared` exists solely to create `std::shared_ptr` objects

We can write it more simply as:

```cpp
auto str = std::make_shared<std::string>("Test");
```
DYR example

- Using `auto` rather than repeating the type is indicated most strongly when:
  - the type names are long or complex
  - the types are identical or closely related

- `auto` is less useful when:
  - the type name is simple - or important
  - The cognitive overhead on the reader of the code is higher
DRY example

- So I think `auto` is less useful here:

  ```
  // in some header
  struct X {
    int *mem_var;
    void aMethod();
  };

  // in a cpp file
  void X::aMethod() {
    auto val = *mem_var; // what type is `val`?
    ...
  }

- YMMV – opinions differ here (also on whether you are using an IDE with type info)
DRY example

/*
 // in some header
 struct X {
   int *mem_var;
   void aMethod();
 };

 // in a cpp file
 void X::aMethod() {
   auto val = *mem_var; // what type is val?
 }
 int val

Dependent return type example

- `auto` can simplify member function definitions

```cpp
class Example
{
public:
    typedef int Result;

    Result getResult();
};

Example::Result Example::getResult()
{ return ...; }
```
Dependent return type example

- `auto` allows removal of the class name from the return type
  
  ```cpp
  auto Example::getResult() -> Result
  { return ...; }
  ```

- Whether or not this makes the code **clearer** depends on factors including:
  - familiarity
  - consistent use of this style

- I personally still can't decide on this one
Polymorphism?

• One problem with `auto` is the temptation to code to the implementation rather than the interface:

```cpp
auto shape = make_shared<ellipse>(2, 5);
shape->minor_axis(3);
```

• When the type of shape is the abstract base class you can't make this mistake

• (Aside: I think this is a bigger problem with `var` in C# than with `auto` in C++)
Polymorphism?

- **auto** is too "plastic" – it fits the *exact* type that matches
- Without **auto** the author needs to make a decision about the most appropriate type to use
- This doesn't only affect polymorphism: **const**, signed/unsigned integer types and sizes are other possible pinch points
What type is it?

- What does this do:

```cpp
auto main() -> int {
    auto i = '1';
    auto j = i * 'd';
    auto k = j * 100l;
    auto l = k * 100.;
    return l;
}
```

- Easy to assume the auto types are all the same – miss the promotion, the 'l' or the '.'
What type is it?

- You can use the `auto` rules (on some compilers) to tell you the type:

```cpp
auto main() -> int {
    auto i = '1';
    auto j = i * 'd', x = "x";
    ...

error: inconsistent deduction for 'auto': 'int' and then 'const char*'
```
What type is it?

- You may also be able to get the compiler to tell you the type by using template argument deduction, for example:

```cpp
template <typename T>
void test() { T::dummy(); }

auto val = '1';
test<decltype(val)>();
```

=> “see reference to function template instantiation 'void test<char>(void)' being compiled”
What type is it?

- The meaning of an `auto` variable declaration follows the rules for template argument deduction
  
  ```
  auto val = '1';
  ```

- Consider the invented function template
  
  ```
  template <typename T>
  void f(T t) {}
  ```

- The type of `val` is that deduced in `f('1')`
What type is it?

- `auto` differs from a naïve use of `decltype`:
  ```cpp
  const int ci;
  auto val1 = ci;
  decltype(ci) val2 = ci;
  ```

- `val1` is `int`
- `val2` is `const int`
- (Think about top level `const`)
What sort of reference?

- What's the difference?

  ```
  auto i = <expr>
  auto const ci = <expr>
  auto & ri = <expr>
  auto const & cri = <expr>
  auto && rri = <expr>
  ```

- As above, `auto` uses the same rules as template argument deduction
What sort of reference?

- **Compare:**

  ```cpp
  template <typename T>
  void f(T          i);
  void f(T const    ci);
  void f(T       &  ri);
  void f(T const &  cri);
  void f(T       && rri);
  ```

- **It depends ... especially for the && case**
  (Scott Meyers “Universal Reference”)
What sort of reference?

- `const inference (values)`

  ```cpp
  int i(0); int const ci(0);

  auto v0 = 0;
  auto const v1 = 0;
  auto v2 = i;
  auto const v3 = i;
  auto v4 = ci;
  auto const v5 = ci;
  ```
What sort of reference?

- **const inference (values)**

```cpp
int i(0); int const ci(0);

auto v0 = 0;  // int
auto const v1 = 0;  // int const
auto v2 = i;  // int
auto const v3 = i;  // int const
auto v4 = ci;  // int (as earlier)
auto const v5 = ci;  // int const
```
What sort of reference?

- `const inference (references)`

```cpp
test

int i(0); intconst ci(0);

auto & v0 = 0;
auto const & v1 = 0;
auto & v2 = i;
auto const & v3 = i;
auto & v4 = ci;
auto const & v5 = ci
```
What sort of reference?

- `const` inference (references)

```cpp
int i(0); int const ci(0);

auto & v0 = 0; // error
auto const & v1 = 0; // int const &
auto & v2 = i; // int &
auto const & v3 = i; // int const &
auto & v4 = ci; // int `const` &
auto const & v5 = ci; // int const &
```
What sort of reference?

- Reference collapsing

```cpp
int i(0); int const ci(0);

auto      && v0 = 0;
auto const && v1 = 0;
auto      && v2 = i;
auto const && v3 = i;
auto      && v4 = ci;
auto const && v5 = ci;
```
What sort of reference?

- Reference collapsing

    int i(0); int const ci(0);
    
    auto       && v0 = 0;  // int       &&
    auto const && v1 = 0;  // int const &&
    auto       && v2 = i;  // int       &
    auto const && v3 = i;  // error
    auto       && v4 = ci;  // int const &
    auto const && v5 = ci;  // error

- Note `const` disables reference collapsing
What sort of reference?

- This is the complicated one:
  
  ```
  auto && var = <expr>;
  ```

- Depending on `<expr>` `var` could be
  - `T &`
  - `T &&`
  - `T const &`
  - Is that all?
What sort of reference?

- This is the complicated one:
  ```
  auto && var = <expr>;
  ```

- Depending on `<expr>` `var` could be
  - `T &`
  - `T &&`
  - `T const &`
  - `T const &&` (I didn't show that one)
What sort of reference?

- But it is a bit obscure ... (*compiler* is wrong here)

```cpp
const int x() { return 0; }

int main()
{
    int i(0); int const ci(0);
    auto && v0 = 0;
    auto const && v1 = 0;
    auto && v2 = i;
    auto const && v3 = i;
    auto && v4 = ci;
    auto const && v5 = ci;
    auto && v6 = x();
    v6 = 10;   int & & v6
}
```

The output from the build process indicates a compilation error. The error message is:

```
C:\projects\cpp\temp\virtualqueryimpl.cpp(164): error C3892: 'v6' : you cannot assign to a variable that is const
VirtualQuery.cpp
Summary.cpp
Generating Code...
```
What sort of reference?

• Here's an example of deducing const &&

    class T{};
    const T x() { return T(); } 

    auto && var = x();

    - var is of type T const &&
    - (non-class types, like int, decay to &&)
More dubious cases

- **auto** does not work well with initializer lists

```c
int main() {
    int var1{1};
    auto var2{1};
}
```

- You might expect `var1` and `var2` to have the same type.

- Sadly the C++ rule have introduced a new 'vexing parse' into the language
More dubious cases

- **auto** does not work well with initializer lists

  ```cpp
  int main() {
      int var1{1};
      auto var2{1};
      auto p1 = &var1, p2 = &var2;
  }
  ```

- **Produces**

  ```cpp
  error: inconsistent deduction for 'auto': 'int*' and then 'std::initializer_list<int>**'
  ```
More dubious cases

- Mix of signed/unsigned integers – or different sizes – can cause problems with `auto`
- In many cases the compiler generates a warning, if you set the appropriate flag(s)
- But not all ....

```c++
for (int i = v.size() - 1; i > 0; i -= 2)
{
    process(v[i], v[i-1]);
}
```
- Change `int` to `auto` and the code breaks
Future directions

- Polymorphic lambda (N3559)
  
  ```cpp
  auto Identity = [](auto a) { return a; };
  ```

- Generates a family of lambdas – much like a template does.
- `Identity(17)` **instantiates the lambda for int**
- `Identity(3.14159)` **for double**

- Agreed in principle
Future directions

- **auto** in function arguments
  
  ```cpp
  void func(auto a);
  ```

- Generates an *implicit* function template

- Equivalent to something like this
  
  ```cpp
  template<typename __T1>
  void func(__T1 a);
  ```

- This may or may not get standardised
Future directions

- Auto function return type (N3582)

- Currently `auto` for function return type requires a return type declaration, this proposal allows for:

```cpp
auto g() { return 'X'; } // implicit
struct A { auto f(); }; // fwd declare
...
auto A::f() { return 42; }
```
Future directions

- The implicit deduction is agreed in principle
- The paper includes an extension to allow reference return types:
  ```
  auto const & log() { return theLogger; }
  ```
Future directions

- **auto** for member variables
- The difficulty is that the type of **auto** is only known when the variable is initialised. This occurs *after* the class has been parsed.
- **decltype** can be used instead (as this can be processed during the initial parse):

```cpp
class X {
    decltype(foo()) aFoo;
};
```
Conclusion

- **auto** is a new tool in the C++ programmer's arsenal.
- Use of it can make code easier to write, to understand and to maintain
- However, over-use or careless use can result in code that is hard to follow or contains subtle bugs
- Know your tools!