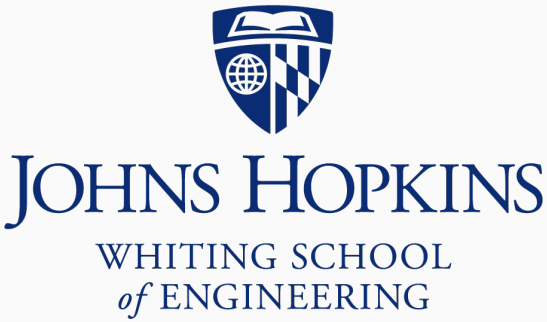
auto

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

auto

Use auto in a place where you would otherwise have written a type:

```
for(auto it = vc.begin(); it != vc.end(); it++) {
    ...
}
```

Doing so leaves type unspecified; up to the C++ compiler to *infer* appropriate type

auto

Sometimes, this is an easy inference to make:

```
int a = 7;
auto b = a; // b clearly an int
```

Made slightly trickier by type promotion:

```
int c = 7;
double d = 11.1;
auto e = c * d; // e is a double
```

auto

auto variable must be initialized; compiler must be able to infer type immediately

```
auto x = 0;
// above is OK

auto y;
y = 7;
// compile error:
// declaration of ‘auto y’ has no initializer
```

auto

```
#include <iostream>

using std::cout;  using std::endl;

int main() {
    int a = 7;
    auto b = a;
    cout << "b=" << b << ", size=" << sizeof(b) << endl;

    int c = 7;
    double d = 11.1;
    auto e = c * d; // e is a double
    cout << "e=" << e << ", size=" << sizeof(e) << endl;

    return 0;
}
```

auto

```
$ g++ -c auto1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o auto1 auto1.o
$ ./auto1
b=7, size=4
e=77.7, size=8
```

auto

We saw auto in the context of iterators:

```
int sum_every_other(const vector<int>& ls) {
    int total = 0;
    for(vector<int>::const_iterator it = ls.cbegin();
        it != ls.cend(); it++)
    {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

auto

```
int sum_every_other(const vector<int>& ls) {
    int total = 0;
    // using auto instead
    for(auto it = ls.cbegin(); it != ls.cend(); it++) {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

auto

Con: types aren't as clear from the source code

Pro: Source is more concise and, arguably, easier to maintain

- Changes to types on right-hand side propagate to left:

```
int c = 7;
double d = 11.1;
// changing d to int would also change e to int!
auto e = c * d;
```

auto

E.g., changing parameter type from vector<int> to list<int> automatically propagates to iterator type:

```
int sum_every_other(const list<int>& ls) {
    int total = 0;
    // it *was* vector<int>::const_iterator
    // *now* it's list<int>::const_iterator
    for(auto it = ls.cbegin(); it != ls.cend(); it++) {
        total += *it;
        if(++it == ls.cend()) { break; }
    }
    return total;
}
```

auto

Caution: compiler-inferred type can be unexpected:

```
vector<int> vec = {1, 2, 3};
const vector<int>& vec_ref;
auto vec2 = vec_ref;
```

Is vec2 a *copy* of vec, or an alias?

auto

```cpp
#include <iostream>
#include <vector>

using std::cout;  using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    const vector<int>& vec_ref = vec;
    auto vec2 = vec_ref;

    vec[0] = 10;
    cout << "vec[0] = " << vec[0] << endl;
    cout << "vec2[0] = " << vec2[0] << endl;

    return 0;
}
```

auto

```
$ g++ -c auto2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o auto2 auto2.o
$ ./auto2
vec[0] = 10
vec2[0] = 1
```

It's a copy

Compiler inferred type vector<int> for vec2, *not* const vector<int>&