Standard Library Improvements

Giovanni Dicanio
AUTHOR, SOFTWARE ENGINEER

https://blogs.msmvps.com/gdicanio

Overview

make_unique and smart pointers

Standard-defined literals

Fetching items from tuples by types

(Raw) Pointers in Modern C++

Basic Level

No need

Intermediate Level

May need

Pointers and Addresses in Computer Memory

Human Address

Connie White

113 Awesome Street

98101 Seattle, WA

USA

Address in Computer Memory

`0x7ffca2e06bdc`

```
int n{64};
```

Address of n

```
&n
```

Variable: Where Do You Live?

```
int n{64};
```

Pointer to n

```
p = &n;
```

Variable: Where Do You Live?

```
int n{64};
int *p = &n;
```

Pointer type

Variable: Where Do You Live?

`nullptr`

The Null Pointer

Points to *nothing*

```
void DoSomething() {
  int n{64};
  double speed{10.0};
  bool success{true};

  // More code ...
```

Allocated on the *stack*

Variables Allocated on the Stack

Stack: Pros and Cons

Very fast allocations

Limited amount of space

Larger Amount of Data: Allocate on the **Heap**

Heap: Pros and Cons

Slower allocations

Can serve much *larger* chunks of memory

Allocate from the *heap*

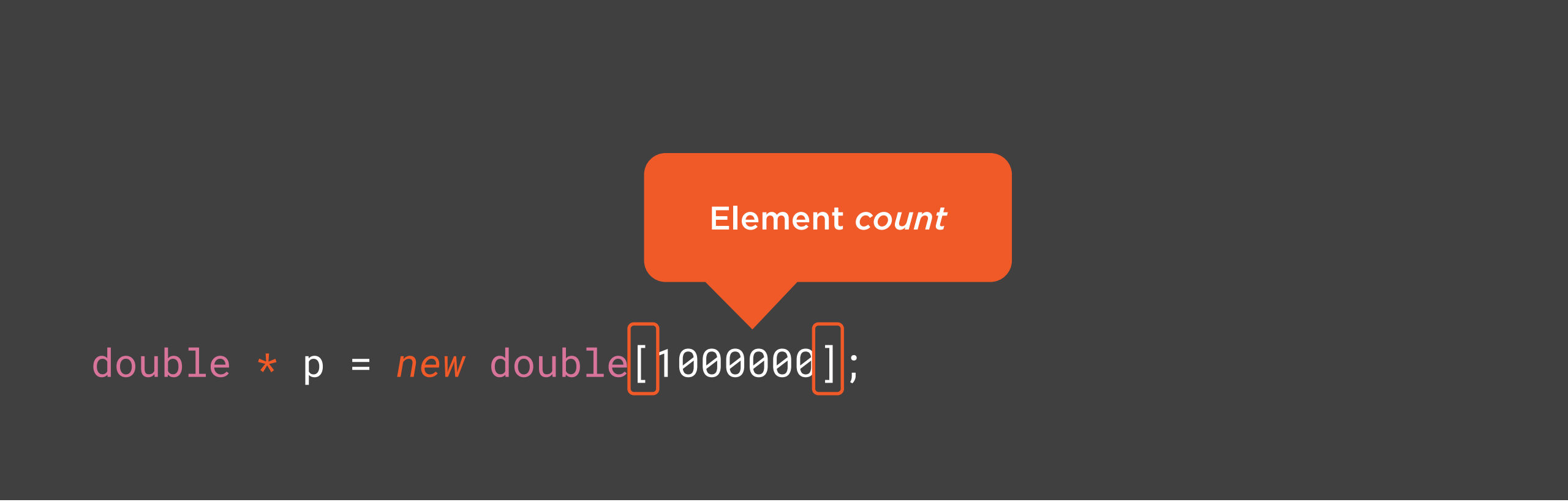
```
double * p = new double[1000000];
```

Requesting Memory from the Heap

Element *type*

```
double * p = new double[1000000];
```

Requesting Memory from the Heap



Requesting Memory from the Heap

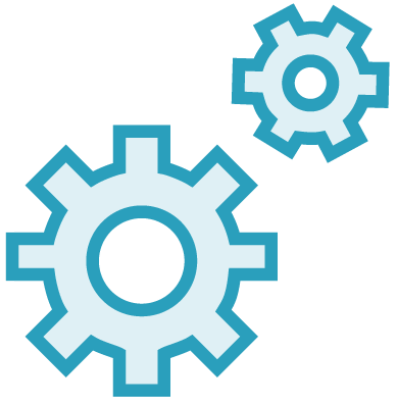
Points to the *beginning* of the allocated memory

```
double * p = new double[1000000];
```

Requesting Memory from the Heap

Memory Lifecycle

Allocate



Use

Allocated memory must be *released*

```
// 1. Allocate big chunk of memory using new[]
// 2. Use the memory
// 3. Must release it!
delete [] p;
p = nullptr; // Avoid dangling references
```

Memory Lifecycle with Raw Owning Pointers

```
{
  ResourceManager x{/* ... */};
  x.DoSomething();
  // More code...
}
```

Resource allocation

Resource use

Resource *automatically* released

~ResourceManager

«C++11 from Scratch»

Automatic Resource Cleanup with Destructors

bit.ly/CppAutoResMgm

Resource Manager

Automatically releases the owned resource

Raw vs. Smart Pointers

Raw (Owning) Pointers

Must *explicitly* delete/release

***Smart* Pointers (`unique_ptr`)**

Automatically deleted

```cpp
#include <memory> // For unique_ptr and make_unique

// Create a unique_ptr to an array of 1,000,000 doubles
auto p = std::make_unique<double[]>(1000000);
```

«Smart» *new*

Creating unique_ptr's with make_unique

```cpp
#include <memory> // For unique_ptr and make_unique

// Create a unique_ptr to an array of 1,000,000 doubles
auto p = std::make_unique<double[]>(1000000);
```

Element count

Creating unique_ptr's with make_unique

```cpp
#include <memory> // For unique_ptr and make_unique

// Create a unique_ptr to an array of 1,000,000 doubles
auto p = std::make_unique<double[]>(1000000);
```

Array of doubles

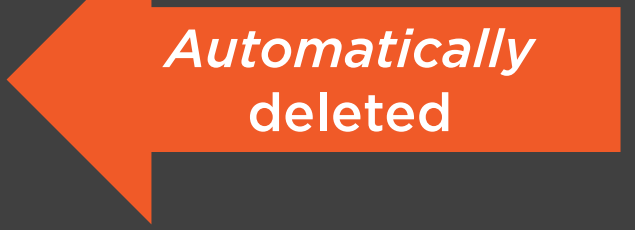
Creating unique_ptr's with make_unique

```cpp
#include <memory> // For unique_ptr and make_unique


// Create a unique_ptr to an array of 1,000,000 doubles
auto p = std::make_unique<double[]>(1000000);
```

Automatically deduce the type

Creating unique_ptr's with make_unique

```cpp
#include <memory> // For unique_ptr and make_unique

// Create a unique_ptr to an array of 1,000,000 doubles
auto p = std::make_unique<double[]>(1000000);
```



Creating unique_ptr's with make_unique

A *smart* pointer like unique_ptr does *not* require explicit delete

In modern C++ code, in most cases,
there should be no *explicit*
calls to *new* and *delete.*

```
class Image {
   ...
   int mWidth;
   int mHeight;
   unique_ptr<byte[]> mPixels;
};
```

Some Applications of unique_ptr

Safe owning-pointer to image pixels

```
// Factory function
unique_ptr<SomeObject> MakeObject(/* settings… */) {
   ...
}
```

Some Applications of unique_ptr

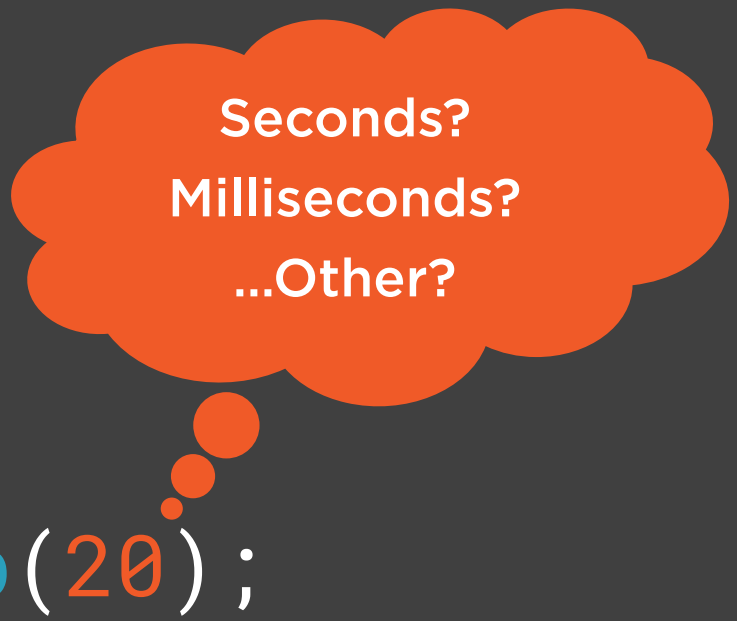
Safe owning-pointer returned by a factory function

Demo

Raw owning pointers (and leaks)

Smart pointers

- make_unique and unique_ptr

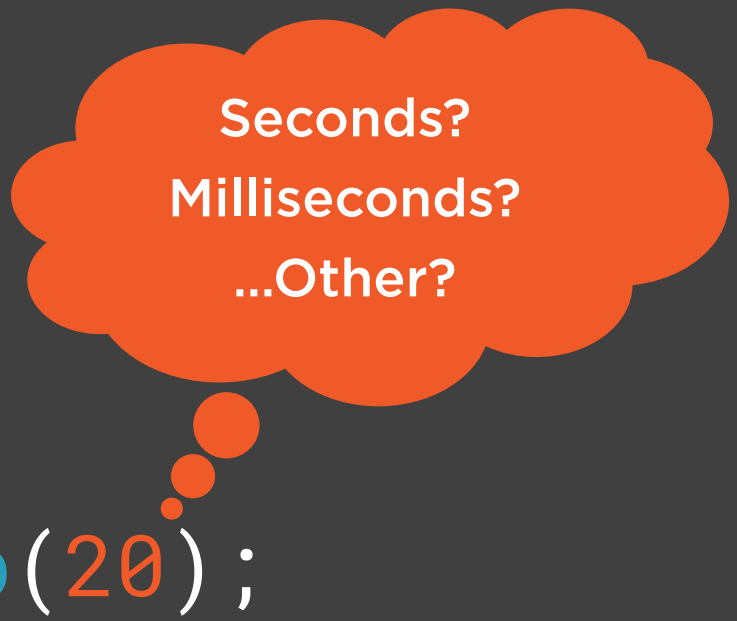


Units of Measurement

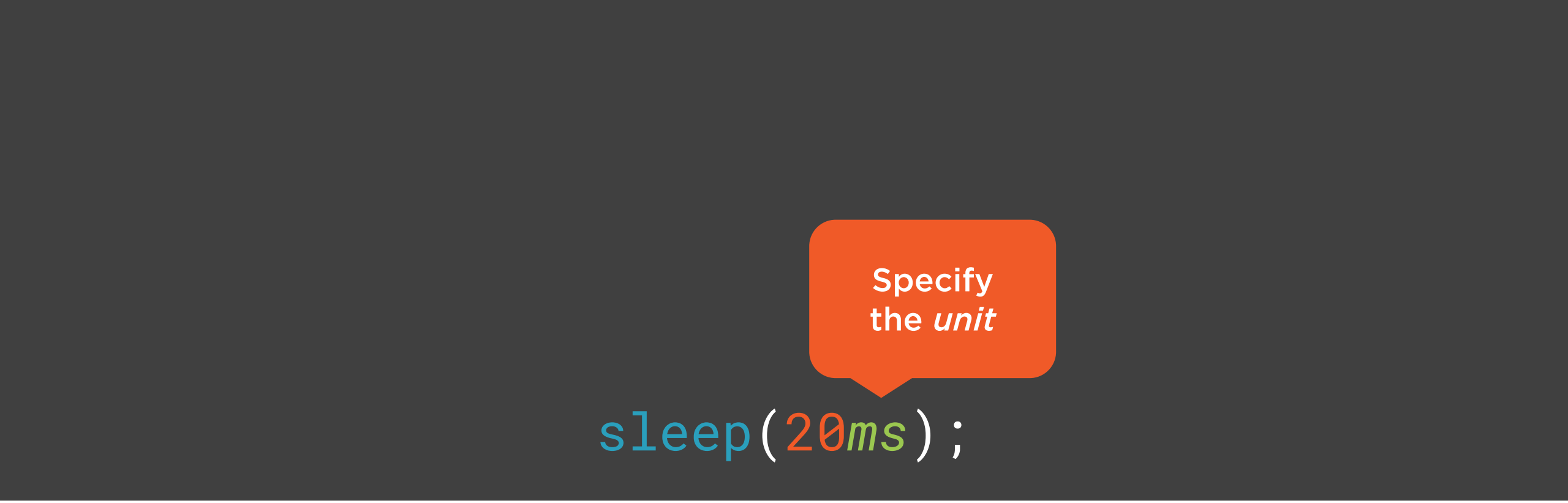
1999: $125 *million* satellite burned up

English/metric unit *mismatch*

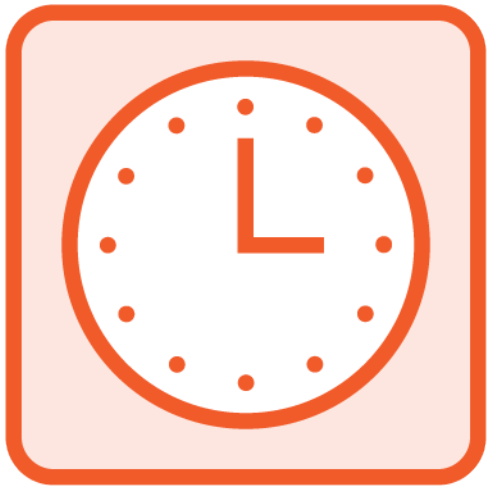
Units *are* important



Units of Measurement



Units of Measurement



Header <chrono>

- namespace **std::chrono**

Time duration

- hours
- minutes
- seconds
- milliseconds
- microseconds
- nanoseconds

Automatic
unit conversions

Zero initialization

```
seconds s{};     // 0-initialized
seconds s;       // No initialization
```

Just like *int* ...

Time Duration: Seconds

Initialization

«C++11 from Scratch»

Declaring and Initializing Variables

bit.ly/CppInitVar

```
seconds s{10};   // 10 seconds
seconds s = 10; // Will not compile
auto s = 10s;    // 10 seconds (C++14!)
```

auto deduces chrono::seconds

Time Duration: Seconds

Initialization

Standard-defined Literals

Literal Suffix	std::chrono::duration
h	hours
min	minutes
s	seconds
ms	milliseconds
us	microseconds
ns	nanoseconds

```
auto s = "Connie";
```

s deduced *pointer (const char*)*

```
auto s = "Connie"s;
```

s deduced *std::string*

Deducing Strings

```
auto x = "Connie"s; // auto deduces std::string

auto y = 10s;       // auto deduces std::chrono::seconds
```

Same Suffix for Different Types

Demo

Chrono standard-defined literals in action

std::tuple

```
#include <tuple>
```

First Name	Last Name
`string`	`string`
GPA	**Grade**
`double`	`char`

Tuple Is a Generalization of Pair

`std::pair`

2 elements

`std::tuple`

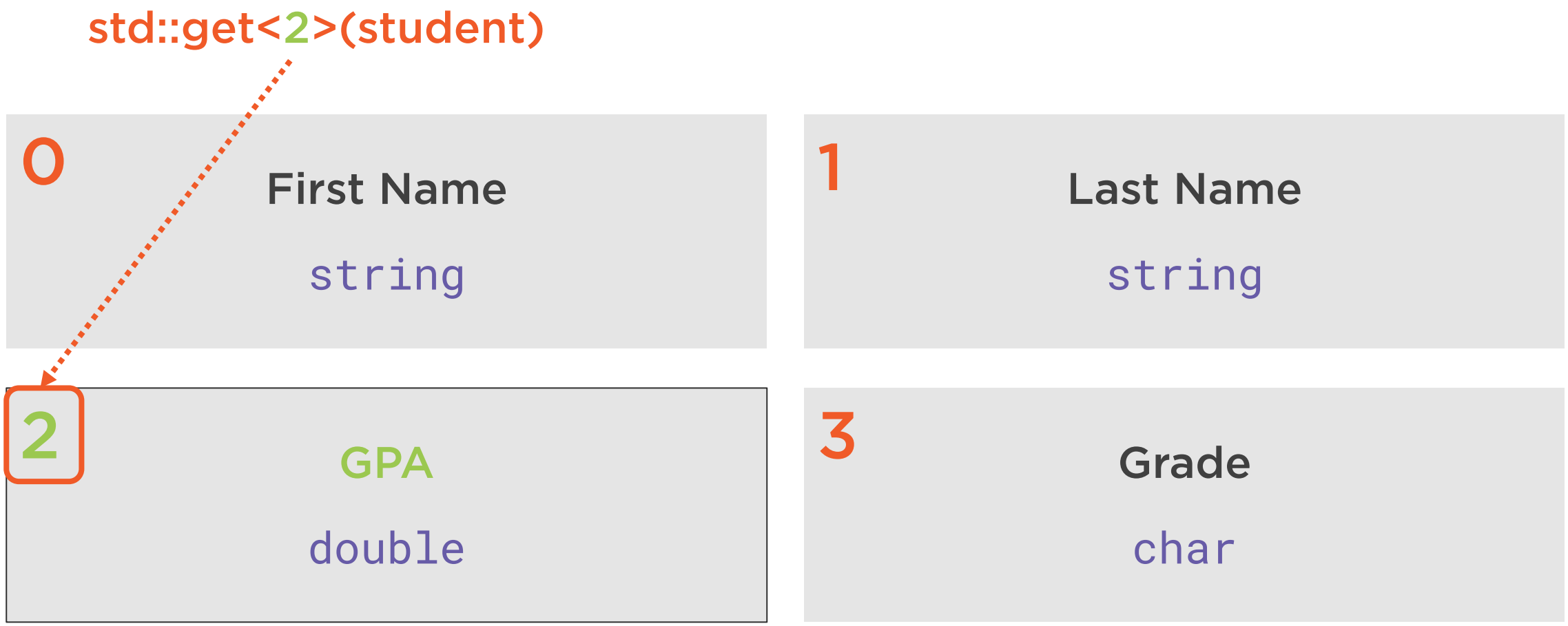
***N* elements**

Fetching Elements from std::tuple

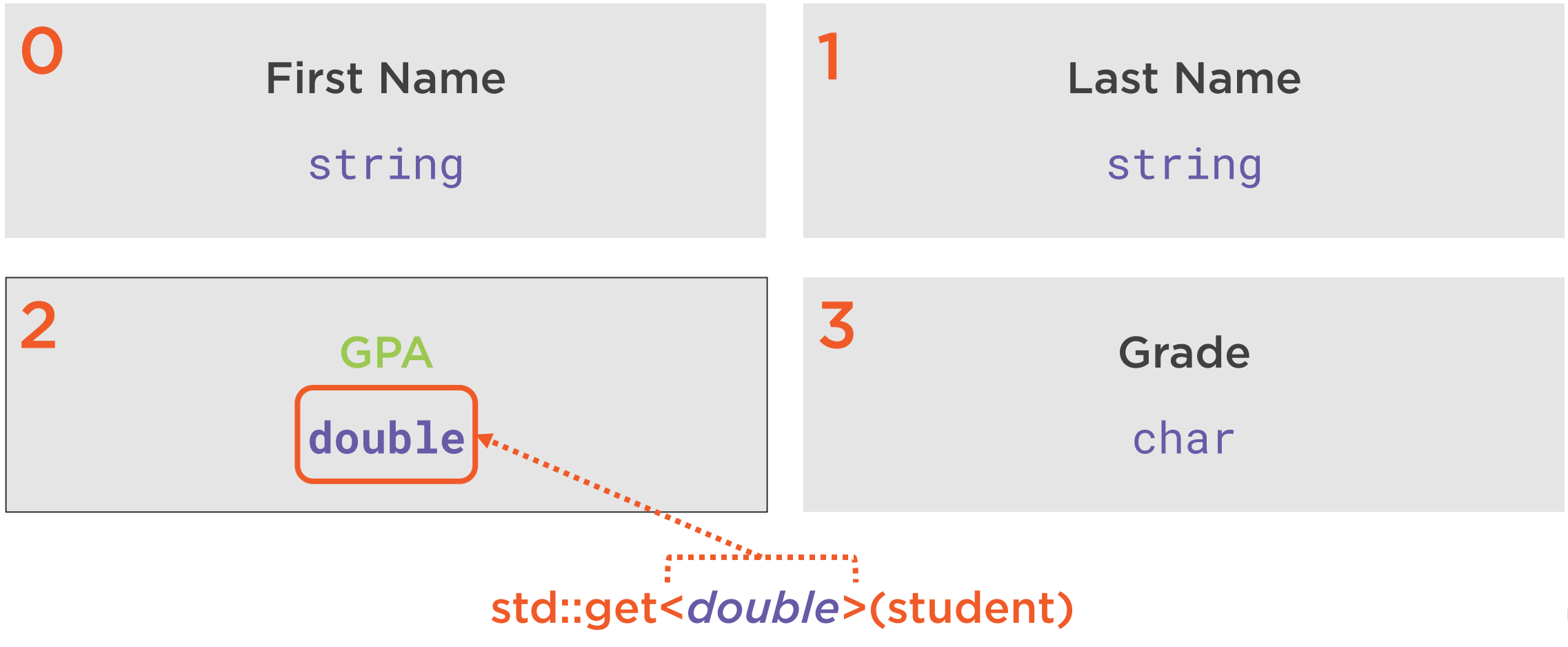
std::get<> with 0-based Index

0	First Name `string`	1	Last Name `string`
2	GPA `double`	3	Grade `char`

Fetching Elements from std::tuple



Fetching Elements from std::tuple



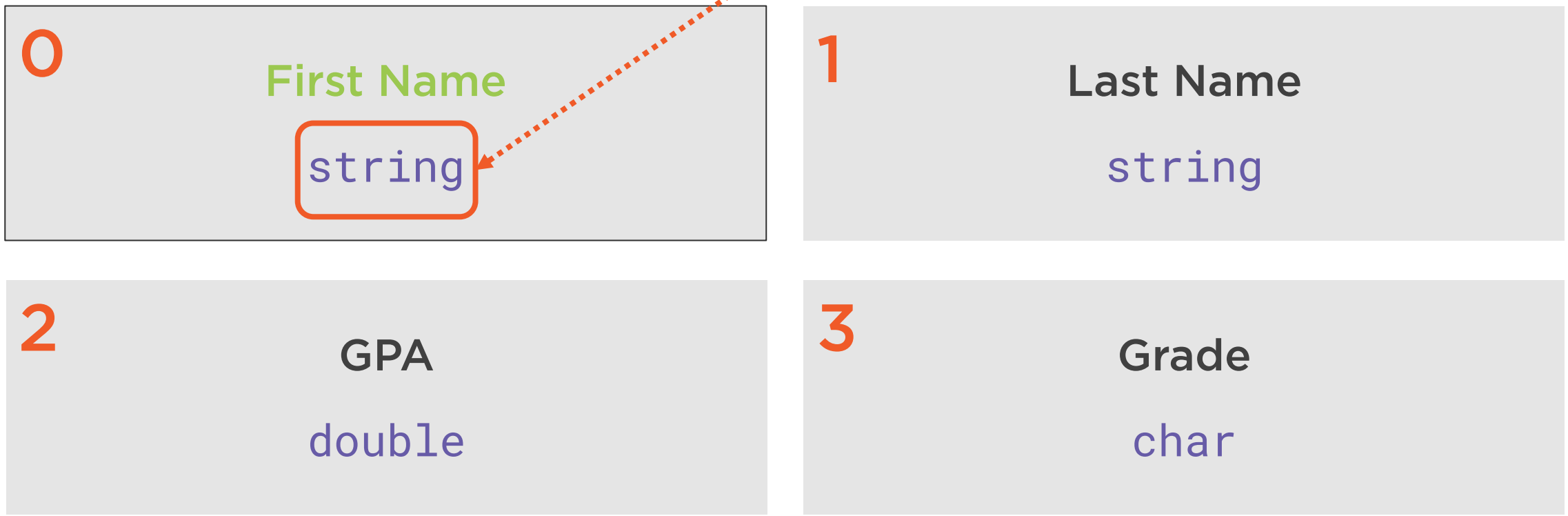
Fetching Elements from std::tuple

std::get<*string*>(student)

0	First Name `string`
1	Last Name `string`
2	GPA `double`
3	Grade `char`

Fetching Elements from std::tuple

std::get<*string*>(student)



Fetching Elements from std::tuple

std::get<*string*>(student)

0	First Name `string`	1 Last Name `string`
2	GPA `double`	3 Grade `char`

Fetching Elements from std::tuple

std::get<*string*>(student)

0	1	2	3
First Name	Last Name	GPA	Grade
string	string	double	char

Summary

make_unique and unique_ptr

Standard-defined literals

Fetching tuple elements by type