CXX 3 - Session 3

May’24

EPITA Research & Development Laboratory (LRDE)

C++ and safety

C++ and safety

Memory safety

- C and C++ are **not** inherently memory safe
- This provokes many bugs and discussion

Memory safety

- C and C++ are **not** inherently memory safe
- This provokes many bugs and discussion
- So much so that there is an official warning: white house

Memory safety

- C and C++ are **not** inherently memory safe
- This provokes many bugs and discussion
- So much so that there is an official warning: white house

Why does this happen?

Memory safety

- C and C++ are **not** inherently memory safe
- This provokes many bugs and discussion
- So much so that there is an official warning: white house

Why does this happen?

- C++: Pay for what you ask
- Even with `-fsanitize=address -fsanitize=bounds-strict -fanalyzer`

```
auto v = std::vector<int>();
v.reserve(10);
v.push_back(111);
v[5] = 10;
std::cout << v[5] << ':' << v[9] << '\n';
//std::cout << v[50] << '\n'; // Only this is detected
```

Memory safety 2

- In Python, you can not modify **and** iterate over a list at the same time

```
for (unsigned e : v)
    if (e%2 == 0)
        v.push_back(3*e+1);
```

Memory safety 2

- In Python, you can not modify **and** iterate over a list at the same time

```
for (unsigned e : v)
    if (e%2 == 0)
        v.push_back(3*e+1);
```

- In C++ you can, but you need to be careful.
- The same holds for iterators and references
- Ranges are not safer either (*pair of iterators*)

Memory safety 2

- In Python, you can not modify **and** iterate over a list at the same time

```
for (unsigned e : v)
    if (e%2 == 0)
        v.push_back(3*e+1);
```

- In C++ you can, but you need to be careful.
- The same holds for iterators and references
- Ranges are not safer either (*pair of iterators*)

There are other safety issues (casts, memory leaks, …)

We will focus on **out-of-bounds access**.

Usage of `std::vector`

- Using `.at()` instead of `operator[]`[1]

```
auto v = std::vector<int>();
v.reserve(10);
v.push_back(111);
v.at(5) = 10; // Throws here even without flags
std::cout << v.at(5) << ':' << v.at(9) << '\n';
std::cout << v.at(50) << '\n'; // Only this is detected
```

Usage of `std::vector`

- Using `.at()` instead of `operator[]`[1]

```
auto v = std::vector<int>();
v.reserve(10);
v.push_back(111);
v.at(5) = 10; // Throws here even without flags
std::cout << v.at(5) << ':' << v.at(9) << '\n';
std::cout << v.at(50) << '\n'; // Only this is detected
```

- However

```
auto v = std::vector<int>(10, 123);
auto r = v.at(5);
v.clear();
std::cout << r << '\n';
```

Works just “fine”!

Usage of `std::vector`

- Using `.at()` instead of `operator[]`[1]

```
auto v = std::vector<int>();
v.reserve(10);
v.push_back(111);
v.at(5) = 10; // Throws here even without flags
std::cout << v.at(5) << ':' << v.at(9) << '\n';
std::cout << v.at(50) << '\n'; // Only this is detected
```

- However

```
auto v = std::vector<int>(10, 123);
auto r = v.at(5);
v.clear();
std::cout << r << '\n';
```

Works just “fine”!

The same holds for iterators and ranges

```
auto v = std::vector<int>(10, 2);
auto it = v.begin() + 2; *it += 2;
auto s = std::span(v.begin(), v.end());
v.clear();
std::cout << *it << std::endl;
for (auto&& x : s)
    std::cout << x << ' ';
```

Why is python *memory safe*

The reasons

- All[a] objects are basically wrapped in a shared pointer
- Out-of-bounds access is systematically checked for
- There is no notion of iterator into a list

Why is python *memory safe*

The reasons

- All[a] objects are basically wrapped in a shared pointer
- Out-of-bounds access is systematically checked for
- There is no notion of iterator into a list

[a] Almost all ...

The drawbacks

- Large memory overhead, especially for small objects
- Less efficient due to additional checks
- Operations can not be optimized (e.g. vectorized)