



KDAB



Embedded Days

13-14 April 2021

Demystifying C++ for C developers

Qt Embedded Days 2021

Giuseppe D'Angelo

giuseppe.dangelo@kdab.com

The Qt, OpenGL and C++ experts

About me

- Senior Software Engineer, KDAB
- Developer & Trainer
- Qt Approver
- Ask me about QtCore, QtGui, QtQuick, ...
 - And also about Modern C++, 3D graphics



C++ usage in Embedded Systems

- C++ usage has consistently been behind C usage in all rankings.
- Since... **forever!**

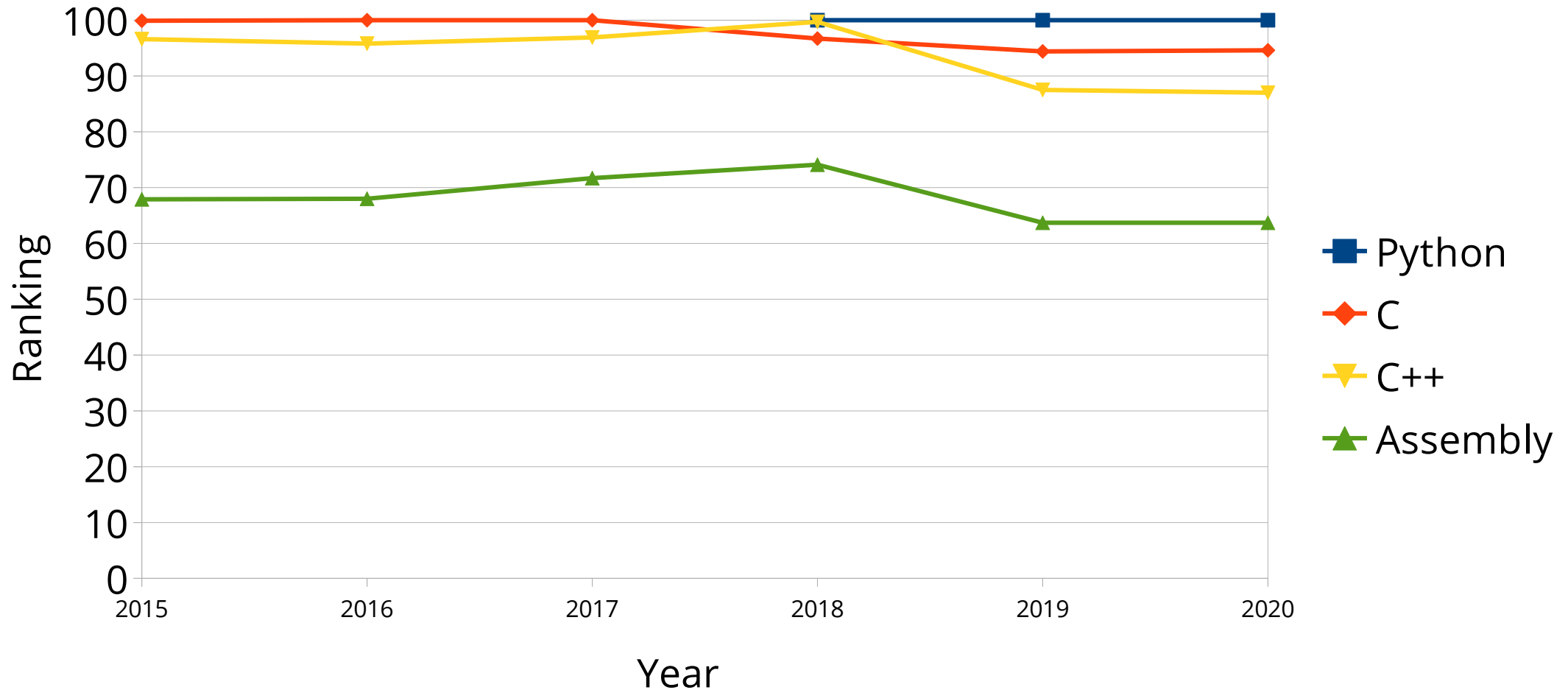
C/C++ usage in Embedded Systems

Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python▼	  	100.0
2	C▼	  	94.6
3	C++▼	  	87.0
4	Arduino▼		73.2
5	Assembly▼		63.7
6	Rust▼	  	55.6

Source: IEEE Spectrum Top Programming Languages, Embedded, 2020

C/C++ usage in Embedded Systems



Source: IEEE Spectrum Top Programming Languages, Embedded, 2015-2020

Why is that ?

Why is that?

- Is C++ simply not available for some developers?
 - E.g. embedded vendors still not providing C++ toolchains.
 - (Or providing outdated, bugged ones)
- Luckily, this bad trend has disappeared.

Why is that?

- Is C++ not suited for embedded development?
 - Does it perform too poorly?
 - Does it not offer (at least) the same good qualities of C++, if not more?
- Simply untrue
 - “Don’t pay for what you don’t use” is a C++ mantra
 - Many times C++ is more stringent than C, not vice versa
 - Ask your closest C++ advocate: they’ll swear that **C++ is better than C.**

Why is that?

- Do C developers simply not *like* C++?
- If so:
 - How do we make C++ development more encouraging?
 - How do we find out what the “pain points” are?
 - Is there anything we can do about them?



*Unrelated stock photo
of a C developer who
may not like C++ very much.*

IN A NUTSHELL

Poor adoption of C++ by C programmers
is not a one-dimensional problem.

And it's hardly a technological problem.

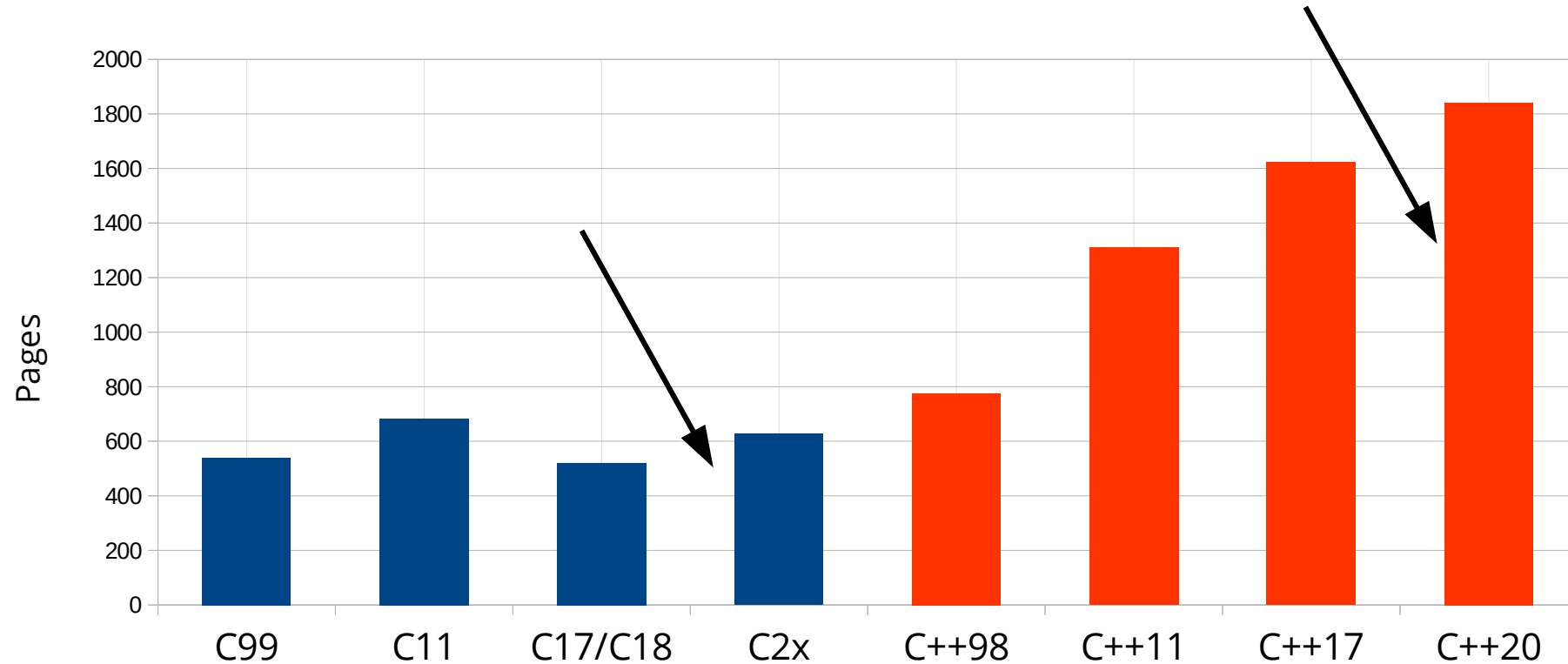
Is there even a choice?

- For the overwhelming majority of developers, programming languages are a **tool** to **get the job done**.
- Not everyone can afford spending time to learn a new tool. They've got projects to deliver, honoring the specs, on time, within budget.
 - So please don't harass your fellow C developer with "*how great C++ is!*"

Is there even a choice?

- Even if some developer could afford learning C++, why should they?
- Getting out of your comfort zone is a huge demand.
 - People are willing to do so only for extremely good reasons.
 - Seeing the “advantages over there” is usually not enough; being scared of the “disadvantages over here” is a much better approach.
- C++ complexity is daunting!
Even *seeing* the advantages is not so obvious.

Page count of each published C/C++ standard



The size of the C++20 Standard is ***three times*** the size of the latest C draft !

What do people do when facing
hard-to-explain complexity?

They invent stories.

Myths and legends are all born this way.

“Use C++, and after a few days you’ll face this...!”

```

template <typename F, typename IndexSequence, typename TP>
class FunctionObjectSlotArgumentsHelper;
template <typename F, std::size_t ... Ns, typename ... Args>
class FunctionObjectSlotArgumentsHelper<F, std::index_sequence<Ns...>, TypePack<Args...>>
{
    using Arguments = TypePack<Args...>;

    using ListOfCompatibleArguments = BoolPack<
        (IsInvocable<F, typename TruncateTypePack<Ns, Arguments>::type>::value) ...
    >;

    // Get the index of the last compatible one, and use that one to determine the arguments
    static constexpr inline auto IndexOfLastCompatible = IndexOfLastTrueInPack<ListOfCompatibleArguments>::value;
    static_assert(IndexOfLastCompatible ≥ 0,
        "There is no subset of arguments compatible with the callable");

    using type = typename TruncateTypePack<IndexOfLastCompatible, Arguments>::type;
    ...

```

Guess what? **That’s a myth.** There’s no *need* of using any of these C++ facilities, especially when coming from C.

The origin of “Fake news”

- Stories are born to let people stay in their own comfort zone.
 - Mocking the “adversary” is the perfect example of this.
- **“False news is probably born of *imprecise individual observations* or imperfect eyewitness accounts, but the original accident is not everything: by itself, it really explains nothing.**

The error propagates itself, grows, and ultimately survives only on one condition—that it finds a favorable cultural broth in the society where it is spreading. **Through it, people unconsciously express all their prejudices, hatreds, fears, all their strong emotions.”**

- *Réflexions d’un historien sur les fausses nouvelles de la guerre*, Marc Bloch, 1921

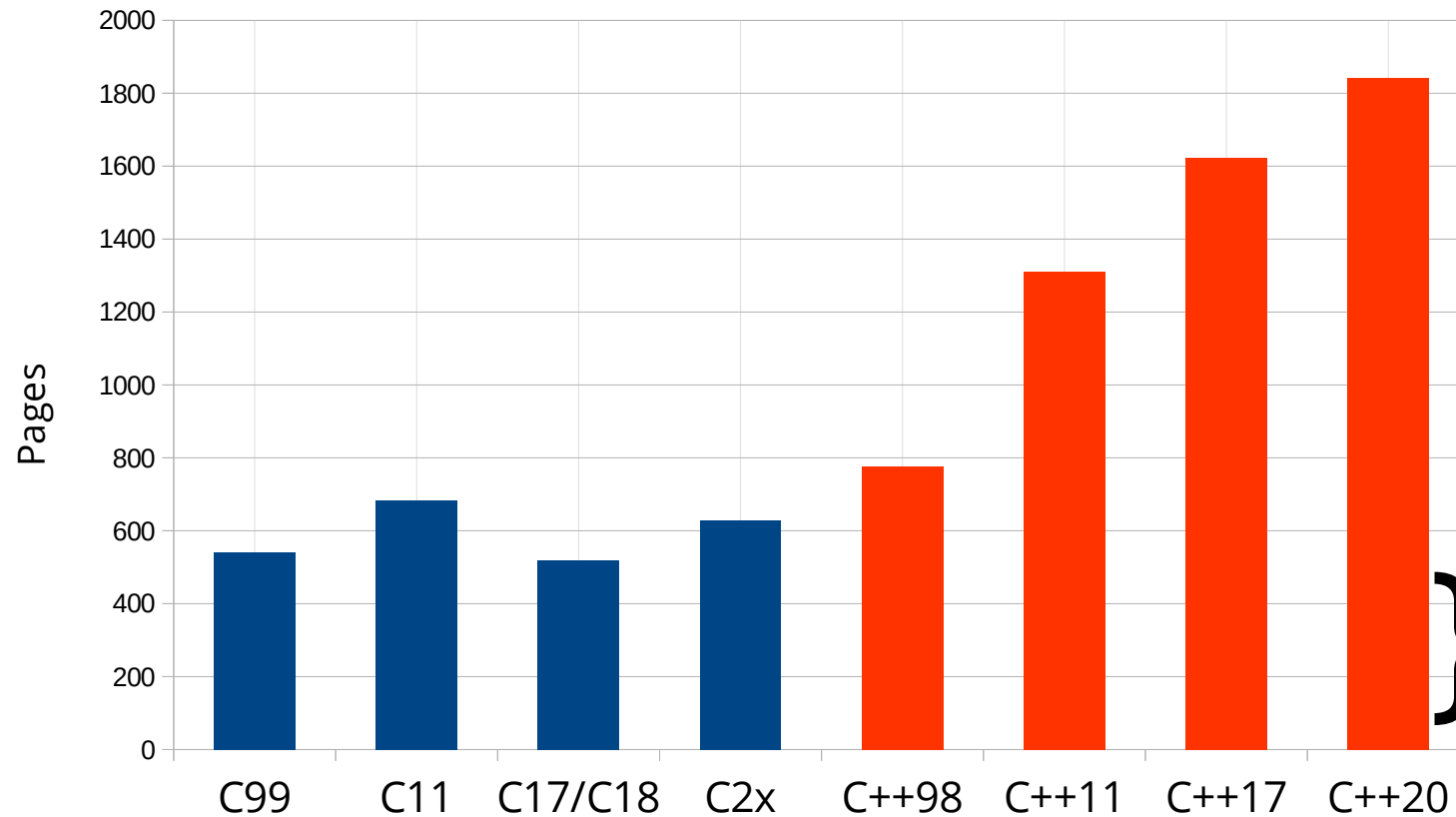
Literally scientific people ; engineers ; developers :
they're all humans, and therefore,
they are *not immune* from this!

(Do we want to go in a emacs / vi flamewar ?)

So how to make C++ more pleasant for C developers?

- I don't still fully know.
- My advice, if you are a C developer and want to start exploring C++: stick to **simple things**.
 - Identify simple patterns that you can fully control.
 - Maybe that are perfect, free replacements for patterns you've already been using.
 - Get confidence using those.

Page count of each published C/C++ standard



The bits you want to focus on when you start

C++ is C++

- Also please ignore anyone claiming that “using a subset of C++ is not using C++”.
 - **Programming languages are tools.**
- Using C++ as “C with classes” isn’t wrong. At bare minimum, you get proper syntax for OOP, and destructors!

Let me use the rest of this talk to show you some appreciation of C++ features, and hopefully, debunk some of the “false stories” you may have heard.

“C++ generates bigger code”

- (An overreaching statement; are the input conditions equal?)
- In C++ you **don't pay for what you don't use**, not even in terms of code size.
- C++ compilers can optimize for size just as well as C compilers.
- But: *some* C++ features may cost you in terms of code size. Don't use/disable them if you're not fine with this.

Remove functionalities that may add code overhead

- If you don't use RTTI (`dynamic_cast`, `typeid`, exceptions), disable it:
 - `-fno-rtti`
- If you don't use exceptions, disable their support:
 - `-fno-exceptions -fno-asynchronous-unwind-tables`

Don't link to the C++ Standard Library if not used

- g++ and clang++ by default link against the respective C++ standard libraries (libstdc++ / libc++). Those libraries may be huge!
- If you not use anything but operator new / delete and similar low level functions, you may want to link to the “core” subsets instead:
 - libsupc++ for libstdc++ (-lsupc++)
 - libc++abi for libc++ (-lc++abi)
- Use gcc (not g++) and clang (not clang++) to perform linking.

Limit the usage of templates to combat code bloat

- Templates may generate more code, trading speed for code size.
- Use whatever solutions you have from C to write “generic” code if size is a problem. Voilà!
 - When using templates then? To increase correctness and being even more generic than a solution achievable in C.

C can be pretty ugly.
C++ solves lots of the « superficial » problems.

C99 can be ugly

- C99 does not have overloads: similar functions require different names. Using the wrong name may even work.

```
double      sin (double angle);
float       sinf(float angle);
long double sinl(long double angle);

double d = 3.14;
printf("sin %f\n", sin(d));
float f = 1.0f;
printf("sin %f\n", sinf(f));
```

C11 can be better...

- C11 improves by introducing generics:

```
double      sin (double angle);
float       sinf(float angle);
long double sinl(long double angle);

#define sin(angle) _Generic((angle), \
    long double: sinl, \
    float: sinf, \
    default: sin \
)(angle)

double d = 3.14;
printf("sin %f\n", sin(d));
float f = 1.0f;
printf("sin %f\n", sin(f));
```

C98 is best

- C++ has *always* had overloading, without an awkward syntax.

```
double      sin(double angle);
float       sin(float angle);
long double sin(long double angle);

double d = 3.14;
printf("sin %f\n", sin(d));
float f = 1.0f;
printf("sin %f\n", sin(f));
```

Classes as... first-class citizen

- Everyone *already* does OOP in C; why not using a more convenient syntax?

```
pthread_mutex *mutex;  
pthread_mutex_init(mutex, NULL); // create  
pthread_mutex_lock(mutex);      // use it  
// ...  
pthread_mutex_unlock(mutex);  
pthread_mutex_destroy(mutex); // destroy
```

```
GString *str = g_string_new("Hello");  
str = g_string_append(str, ", world!");  
g_print(str);  
g_string_free(str);
```

```
std::mutex mutex;  
mutex.lock();  
// ...  
mutex.unlock();
```

```
QByteArray ba("Hello");  
ba.append(", world");  
printf("%s", ba.data());
```

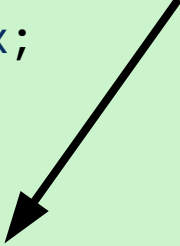
Classes as... first-class citizen

By the way: did you notice the lack of the “destroy” step in C++?

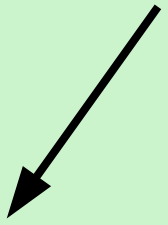
```
pthread_mutex *mutex;
pthread_mutex_init(mutex, NULL);
pthread_mutex_lock(mutex);
// ...
pthread_mutex_unlock(mutex);
pthread_mutex_destroy(mutex);

GString *str = g_string_new("Hello");
str = g_string_append(str, ", world!");
g_print(str);
g_string_free(str);
```

```
std::mutex mutex;
mutex.lock();
// ...
mutex.unlock();
```



```
QByteArray ba("Hello");
ba.append(", world");
printf("%s", ba.data());
```



Destructors

- C++ destroys local objects *automatically* when control leaves their scope.
- No need to remember to do so manually. A bless.
- And, each class can define what to do when it gets destroyed:
 - free memory? ✓
 - close a file descriptor? ✓
 - open a mutex? ✓
 - disconnect from the database? ✓
 - ...

Structured error handling in C

```
int doSomething()
{
    int result = FAILURE;
    int *buffer = (int *)malloc(1024 * sizeof(int));
    if (!buffer)
        goto err1;
    int fd = open("file", O_RDONLY | O_CLOEXEC);
    if (fd < 0)
        goto err2;
    pthread_mutex_lock(mtx);
    // ... do the actual work here ...
    result = SUCCESS;
    pthread_mutex_unlock(mtx);
    close(fd);
err2:
    free(buffer);
err1:
    return result;
}
```

Structured error handling in C

```
int doSomething()
{
    int result = FAILURE;
    int *buffer = (int *)malloc(1024 * sizeof(int));
    if (!buffer)
        goto err1;
    int fd = open("file", O_RDONLY | O_CLOEXEC);
    if (fd < 0)
        goto err2;
    pthread_mutex_lock(mtx);
    // ... do the actual work here ...
    result = SUCCESS;
    pthread_mutex_unlock(mtx);
    close(fd);
err2:
    free(buffer);
err1:
    return result;
}
```

Structured error handling in C++: RAII

- Resource Acquisition Is Initialization

```
int doSomethingBetter()
{
    std::unique_ptr<int[]> buffer(new int[1024]);
    if (!buffer)
        return FAILURE;

    FdHandler fd("file", O_RDONLY | O_CLOEXEC); // not in std
    if (!fd)
        return FAILURE;

    std::scoped_lock<std::mutex> lock(mutex);

    // ... do the work ...

    return SUCCESS;
}
```

Nothing to do to clean up the resources acquired, in either success or in failure paths.

Structured error handling in C++: RAII+Exceptions

- If the holder objects throw an exception if they fail to initialize, the code can be simplified even further:

```
void doSomethingWithExceptions()
{
    std::unique_ptr<int[]> buffer(new int[1024]);
    FdExceptionHandler fd("file", O_RDONLY | O_CLOEXEC);
    std::scoped_lock<std::mutex> lock(mutex);

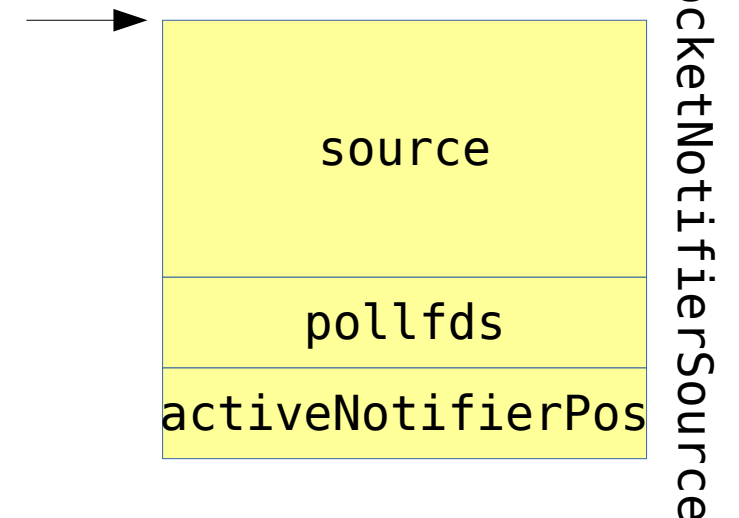
    // ... do the work ...
}
```

- HOWEVER: now you pay for the convenience. An experiment on i386 (GCC 10.2, -m32) shows ~100 more bytes generated for the above code.

Inheritance in C

- Libraries allow you to define “subclasses” for their types
 - “Put an object of this type as the *first member* of your struct”
 - “Put this macro as the *first thing* in your struct”
- Why? So that the memory layout of your subclass is fixed, and a pointer to your subclass can be converted to a pointer to the base class.

```
typedef struct
{
    GSource source;
    QList<GPollFDWithQSocketNotifier *> pollfds;
    int activeNotifierPos;
} GSocketNotifierSource;
```



Inheritance in C

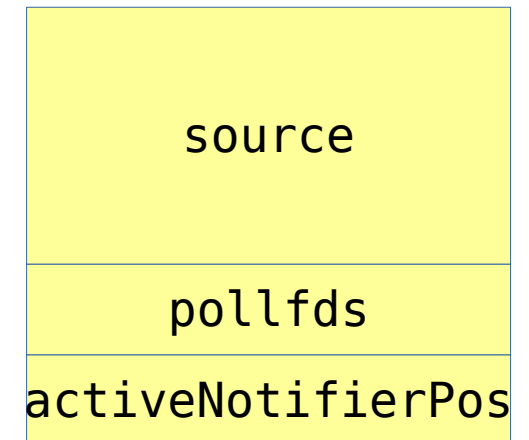
- The type safety still brittle
- How do the library functions accept a custom object of yours?
- That's right, a `void *`. There's no special conversion between a pointer-to-derived and a pointer-to-base, so you need to lose type safety.
 - At least, Glib does this a bit better.

Inheritance in C++

- Explicit syntax
- Memory layout is the same, but that's not so important
- The language guarantees the safety of pointer convertibility: pointers to the derived class automatically convert the pointers to the base class.

```
struct GSocketNotifierSource : GSource
{
    QList<GPollFDWithQSocketNotifier *> pollfds;
    int activeNotifierPos;
};

void fun(GSource *s);
GSocketNotifierSource *object = ... ;
fun(object);
```

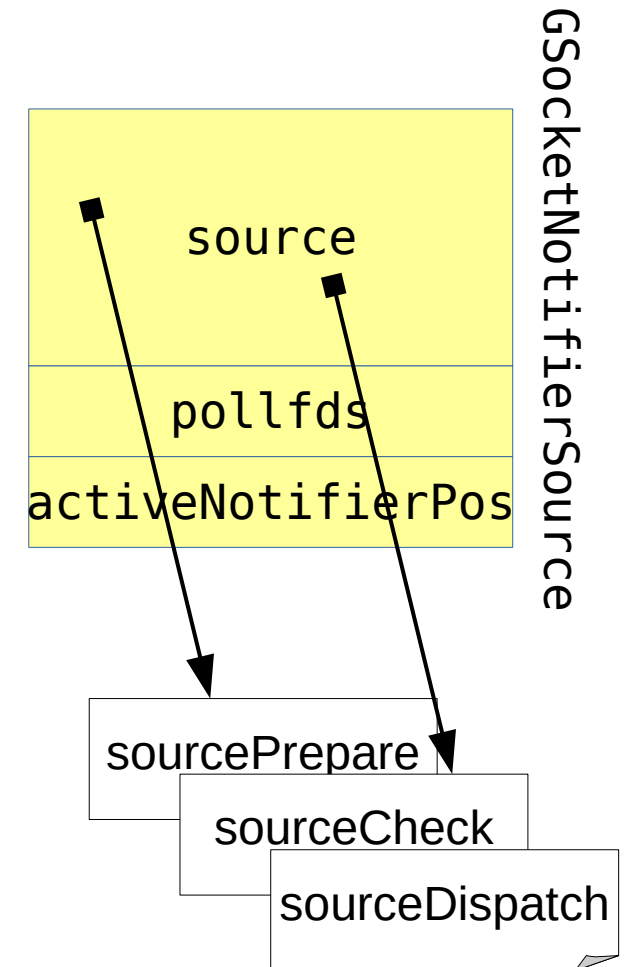


GSocketNotifierSource

Inheritance in C again

- To customize behavior, we *install* pointers to functions:

```
typedef struct
{
    GSource source;
    QList<GPollFDWithQSocketNotifier *> pollfds;
    int activeNotifierPos;
} GSocketNotifierSource;
GSourceFuncs socketNotifierSourceFuncs = {
    socketNotifierSourcePrepare,
    socketNotifierSourceCheck,
    socketNotifierSourceDispatch,
    nullptr,
    nullptr,
    nullptr
};
src = g_source_new(&socketNotifierSourceFuncs,
                 sizeof(GSocketNotifierSource));
```

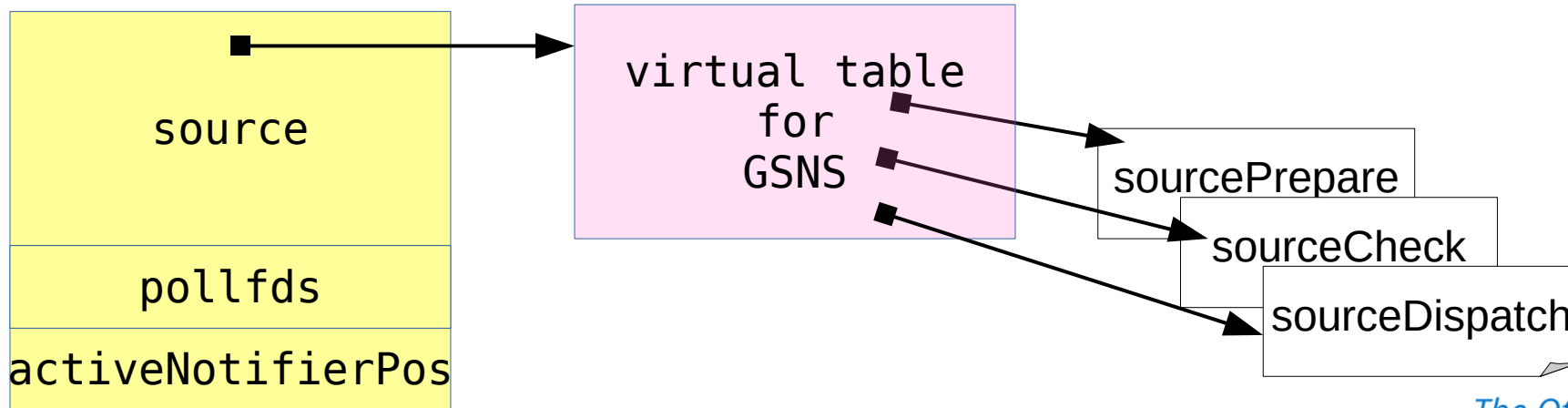


Runtime polymorphism in C++

- The same is achieved via *virtual* functions in C++.

```
struct GSocketNotifierSource : GSource
{
    void sourcePrepare() override;
    void sourceCheck() override;
    void sourceDispatch() override;
    QList<GPollFDWithQSocketNotifier *> pollfds;
    int activeNotifierPos;
};
```

GSocketNotifierSource



Last but not least...

C type system is way too lax.

None of this should compile. Yet it does. It doesn't when using the equivalent C++ facilities. **How many times you hit a bug because of these?**

```
// implicit prototype (may not match the actual function)
undeclared_function(123);

void f(int *);
void *v = malloc(32);
f(v);
short *s = malloc(32);
f(s); // warning, not error

int *sourceBuffer;
double *destinationBuffer;
memcpy(destinationBuffer, sourceBuffer, size*sizeof(int));
```

In C++:

- Conversions are more strict
- Everything must be declared before use
- Algorithms provide more type-safety at same or equal speed

```
undeclared_function(123);    // ERROR. Missing declaration

void f(int *);
void *v = malloc(32);
f(static_cast<int *>(v));    // OK, but must add a cast

short *s = malloc(32);
f(static_cast<int *>(s));    // ERROR: pointer types not compatible

int *source;
double *destination;
std::copy_n(source, size, destination); // OK: copy-converts each int to double
```

Goodies that are in C++ and not in C

- For... no reason. C is sabotaging your productivity!

```
int bigValue = 1'000'000; // grouping

uint mask = 0b000000100; // binary literals

// range based for loop
static const int primes[] = {
    1, 2, 3, 5, 7, 11, 13,
    17, 19, 23, 29, 31, 37
};
for (int p : primes)
    print(p);
```

Keep the conversation open.

- C still dominates embedded development
- But C++ is by far the easiest path forward for existing C developers.
- Don't *jump blindly* into C++.
 - There's no need of abandon all of your experience from C!
 - Adopt only what you need, and when you need it.
- Legends about programming languages will stay with us for a long, long time. Try to understand them for what they are.

Thank you!

Questions?

References

- IEEE Spectrum, The Top Programming Languages
- Embedded.com
- GLib Reference Manual 2.66
- Michael Abrash, The Graphics Programming Black Book
- Dan Saks, "Extern C", CppCon 2016
- Reflections of a Historian on the False News of the War, Marc Bloch