MANCHESTER 1824
The University of Manchester

COMP26020 Programming Languages and Paradigms

Lecture 44: A smaller and clearer language

Pavlos Petoumenos
pavlos.petoumenos@manchester.ac.uk

C++

C++98: Already a massive language

C++11: Added tons of stuff while maintaining backward compatibility

→ An even larger language

Very few people understand the whole thing

You don’t have to understand the whole thing

Baggage from C and C++98

Capabilities only useful for implementing the standard library

Capabilities only useful to other library developers

Capabilities used only in certain sub-domains

A smaller and clearer language

Most new code needs only a small subset of C++

C++ Core Guidelines

- Not fully mechanically enforced
- Not perfect
- Not complete
- But a good start for writing better C++

Core Guidelines Philosophy

P.1: Express ideas directly in code
P.2: Write in ISO Standard C++
P.3: Express intent
P.4: Ideally, a program should be statically type safe
P.5: Prefer compile-time checking to run-time checking
P.6: What cannot be checked at compile time should be checkable at run time
P.7: Catch run-time errors early
P.8: Don't leak any resources
P.9: Don't waste time or space
P.10: Prefer immutable data to mutable data
P.11: Encapsulate messy constructs, rather than spreading through the code
P.12: Use supporting tools as appropriate
P.13: Use support libraries as appropriate

Previously discussed

F.4: If a function might have to be evaluated at compile time, declare it constexpr
F.21: To return multiple "out" values, prefer returning a struct or tuple
C.3: Represent the distinction between an interface and an implementation using a class
C.9: Minimize exposure of members
C.33: If a class has an owning pointer member, define a destructor
C.49: Prefer initialization to assignment in constructors
C.51: Use delegating constructors to represent common actions for all constructors of a class
C.132: Don't make a function virtual without reason
C.160: Define operators primarily to mimic conventional usage
R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)
R.3: A raw pointer (a T*) is non-owning
R.10: Avoid malloc() and free()
R.11: Avoid calling new and delete explicitly
R.20: Use unique_ptr or shared_ptr to represent ownership
ES.1: Prefer the standard library to other libraries and to "handcrafted code"
ES.11: Use auto to avoid redundant repetition of type names
ES.23: Prefer the {}-initializer syntax
ES.47: Use nullptr rather than 0 or NULL
SF.1: Use a .cpp suffix for code files and .h for interface files if your project doesn't already follow another convention
SL.con.1: Prefer using STL array or vector instead of a C array
SL.con.2: Prefer using STL vector by default unless you have a reason to use a different container

Not discussed before (1)

I.11: Never transfer ownership by a raw pointer (T*) or reference (T&)

I.13: Do not pass an array as a single pointer

F.15: Prefer simple and conventional ways of passing information

<table>
<tr><th></th><th>Cheap or impossible to copy (e.g., int, unique_ptr)</th><th>Cheap to move (e.g., vector<T>, string)
or Moderate cost to move (e.g., array<vector>, BigPOD)
or Don't know (e.g., unfamiliar type, template)</th><th>Expensive to move (e.g., BigPOD[], array<BigPOD>)</th></tr>
<tr><td>Out</td><td colspan="2">X f()</td><td rowspan="2"></td></tr>
<tr><td>In/Out</td><td colspan="2">f(X&)</td></tr>
<tr><td>In</td><td rowspan="2">f(X)</td><td rowspan="2">f(const X&)</td><td rowspan="2"></td></tr>
<tr><td>In & retain "copy"</td></tr>
</table>

Not discussed before (2)

C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead

BAD

```
class Obj {
  int member1;
  int member2;
public:
  Obj() : member1{-1}, member2{-1} {};
```

GOOD

```
class Obj {
  int member1{-1};
  int member2{-1};
public:
  // auto default constructor
```

Not discussed before (3)

ES.25: Declare an object const or constexpr unless you want to modify its value later on

ES.30-31: Don't use macros

ES.45: Avoid "magic constants"; use symbolic constants

ES.71: Prefer a range-for-statement to a for-statement when there is a choice

Not discussed before (4)

SL.con.3: Avoid bounds errors

- Reason
- Notes
- Example, bad
- Example, good
- Enforcement
- (Exceptions?)

SL.con.3: Avoid bounds errors

Reason

Read or write beyond an allocated range of elements typically leads to bad errors, wrong results, crashes, and security violations.

Note

The standard-library functions that apply to ranges of elements all have (or could have) bounds-safe overloads that take `span`. Standard types such as `vector` can be modified to perform bounds-checks under the bounds profile (in a compatible way, such as by adding contracts), or used with `at()`.

Ideally, the in-bounds guarantee should be statically enforced. For example:

- a range-`for` cannot loop beyond the range of the container to which it is applied
- a `v.begin(),v.end()` is easily determined to be bounds safe

Such loops are as fast as any unchecked/unsafe equivalent.

Often a simple pre-check can eliminate the need for checking of individual indices. For example

- for `v.begin(),v.begin()+i` the `i` can easily be checked against `v.size()`

Such loops can be much faster than individually checked element accesses.

Example, bad

```
void f()
{
    array<int, 10> a, b;
    memset(a.data(), 0, 10);        // BAD, and contains a length error (length = 10 * sizeof(int))
    memcmp(a.data(), b.data(), 10); // BAD, and contains a length error (length = 10 * sizeof(int))
}
```

Also, `std::array<>::fill()` or `std::fill()` or even an empty initializer are better candidates than `memset()`.

Example, good

```
void f()
{
    array<int, 10> a, b, c{};      // c is initialized to zero
    a.fill(0);
    fill(b.begin(), b.end(), 0);   // std::fill()
    fill(b, 0);                    // std::ranges::fill()

    if ( a == b ) {
      // ...
    }
}
```

Example

If code is using an unmodified standard library, then there are still workarounds that enable use of `std::array` and `std::vector` in a bounds-safe manner. Code can call the `.at()` member function on each class, which will result in an `std::out_of_range` exception being thrown. Alternatively, code can call the `at()` free function, which will result in fail-fast (or a customized action) on a bounds violation.

```
void f(std::vector<int>& v, std::array<int, 12> a, int i)
{
    v[0] = a[0];       // BAD
    v.at(0) = a[0];    // OK (alternative 1)
    at(v, 0) = a[0];   // OK (alternative 2)

    v.at(0) = a[i];    // BAD
    v.at(0) = a.at(i); // OK (alternative 1)
    v.at(0) = at(a, i); // OK (alternative 2)
}
```

Enforcement

- Issue a diagnostic for any call to a standard-library function that is not bounds-checked. ??? insert link to a list of banned functions

This rule is part of the bounds profile.

Do I need to remember all of them?

Will the guidelines be in the exam?

Core Guidelines

Exam-wise: only the ones, I have mentioned in the lectures or the live sessions

Lab-wise and in real life

- Don’t remember them
- Learn from them

Guidelines Enforcement

Guidelines are not mandatory

Many cannot be enforced

Some important ones can be enforced

clang-tidy, MS CppCoreCheck, your favourite IDE

clang-tidy

```
$ clang-tidy test.cpp -checks=cppcoreguidelines-*

1941 warnings generated (most in library code and suppressed)

test.cpp:12:3: warning: do not declare C-style arrays, use std::array<> instead
[cppcoreguidelines-avoid-c-arrays]

  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26}; // Fast

  ^

test.cpp:12:31: warning: 5 is a magic number; consider replacing it with a named constant
[cppcoreguidelines-avoid-magic-numbers]

  int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26}; // Fast

                               ^

...

test.cpp:19:6: warning: variable 'cube' is non-const and globally accessible, consider making it
const [cppcoreguidelines-avoid-non-const-global-variables]

auto cube = [](int x) {return x * x * x;};

     ^
```

clang-tidy

Many more automated checks

bugprone-	bug-prone code constructs
clang-analyzer-	clang Static Analyzer checks
concurrency-	concurrent programming (including threads, fibers, coroutines, etc.)
google-	Google coding conventions
misc-	
modernize-	advocate usage of modern language constructs
performance-	performance-related issues
portability-	portability-related issues
readability-	readability-related issues

Recap

Less is more

C++ Core Guidelines

Checkers to partially enforce

Up Next

Part 1 Epilogue