# Move Semantics; String Views

ITP 435
Week 1, Lecture 2

# Lvalue

- An ***lvalue*** is a variable or object that persists beyond an expression

- An easy way to determine if something is an lvaue is whether or not you can take the address of it…

```
int a = 5;
&a; // valid, because a is an lvalue
&(a + 1); // invalid, because (a + 1) is not an lvalue

int* p1 = &a;
&p1; // valid, because p1 is an lvalue
&(++p1); // valid, because preincrement modifies lvalue
&(p1++); // invalid, because p1++ creates copy that's not an lvalue
```

- Error message is: error C2102: '&' requires l-value

# Lvalues, cont'd

- Lvalues also can appear on the left side of an assignment, like:

```
int i, j, *p;

// Correct usage: the variable i is an lvalue.
i = 7;

// Incorrect usage: The left operand must be an lvalue (C2106).
7 = i; // C2106
j * 4 = 7; // C2106

// Correct usage: the dereferenced pointer is an lvalue.
*p = i;

// Correct usage: the conditional operator returns an lvalue.
((i < 3) ? i : j) = 7;
```

# References and lvalues

- A normal reference (herein called an ***lvalue reference***) can only refer to an lvalue

- That's because references work based on memory addresses!

```
int a = 5;
int& ref = a; // valid, because a is an lvalue
int& ref2 = (a + 1); // invalid -- (a + 1) is not an lvalue
```

- Error: C2440: 'initializing' : cannot convert from 'int' to 'int &'

# Rvalue

- The opposite of an lvalue is an ***rvalue***
- Any hidden variables that are created as a result of expressions or function calls are rvalues:

```
int a = 5;
&a; // a is an lvalue
&(a + 1); // invalid, because (a + 1) is an rvalue

int* p1 = &a;
&p1; // valid, because p1 is an lvalue
&(++p1); // valid, because preincrement modifies lvalue
&(p1++); // invalid, because p1++ creates an rvalue copy
```

# More Examples

```
// lvalues:
int i = 42;
i = 43; // ok, i is an lvalue
int* p = &i; // ok, i is an lvalue
int& foo();
foo() = 42; // ok, foo() is an lvalue
int* p1 = &foo(); // ok, foo() is an lvalue

// rvalues:
int foobar();
int j = 0;
j = foobar(); // ok, foobar() is an rvalue
int* p2 = &foobar(); // error, cannot take the address of an rvalue
j = 42; // ok, 42 is an rvalue
```

# Simple String Class

```
class string
{
   char* mData;
   size_t mSize;
   size_t mCapacity;
public:
   // Assume we have regular constructor, destructor, and
   // operator+ defined also
   string(const string& rhs) // copy constructor
   {
      mSize = rhs.mSize;
      mCapacity = rhs.mCapacity;
      mData = new char[mCapacity];
      for (size_t i = 0; i < mSize + 1; i++)
      {
         mData[i] = rhs.mData[i];
      }
   }
};
```

# Unnecessary Copying

- Suppose you have the following strings:

```
string a("1234");
string b("5678");
```

- What happens when you do this?

```
string c(a + b);
```

1. operator+ will construct a new string (“12345678”) which it returns by *value*
2. c will then call the copy constructor which will allocate more memory and then copies “12345678” into it

This wastes time and memory!!!

The string in step 1 is an rvalue, so we should steal it’s m_str

# Move Constructor

- Relies on && which is an ***rvalue reference***

```
// Add this "move constructor" to string class
// Steal the rvalue's data!!!
string(string&& rvalue)
{
   mData = rvalue.mData;
   mSize = rvalue.mSize;
   mCapacity = rvalue.mCapacity;

   // Null so rvalue destructor won't delete the data
   rvalue.mData = nullptr;
   // Optional, but good to do
   rvalue.mSize = 0;
   rvalue.mCapacity = 0;
}
```

# Try this again

- Suppose you have the following strings:

```
string a("1234");
string b("5678");
```

- Now that we have a move constructor, what happens here?

```
string c(a + b);
```

1. operator+ will construct a new string ("12345678") which it returns by *value*
2. c will then call the move constructor, which will steal the data from the (a + b) rvalue

- Success (in this case)!!
- ***But more generally there is one other thing to do for moves…***

# Another Example…

```cpp
struct Test {
    // Default constructor
    Test() {
        std::cout << "Default" << std::endl;
        mValue = 0;
    }
    // Copy constructor
    Test(const Test& rhs) {
        std::cout << "Copy" << std::endl;
        mName = rhs.mName;
        mValue = rhs.mValue;
    }
    // Move constructor
    Test(Test&& rhs) {
        std::cout << "Move" << std::endl;
        mName = rhs.mName;
        mValue = rhs.mValue;
    }

    std::string mName;
    int mValue;
};
```

# Then I use it...

- Then if I have this:

```
Test doStuff() {
    Test temp;
    temp.mName = "Hello World!";
    return temp;
}

int main() {
    Test a(doStuff());
    std::cout << a.mName << std::endl;
    return 0;
}
```

# Output

- The output* is:

```
Default
Move
Hello World!
```

**With this specific configuration in clang:*

```
-std=c++20 -fno-elide-constructors
```

# The move

```
Test doStuff() {
    Test temp;
    temp.mName = "Hello World!";
    return temp;
}

int main() {
    Test a(doStuff());
    std::cout << a.mName << std::endl;
    return 0;
}
```

This object is moved to a

# Let's add another Test constructor

```
Test(const std::string& name)
{
    std::cout << "Default" << std::endl;
    mValue = 0;
    mName = name;
}
```

# A different doStuff

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test a(doStuff());
    std::cout << a.mName << std::endl;
    return 0;
}
```

# Output

- The output* is:

```
Default
Hello World!
```

**With this specific configuration in clang:*

```
-std=c++20 -fno-elide-constructors
```

# A different doStuff

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test a(doStuff());
    std::cout << a.mName << std::endl;
    return 0;
}
```

Because this is an "unnamed" return value object, it's guaranteed to have its constructor elided in C++17 or later.

# A different doStuff

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test a(doStuff());
    std::cout << a.mName << std::endl;
    return 0;
}
```

That unnamed Test object is constructed directly in the memory of a.

# Constructor Elision in C++17 and beyond

- There are some constructors the compiler is REQUIRED to ***elide*** (remove) in certain conditions

- There are some constructors the compiler has the *option* to elide, but it's not guaranteed

- With clang, if you use the **`-fno-elide-constructors`** flag, it will always choose "no" on the optional elision but it will still do the required elision

# Guaranteed Copy/Move Elision in C++17

```
T f() {
   return T{}; // no copy/move here (C++17)
}

T x = f();    // no copy/move here either (C++17)

T g() {
   T t;
   return t;   // one copy/move, can be elided but elision is not guaranteed
}

T y = g();    // no copy/move here (C++17)

T h(T &t) {
   return t;   // one guaranteed copy (by necessity)
}

T z = h(x);   // no copy/move here (C++17)
```

# What’s wrong with this move constructor?

```
// Move constructor
Test(Test&& rhs) {
    std::cout << "Move" << std::endl;
    mName = rhs.mName;
    mValue = rhs.mValue;
}
```

What happens with the string member?

# What’s wrong with this move constructor?

```
// Move constructor
Test(Test&& rhs) {
    std::cout << "Move" << std::endl;
    mName = rhs.mName;
    mValue = rhs.mValue;
}
```

First mName will be constructed w/ default constructor…

Then we call the assignment operator…

# First, we should use object initializer syntax!

```
struct Test {
    // Default Constructor
    Test()
        : mValue(0)
    { std::cout << "Default" << std::endl; }

    // Copy constructor
    Test(const Test& rhs)
        : mName(rhs.mName)
        , mValue(rhs.mValue)
    { std::cout << "Copy" << std::endl; }

    // Move constructor
    Test(Test&& rhs)
        : mName(rhs.mName)
        , mValue(rhs.mValue)
    { std::cout << "Move" << std::endl; }

    std::string mName;
    int mValue;
};
```

# Initializer

- Now what happens?

```
// Move constructor
Test(Test&& rhs)
    : mName(rhs.mName)
    , mValue(rhs.mValue)
{
    std::cout << "Move" << std::endl;
}
```

mName is constructor with rhs.mName, which calls which constructor?

# std::move

- Defined in `<utility>`
- std::move casts from an lvalue to an rvalue reference:

```
Test(Test&& rhs)
: mName(std::move(rhs.mName))
, mValue(std::move(rhs.mValue))
{
   std::cout << "Move" << std::endl;
}
```

- This way, mName will be constructed with a move constructor, if one exists for std::string (which it does!)

# xvalue (Not important, very pedantic)

- When we do a std::move we actually get an xvalue.

- From the C++ standard:

"An ***xvalue*** is an expression that identifies an "eXpiring" object, that is, the object that may be moved from. The object identified by an xvalue expression may be a nameless temporary, it may be a named object in scope, or any other kind of object, but if used as a function argument, xvalue will always bind to the rvalue reference overload if available"

- You can think of an xvalue as a subset of rvalue (though technically it isn't)

# The “Rule of three”

The “Rule of three” in C++ states that if you are compelled to implement any of the following three class members:

- Destructor
- Copy Constructor
- Assignment Operator

…you should most likely implement all three of them – otherwise bad things can happen!

# The “Rule of ~~three~~ five”

The “Rule of ~~three~~ five” in C++ states that if you are compelled to implement any of the following three class members:

- Destructor
- Copy Constructor
- Assignment Operator
- **Move Constructor**
- **Move Assignment Operator**

…you should most likely implement all five of them – otherwise bad things can happen!

# The "Rule of zero"

There's also the "rule of zero" which says that in modern C++, you shouldn't have to overload any of the five member functions!

This can only be the case if you avoid using `new` altogether and instead use:

- STL collections
- Smart pointers

We'll get here eventually

# Destructor, Assignment and Move Assignment

```cpp
// Destructor
~Test() { std::cout << "Destructor" << std::endl; }
// Assignment
Test& operator=(const Test& rhs) {
   std::cout << "Assignment" << std::endl;
   if (&rhs != this) {
      mName = rhs.mName;
      mValue = rhs.mValue;
   }
   return *this;
}
// Move Assignment
Test& operator=(Test&& rhs) {
   std::cout << "Move Assignment" << std::endl;
   if (&rhs != this) {
      mName = std::move(rhs.mName);
      mValue = std::move(rhs.mValue);
   }
   return *this;
}
```

# Another Test…

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

# Another Test...

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

| Output |
|---|
| 1. Default (b) |

# Another Test…

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

| Output |
| --- |
| 1. Default (b) |
| 2. Default (temporary) |

# Another Test…

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

| Output |
|---|
| 1. Default (b) |
| 2. Default (temp) |
| 3. Move Assignment (temporary to b) |

# Another Test...

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

| Output |
|---|
| 1. Default (b) |
| 2. Default (temp) |
| 3. Move Assignment (temporary to b) |
| 4. Destructor (temp) |

# Another Test...

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

| Output |
|---|
| 1. Default (b) |
| 2. Default (temp) |
| 3. Move Assignment (temporary to b) |
| 4. Destructor (temp) |
| 5. Hello World! (b.Name) |

# Another Test...

```
Test doStuff() {
    return Test("Hello World!");
}

int main() {
    Test b;
    b.mName = "Goodbye!";
    b = doStuff();
    std::cout << b.mName << std::endl;
    return 0;
}
```

| Output |
|---|
| 1. Default (b) |
| 2. Default (temp) |
| 3. Move Assignment (temporary to b) |
| 4. Destructor (temp) |
| 5. Hello World! (b.Name) |
| 6. Destructor (b) |

# In-class Week 1 Lecture 2

- Given the following declarations:

```
struct S {
   S() { printf("Default\n"); }
   S(const S& other) { printf("Copy\n"); }
   S(S&& other) { printf("Move\n"); }
   S& operator=(const S& other) { printf("Assignment\n"); return *this; }
   S& operator=(S&& other) { printf("Move Assignment\n"); return *this; }
   ~S() { printf("Destructor\n"); }
};
```

- What’s the output of (assuming C++17 and no elide):

```
const S& f1(const S& s) {
   return s;
}

S f2(const S& s) {
   S x(s); return x;
}

int main() {
   S a;
   S b = a;
   S c = f2(a);
   const S& d = f1(c);
   return 0;
}
```

# In-class Week 1 Lecture 2

- Given the following declarations:

```
struct S {
   S() { printf("Default\n"); }
   S(const S& other) { printf("Copy\n"); }
   S(S&& other) { printf("Move\n"); }
   S& operator=(const S& other) { printf("Assignment\n"); return *this; }
   S& operator=(S&& other) { printf("Move Assignment\n"); return *this; }
   ~S() { printf("Destructor\n"); }
};
```

- What's the output of (assuming C++17 and no elide):

```
const S& f1(const S& s) {
   return s;
}

S f2(const S& s) {
   S x(s); return x;
}

int main() {
   S a;
   S b = a;
   S c = f2(a);
   const S& d = f1(c);
   return 0;
}
```

Default (a)
Copy (a to b)
Copy (s to x)
Move (x to c)
Destructor (x)
Destructor (c)
Destructor (b)
Destructor (a)

# A quote from Effective Modern C++ (Item #29)

"Move semantics is arguably *the* premier feature of C++11. 'Moving containers is now as cheap as copying pointers!' you're likely to hear, and 'Copying temporary objects is now so efficient, coding to avoid it is tantamount to premature optimization!' Such sentiments are easy to understand. Move semantics is truly an important feature. It doesn't just allow compilers to replace expensive copy operations with comparatively cheap moves, it actually *requires* that they do so (when the proper conditions are fulfilled). Take your C++98 code base, recompile with a C++11-conformant compiler and Standard Library, and—*shazam!*—your software runs faster."

# More From Item #29…

"There are thus several scenarios in which C++11's move semantics do you no good:

- **No move operations:** The object to be moved from fails to offer move operations. The move request therefore becomes a copy request.
- **Move not faster:** The object to be moved from has move operations that are no faster than its copy operations.
- **Move not usable:** The context in which the moving would take place requires a move operation that emits no exceptions, but that operation isn't declared noexcept.

It's worth mentioning, too, another scenario where move semantics offers no efficiency gain:

- **Source object is lvalue:** With very few exceptions (see e.g., Item 25) only rvalues may be used as the source of a move operation."

# std::string_view

# std::string_view

```
int count_matches_string(const std::string& str, char ch) {
  int count = 0;
  for (auto c : str) {
    if (c == ch) {
      count++;
    }
  }
  return count;
}
```

```
int count_matches_string_view(std::string_view str, char ch) {
  int count = 0;
  for (auto c : str) {
    if (c == ch) {
      count++;
    }
  }
  return count;
}
```

**Use string_view**

# string_view is not a copy!

```
char c[] = "This is a test.";
```

This is a test.

Copy

Heap Allocation

```
std::string s = c;
```

This is a test.

~~Copy~~
Reference

```
std::string_view sv = c;
```

Pointer

Size

# Can reduce unneeded copies

```
int count_matches_string(const std::string& str, char ch);
```

```
std::string s = "This is a test.";
r = count_matches_string(s, 't');
```

```
char c[] = "This is a test.";
r = count_matches_string(c, 't');
```

**Extra copy here**

```
int count_matches_string_view(std::string_view str, char ch);
```

```
std::string s = "This is a test.";
r = count_matches_string_view(s, 't');
```

```
char c[] = "This is a test.";
r = count_matches_string_view(c, 't');
```

**We are saving here!**

# Why's it called string_view?

```
char c[] = "This is a test.";
```

```
std::string s = c;
```

This is a test.

Copy

This is a test.

Reference

```
std::string_view sv = c;
```

Pointer

Size

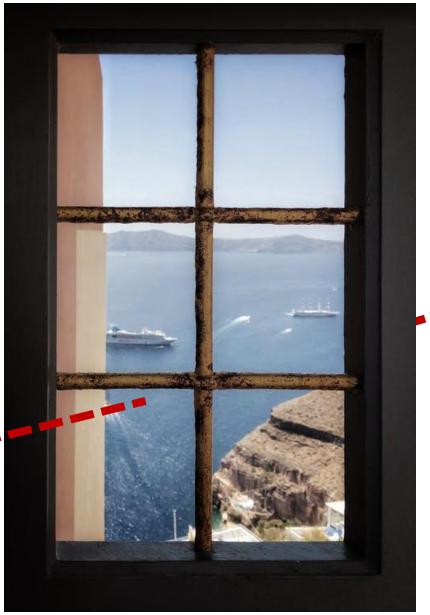
A view to the original

👉 string_view is just a view of a string (like a pointer)

# Why's it called string_view

- You're "looking" at the string

```
std::string_view sv = c;
```



A view to the original

```
char c[] = "This is a test.";
```

Pointer
Size

# Modifying the Original

- You can’t modify the string_view, but if you modify the original, what the view sees updates

```
int main() {
{
  char c[] = "This is a test.";
  std::string s = c;        <-------- Causes copy and heap allocation
  std::string_view sv = c;  <--- No heap allocation

  // Modify original
  c[0] = '$';

  std::cout << "c: " << c << std::endl;
  std::cout << "s: " << s << std::endl;
  std::cout << "sv: " << sv << std::endl;
}
```

```
Original          c: $his is a test.
Copy              s: This is a test.
View to Original  sv: $his is a test.
```

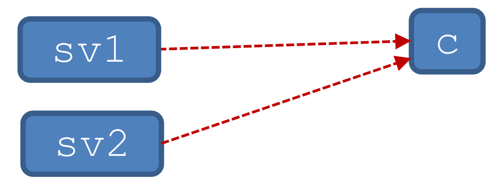
# Modifying the Original

```
char c[] = "This is a test.";
std::string s = c;
std::string_view sv1 = c;
// Copy the view.
std::string_view sv2 = sv1;

// Modify original.
c[0] = '$';

std::cout << "c: " << c << std::endl;
std::cout << "s: " << s << std::endl;
std::cout << "sv1: " << sv1 << std::endl;

std::cout << "sv2: " << sv2 << std::endl;
```

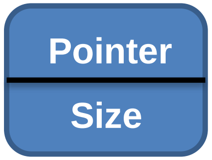



👉 All views change if original changes

👉 Copying the view is **shallow**

| | |
|---|---|
| Original | `c: $his is a test.` |
| Copy | `s: This is a test.` |
| View to original | `sv1: $his is a test.` |
| Copy of view to original | `sv2: $his is a test.` |

# How does it know the size?




```
char c1[] = "This is a test.";
std::string_view sv1{c1};

char c2[]{'t', 'e', 's', 't', 's'};
std::string_view sv2{c2, 5};

std::cout << "sv1: " << sv1 << std::endl;
std::cout << "sv2: " << sv2 << std::endl;
```

Null terminated, doesn't need the size, but will calculate size in O(n) time

Not null terminated, needs the size, but is O(1) since it doesn't calculate it

```
sv1: This is a test.
sv2: tests
```

# Let's Benchmark It

```
char c[] = "This is a test.";

last(c);
last_string_view(c);

auto c_size = strlen(c);
last_string_view({c, c_size});
```

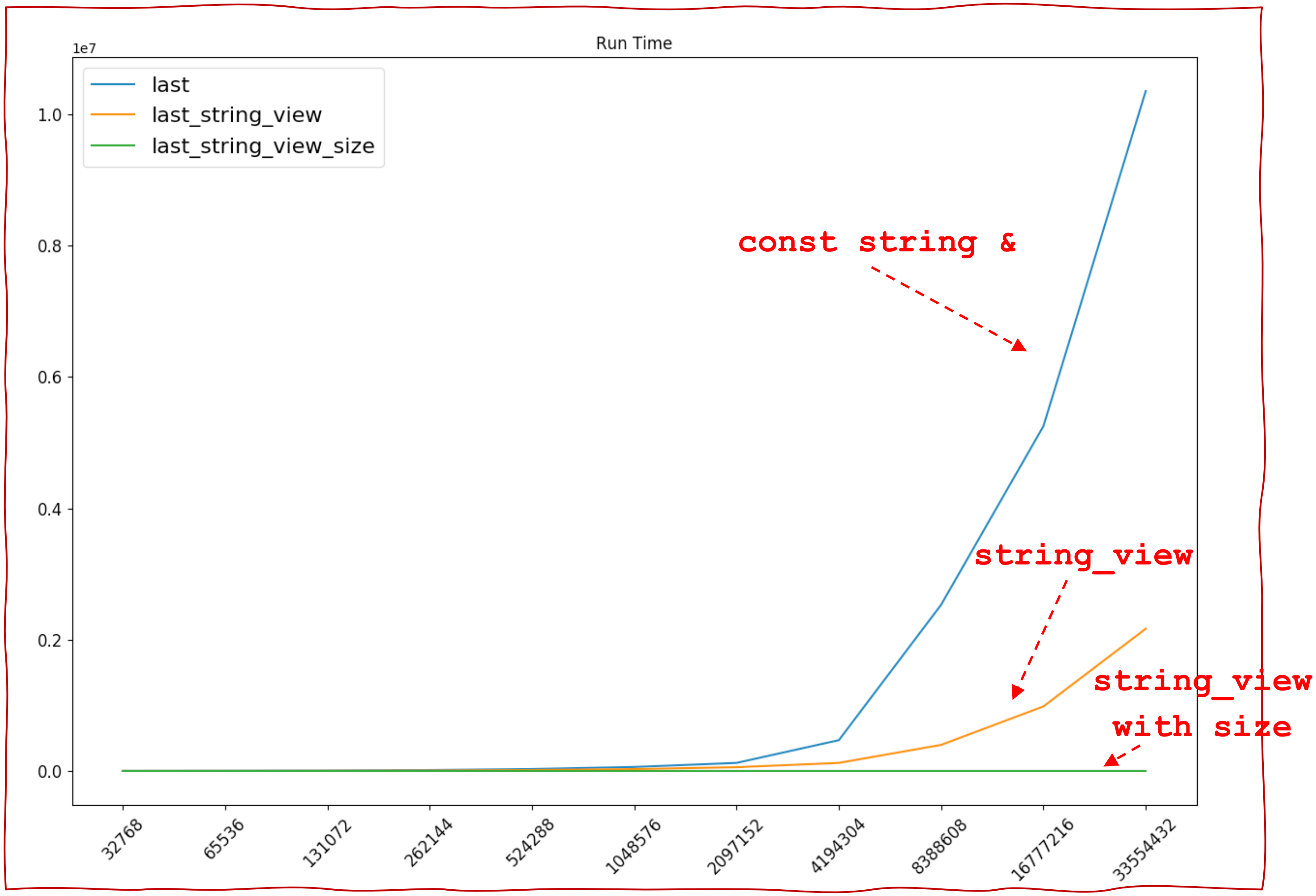
```
char last(const std::string& str) {
  if (!str.empty()) {
    return str[str.size() - 1];
  } else {
    return ' ';
  }
}
```

```
char last_string_view(std::string_view sv) {
  if (!sv.empty()) {
    return sv[sv.size() - 1];
  } else {
    return ' ';
  }
}
```

**Extra copy here**

**No extra copy here**

# Benchmark



Run Time
1e7
last
last_string_view
last_string_view_size
1.0
0.8
0.6
0.4
0.2
0.0
const string &
string_view
string_view
with size
32768
65536
131072
262144
524288
1048576
2097152
4194304
8388608
16777216
33554432

```
char c[] = "This is a test.";

last(c);
last_string_view(c);

auto c_size = strlen(c);
last_string_view({c, c_size});
```

```
char last(const std::string& str) {
  if (!str.empty()) {
    return str[str.size() - 1];
  } else {
    return ' ';
  }
}
```

```
char last_string_view(std::string_view sv) {
  if (!sv.empty()) {
    return sv[sv.size() - 1];
  } else {
    return ' ';
  }
}
```

**Extra copy here**

**No extra copy here**

# Dangling Views

```
std::string_view sv;

char *c2, *c3;
c2 = new char[2];
c2[0] = 'a';
c2[1] = '\0';
sv = c2;
std::cout << "sv: " << sv << std::endl;
delete c2;

c3 = new char[2];
c3[0] = '?';
c3[1] = '\0';
std::cout << "c3: " << c3 << std::endl;

std::cout << "sv: " << sv << std::endl;
```


**sv becomes a dangling view -- Undefined behavior!!**

```
sv: a
c3: ?
sv: ?
```

👉 Make sure the original will outlive the view!!

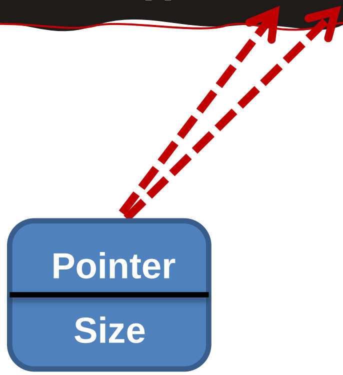
# Modifying the View

```
char c[] = "This is a test.";
std::string_view sv = c;

// Remove the first 2 characters from the view.
sv.remove_prefix(2);
std::cout << "sv: " << sv << std::endl;

// Remove the last 2 characters from the view.
sv.remove_suffix(2);
std::cout << "sv: " << sv << std::endl;
std::cout << "c: " << c << std::endl;
```

```
char c[] = "This is a test.";
```




```
sv: is is a test.
sv: is is a tes
c: This is a test.
```

👉 Modifying the view doesn't change the original

# Converting to a C-Style String

```
char c[] = "This is a test.";
```

```
std::string s{c};
std::cout << "std::strlen(s): " <<
  std::strlen(s.c_str()) << std::endl;
```

```
std::string_view sv{c};
std::cout << "std::strlen(sv.data()): " <<
  std::strlen(sv.data()) << std::endl;
```

**`std::string: c_str()`**

**`std::string_view: data()`**

👉 data() is valid only if:
- The view hasn't been modified.
- The original is null terminated.

# Converting to a C-Style String

```
char c[] = "This is a test.";
std::string_view sv{c};
c[std::strlen(c)] = 'a';
std::cout << "sv: " << sv << std::endl;
std::cout << "sv.data(): " << sv.data() << std::endl;
```

```
sv: This is a test.
sv.data(): This is a test.a�T��
```

```
std::string_view sv{"This is a test."};
sv.remove_prefix(5);
sv.remove_suffix(5);

std::cout << "sv: " << sv << std::endl;
std::cout << "sv.data(): " << sv.data() << std::endl;
```

```
sv: is a
sv.data(): is a test.
```

👉 data() is valid only if:

- The view hasn't been modified.
- The original is null terminated.

# Other string_view methods

| Element access |
|---|
| operator[] |
| at |
| front |
| back |
| data |
| **Capacity** |
| size<br>length |
| max_size |
| empty |

| Operations |
|---|
| copy |
| substr |
| compare |
| starts_with (C++20) |
| ends_with (C++20) |
| contains (C++23) |
| find |
| rfind |
| find_first_of |
| find_last_of |
| find_first_not_of |
| find_last_not_of |

| Modifiers |
|---|
| remove_prefix |
| remove_suffix |
| swap |

```
auto s1{std::string_view{"aaaa"}};
auto s2{std::string_view{"bbbb"}};

std::cout << "s1, s2 Before: " << s1 << ", " << s2 << "\n";
s1.swap(s2);
std::cout << "s1, s2 After: " << s1 << ", " << s2 << "\n";
```

```
s1, s2 Before : aaaa, bbbb
s1, s2 After  : bbbb, aaaa
```

# Summary

- Use **`string_view`** instead of **`const string &`**.
- The view ***does not own*** the data.
- The original ***must outlive*** the view.
- Pass **`string_view`** by value. No need for **`const`** or **`&`**.
- Include **`<string_view>`**

```
char last_string_view(std::string_view sv) {
  if (!sv.empty()) {
    return sv[sv.size() - 1];
  } else {
    return ' ';
  }
}
```