Systems Programming (& Safety)

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka
Andy Jiang
Charis Gao
Daniel Da
Emily Shen
Haoyun Qin
Jeff Yang
Jerry Wang
Jinghao Zhang
Julius Snipes
Kyrie Dowling
Oliver Hendrych
Maxi Liu
Rohan Verma
Ryan Boyle
Ryoma Harris
Shyam Mehta
Tom Holland
Tina Kokoshvili
Zhiyan Lu

Poll Everywhere

pollev.com/tqm

- How is PennOS going?

Administrivia

- PennOS
  - Everyone should have already contacted their group, and should get started working on it.
  - Full Thing due ~April 22nd (Monday)
    - Can still use late tokens, so late deadline is April 26th
    - After you submit, you need to schedule a meeting with your TA to demonstrate that it is working
  - I am told some people are splitting the kenrnel into shell vs non-shell.
    - This is usually a terrible Idea. The Shell depends a lot on the Kernel and know how the kernel works will help A LOT. Shell can't be tested much until Kernel is implemented.

Administrivia

- Check-in released: due before lecture Thursday next week
  - Don't forget to do it!

- We released stress.c and stress.h for testing your PennOS kernel
  - Note: there was originally an error when first released. It should be calling the linux system call usleep in the provided code and NOT s_sleep

- CIS TA Application is out now!
  - Intro courses are due Tomorrow night @ midnight
  - 2400 is "due" April 26th @ midnight

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- Systems Programming
- C & C++
- Intro to C++
  - std::string & iostreams
  - std::vector
  - References
  - std::optional
- Safety
- What’s Next?

Poll Everywhere

pollev.com/tqm

- On a scale of 1 (hate) to 5 (love), how do you feel about C as a programming language?

Poll Everywhere

pollev.com/tqm

- Why do you think we chose C as the programming language for this course?

Poll Everywhere

pollev.com/tqm

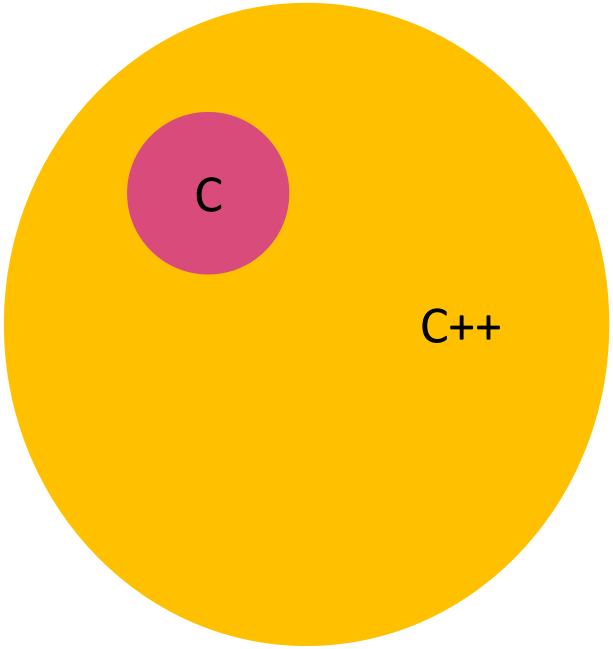
- Why do you think we chose C as the programming language for this course?

- What comes to my mind:
  - C is fast
  - C exposes you to the low-level features that other languages abstract away. (Even if we did not use them all)
    - addresses
    - Memory management
    - System Calls
    - Assembly
  - Operating System Kernels and Systems have been written in C for a long time. In some ways it would be blasphemous to choose something like python

C/C++?

- Common way of listing the languages: C/C++

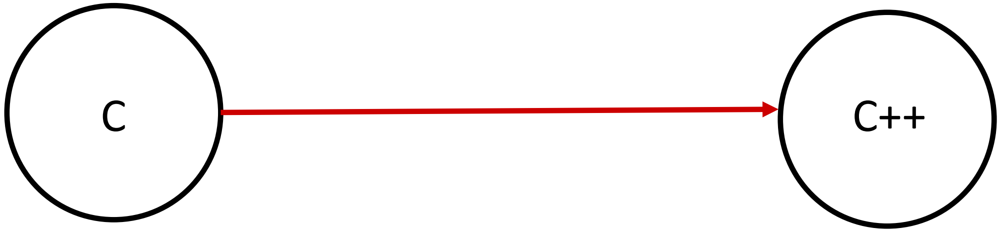
- Common understanding of the language
  - C++ is C but more
  - C++ is a super set of C



- This understanding is a pet-peeve of mine

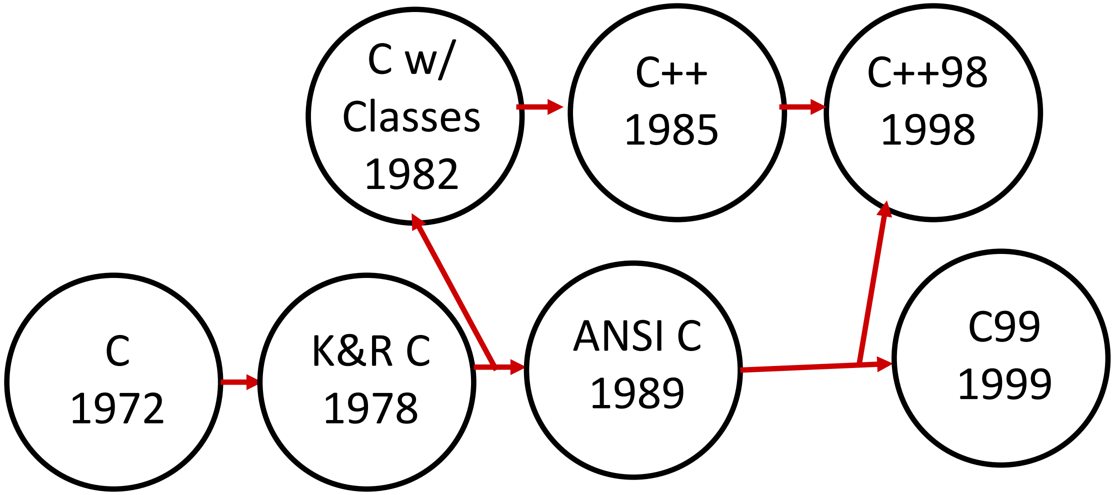
C vs C++ (Timeline)

- What People Think



C vs C++ (Timeline)

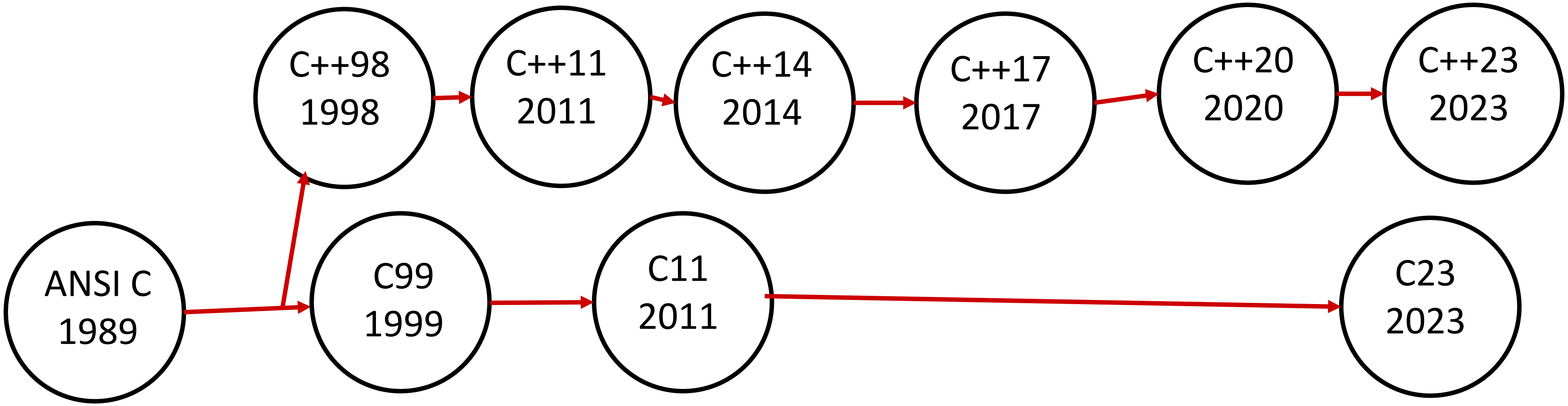
- More Detail (but a lot left out)



THE LANGUAGES "FORK" around 1999
Not all C99 features are legal C++, but most of them are.

C vs C++ (Timeline)

- More Detail (but a lot left out)



THE LANGUAGES "FORK" around 1999
Not all C99 features are legal C++, but most of them are.

C has adopted changes from C++
example: `auto` and `nullptr` in C23

C vs C++ Examples

- old_c.c
  - C has evolved since it was introduced in 1972
- c23.c
  - C still gets updates adding new features
  - Admittedly, the updates are small relative to other language updates
- cpp23.cpp and stdin_echo.cpp
  - Modern C++ is very different from C (Though most C is still legal!)
- cpp23_hello.cpp
  - The fundamentals of the language are changing as well

Hello World in C++

helloworld.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS

using namespace std;

int main() {
  cout << "Hello, World!" << endl;
  return EXIT_SUCCESS;
}
```

- Looks simple enough…
  - Compilation command if you want to compile this yourself:

```
g++-12 -Wall -g -std=c++23 -o helloworld helloworld.cpp
```

  - Let’s walk through the program step-by-step to highlight some differences

Poll Everywhere

pollev.com/tqm

- What does this code print?

```
#include <iostream>
#include <cstdlib>

using namespace std;

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  cout << num << endl;
  return EXIT_SUCCESS;
}
```

Let’s do a slightly more complex program

Greeting C++

greeting.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for string

using namespace std;

int main() {
  string expected {"Travis"};
  // ...
}
```

- `string` is part of the ***C++*** standard library
  - We still have to #include it
    - No more char* !
  - When we initialize any variable in C++, we use the {}

Greeting C++

greeting.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;
  // ...
}
```

Greeting C++

greeting.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;

  string input {};
  // ...
}
```

- Declares an empty string ("")
  - You ~~should~~ **must** always initialize a variable with {} even if empty

Greeting C++

greeting.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>     // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;

  string input {};
  cin >> input;

  // ...
}
```

- Reads from stdin (terminal input) into "`input`"
  - This works for reading in numbers too

Greeting C++

greeting.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>     // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;

  string input {};
  cin >> input;

  // ...
}
```

- Reads from stdin (terminal input) into “`input`”
  - This works for reading in numbers too

Greeting C++

greeting.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>     // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;

  string input {};
  cin >> input;

  if (input == expected) {

  }
}
```

- Can use == to compare strings

Greeting C++

greeting.cpp

```
#include <iostream>    // for cout, endl
#include <cstdlib>     // for EXIT_SUCCESS
#include <string>      // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;

  string input {};
  cin >> input;

  if (input == expected) {
    cout << "Hello " << input << "!" << endl;
  }
}
```

- Can chain << repeatedly
  - Would also work for printing a lot of types
    - (including integer and floating point types)

Greeting C++

- Print to `cerr` when there is an error.
- Also known as stderr
- This case is debatably an error.

greeting.cpp

```
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS
#include <string>     // for string

using namespace std;

int main() {
  string expected {"Travis"};
  cout << "Who are you?" << endl;

  string input {};
  cin >> input;

  if (input == expected) {
    cout << "Hello " << input << "!" << endl;
  } else {
    cerr << "Who the hell are you????" << endl;
  }
  return EXIT_SUCCESS;
}
```

Lecture Outline

STL Containers ☺

- A container is an object that stores (in memory) a collection of other objects (elements)
  - Implemented as class templates, so hugely flexible
  - More info in C++ Primer §9.2, 11.2

- Several different classes of container
  - Sequence containers (vector, deque, list, ...)
  - Associative containers (set, map, multiset, multimap, bitset, ...)
  - Differ in algorithmic cost and supported operations

`vector`

C++ equivalent of ArrayList

- A generic, dynamically resizable array
  - https://cplusplus.com/reference/vector/vector/
  - Elements are store in contiguous memory locations
    - Can index into it like an array
    - Random access is O(1) time
  - Adding/removing from the end is cheap (amortized constant time)
  - Inserting/deleting from the middle or start is expensive (linear time)

- Most common member function: **`push_back()`**
  - Adds an element to the end of the vector

Vector example

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

using namespace std;

int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};
  vec.push_back(3);
  vec.push_back(2);
  vec.push_back(1);

  cout << "vec.at(0)" << endl << vec.at(0) << endl;
  cout << "vec.at(1)" << endl << vec.at(1) << endl;

  // iterates through all elements
  for (size_t i = 0U; i < vec.size(); ++i) {
    cout << vec.at(i) << endl;
  }

  return EXIT_SUCCESS;
}
```

Most containers are in a module of the same name

Constructs a vector with three initial elements

Add three integers to the vector

Print all the values in the array

range for loop

- Syntactic sugar similar to Java's foreach

```
for (declaration : expression) {
  statements
}
```

  - `declaration` defines the loop variable
  - `expression` is an object representing a sequence
    - Strings, and most STL containers work with this

```
string str{"hello"};
// prints out each character
for (char c : str) {
  cout << c << endl;
}
```

range for loop vector example

- If you need to iterate over every element in a sequence, you should use a range for loop.
    - Why? It is harder to mess it up that way

```
int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};
  vec.push_back(3);
  vec.push_back(2);
  vec.push_back(1);

  // iterates through all elements
  for (int element : vec) {
     cout << element << endl;
  }

  return EXIT_SUCCESS;
}
```

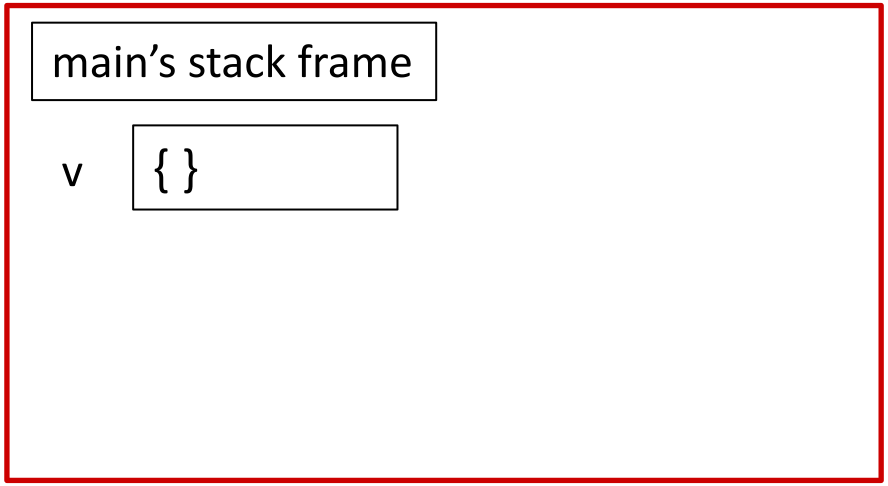
Poll Everywhere

pollev.com/tqm

- What is the final value of v by the end of the main() function?

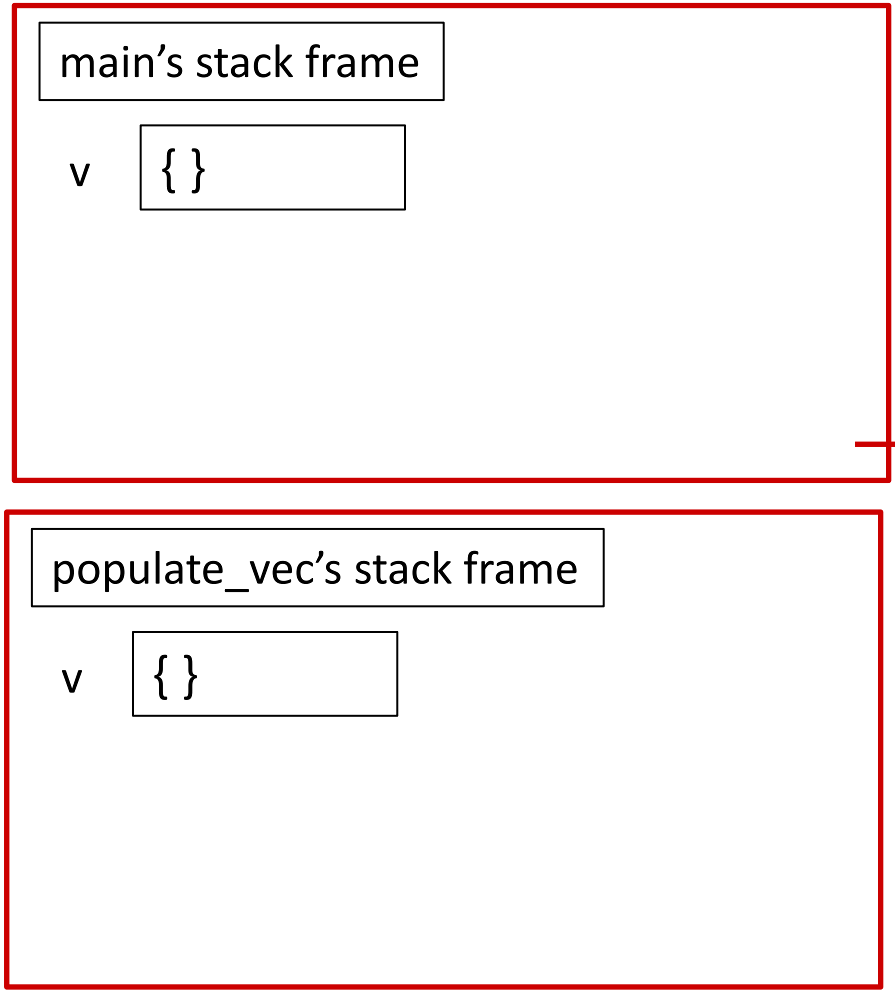
```
#include <vector>
#include <iostream>

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  for (size_t i = 0U; i < v.size(); ++i) {
     cout << v.at(i) << endl;
  }
  return EXIT_SUCCESS;
}
```

Visualization



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

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Visualization



main's stack frame

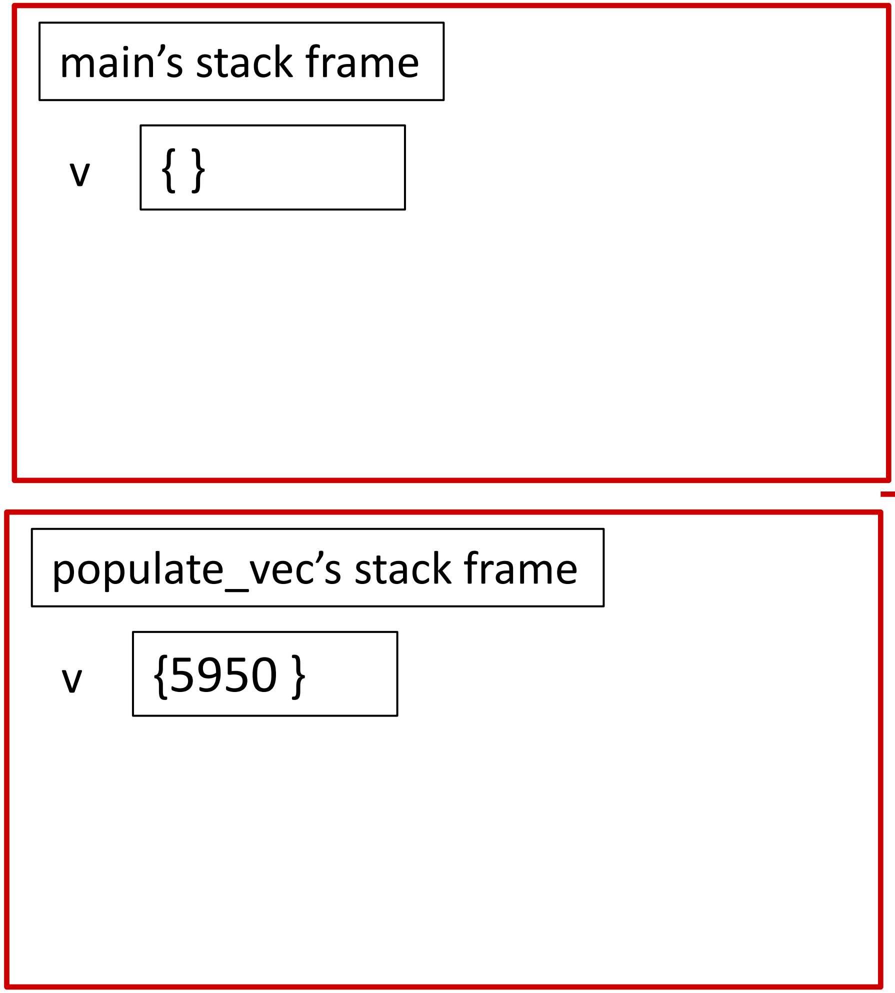
v { }

populate_vec's stack frame

v { }

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

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Visualization



main's stack frame

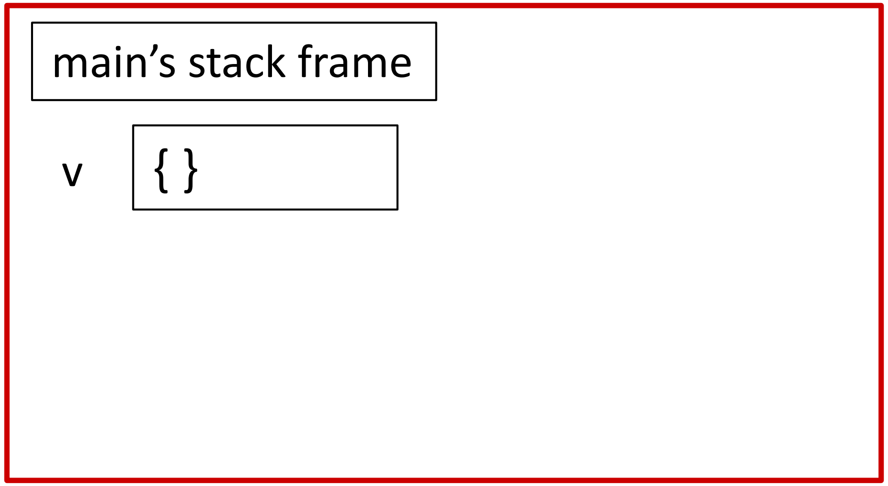
v { }

populate_vec's stack frame

v {5950 }

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

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Visualization



main’s stack frame

v { }

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

using namespace std;

void populate_vec(vector<int> v) {
  v.push_back(5950);
}

int main() {
  vector<int> v {};
  populate_vec(v);
  cout << v.size() << endl;
  // loop removed for space
  return EXIT_SUCCESS;
}
```

Lecture Outline

References

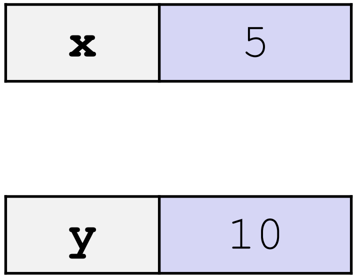
Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;

  z += 1;
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

When we use '&' in a type declaration, it is a reference.

&var still is "address of var"

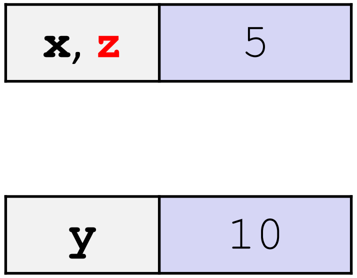


References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```

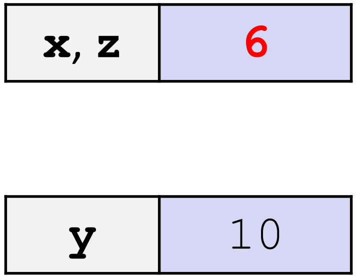


References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;

  z  = y;
  z += 1;

  return EXIT_SUCCESS;
}
```



References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // Normal assignment
  z += 1;

  return EXIT_SUCCESS;
}
```

x, z	7

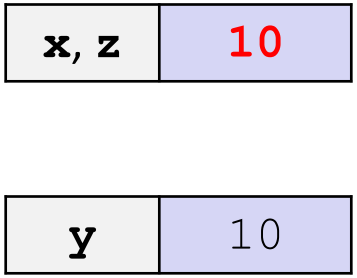
y	10

References

Note: Arrow points to *next* instruction.

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // sets z (and x) to the value of y
  z += 1;

  return EXIT_SUCCESS;
}
```

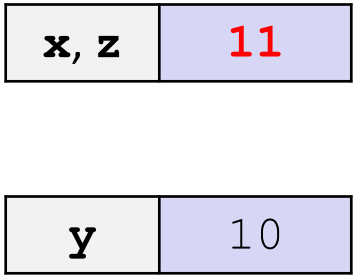


Note: Arrow points to *next* instruction.

References

- A **reference** is an alias for another variable
  - *Alias*: another name that is bound to the aliased variable
    - Mutating a reference ***is*** mutating the aliased variable
  - Introduced in C++ as part of the language

```
int main(int argc, char** argv) {
  int x = 5, y = 10;
  int& z = x;  // binds the name "z" to x

  z += 1;  // sets z (and x) to 6
  x += 1;  // sets x (and z) to 7

  z  = y;  // sets z (and x) to the value of y
  z += 1;  // sets z (and x) to 11

  return EXIT_SUCCESS;
}
```



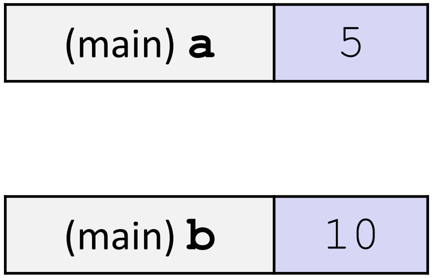
Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller’s argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

Parameters are attached
To variables provided by caller

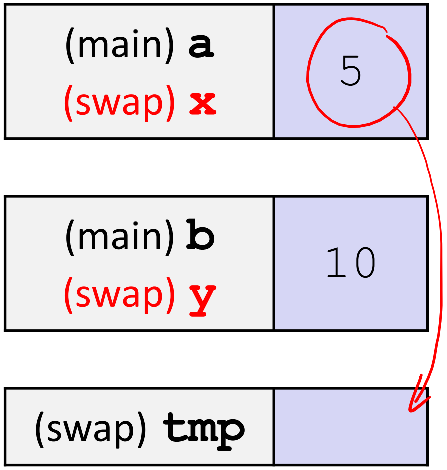


Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```



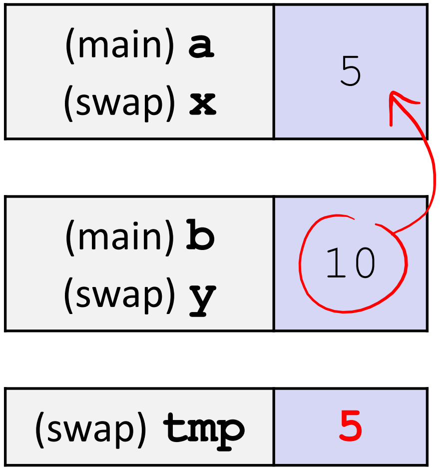
(main) **a** (swap) **x**	5
(main) **b** (swap) **y**	10
(swap) **tmp**	

Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller’s argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

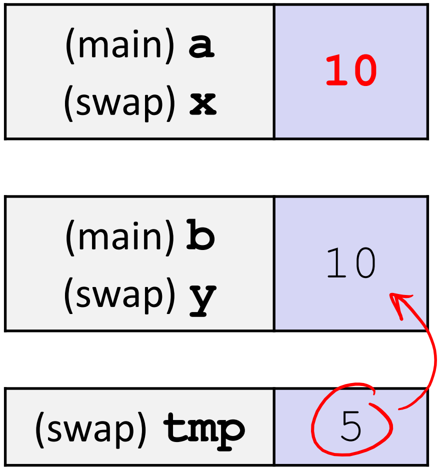


Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```



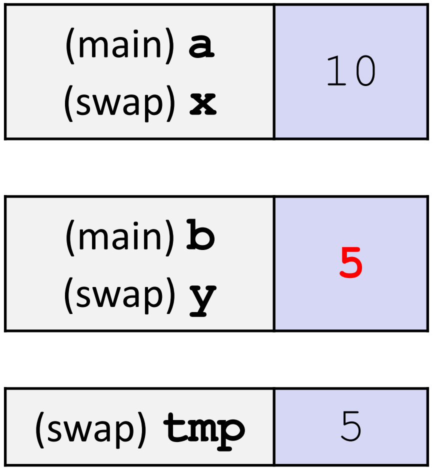
(main) **a** (swap) **x**	**10**
(main) **b** (swap) **y**	10
(swap) **tmp**	5

Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller's argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```

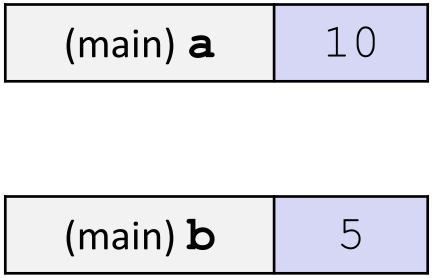


Pass-By-Reference

Note: Arrow points to *next* instruction.

- C++ allows you to use real pass-by-*reference*
  - Client passes in an argument with normal syntax
    - Function uses reference parameters with normal syntax
    - Modifying a reference parameter modifies the caller’s argument!

```
void swap(int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(int argc, char** argv) {
  int a = 5, b = 10;

  swap(a, b);
  cout << "a: " << a << "; b: " << b << endl;
  return EXIT_SUCCESS;
}
```



Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

A. Output "(3,3,3)"

B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

<u>Note</u>: Arrow points to *next* instruction.

A. Output "(3,3,3)"

B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

a, c	1
b	2

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(3,3,3)"

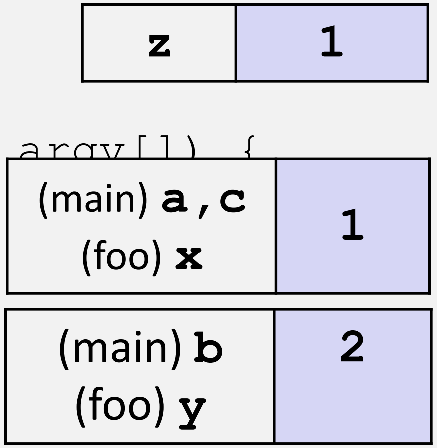
B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

```
void foo(int& x, int& y, int z) {
→ z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



z	1
(main) **a**, **c** (foo) **x**	1
(main) **b** (foo) **y**	2

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(3,3,3)"

B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We’re lost…

```
void foo(int& x, int& y, int z) {
  z = y;
→ x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

z	2
(main) **a,c** (foo) **x**	1
(main) **b** (foo) **y**	2

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(3,3,3)"

B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

z	2

(main) **a**,**c** (foo) **x**	3
(main) **b** (foo) **y**	2

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(3,3,3)"

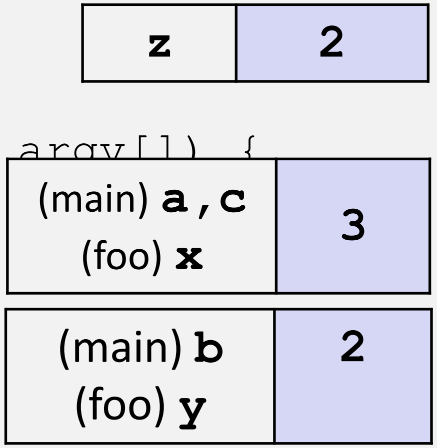
B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```



z	2
(main) **a,c** (foo) **x**	3
(main) **b** (foo) **y**	2

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

<u>Note</u>: Arrow points to *next* instruction.

A. Output "(3,3,3)"

B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We're lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char* argv[]) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, b, c);
  cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

z	2
(main) **a,c** (foo) **x**	3
(main) **b** (foo) **y**	3

Poll Everywhere

pollev.com/tqm

- What will happen when we run this?

Note: Arrow points to *next* instruction.

A. Output "(3,3,3)"

B. Output "(3,3,2)"

C. Compiler error about arguments to foo (in main)

D. Compiler error about body of foo

E. We’re lost…

```
void foo(int& x, int& y, int z) {
  z = y;
  x += 2;
  y = x;
}

int main(int argc, char** argv) {
  int a = 1;
  int b = 2;
  int& c = a;

  foo(a, &b, c);
→ cout << "(" << a << ", " << b
       << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

a,c	3

b	3

Lecture Outline

Functions that sometimes fail

- It is pretty common to write functions that sometimes fail. Sometimes they don't return what is expected
- Consider we were building up a Queue data structure that held strings, that could
  - Add elements to the end of a sequence
    - `void add(string data);`
  - Remove elements from the beginning of a sequence

```
???? remove(????);
```

  - How do we design this function to handle the case where there are no strings in the queue (e.g. it errors?)

Previous ways to handle failing functions

- Return an "invalid" value: e.g. if looking for an index, return -1 if it can’t be found.
  - What if there is no nice "invalid" state?

```
// what is an invalid string?
string remove();
```

- C-style: return an error code or success/failure.
  Real output returned through output param

```
bool remove(string* output);
```

Aside: Java “Object” variables

- Does this java compile?

```
public static String foo() {
   return null;
}
```

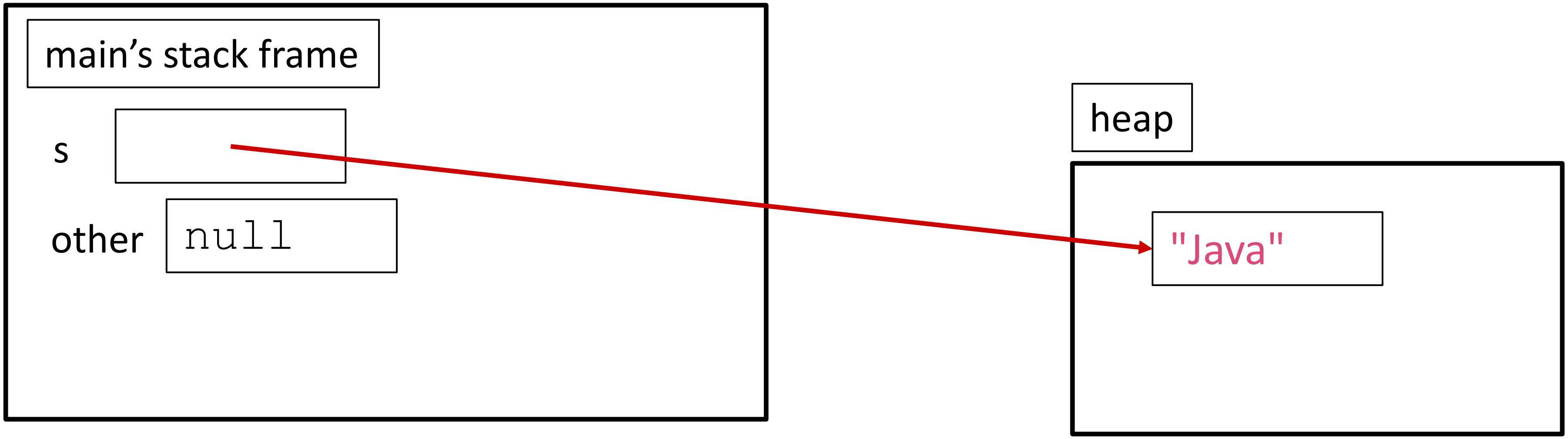
- What about this C++?

```
string foo() {
   return nullptr;
}
```

Aside: Java "Object" variables

- In high level languages (like java), object variables don't actually contain an object, they contain a reference to an object.
  - References in these languages can be null

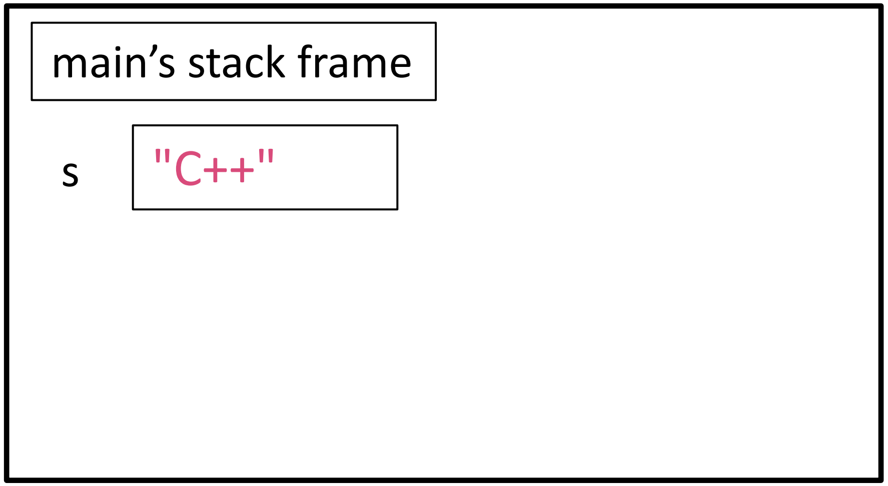
```
String s = new String("Java");
String other = null;
```



Aside: Java "Object" variables

- In C++, a string variable is itself a string object

```
string s{"C++"};

// does not do what you think it does
string other = nullptr;
```



main's stack frame

s "C++"

More on this idea when I talk about pointers later

Previous ways to handle failing functions

- Return a pointer to a heap allocated object, could return `nullptr` on error
  - Uses the heap when it is otherwise unnecessary ☹
  - Need to remember to `delete` the string

```
string* remove();
```

- Java style: throw an exception in the case of an error
  return the value as normal
  - Exceptions not best for performance
  - Exception catching not always the easiest to handle

```
string remove() {
  if (this->size() <= 0U) {
    throw std::out_of_range{"Error!"};
  }
```

std::optional

- **`optional<T>`** is a struct that can either:
  - Have some value T
    (**`optional<string> {"Hello!"}`**)
  - Have nothing
    (**`nullopt`**)

- **`optional<T>`** effectively extends the type **`T`** to have a "null" or "invalid" state

```
optional<string> foo() {
  if (/* some error */) {
    return nullopt;
  }
  return "It worked!";
}
```

Using an optional

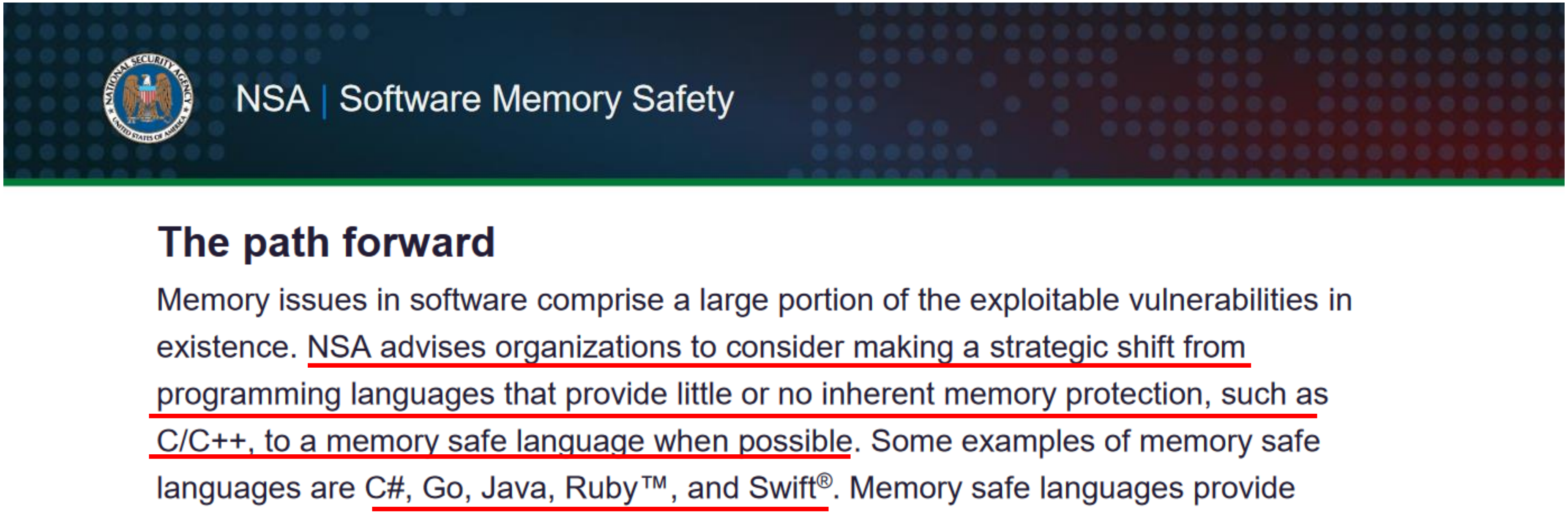
- If we call a function that returns an optional, we need to check to see if it has a value or not

```
optional<string> foo() {
  if (/* some error */) {
    return nullopt;
  }
  return "It worked!";
}

int main() {
  auto opt = foo();
  if (!opt.has_value()) {
    return EXIT_FAILURE;
  }
  string s = opt.value();
}
```

Lecture Outline

What else is going on?

- C++ Seems so cool!!!! What else is going on? ☺
- NSA: 1.5 years ago (Nov 10th, 2022)



NSA | Software Memory Safety

The path forward

Memory issues in software comprise a large portion of the exploitable vulnerabilities in existence. NSA advises organizations to consider making a strategic shift from programming languages that provide little or no inherent memory protection, such as C/C++, to a memory safe language when possible. Some examples of memory safe languages are C#, Go, Java, Ruby™, and Swift®. Memory safe languages provide

Rust is not mentioned in this snippet, but mentioned somewhere else in the announcement

What else is going on?

- C++ Seems so cool!!!! What else is going on? ☺

- White House: 2 **months** ago (Feb 26th, 2024)

FEBRUARY 26, 2024

Press Release: Future Software Should Be Memory Safe

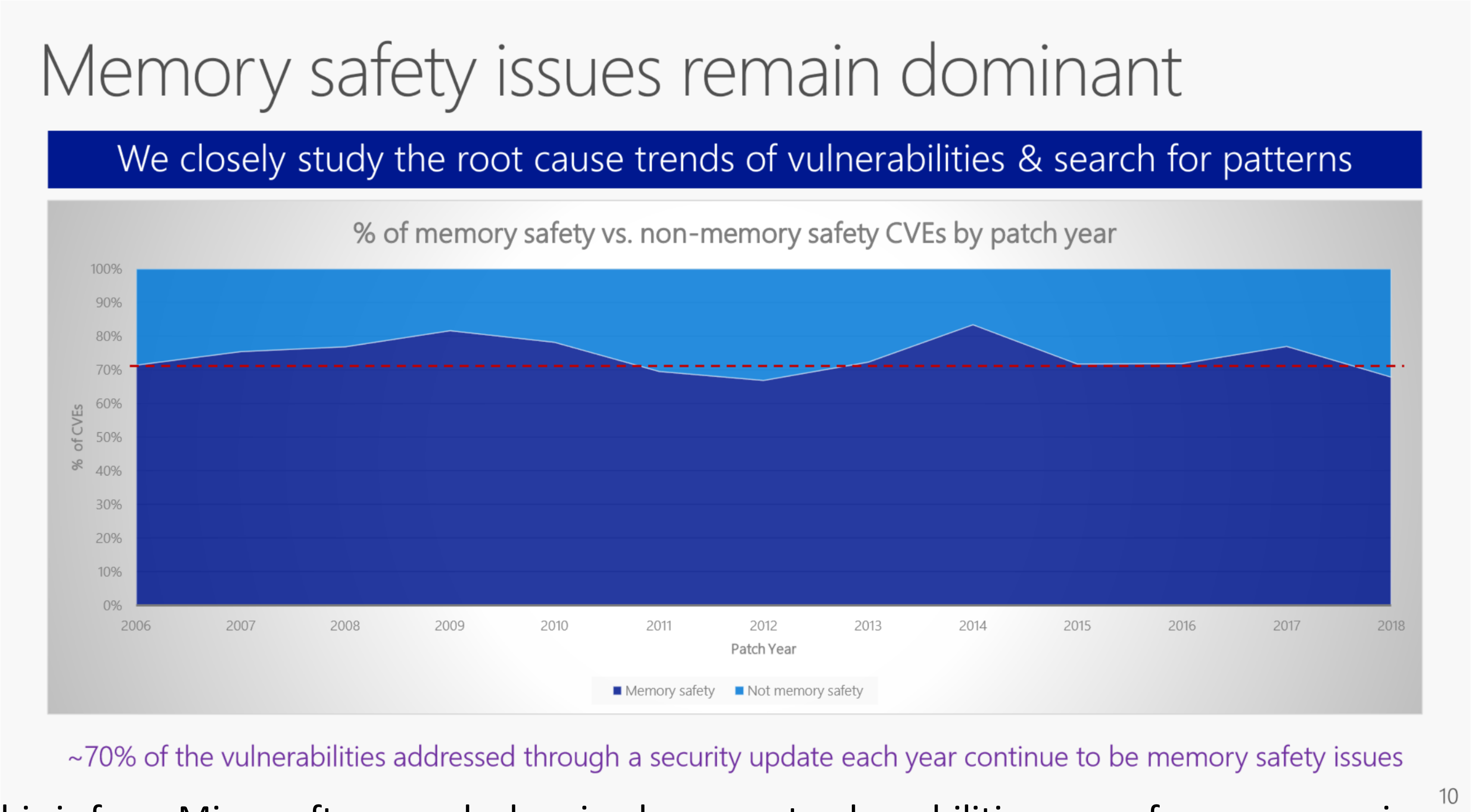
ONCD ▸ BRIEFING ROOM ▸ PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Read the full report here

Memory Safety CVE

- CVE = Common Vulnerabilities and Exposures



Memory safety issues remain dominant

We closely study the root cause trends of vulnerabilities & search for patterns

% of memory safety vs. non-memory safety CVEs by patch year

~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues

10

This is from Microsoft research showing how most vulnerabilities come from memory issues

Memory Safety

- Memory Safety is dominating discussion on Systems programming languages (C, C++, Rust, Zig, Nim, D, …)

- What is memory safety?

- Broadly two types:
  - Temporal Safety: making sure you don't access "objects" that are destroyed, or in invalid states

  - Spatial Safety: making sure you do not access memory you either shouldn't access or accessing them in the wrong ways

Temporal Safety C Example

- Here is an example in C where is the issue?

```c
int main(int argc, char** argv) {
  int* ptr = malloc(sizeof(int));
  assert(ptr != NULL);
  *ptr = 5;

  // do stuff with ptr

  free(ptr);

  printf("%d\n", *ptr);
}
```

Temporal Safety

- Here is an example in C++ where is the issue?

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

using namespace std;


int main(int argc, char** argv) {
  vector<int> v {3, 4, 5};
  int& first = v.front();

  cout << first << endl;

  v.push_back(6);

  cout << v.size() << endl;
  cout << first << endl;
}
```

Temporal Safety

- Here is an example in C++ where is the issue?

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

using namespace std;


void func(vector<int>& v1, vector<int>& v2) {
  v1.push_back(v2.front());
}

int main() {
  vector<int> x{3, 4, 5};
  func(x, x);
}
```

Temporal Safety

- Here is an example in C++ where is the issue?

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

using namespace std;


void func(vector<int>& v1, vector<int>& v2) {
  v1.push_back(v2.front());
}

int main() {
  vector<int> x{3, 4, 5};
  func(x, x);
}
```

push_back takes in an int&

push_back may need to resize, if it does, the reference to its front becomes invalid

Spatial Safety

- C (and C++) enforce types on variables, they are statically typed
- C and C++ can easily get around the type system though:

```
int main() {
  int x = 3;
  float f1 = x;  // converts bits to floating point rep
  float f2 = *(float*)&x; // copies bits

  printf("%f\n", f1); // these two print
  printf("%f\n", f2); // different things
}
```

Spatial Safety

- C (and C++) enforce types on variables, they are statically typed
- C and C++ can easily get around the type system though:

```
int main() {
  string s = "Howdy :)";
  vector<int> v = *retinterpret_cast<vector<int>*>(&s);

  v.push_back(3);

  // this code probably crashes before getting here
}
```

Aside: unions

- A union is a type that can have more than one possible representations in the same memory position

```
union {
  float f;
  int i;
};

f = 3.14;  // assigns a float value to the union

printf("%d\n", i);  // try to interpret the same memory as an int

// this is not type checked ☹
```

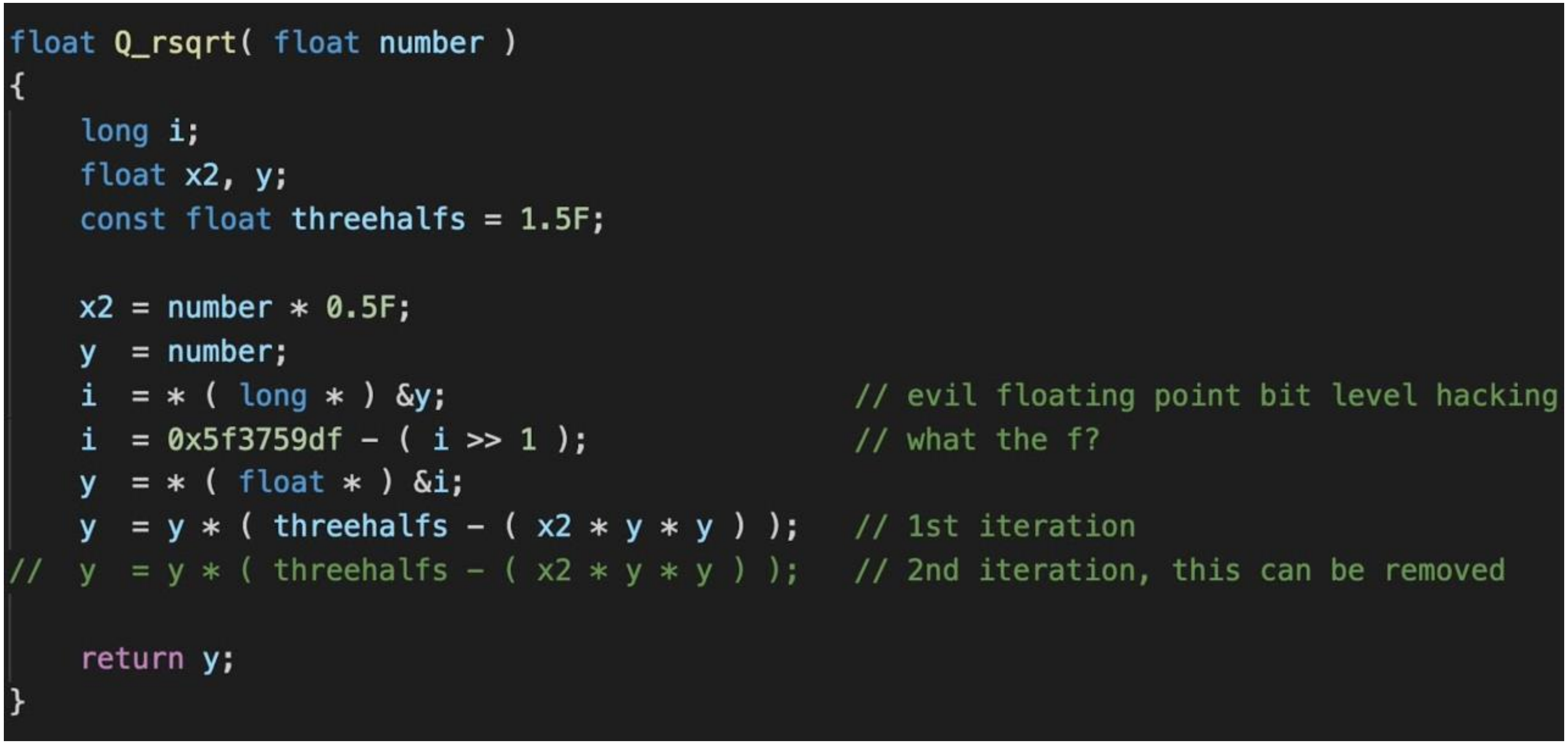
Spatial Safety

- A union is a type that can have more than one possible representations in the same memory position

```
// common design pattern, return a struct that either holds
// an error or the expected value, with a bool to indicate
struct parer_result {
  bool is_valid;
  union {
    char* error message;
    struct parsed_command* cmd;
  };
};
struct parser_result parse_cmd(const char* input);

int main() {
  struct parser_result = parse_cmd("…");
  struct parsed_command = *(parser_result.cmd)
}
// We didn't check if the result was valid, may be violating
spatial safety
```

Spatial Safety

- Sometimes violating spatial safety is "needed"
    - To support “Generics” in C, we often cast to/from `void*`
    - Can be used for some cool stuff like this fast inverse square root algorithm (don’t do this, it is not fast anymore):



```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the f?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

Spatial Safety

- Spatial safety includes index out of bounds.

```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;  // memory smash!
```

No IndexOutOfBounds
Hope for segfault

- What is wrong here?

```
write(STDERR_FILENO, "Hello!\n", PAGE_SIZE);
```

- Here?

```
char buf[6];
strcpy(buf, "Hello!\n");
```

Has C++ Been Fixing These?

- C++ has been giving replacements for these features that are safer.
  - Instead of union, C++ has optional, variant, any and others
  - Instead of C arrays, there is the vector and array type

- Is this C++ safe?

```
vector<int> v {2, 3, 5, 6, 11, 13};
v[1000] = 7;        // is this safe?
v.at(1000) = 0;     // above: no, this: yes
```

- C++ Keeps adding new features that are better and safer but adding in unchecked-unsafe ways to use them. Usually, the argument is for performance

C++ Backwards compatibly

- Even with Modern C++ adding new features to get better and safer, many people stick to bad habits that are kept in C++ for backwards compatibility

Counter Point: How serious is this safety?

- A counterpoint to the safety stuff is that:
  - There is already a lot of tools to help detect these issues (Valgrind, Address Sanitizer, UB Sanitizer, etc.)
  - These issues are common, but they are not the biggest issues of Security

- Notable Recent Security Issues:
  - Heartbleed
  - Spectre & Meltdown
  - Log4j
  - XZ utils backdoor
  - Social Engineering in general

Other Point: Productivity

- These issues also affect how productive C++ developers are. These are added spots for bugs and can make coding more difficult

- Some initial studies report improved productivity from moving from C++ to Rust

- Other languages also have more modern tooling support
  - Compilation
  - Package Management
  - Etc.

Lecture Outline

- What’s Next?

C++ Successor Languages

- Because of the issue with safety, 2022 has been called "the year of the C++ successor Languages"

- Just in 2022, three successor languages were announced:
  - Val (now called Hylo)
  - Carbon
  - cppfront (sometimes called cpp2)

- There have been many languages before:
  - D
  - Go
  - Rust
  - Others: Nim, Zig, Swift, etc.

C and C++ are used everywhere

- Many things are written largely/primarily in C++ or C
  - The Adobe suite (Photoshop, etc)
  - The Microsoft office suite (word, PowerPoint, etc.)
  - The libre office suite (FOSS word, PowerPoint, etc)
  - Chromium (Core of most web browsers, Edge, Opera, Chrome, etc)
  - Firefox
  - Most Database implementations
  - Tensorflow & Pytorch
  - gcc, clang & llvm (which is the backbone for many compilers)
  - Game Engines (Unreal, Unity, etc.)

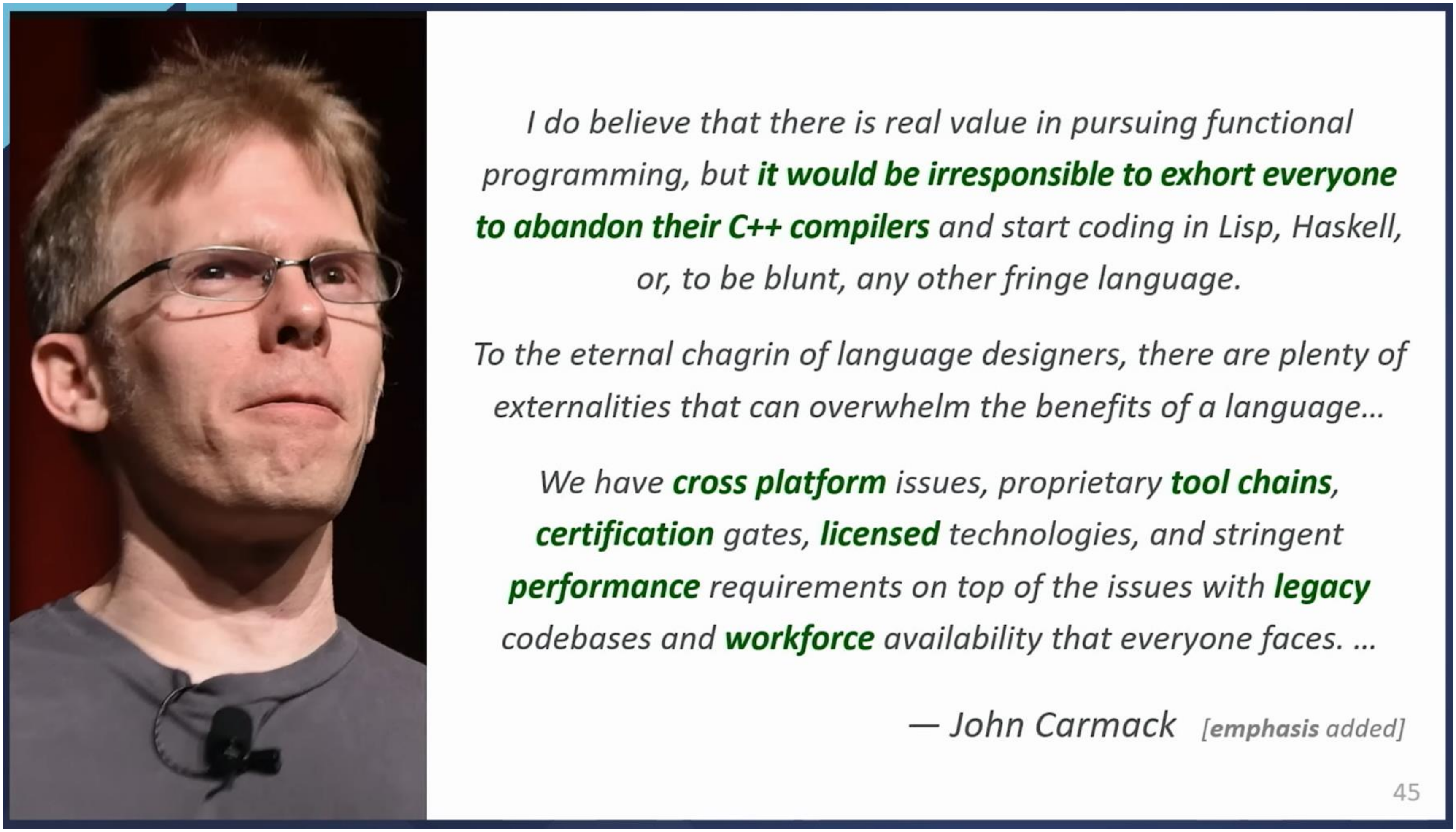
Most of this information is from Jason Turner's "C++ is 40… Is C++ DYING?" video
https://www.youtube.com/watch?v=hxjSpasg3gk

C and C++ are used everywhere

- Regularly ranks in top used ~5-10 programming languages

- Many people still use C++
  - Estimates from JetBrains
  - ~1,157,000 professional developers use C++ as their primary language
  - ~2,492,000 professional developers regularly use C++

Programming Language Adoption



I do believe that there is real value in pursuing functional programming, but ***it would be irresponsible to exhort everyone to abandon their C++ compilers*** *and start coding in Lisp, Haskell, or, to be blunt, any other fringe language.*

To the eternal chagrin of language designers, there are plenty of externalities that can overwhelm the benefits of a language…

We have ***cross platform*** *issues, proprietary* ***tool chains****,* ***certification*** *gates,* ***licensed*** *technologies, and stringent* ***performance*** *requirements on top of the issues with* ***legacy*** *codebases and* ***workforce*** *availability that everyone faces. …*

— John Carmack [**emphasis** added]

45

For better or for worse, C++ already exists and has a bunch of work behind it.

Moving to another thing is going to take time and money, but is not impossible

Screenshot from Herb Sutter's Plenary in cppcon 2023: https://www.youtube.com/watch?v=8U3hl8XMm8c

It is an interesting talk, but his cppcon 2022 or c++now 2023 talks may be better starting points for those interested

Migration

- Some organizations are (at least in part) trying to move from C / C++

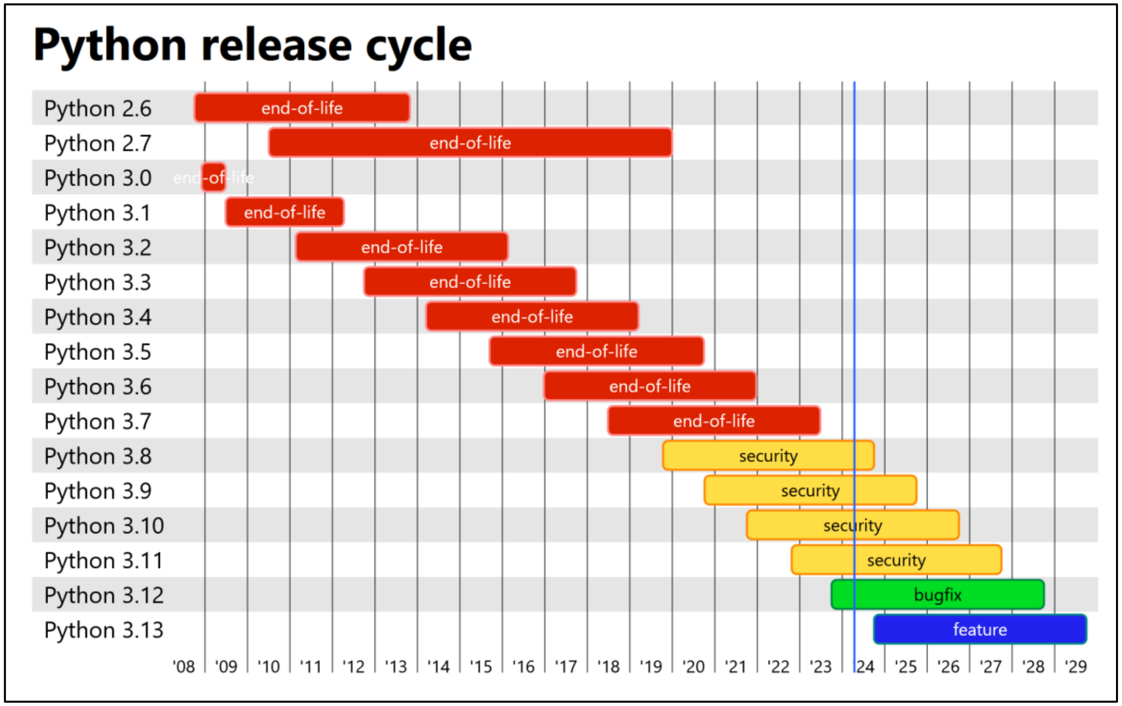
- The Linux kernel has incorporated Rust into it
  - It never allowed C++ into the kernel

- Microsoft and Mozilla Firefox are putting in a lot of effort to start training some employees to program in Rust.

- Some places are investigating the languages “Zig”

Example: Python

- Python made breaking changes just moving from version 2.7 to 3.0
- Python 2.7 was extended in support for a long time. ~10 years



- It took a REALLY long time for many people to give up Python 2.7 and move to Python 3.
- How long will it take to move away from C++?

Evolution

- C++ is evolving to try and accommodate for some of these issues
  - Epochs & safety profiles

- Some passionate C++ developers are trying to make a new language/syntax.
  - Cppfront (cpp2) by Herb Sutter: a new syntax on C++ that fixes a lot of broken defaults and makes writing C++ simpler. Still compiles with and can directly invokes existing C++ code
  - Circle: a C++ compiler that supports many new features including ones related to safety, but these features are not std C++
  - Carbon by Google: a new language with strong C++ interoperability. Still very early on and not runnable

What’s next?

- The situation is developing, we will see how things evolve over time ☺

- There is a lot of inertia towards moving away from C++ and a lot of things look promising
  - I think Rust and Zig both look very very cool and I wish I could teach you one of those languages and we could just use them.
  - Cppfront (or carbon or circle) looks the most promising. They have the advantage of easier integration into existing C++ ecosystems and making C++ safer and easier to use. It is compatible with most existing C++ tools and code-bases.