This is why we can’t have nice things

Timo van der Kuil
Who am I

• Student
• Student assistant
• Research intern

• Saxion University of Applied Sciences
Disclaimer

• First talk at a conference
  • Feedback
• Opinions are my own
Outline

• Weird things
• Origins
• Design and philosophy
• Flexibility
• Weird things explained
Weird things in C++
Initialization

```cpp
int a; // a is not initialized, only declared

int a{}; // a is initialized with 0

std::array<int, 100> array; // array is not initialized, only declared

std::array<int, 100> array{}; // array is initialized with 0’s
```
Unspecified behaviour

```cpp
int a() { return std::puts("a"); } 
int b() { return std::puts("b"); } 
int c() { return std::puts("c"); } 
void f(int, int, int) {} 

int main() {
    f(a(), b(), c());
    // a b c?
    // c b a?
}
```
More unspecified behaviour

```cpp
int a() { return std::puts("a"); }  
int b() { return std::puts("b"); }  
int c() { return std::puts("c"); }

int main() {
    return a() + b() + c();
}
```
void f0(int i) { } // Void as return type -> no return

int f1(void) { return 1; } // Void as parameter -> no parameters

int f2(void* i) {
    return *static_cast<int*>(i);
} // Void* as parameter -> pointer to anything

(void) some_unused_var // Void as cast -> cast to nothing
mutable lambdas

```cpp
int i = 2;

auto ok = [&i](){ ++i; }; // i captured by reference

auto err = [i](){ ++i; }; // increment of read-only variable 'i'

auto err2 = [x{22}](){ ++x; }; // increment of read-only variable 'x'

auto ok2 = [i, x{22}]() mutable { ++i; ++x; }; // Using mutable keyword
```
future.h

```cpp
std::async(std::launch::async,[]{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "first thread" << '\n';
});

std::async(std::launch::async,[]{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "second thread" << '\n';
});

std::cout << "main thread" << '\n';
```

first thread
second thread
main thread
Type punning through **unions**

```cpp
union Pun {
    int x;
    unsigned char c[sizeof(int)];
};

void bad(Pun& u)
{
    u.x = 'x';
    std::cout << u.c[0] << '
';  // undefined behaviour
}
```
Origins of C++

- A brief history -
Idea for a suitable tool

• Best of both worlds
• Simula
  • Classes
  • Hierarchies
  • Concurrency
  • Static type checking
• BCPL
  • Efficiency
  • Combining compiled programs
• Portable implementation
C with classes

Issue that called for a new tool

Cpre

C with classes
From C with classes to C++

• C with classes was a medium success
• Paid for itself and developer
• Not for support and development

• Two choices:
  1. Stop supporting the language to be able to do something else
  2. Develop a new language that appeals to a larger audience to pay for its support and further development
Usage of C++

(github.com, 2019)
Javascript 1995
Python 1991
Java 1995
PHP 1995
C# 2001
Typescript 2012
Shell 1989 (bash)
C 1972
"C is clearly not the cleanest language ever designed nor the easiest to use, so why do so many people use it?"

- Bjarne Stroustrup, 1987
Why C?

- C is flexible
  - Almost every application
- C is efficient
  - C is low level, relatively easy to make the most out of resources
- C is available
  - There is a compiler for pretty much every platform
- C is portable
  - Porting from OS to OS is typically feasible, but not trivial
Design and philosophy of C++
Aims of C++

• Make programming more enjoyable
• General purpose programming language that
  • Is a better C
  • Supports data abstraction
  • Supports object-oriented programming
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. **Must be useful now**
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. **Every feature must have a reasonably obvious implementation**
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. **Always provide a transition path**
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. **Provide comprehensive support for each supported style**
8. Don’t try to force people
Development rules of C++

1. Evolution must be driven by real problems
2. Don’t get involved in a sterile quest for perfection
3. Must be useful now
4. Every feature must have a reasonably obvious implementation
5. Always provide a transition path
6. It’s a language, not a system
7. Provide comprehensive support for each supported style
8. Don’t try to force people
Flexibility
Type system

- Strongly and statically typed
- Type specifiers -> Compile time checking
- Fundamental types
  - \texttt{char}, \texttt{double}, \texttt{int}
- Compound types
  - ‘Defined in terms of another type’
- Every type is treated equally
- \textbf{The C++ Type System is your Friend} by Hubert Matthews
class Date{  
    Date(int, int, int) {};  
};

class Year {};  
class Month {};  
class Day {};

class Date{  
    Date(Year, Month, Day) {};  
};

// Ambiguous: d/m/y  y/m/d  m/d/y?  
// -> bug at runtime

// Umambiguous: y/m/d  
// -> bug at compile time
Memory model

• ‘The Memory Model’ by Rainer Grimm

• First only sequential execution -> no need for memory model
• C++11 multi-threading
• Race conditions
  • Every thread has r/w to memory
• 6 memory orders
  • \texttt{std::atomic}
Memory model

- `memory_order_seq_cst` -> Default, strict
- `memory_order_acq_rel` -> No reordering before and after
- `memory_order_acquire` -> No reordering before
- `memory_order_release` -> No reordering after
- `memory_order_consume` -> No reordering before and after (of this atomic)
- `memory_order_relaxed` -> Weak

- Less rules -> more optimization
- Up to the programmer
Why things are weird in C++
int a;  
// a is not initialized, only declared

int a{};  
// a is initialized with 0

std::array<int, 100> array;  
// array is not initialized, only declared

std::array<int, 100> array{};  
// array is initialized with 0’s
Initialization

- Inherited from C
- Initialization can lead to performance hits
  - Mostly on older systems
- `std::array` -> implicit, default, trivial constructor (POD)
  - Empty ctor but value initialized with `{}` or `{}`
  - Wrapper for C-style `array`
- MSVC debug vs. release mode
- ‘Initialization in C++’ by Timur Doumler
- ‘The nightmare of initialization in C++’ by Nicolai Josuttis
Unspecified behaviour

```cpp
int a() { return std::puts("a"); }
int b() { return std::puts("b"); }
int c() { return std::puts("c"); }
void f(int, int, int) {}  

int main() {
    f(a(), b(), c());  
    a b c?
    c b a?
}
```
More unspecified behaviour

```cpp
int a() { return std::puts("a"); }
int b() { return std::puts("b"); }
int c() { return std::puts("c"); }

int main() {
    return a() + b() + c();
}
```
Unspecified behaviour

- Not specified in ISO standard
- Comma is not a sequence point
- Mistakes can easily be avoided
  - Warnings
- Don’t force compilers

\[ f(a(), b(), c()); \]
```c
void f0(int i) { } // Void as return type -> no return

int f1(void) { return 1; } // Void as parameter -> no parameters

int f2(void* i) {
    return *static_cast<int*>(i); // Void* as parameter -> // pointer to anything
}

(void) some_unused_var // Void as cast -> cast to nothing
```
void

- Inherited from C
- Used for polymorphism in C
- C++ has templates, `std::optional, std::variant`
- Certain platforms still need `void*`; limited access to header
void

C:
  f()  -> Any amount of arguments (marked obsolescent)
  f(void) -> No arguments

C++:
  f()  -> No arguments
  f(void) -> No arguments
mutable lambdas

```cpp
int i = 2;

auto ok = [&i](){ ++i; }; // i captured by reference

auto err = [i](){ ++i; }; // increment of read-only variable 'i'

auto err2 = [x{22}]{() { ++x; } }; // increment of read-only variable 'x'

auto ok2 = [i, x{22}][() mutable { ++i; ++x; }]; // Using mutable keyword
```
mutable lambdas

- Unnamed `class`
- `::operator()` is implicitly defined `const`
  - UNLESS `mutable` is used
- Member variables
- Prevent unwanted mutation of lambdas
mutable lambdas

```cpp
int x{5};
int y{};

auto lambda = [x]() mutable { return x++ + 5; };

y = lambda();
std::cout << y << ',' << x << '
';

x = 20;
y = lambda();
std::cout << y << ',' << x << '
';
```
mutable lambdas

auto lambda = [x]() mutable { return x++ + 5; };

struct UnnamedType {
    // Member variable that stores captured val
    int x;
    // Ctor initializes x with captured val (5)
    UnnamedType(int x) : x{x} {}  
    // operator() executes the passed statements
    int operator() () { return x++ + 5; }
};
mutable lambdas

```cpp
auto lambda = [x](){ mutable {return x++ + 5; }};

struct UnnamedType {
  // Member variable that stores captured val
  const int x;
  // Ctor initializes x with captured val (5)
  UnnamedType(int x) : x{x} {}  
  // operator() executes the passed statements
  int operator() () const { return x++ + 5; }  
};
```
mutable lambdas

```cpp
int x{5};
int y{};

auto lambda = [x](){ mutable { return x++ + 5; });

y = lambda();
std::cout << y << ',' << x << '
'; //outputs 10,5

x = 20;
y = lambda();
std::cout << y << ',' << x << '
'; //outputs 11,20
```
mutable lambdas

- Only two ways to capture -> explicit choice by the programmer
  - Copy-by-value [\=, val]
  - Reference [&, &val]

```cpp
template
auto lambda = [x]() { return x++ + 5; }; 
```

```cpp
template
auto lambda = [x]() const { return x + 5; }; 
```

- N3424: Lambda Correctness and Usability Issues by Herb Sutter
std::async(std::launch::async,[]{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "first thread" << '
';
});

std::async(std::launch::async,[]{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "second thread" << '
';
});

std::cout << "main thread" << '
';
auto first = std::async(std::launch::async, []{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::cout << "first thread" << '\n';
});

auto second = std::async(std::launch::async, []{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "second thread" << '\n';
});

std::cout << "main thread" << '\n';
• N2802: A plea to reconsider detach-on-destruction for thread objects by Hans Boehm
• N3630: async, ~future, and ~thread (Revision 1) by Herb Sutter
• N3636: ~thread Should Join by Herb Sutter
• N3637: async and ~future (Revision 3) by Herb Sutter, Chandler Carruth, Niklas Gustafsson
• N3679: Async() future destructors must wait by Hans Boehm
• N3773: async and ~future (Revision 4) by Herb Sutter, Chandler Carruth, Niklas Gustafsson
• N3776: Wording for ~future by Herb Sutter
• N3777: Wording for deprecating async by Herb Sutter
future.h

- One proposal to save us all
- Initially C++20, now C++23 or even C++26
  - Relies on executors

- P1054r0: A Unified Futures Proposal For C++ by Lee Howes, Bryce Adelstein Lelbach, David S. Hollman and Michal Dominiak
Type punning through **unions**

```cpp
union Pun {
    int x;
    unsigned char c[sizeof(int)];
};

void bad(Pun& u)
{
    u.x = 'x';
    std::cout << u.c[0] << '\n'; // undefined behaviour
}
```
Type punning through **unions**

- Popular in C; no alternative
- Used on systems with limited capacity
- (mis)Used for type punning
- C++ has `static_cast<>()`
- `std::byte` with `static_cast<char>()`

```cpp
std::vector<std::byte> i_buffer;
i_buffer.push_back(std::byte(0b01000011));

std::cout << static_cast<char>(i_buffer[0]) << '\n';
```
Closing thoughts
Many ways to do the same
Hard to learn
Hard to teach
Versatile
Fun
This is why we can’t have nice things

Timo van der Kuil