

AAAARGH!?

...

Adopting Almost-Always-Auto
Reinforces Good Habits!?

Save your pitchforks & torches!

78%

of angry mobs suffer injury while attempting to accomplish their goal.

Agenda

- Idiom
- Case Study
- Variables
- Functions
- Lambdas
- Branching
- Looping
- Takeaway

The Almost-Always-Auto Idiom

Prefer deduced rather than explicit types.

Leverages `auto`, `decltype`, and `decltype(auto)` keywords.

Promoted by Herb Sutter & Scott Meyers.

“write code against interfaces, not implementations” - Herb Sutter.

Controversial effect on code readability vs flexibility.

Automagic: A Spelling Game

Start at max life.

Cast a random spell each turn.

Each spell changes life differently.

Game ends when you have no life.

```
1.  int main () {  
2.      Game game{};  
3.      int turnCount{};  
4.      while (game.Turn())  
5.          ++turnCount;  
6.      return turnCount;  
7.  }
```

Automagic version 0.0 - No AAA

```
1.  template <typename T> using UD =
2.      std::uniform_int_distribution<T>;
3.  struct Game {
4.      using Life = unsigned int;
5.      static const Life c_lifeMax{
6.          std::numeric_limits<Life>::max()
7.      };
8.      std::mt19937 m_engine{};
9.      Life m_life{c_lifeMax};
10.     Life& Heal (Life& life);
11.     Life& Hurt (Life& life);
12.     bool Turn ();
13. };
```

Automagic version 0.0 - No AAA

```
1. Life& Game::Heal (Life& life) {  
2.     UD<Life> dis{0, c_lifeMax - life};  
3.     return life += dis(m_engine);  
4. }  
5.  
6. Life& Game::Hurt (Life& life) {  
7.     UD<Life> dis{0, life};  
8.     return life -= dis(m_engine);  
9. }
```

Automagic version 0.0 - No AAA

```
1.  bool Game::Turn () {
2.      using Spell = Life& (Game::*)(Life&);
3.      static const Spell c_spells[] = {
4.          &Game::Heal, &Game::Hurt,
5.      };
6.
7.      UD<size_t> dis{0, size_array(c_spells) - 1};
8.      Spell spell = c_spells[dis(m_engine)];
9.
10.     return (this->*spell)(m_life) > 0;
11. }
```


Variables

Version 0

- **auto**&& for general purpose locals.
 - Works with l- and r-values.
 - It extends rvalue lifetime.
 - Works with non-copyable and non-movable types.
- **const auto**& for general purpose immutable locals.
- **auto** when copy or move construction is desired.

auto*, **auto**&, and **decltype(auto)** are avoided.

AAA for Member Variables

Static members can use `auto`, `decltype`, and `decltype(auto)`.

- `const` members are limited to integral in-class initializers.
- `constexpr` members require literal type in-class initializers.

But non-static member variables can only use `decltype`!

```
1. struct Game {  
2.     static const auto c_lifeMax{  
3.         std::numeric_limits<Life>::max()  
4.     };  
5.     std::mt19937 m_engine{};  
6.     decltype(c_lifeMax) m_life{c_lifeMax};  
7. };
```

AAA for Non-Member Variables

Local variables can use `auto`, `decltype`, and `decltype(auto)`.

`auto&&` guaranteed to work for any non-void statement on right.

```
1. Life& Game::Hurt (Life& life) {  
2.     UD<Life> dis{0, life};  
3.     return life -= dis(m_engine);  
4. }  
5.  
6. Life& Game::Hurt (Life& life) {  
7.     auto&& dis = UD<Life>{0, life};  
8.     return life -= dis(m_engine);  
9. }
```

AAA for Arrays

auto deduces untyped braces as a std::initializer_list

```
1. auto&& c_spells = {&Game::Heal, ...}; // meh
```

arrays of auto are not valid.

```
2. auto&& c_spells[] = {&Game::Heal, ...}; // no
```

```
3. auto&& c_spells[1] = {&Game::Heal, ...}; // no
```

auto deduces typed braces as a constructor.

```
4. using Spells = Spell[]; // meh
```

```
5. auto&& c_spells = Spells{&Game::Heal, ...};
```

AAA for Arrays

C++17 Library TS `std::experimental::make_array` can help.

```
1. using std::experimental::make_array;
2. auto&& c_spells = make_array(
3.     &Game::Heal, &Game::Hurt
4. ); // std::array<Life&(Game::*)(Life&), 2>
5.
6. auto&& arrDb1 = make_array<double>(
7.     0, 1, 2
8. ); // std::array<double, 3>
```

Still useful even with C++17 constructor deduction to avoid explicit size.

Functions

Version 0

- **auto** for functions returning locals or rvalues.
 - Type deduced as common type of all return statements.
- **auto&** for class member ‘getters’.
 - const member functions will return reference to const.
- **decltype(auto)** for forwarding functions.
- **auto (...)** -> ... { ... } to SFINAE.

AAA for Functions

Create auxiliary template functions to perform additional deduction.

```
1.  template <
2.      typename A,
3.      typename B,
4.      typename C = std::common_type_t<A, B>,
5.      typename D = std::uniform_int_distribution<C>
6.  >
7.  auto make_ud (const A& lo, const B& hi) ->
8.      std::enable_if_t<std::is_integral<C>{}, D>
9.  {
10.     return D{C{lo}, C{hi}};
11. }
```

AAA for Functions

```
1.  auto& Heal (Life& life) {
2.      auto&& dis = make_ud(0, c_lifeMax - life);
3.      return life += dis(m_engine);
4.  }
5.  auto Turn () {
6.      ...
7.      auto&& dis = make_ud(
8.          0, size_array(c_spells) - 1
9.      );
10.     Spell spell = c_spells[dis(m_engine)];
11.     return (this->*spell)(m_life) > 0;
12. }
```


AAA for Functions

We can also wrap selection of a random element in a range.

```
1.  template <typename InputIt, typename G>
2.  auto random_element (
3.      InputIt first,
4.      InputIt last,
5.      G&& g
6.  ) {
7.      auto&& dis = make_ud(
8.          0, std::distance(first, last) - 1
9.      );
10.     return std::next(first, dis(g));
11. }
```

AAA for Functions

This hides the distribution details in `random_element`.

```
1.  auto Turn () {  
2.      ...  
3.      auto&& s = random_element(  
4.          std::begin(c_spells),  
5.          std::end(c_spells),  
6.          m_engine  
7.      );  
8.      return (this->*s)(m_life) > 0;  
9.  }
```

Lambdas

Version 0

- Use `auto&&` to name lambdas with local variables.
- Use `auto&&` for input parameters.
- Use `auto&` for output parameters.
- Use `[] (...) -> auto& { }` for lambdas meant to be chained.
- Use `[] (...) -> decltype(auto) { }` for forwarding lambdas.

AAA for Lambdas

We can implement spells via lambdas to leverage additional deduction.

```
1. auto Turn () {  
2.     auto&& heal = [] (  
3.         auto& life, auto&& engine  
4.     ) -> auto& {  
5.         auto&& dis = make_ud(0, c_lifeMax - life);  
6.         return life += dis(engine);  
7.     };  
8.     ...  
9. }
```

AAA for Lambdas

We can also unify the spell implementation and array definitions.

```
1. using Spell = decltype(m_life)& (*)(  
2.     decltype(m_life)&,   
3.     decltype(m_engine)&  
4. );  
5. static const auto c_spells = make_array<Spell>(  
6.     [] (auto& life, auto& engine) -> auto& {  
7.         auto&& dis = make_ud(0, c_lifeMax - life);  
8.         return life += dis(engine);  
9.     }  
10. );  
11. auto&& spell = random_element(...);
```

Automagic version 0.1 - Observations

Added a few reusable deducing functions.

Lots of difficulty converting arrays to AAA style.

Adding a spell would just require adding a lambda.

Spells are no longer visible outside of Turn.

Individual spells are harder to spot at a glance.

Automagic version 0.1 - Statistics

Explicit types **decreased** from 18 to 4.

Deduced types **increased** from 0 to 13.

Words **increased** from 74 to 86.

Characters **increased** from 720 to 856.

Elapsed time for 11,730,100 turns **decreased** from 530ms to 505ms.

Branching

Version 1

Add a Maim spell:

- If life is between (80, 100]%, deal 25% damage.
- If life is between (60, 80]%, deal 20% damage.
- If life is between (40, 60]%, deal 15% damage.
- If life is between (20, 40]%, deal 10% damage.
- Else, do nothing.

Automagic version 1.0 - No AAA

```
1. Life& Game::Maim (Life& life) {  
2.     Life change;  
3.     if (life > c_lifeMax / 100 * 80)  
4.         change = c_lifeMax / 100 * 25;  
5.     else if (life > c_lifeMax / 100 * 60)  
6.         change = c_lifeMax / 100 * 20;  
7.     else if (life > c_lifeMax / 100 * 40)  
8.         change = c_lifeMax / 100 * 15;  
9.     else if (life > c_lifeMax / 100 * 20)  
10.         change = c_lifeMax / 100 * 10;  
11.     else change = Life{};  
12.     return life -= change;  
13. }
```

AAA for branching

`auto` needs initializers, but each branch may use a different type.

```
1.  template <
2.      typename If, typename Then, typename... Elses
3.  >
4.  decltype(auto) choose (
5.      If&& cif,
6.      Then&& cthen,
7.      Elses&&... celses
8.  ) {
9.      return cif() ? cthen()
10.         : choose(std::forward<Elses>(celses)...);
11. }
```

AAA for branching

Here's the terminating overload and a small example.

```
1.  template <typename If, typename Then>
2.  decltype(auto) choose (If&& cif, Then&& cthen) {
3.      return
4.          cif()
5.          ? cthen()
6.          : std::result_of_t<Then()>{};
7.  }
8.  auto&& result = choose(
9.      [] { return false; }, [] { return 0; },
10.     [] { return true; },  [] { return 1.0; }
11. ); // double (1.0)
```

AAA for branching

Each pair of lambdas forms an if->then branch.

```
1.  auto&& change = choose(  
2.    [&life]{ return life > c_lifeMax/100*80; },  
3.    [] { return c_lifeMax/100*25; },  
4.    [&life] { return life > c_lifeMax/100*60; },  
5.    [] { return c_lifeMax/100*20; },  
6.    [&life] { return life > c_lifeMax/100*40; },  
7.    [] { return c_lifeMax/100*15; },  
8.    [&life] { return life > c_lifeMax/100*20; },  
9.    [] { return c_lifeMax/100*10; }  
10. );
```

Branching Observations

choose eschews raw branches like `for_each` does raw loops.

It expects and enforces a regular pattern for each ‘branch’.

However, the branching behavior is now completely hidden.

Errors due to mismatched arguments can be harder to diagnose.

Passing a `std::pair<condition, result>` is safer but more verbose.

A lambda IIFE is a good non-algorithmic alternative.

Branching Statistics

Explicit types **decreased** from 3 to 0.

Deduced types **increased** from 0 to 12.

Words **increased** from 81 to 97.

Characters **increased** from 427 to 493.

Elapsed time to process 10,203,900 turns **equivalent** at 407ms.

Optimized assembly is identical.

Looping

Version 2

Add multiplayer (array of Life).

Turns iterate over the array.

Skip array element if Life is already 0.

End game when all Life values are 0.

Automagic version 2.0 - No AAA

```
1. struct Game {  
2.     std::array<Life, 3> m_life;  
3.     Game () { m_life.fill(c_lifeMax); }  
4.     bool Turn () {  
5.         bool anyAlive = false;  
6.         for (LifeArray::value_type& life : m_life) {  
7.             if (life > 0)  
8.                 anyAlive = spell(life) > 0 || anyAlive;  
9.         }  
10.        return anyAlive;  
11.    }  
12. };
```


AAA for looping

Swapping raw loops for algorithms is straightforward.

```
1.  auto&& anyAlive = false;
2.  std::for_each(
3.      std::begin(m_life),
4.      std::end(m_life),
5.      [this, &anyAlive] (auto& life) {
6.          if (life > 0)
7.              anyAlive = spell(life) > 0 || anyAlive;
8.      }
9.  );
```

AAA for looping

We can avoid initializing outside of the loop with additional deduction.

```
1.  template <
2.      typename InIt, typename UnOp,
3.      typename BiOp, typename T
4.  >
5.  T accumulate (InIt f, InIt l, UnOp&& uop,
6.      BiOp&& bop, T&& init = T{})
7.  {
8.      for (; f != l; ++f)
9.          init = bop(std::forward<T>(init), uop(*f));
10.     return init;
11. }
```

AAA for looping

```
1.  auto&& anyAlive = accumutate(  
2.      std::begin(m_life),  
3.      std::end(m_life),  
4.      [this] (auto& life) {  
5.          if (life > 0)  
6.              spell(life, m_engine);  
7.          return life > 0;  
8.      },  
9.      [] (auto&& anyAlive, auto&& result) {  
10.          return anyAlive || result;  
11.      }  
12. );
```

Looping Observations

Eschewing raw loops requires additional work for accumulation.

Most standard algorithms do not support mutating range contents.

i.e. `std::any_of` would be ideal here.

`accumulate` unlike `std::transform_reduce` can deduce `T` and default it.

Generic algorithm parameter order & side effects are important.

Looping statistics

Explicit types **decreased** from 21 to 5.

Deduced types **increased** from 0 to 27.

Words **increased** from 52 to 64.

Characters **increased** from 431 to 654.

Elapsed time for 22,018,700 turns **increased** from 2150ms to 2198ms.

Optimized assembly is similar but not identical.

Benefits

- auto guarantees initialization
- Avoids accidental conversion
- Avoids accidental slicing
- Consistent type placement
- Explicit type dependency
- Promotes encapsulation
- Promotes generic code
- Simplifies iterator usage
- Use proxy types (`std::bitset`)
- Use unspecified types

Drawbacks

- Accidental copying
- Accidental references
- Algorithm complexity
- Incomplete/wrong IDE info
- Language arcana
- No deleted/defaulted funcs
- Not for non-static members
- Opacity
- Paradigm shift
- Searchability

AAAARGH!?

Apparently
Almost-Always-Auto
Requires Generic
Heuristics!?

More (keyboard) typing for
less (explicit) typing.

- Case Study
- Variables
- Functions
- Lambdas
- Branching
- Looping

Thanks!

Ben Deane

Ben Wooller

Carl Chimes

Geoff Tucker

Grant Mark

Otmar Schlunk

Andy Bond

<https://git.io/vi7EE>

Resources

[Effective Modern C++](#), by Scott Meyers. (Chapter 1: Deducing Types)

[Effective Modern C++](#), by Scott Meyers. (Chapter 2: auto)

[Sutter's Mill: AAA Style](#), by Herb Sutter.

[Sutter's Mill: Elements of Modern C++ Style](#), by Herb Sutter.

[auto considered awesome](#) & [auto considered harmful](#), by Jarryd Beck.

[auto specifier](#), from cppreference.com

[C++ Seasoning](#), by Sean Parent.

[decltype specifier](#), from cppreference.com

[Folds \(ish\) In C++11](#), by Jason Turner.

[Gotchas of Type Inference](#), by Andrzej Krzemiński.

[Lambda Overloading & Recursion](#), by Jamboree.

[Significant Parentheses](#), by KrzaQ.

[SFINAE std::result_of? Yeah right!](#), by Scott Prager.

[Value Categories](#), from cppreference.com

[Variadic templates in C++](#), by Eli Bendersky.

Appendix

AAA for algorithms

Passing template functions to algorithms can be difficult.

```
1. auto&& begin = std::begin(container);
2. auto&& end = std::end(container);
3. auto&& initial =
4.     std::remove_reference_t<
5.         decltype(*std::begin(container))
6.     >{};
7. auto&& maximum = std::accumulate(
8.     begin, end, initial, ???
9. ); // std::max ambiguous
```

AAA for algorithms

Use generic forwarding lambdas to defer overload resolution.

```
1. inline auto DeducedMax () {  
2.     return [] (auto&&... args) -> decltype(auto) {  
3.         using std::max; // assist with ADL  
4.         return max(  
5.             std::forward<decltype(args)>(args)...  
6.         );  
7.     };  
8. }  
9. auto&& result = std::accumulate(  
10.     begin, end, initial, DeducedMax()  
11. );
```

AAA for looping

std::arrays need helpers to be filled when initialized.

```
1. template <typename C, typename T>
2. auto make_filled_array (const C&, const T& t) {
3.     C copy{};
4.     copy.fill(t);
5.     return copy;
6. }
7. struct Game {
8.     std::array<Life, 3> m_life{
9.         make_filled_array(m_life, c_lifeMax)
10.    };
11. };
```

Recursion

Version 3

Add a Rend spell.

Pick a random number between 0 and max life.

Find greatest common divisor (GCD) between it and current life.

Divide life by the GCD.

Automagic version 3.0 - No AAA

Standard recursion, CalcGCD calls itself by name.

```
1. Life Game::CalcGCD (Life&& a, Life&& b) {  
2.     return (b == Life{}) ? a : CalcGCD(b, a % b);  
3. }  
4.  
5. Life& Game::Rend (Life& life) {  
6.     UD<Life> dis{0, c_lifeMax};  
7.     return life /= CalcGCD(life, dis(m_engine));  
8. }
```

AAA for recursion

Use a generic parameter to pass the lambda.

```
1. auto&& gcd = [] (auto&& f, auto&& a, auto&& b) {  
2.     if (b == decltype(b){}) return a;  
3.     return f(  
4.         std::forward<decltype(f)>(f),  
5.         std::forward<decltype(b)>(b),  
6.         a % b  
7.     );  
8. };  
9. auto&& result = gcd(gcd, 62880, 92680);
```

AAA for recursion

```
1.  template <typename F, typename... Ts>
2.  decltype(auto) recurse (F&& f, Ts&&... ts) {
3.      return f(
4.          std::forward<F>(f), std::forward<Ts>(ts)...
5.      );
6.  }
7.  [] (auto& life, auto& engine) -> auto& {
8.      auto&& dis = make_ud(0, c_lifeMax);
9.      return life /= recurse(
10.         [] (auto&& f, auto&& a, auto&& b) { ... },
11.         life,
12.         dis(engine)
13.     );
14. }
```


Recursion Observations

The lambda version looks markedly different.

The lambda version requires passing to a helper or named variable to recurse.

Forwarding is very verbose since auto variables require using decltype.

VS2015 needs a separate terminating return.

Recursion Statistics

Explicit types **decreased** from 7 to 0.

Deduced types **increased** from 0 to 12.

Words **increased** from 34 to 41.

Characters **increased** from 248 to 352.

Elapsed time to process 9,773,100 turns **increased** from 818ms to 831ms.

Optimized assembly is fairly divergent.

AAA for arguments

Traits can deduce the type of arguments from callable object parameters.

```
1.  template <typename T>
2.  struct parameters_of;    // Not defined
3.  template <typename F, typename... Args>
4.  struct parameters_of<F(Args...)> {
5.      using type = std::tuple<std::decay_t<Args>...>;
6.  };
7.
8.  bool test (int& x, int y);
9.  using params_t = typename parameters_of<decltype(test)>::type;
10.
11. auto&& x = std::tuple_element_t<0, params_t>{}; // int
12. auto&& y = std::tuple_element_t<1, params_t>{}; // int
13. auto&& result = test(x, y);
```

Variables - auto

```
1.  int example{};
2.
3.  // lvalues
4.  auto a = example;
5.  //auto* b = example;
6.  auto& c = example;
7.  auto&& d = example;
8.
9.  // prvalues
10. auto e = &example;
11. auto* f = &example;
12. //auto& g = &example;
13. auto&& h = &example;
14.
15. // xvalues
16. auto i = std::move(example);
17. //auto& j = std::move(example);
18. auto&& k = std::move(example);
```

```
1.
2.
3.
4.  int
5.  <n/a>
6.  int&
7.  int&
8.
9.
10. int*
11. int*
12. <n/a>
13. int*&&
14.
15.
16. int
17. <n/a>
18. int&&
```

Variables - auto

1. <code>const int</code> example{};	1.
2.	2.
3. <code>// lvalues</code>	3.
4. <code>auto</code> a = example;	4. <code>int</code>
5. <code>//auto* b = example;</code>	5. <code><n/a></code>
6. <code>auto&</code> c = example;	6. <code>const int&</code>
7. <code>auto&&</code> d = example;	7. <code>const int&</code>
8.	8.
9. <code>// prvalues</code>	9.
10. <code>auto</code> e = &example;	10. <code>const int*</code>
11. <code>auto*</code> f = &example;	11. <code>const int*</code>
12. <code>//auto& g = &example;</code>	12. <code><n/a></code>
13. <code>auto&&</code> h = &example;	13. <code>const int*&&</code>
14.	14.
15. <code>// xvalues</code>	15.
16. <code>auto</code> i = std::move(example);	16. <code>int</code>
17. <code>auto&</code> j = std::move(example);	17. <code>const int&</code>
18. <code>auto&&</code> k = std::move(example);	18. <code>const int&&</code>

Variables - decltype

1. <code>int</code> example{};	1.
2.	2.
3. <code>//</code> entities	3.
4. <code>decltype</code> (example) a;	4. <code>int</code>
5. <code>decltype</code> (example)& b{example};	5. <code>int&</code>
6. <code>decltype</code> (example)&& c{...};	6. <code>int&&</code>
7.	7.
8. <code>//</code> lvalues	8.
9. <code>decltype</code> ((example)) d{example};	9. <code>int&</code>
10. <code>decltype</code> (*(&example)) e{example};	10. <code>int&</code>
11.	11.
12. <code>//</code> prvalues	12.
13. <code>decltype</code> (example + 0) f;	13. <code>int</code>
14. <code>decltype</code> ((example + 0)) g;	14. <code>int</code>
15. <code>decltype</code> (&example) h;	15. <code>int*</code>
16.	16.
17. <code>//</code> xvalues	17.
18. <code>decltype</code> (std::move(example)) i{...};	18. <code>int&&</code>

Variables - decltype

```
1.  const int example{};
2.
3.  // entities
4.  decltype(example) a{example};
5.  decltype(example)& b{example};
6.  decltype(example)&& c{...};
7.
8.  // lvalues
9.  decltype((example)) d{example};
10. decltype(*(&example)) e{example};
11.
12. // prvalues
13. decltype(example + 0) f;
14. decltype((example + 0)) g;
15. decltype(&example) h;
16.
17. // xvalues
18. decltype(std::move(example)) i{...};
```

```
1.
2.
3.
4.  const int
5.  const int&
6.  const int&&
7.
8.
9.  const int&
10. const int&
11.
12.
13. int
14. int
15. const int*
16.
17.
18. const int&&
```

Variables - decltype(auto)

1. <code>int</code> example{};	1.
2.	2.
3. <code>// entities</code>	3.
4. <code>decltype(auto)</code> a = example;	4. <code>int</code>
5. <code>//decltype(auto)& b = example;</code>	5. <code><n/a></code>
6. <code>//decltype(auto)&& c = example;</code>	6. <code><n/a></code>
7.	7.
8. <code>// lvalues</code>	8.
9. <code>decltype(auto)</code> d = (example);	9. <code>int&</code>
10. <code>decltype(auto)</code> e = (&example);	10. <code>int&</code>
11.	11.
12. <code>// prvalues</code>	12.
13. <code>decltype(auto)</code> f = example + 0;	13. <code>int</code>
14. <code>decltype(auto)</code> g = (example + 0);	14. <code>int</code>
15. <code>decltype(auto)</code> h = &example;	15. <code>int*</code>
16.	16.
17. <code>// xvalues</code>	17.
18. <code>decltype(auto)</code> i = std::move(a);	18. <code>int&&</code>

Variables - decltype(auto)

```
1.  const int example{};
2.
3.  // entities
4.  decltype(auto) a = example;
5.  //decltype(auto)& b = example;
6.  //decltype(auto)&& c = example;
7.
8.  // lvalues
9.  decltype(auto) d = (example);
10. decltype(auto) e = (&example);
11.
12. // prvalues
13. decltype(auto) f = example + 0;
14. decltype(auto) g = (example + 0);
15. decltype(auto) h = &example;
16.
17. // xvalues
18. decltype(auto) i = std::move(a);
```

```
1.
2.
3.
4.  const int
5.  <n/a>
6.  <n/a>
7.
8.
9.  const int&
10. const int&
11.
12.
13. int
14. int
15. const int*
16.
17.
18. const int&&
```