auto, range-based for, smart pointers

Using materials from
Stanford CS 106L Spring 2022 (Instructors: Frankie Cerkvenik and Sathya Edamadaka)
https://learn.microsoft.com/en-us/cpp/cpp/range-based-for-statement-cpp?view=msvc-170

Definition

`auto`: Keyword used in lieu of type when declaring a variable, tells the compiler to deduce the type.

Type Deduction using `auto`

```
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = ‘X’;
auto d = “Hello”;
auto e = std::make_pair(3,  “Hello”);
```

📝 `auto` **does not mean that the variable doesn’t have a type.**
It means that the type is **deduced** by the compiler.

Type Deduction using `auto`

```
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = 'X';
auto d = "Hello";
auto e = std::make_pair(3,  "Hello");
```

Answers: int, double, char, char* (a C string), std::pair<int, char*>

📝 `auto` **does not mean that the variable doesn't have a type.**

It means that the type is **deduced** by the compiler.

‼️ `auto` does not mean that the variable doesn't have a type.

It means that the type is **deduced** by the compiler.

Type Deduction using auto

WARNING: auto drops references and const:

```
const int x = 5;
auto y = x; //const is dropped, y is an int
y = 1; // no problem!

int a = 5;
int & ra = a;
auto b = ra; //reference is dropped, b is an int
b = 10;
cout << "a: " << a << endl; //prints “a: 5”
cout << "ra: " << ra << endl; //prints “ra: 5”
cout << "b: " << b << endl; //prints “b: 10”
```

`auto` CAN'T....

…be used as a template argument:

```
vector<auto> v{2,3}; //not allowed, won't compile!
```

…be used as a function parameter type*:

```
void func(auto a){...} //won't compile on some systems!
```

*this can be done in C++ 20, but you have to compile with a special flag right now on some systems

`auto` CAN....

…be the **return type** of a function:

```
auto sum(int a, int b)
{
    return a + b;
}
//...
auto out = sum(4,5);
```

This is especially useful for templated functions:

```
template <typename T1, typename T2>
auto sum_template(T1 a, T2 b)
{
    return a + b;
}
//...
auto out1 = sum_template(4,5);     //out1 is an int
auto out2 = sum_template(1.3,5.1); //out2 is a double
auto out3 = sum_template(9,2.1);   //out3 is a double
```

range-based for

Recall: Printing out elements of `std::map`

```
#include <map>
using namespace std;

//...

map<string, string> dictionary;
//add some entries to dictionary...

cout << "The Entire Dictionary:" << endl;
for(const pair<string,string>& elem : dictionary)
{
   cout << "Word: " << elem.first << endl;
   cout << "Definition: " << elem.second << endl;
}
```

Ranged-based for loop

```
for (range_declaration : range_expression) {
    //loop body
}
```

range_declaration :

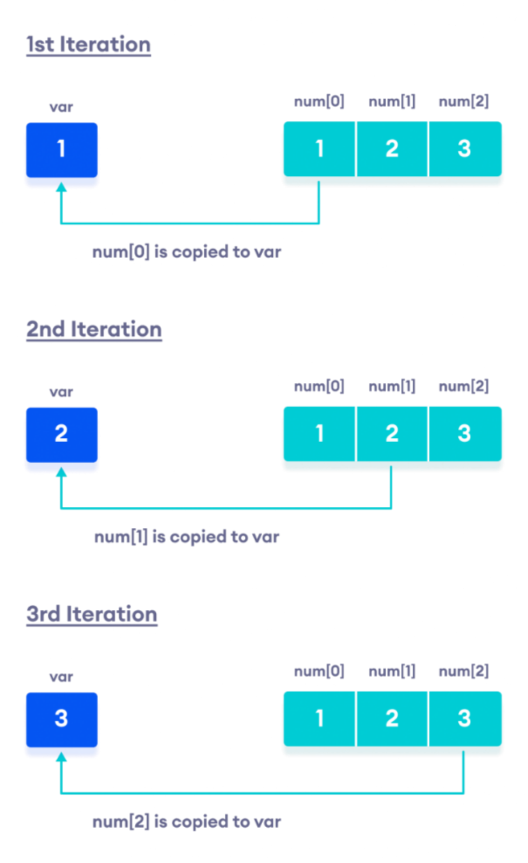
- a declaration of a named variable
- the type must be the type of the element of the sequence represented by range_expression (or a reference to that type)
- We'll often use auto here

range_expression :

- any expression that represents a suitable sequence
- or a braced-init-list, e.g. {1,3,5,6}

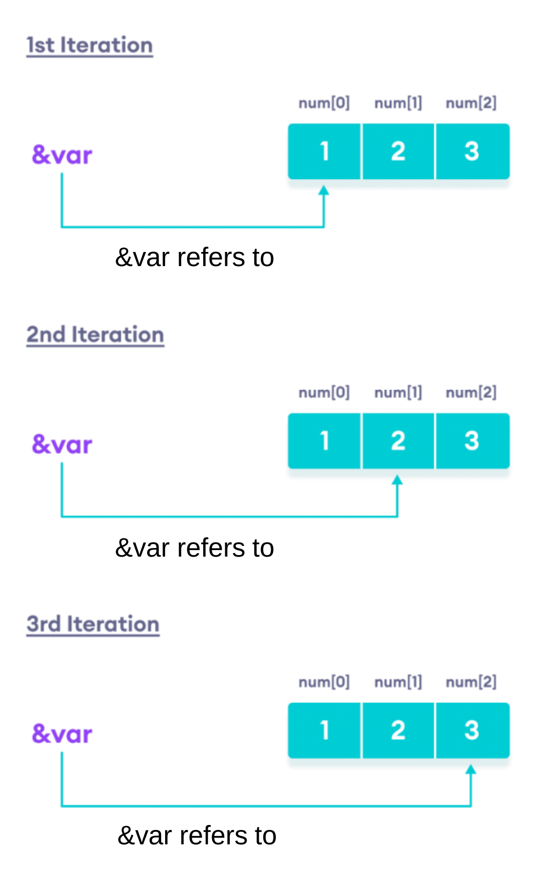
Range-based for loop using <u>copy</u>

```
// 3-element integer array.
int num[3] = { 1, 2, 3};

// Range-based for loop to iterate through the array.
for( int var : num ) {
    cout << var << " ";
}
```



Range-based for loop using references

```
// 3-element integer array.
int num[3] = { 1, 2, 3};

// Range-based for loop to iterate through the array.
for( int &var : num ) {
    cout << var << " ";
}
```



Range-based for loop examples, using auto

```
// 3-element integer array.
int num[3] = { 1, 2, 3};

// Range-based for loop to iterate through the array.
for( int var : num ) { // Access by value using a copy declared as a specific type.
                       // Not preferred.
    cout << var << " ";
}

// The auto keyword causes type inference to be used.
for( auto var : num ) { // Copy of ‘num', almost always undesirable
    cout << var << " ";
}

for( auto &var : num ) { // Type inference by reference.
    // Observes and/or modifies in-place. Preferred when modify is needed.
    cout << var << " ";
}
for( const auto &var : x ) { // Type inference by const reference.
    // Observes in-place. Preferred when no modify is needed.
    cout << var << " ";
}
```

Use this if you want to **read and/or write** to elements

Use this if you want to **only read** elements

Range-based for loop with STL

- Range-based for works with any object that has appropriate .begin() and .end() functions, e.g.:

- What is the type of a?

```
std::vector<int> v{1,2,3,4,5};
// Range-based for loop to iterate through the
// vector, observing in-place.
for( const auto& j : v ) {
    cout << j << " ";
}

// Same idea for map
std::map<int, string> m;
for(const auto& j : m)
{
    cout << “key: " << j.first << endl;
    cout << “value: " << j.second << endl;
}

// Same idea for string
std::string str = "Hello";
for (const auto & a: str)
{
    std::cout << a;
}
```

Spot the logic bug

```
std::vector<int> v{1,2,3};
for(auto i : v)
{
    i = 9;
}

for(auto i : v)
{
    std::cout << i;
}
```

- This will print out “123”, but we were expecting to get “999”

Spot the logic bug

```
std::vector<int> v{1,2,3};
for(auto &i : v)
{
    i = 9;
}

for(auto i : v)
{
    std::cout << i;
}
```

- Need to use `&` (make it a reference) so that you can write to elements of v
- Otherwise, `i` is a copy of an element (which is destroyed after each iteration)

Smart pointers

Smart Pointers

- Two smart pointers in C++ that automatically free underlying memory when destructed:
  - **`std::unique_ptr`**
    - Uniquely owns its resource, can't be copied
  - **`std::shared_ptr`**
    - Can make copies, destructed when underlying memory goes out of scope

To use these, include the <memory> header

std::unique_ptr

Before

```
void rawPtrFn() {
  Node* n = new Node;
  // do things with n
  delete n;
}
```

After!

```
void rawPtrFn() {
  std::unique_ptr<Node> n(new Node);
  // do things with n
  // automatically freed at end!
}
```

What if we could make copies of std::unique_ptr?

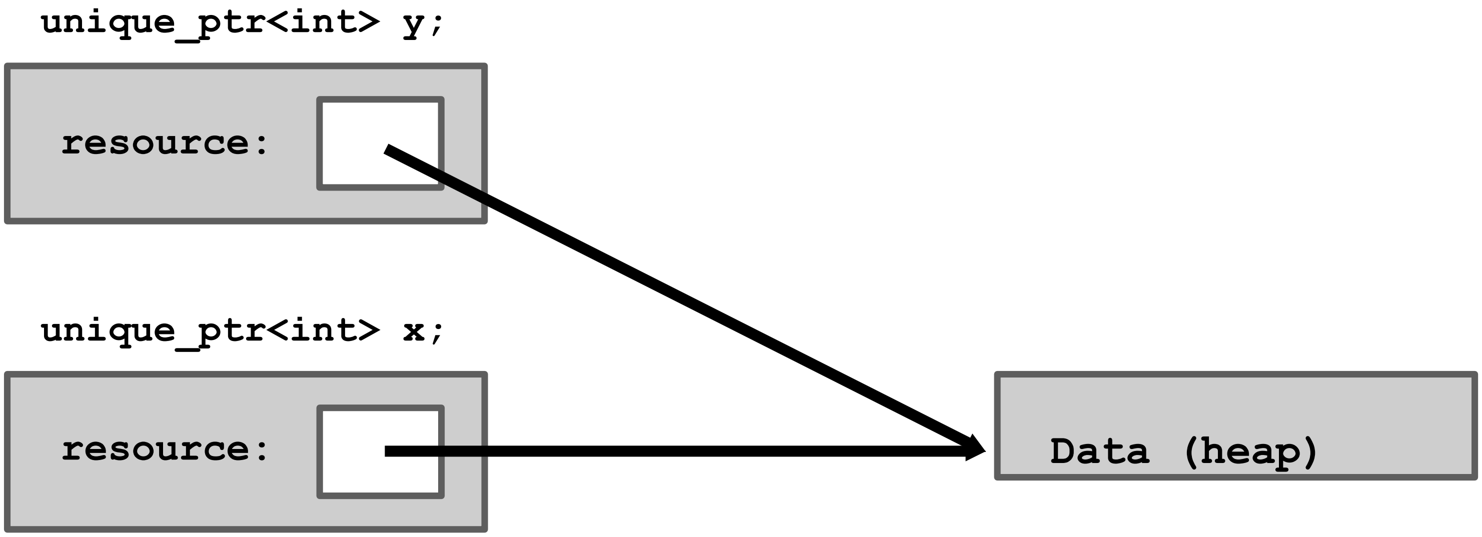
What if we could make copies of std::unique_ptr?

First we make a unique ptr:

```
unique_ptr<int> x;
```

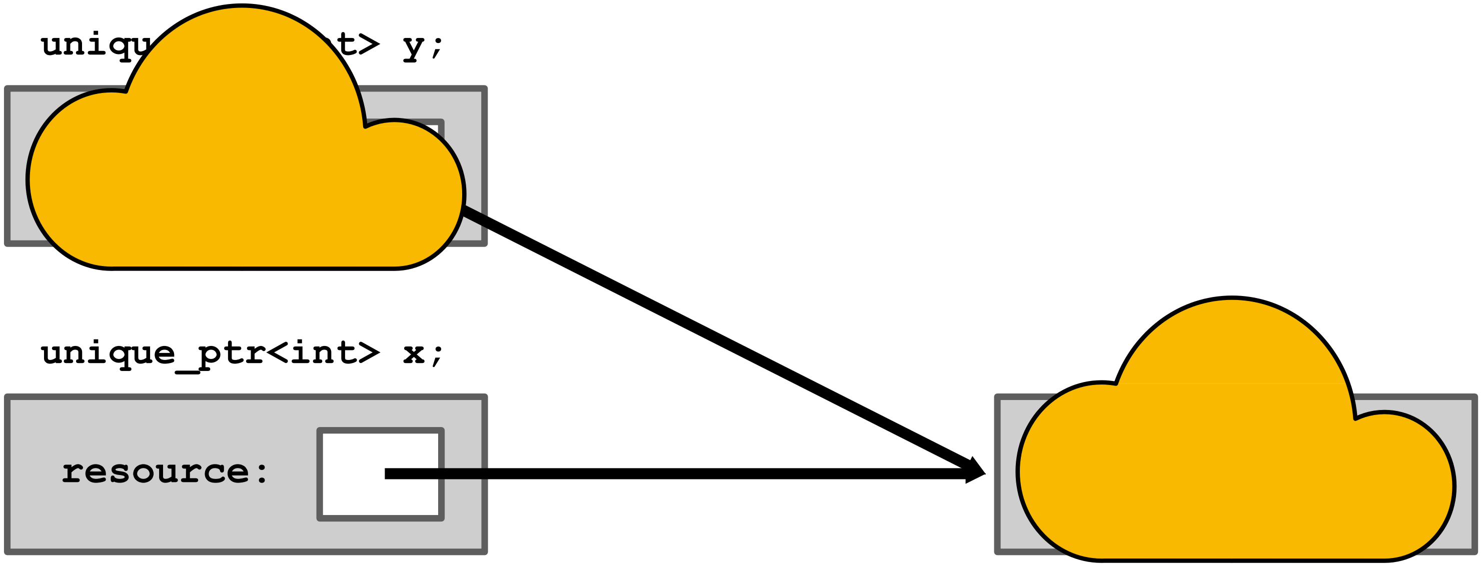
What if we could make copies of std::unique_ptr?

We'd then make a copy of this pointer, pointing to the same resource



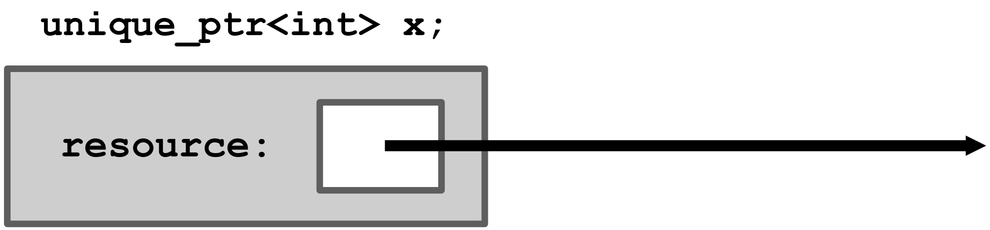
What if we could make copies of std::unique_ptr?

When y goes out of scope, it deletes the heap data



What if we could make copies of std::unique_ptr?

This leaves a hanging pointer x, which points at deallocated data



What if we could make copies of std::unique_ptr?

If we try to access x's data or delete runs the destructor, we crash!

But what if we wanted to have multiple pointers to the same object?

std::shared_ptr!

- Resources can be stored by any number of shared_ptrs
- The resource is **`deleted`** when none of the pointers points to the resource!

std::shared_ptr!

- Resources can be stored by any number of shared_ptrs
- The resource is **deleted** when none of the pointers points to the resource!

```
{
    std::shared_ptr<int> p1{new int(5)};
    // copy p1
    {
    std::shared_ptr<int> p2 = p1;
    }
    // use p1 like so
    cout << *p1<< endl;
}
// the integer is now deallocated!
```

std::shared_ptr!

`std::shared_ptr` manages two entities:

1. the control block (stores meta data such as ref-counts, deletion method, etc)
2. the object being managed

Here are a few useful methods for shared pointers:

```
std::shared_ptr<int> sp{new int(5)};

int* p = sp.get(); // returns the pointer to the object. WARNING: sp WON’T keep
                   // track of p! Don’t do this unless you have no other options.

sp.use_count();  // returns the number of shared_ptr objects that share ownership
                 // over the same pointer as this object (including it).

sp.reset(new int{10});  // deletes managed object, acquires new pointer
```

Smart Pointer Initialization

```
std::unique_ptr<T> up{new T};


std::shared_ptr<T> sp{new T};
```

Smart Pointers Initialization

```
std::unique_ptr<T> up{new T};

      OR
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};

      OR
std::shared_ptr<T> sp = std::make_shared<T>();
```

So which way is better?

```
std::unique_ptr<T> up{new T};
                     OR
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};
                     OR
std::shared_ptr<T> sp = std::make_shared<T>();
```

Answer:

Use std::make_shared<T>() to be more efficient.

So which way is better?

```
std::unique_ptr<T> up{new T};
                     OR
std::unique_ptr<T> up = std::make_unique<T>();

std::shared_ptr<T> sp{new T};
                     OR
std::shared_ptr<T> sp = std::make_shared<T>();
```

- If we don't use make_shared, then we're allocating memory twice (once for sp’s control block, and once for sp’s new T)!
- We should be consistent across smart pointers (do the same thing for unique_ptr)

auto and shared pointers

- Unfortunately can't use auto in shared_pointer template argument:

```
auto i = new int{5}; // auto works for normal pointers, i is an int*
shared_ptr<auto> si{new int{5}}; // won't compile!
```

- Can use auto when copying shared pointers:

```
shared_ptr<int> si{new int{5}};
auto si2 = si; //make another shared pointer
```

Homework

- Homework 5 due Weds
- Read Graphs in computer science (up to Representing graphs in a computer)