Effective Qt: 2017 Edition

CppCon 2017

Giuseppe D'Angelo
giuseppe.dangelo@kdab.com

The Qt, OpenGL and C++ experts

Agenda

- Qt containers redux
- Implicit sharing
- Clazy
- Qt strings classes
- Bonus slides

1. Understand the Qt containers.
Prefer the Standard Library ones.

Don't use the Qt containers
(unless you have to).

Qt containers

- Qt ships with a set of containers
  - Historical reasons: Qt needed to work on platforms without a STL
  - Qt didn't want to expose Standard Library symbols from its ABI
- Since Qt 5 a “working” STL implementation is required
- Qt containers used in Qt APIs, and available for applications

Qt containers: design philosophy

Qt	Standard Library
Good enough for building GUIs	Truly general purpose
Favors ease of use & discoverability of APIs	Favors efficiency and correctness
Uses camelCase	Uses snake_case

```
QVector<int> v;          // no allocator
v << 1 << 2 << 3;        // chained push_back(s)

QVector<int> v2 = v;     // cheap
if (v2.contains(42))     // algorithm as member function
    doSomething();

assert(!v.isEmpty());    // "isEmpty", not "empty"
```

Qt and the Standard Library: linear containers

Qt	Standard Library
QVector	std::vector
QList	—
QLinkedList	std::list
—	std::forward_list
QVarLengthArray	—
—	std::deque
—	std::array

Qt and the Standard Library: associative containers

Qt	Standard Library
QMap	std::map
QMultiMap	std::multimap
QHash	std::unordered_map
QMultiHash	std::unordered_multimap
—	std::set
—	std::multiset
QSet	std::unordered_set
—	std::unordered_multiset

QVarLengthArray

- QVarLengthArray preallocates space for a given number of objects
- Can avoid hitting the heap
- A vector otherwise
  - “a vector with SSO”
  - Similar: Boost's `small_vector`
- Extremely useful if we know in advance that, most of the time, the container will hold up to a certain number of objects

```
QVarLengthArray<Obj, 32> vector;
```

QList

- An array-backed list
  - *Not* a linked list
- Terribly inefficient if the the object stored are bigger than a pointer
  - Allocates every individual object on the heap
- Avoid using it (unless you have to)
  - See bonus slides
- For your own code, use QVector instead

Qt containers: reasons not to use them

- Qt containers are not actively being developed
- STL containers are faster, expand to less code, and are more tested
- Features are greatly inferior to the STL equivalents
  - Datatypes held in Qt containers must be default constructible and copiable
  - No exception safety guarantees
  - Several C++98 APIs still missing (e.g. range construction/insertion)
  - Most C++11 APIs still missing (e.g. emplacement)
  - All post-C++11 APIs missing (e.g. C++17's node-based APIs)
  - No flexibility w.r.t. allocation, comparison, hashing, etc.
- APIs are inconsistent between Qt containers
  - E.g. there is `QVector<T>::append(T &&)`, but not `QList<T>::append(T &&)`
  - Resize / capacity / shrink behaviors
- APIs have not-so-subtle differences w.r.t. STL containers

Which containers to use?

- For many important metrics, the STL containers are better than the Qt containers
- For this reason, Qt is already using STL containers in its own implementation
- Qt containers are still exposed at the API level
  - Can't change it: Qt has strong API and ABI compatibility promises
- Applications should do the same:
  - Prefer STL containers
  - Use Qt containers if there isn't a STL / Boost equivalent (unlikely)
  - Use Qt containers when interfacing with Qt and Qt-based libraries
- Consider *using* the Qt containers, rather than converting back/forth

Towards Qt 6

- Discussion about what to do with Qt containers in Qt 6 is still ongoing
- They still need to be provided for applications
- An massive API break is not acceptable, so they will still need to be used in Qt APIs
- The big question is what to do with QList
  - It's everywhere in Qt APIs
  - It's not the best linear container
  - QList might simply become a typedef for QVector, and a new type (QArrayList?) introduced

2. If you use Qt containers, remember to use Q_DECLARE_TYPEINFO.

Type traits for Qt containers

- Qt uses type traits to optimize handling of data types in its own containers
- The most important optimization is:
  when growing an array of objects, is it OK to use `realloc`?
  - Safe to do iff the type is *relocatable*
  - Huge optimization gain over allocating a new buffer; moving elements; deallocating the old buffer
- Many types are relocatable and could benefit from this optimization
  - E.g. most Qt value classes, thanks to pimpl

Relocatability

- Is this type relocatable?

```
struct IntVector {
    size_t size, capacity;
    int *data;
};
```

- Yes
  - (assuming a “reasonable” implementation)

Relocatability

- Is this type relocatable?

```
struct Pimpld {
    struct Impl *d;
};
```

- *Depends*
- The pimpl may or may not have a pointer back to the “public” class
- If it has a link back, the type is *not* relocatable

Relocatability

- Is this type relocatable?

```
struct TreeNode {
    T data;
    TreeNode *parent;
    TreeNode **children;
};
```

- No: the address of a `TreeNode` is its *identity*
- Moving an object in memory would break pointers from other nodes

Relocatability

- Is this type relocatable?

```
struct String {
    size_t size, capacity;
    char *begin;
    char data[32];
};
```

- If `data` is a short string optimization buffer, then *no*: `begin` may point into `data`
  - Moving an object in memory could break it
- If `data` is used for some ancillary data, then *yes*

Relocatability: author action is needed

- The compiler cannot tell whether a type is relocatable or not
- Type authors must annotate relocatable types by using type traits
- Some libraries let authors add these traits:
  - Qt → `Q_DECLARE_TYPEINFO`
  - EASTL → `EASTL_DECLARE_TRIVIAL_RELOCATE`
  - STL → **crickets**

Qt type traits for containers

- Type traits for a given type are declared using `Q_DECLARE_TYPEINFO(Type, Kind)`, where `Kind` is:
- `Q_PRIMITIVE_TYPE`
  - Every bit pattern is a valid object
  - No need to call constructors or destructors, can `reinterpret_cast` objects from raw memory
- `Q_MOVABLE_TYPE`
  - Objects are relocatable: can be moved in memory using `memmove` / `realloc`
  - (Non copy/move) constructors and destructors still called
- `Q_COMPLEX_TYPE`
  - Default: call constructors, copy/move constructors, destructors

Qt type traits: recommendations

- Every time you define a type that you may end up using in a Qt container, remember to declare its typeinfo

```
struct IntVector {
    int size, capacity;
    int *data;
};
Q_DECLARE_TYPEINFO(IntVector, Q_MOVABLE_TYPE);
```

- Adding a trait “after the fact” is possible, but it's a potential ABI break

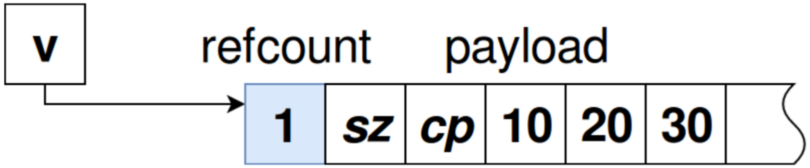
3. Understand implicit sharing, and be careful about hidden detaches.

Implicit sharing?

- Fancy name for reference counting combined with copy on write
- A Qt value class implementation is typically just a pointer to a pimpl, which contains the reference counter and the actual payload
- Reference counter is manipulated during an object's lifetime:
  - On object creation: refcount is 1
  - Copying an object: refcount is incremented by 1
  - Destroying an object: refcount is decremented by 1; if it reaches zero, deallocate the pimpl
  - Calling a const member function: (nothing)
  - Calling a non-const member function: if the refcount is greater than 1, then **detach** (= deep copy the the payload)

Implicit sharing in action

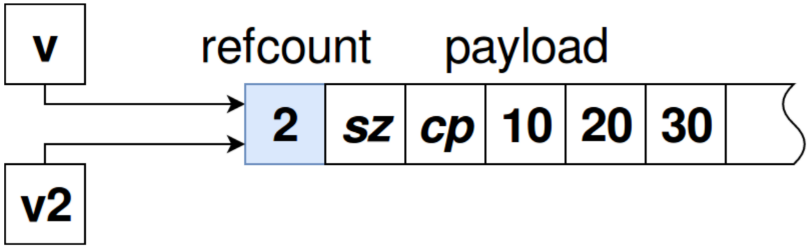
```
QVector<int> v {10, 20, 30};
```



Implicit sharing in action

```
QVector<int> v {10, 20, 30};
```

```
QVector<int> v2 = v;
```

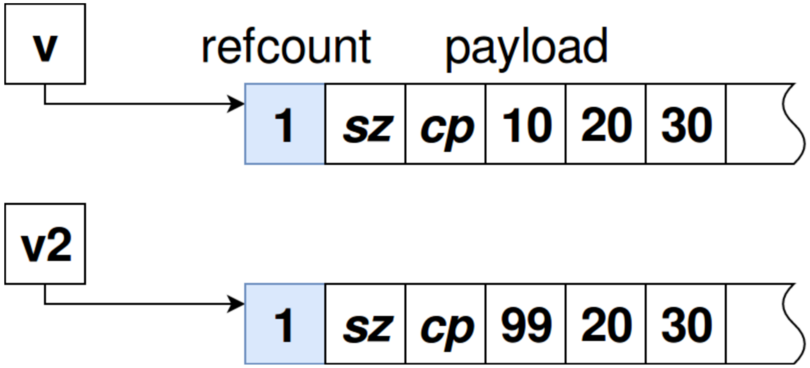


Implicit sharing in action

```
QVector<int> v {10, 20, 30};


QVector<int> v2 = v;


v2[0] = 99;
```



Implicit sharing

- This mechanism makes writing code a lot simpler
  - Take copies, return by value, etc. without thinking twice
- The great majority of Qt value classes are implictly shared
  - Containers (notable exception: QVarLengthArray)
  - QString
  - QByteArray
  - QVariant
  - etc.

Implicit sharing and containers: where's the catch?

- Handing out references to data inside a container does not make the container unshareable
- It's easy to accidentally detach a container
- Accidental detaching can hide bugs
  - IOW, it's not just about performance
- Code polluted by (out-of-line) detach/destructor calls

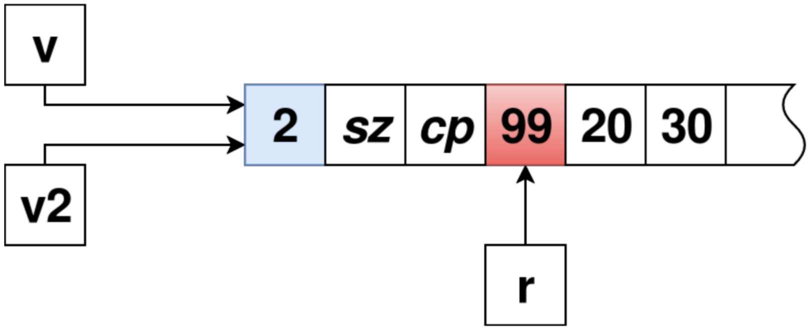
Returning references to data inside a container

- Handing out references to data inside a container does not make the container *unshareable*
  - E.g. of such references: iterators
- Correctness/speed trade off

```
QVector<int> v {10, 20, 30};
auto &r = v[0];

QVector<int> v2 = v;

r = 99;                    // oops!
assert(v2[0] == 10);   // fails
```



Accidental detaches

- "Innocent" code may hide unwanted detaches:

```
QVector<int> calculateSomething();

const int firstResult = calculateSomething().first();
```

- Calls: `T& QVector<T>::first();`
  - Non-const, may detach and deep copy!
- Solution is easy: call `constFirst()`
- Not easy to spot
- Usually appears in heap profilers (heaptrack, massif)

Accidental detaches (2)

- Accidental detaches can actually introduce bugs:

```
QMap<int, int> map;
// …
if (map.find(key) == map.cend()) {
    std::cout << "not found" << std::endl;
} else {
    std::cout << "found" << std::endl;
}
```

- `find(key)` might detach after the call to `cend()`, returning an iterator pointing to a “different” end
- “`found`” is printed, even if the key isn't in the container
- Solution: use `constFind(key)`, don't mix `iterators` and `const_iterators`

Implicit sharing: a double-edged sword

- Definitely good convenience, but beware of what you're doing
  - See later for a solution to some of these problems
- The position of the Standard Library is clear: move away from implicit sharing
  - Actually, *forbid* it
- Qt however will not move away

4. Never use Qt's foreach / Q_FOREACH; use C++11's range-based for. (Be careful with Qt containers.)

foreach / Q_FOREACH

```
foreach ( var, container ) body
```

means:

```
{
    const auto copy = (container);
    auto i = copy.begin(), e = copy.end();
    for (; i != e; ++i) {
        var = *i;
        body
    }
}
```

foreach / Q_FOREACH: pros and cons

- “Pro”: it's always safe to modify the container from the body
  - But don't do it! It makes it extremely hard to reason about the loop
- Con: no mutation possible
  - We are iterating over a const copy
- Con: the container is always copied
  - Cheap if it's a Qt container
  - Expensive and unacceptable if it's a STL container

foreach / Q_FOREACH: wrap up

- Don't use Qt's foreach
- Disable its usage in your code base by defining `QT_NO_FOREACH`
- It will extremely likely be removed in Qt 6

Range-based for loop

```
for ( var : container ) body
```

means:

```
{
    auto &&c = (container);
    auto i = begin(c);
    auto e = end(c);
    for (; i != e; ++i) {
        var = *i;
        body
    }
}
```

- What happens if `container` is `std::vector<T>`?
- `i`, `e` will be mutating iterators: `std::vector<T>::iterator`

Range-based for loop

```
for ( var : container ) body
```

means:

```
{
    auto &&c = (container);
    auto i = begin(c);
    auto e = end(c);
    for (; i != e; ++i) {
        var = *i;
        body
    }
}
```

- What happens if `container` is `QVector<T>`?
- `i`, `e` will be mutating iterators: `QVector<T>::iterator`
- **Possible detach!** Even if we don't actually modify the container through the iterator (in the body of the loop).

Range-based for loop

```
for ( var : container ) body
```

means:

```
{
    auto &&c = (container);
    auto i = begin(c);
    auto e = end(c);
    for (; i != e; ++i) {
        var = *i;
        body
    }
}
```

- What happens if `container` is `const std::vector<T>` or `const QVector<T>`?
- `i`, `e` will be non-mutating iterators: `std::vector<T>::const_iterator` `QVector<T>::const_iterator`
- No detach

Qt foreach vs range-based for loop: summary

Container	Q_FOREACH (const auto &v, c)	for (auto & : c)	for (const auto &v: c)
Qt	OK (cheap)	OK (detach)	Possible detach
const Qt	OK (cheap)	—	OK
STL	**Deep copy**	OK	OK
const STL	**Deep copy**	—	OK

Range-based for loop: conclusions

- Be careful when using non-const Qt containers
- If you are not mutating the container, make the container const
  - `std::as_const(container)` (C++17)
  - `qAsConst(container)` (Qt 5.7)
  - Don't work with rvalues; capture a const-ref in that case

```
QVector<int> vector;
// ...
for ( const auto& v : qAsConst(vector) ) {
    // non-mutating loop
}

const QVector<int> &vector2 = buildVector();
for ( const auto& v : vector2 ) {
    // ditto; cannot just write qAsConst(buildVector())
}
```

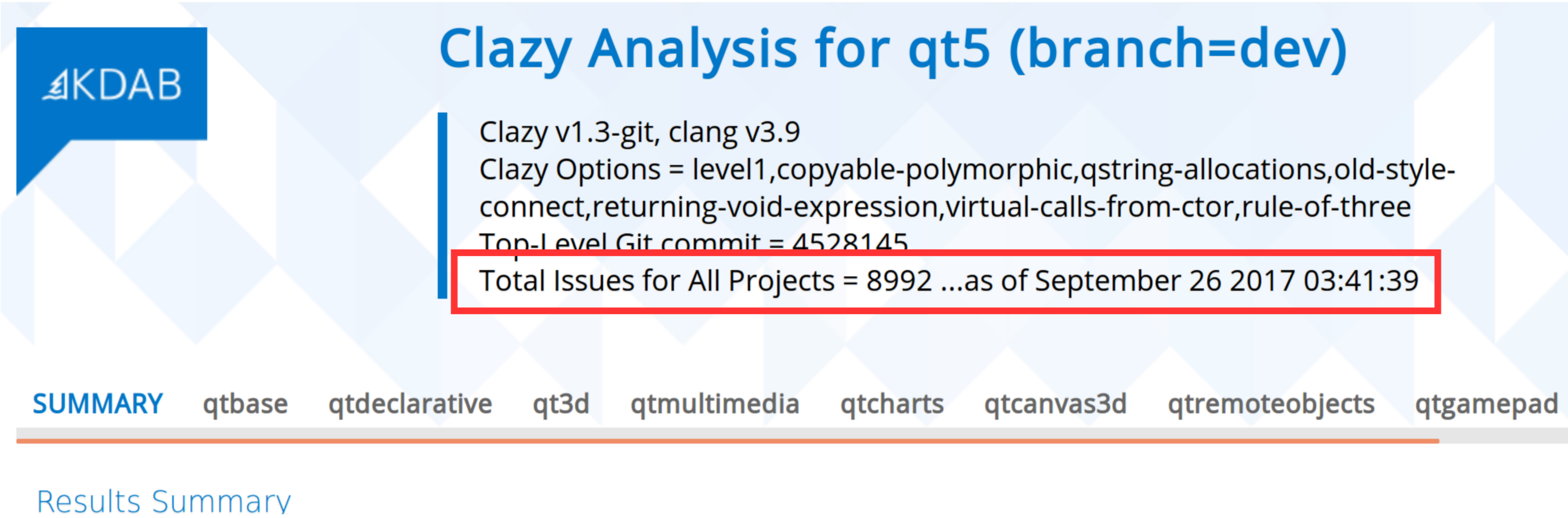
5. Run clazy on your code base, and fix its warnings.

clazy

- An opensource clang-based tool to detect mistakes when using Qt.
  - Akin to clang-tidy
- Ships with 50+ checks, some of which with fix-its for automatic refactoring
- Some of the checks detect the mistakes on implicit sharing we have just discussed:
  - `detaching-temporary`
  - `strict-iterators`
  - `missing-typeinfo`
  - `foreach`

clazy

- Set up runs of clazy over your code base and and fix its warnings
- Even Qt itself is not immune from mistakes:



6. Understand Qt string classes. Embrace QStringView.

In how many ways I can create a string in Qt?

1) “string”

2) QByteArray(“string”)

3) QByteArrayLiteral(“string”)

4) QString(“string”)

5) QLatin1String(“string”)

6) QStringLiteral(“string”)

7) QString::fromLatin1(“string”)

8) QString::fromUtf8(“string”)

9) tr(“string”)

10) QStringView(u”string”)

QByteArray

- A sequence of bytes
- No encoding specified
  - Akin to `std::string`
- Implictly shared
- Its constructors allocate memory
  - QByteArray::fromRawData() to avoid (some) allocation
- `QByteArrayLiteral("string")` never allocates
  - Since Qt 5.9, this is true on all supported platforms
- Use it to store byte arrays (i.e. data)

QString

- A UTF-16 encoded Unicode string
  - Support for Unicode-aware manipulations, unlike `std::u16string`
- Implictly shared
- Its constructors allocate memory
  - Including `QString::fromUtf8()`, `QString::fromLatin1()`
- Clutch: `QString::fromRawData()` as non-allocating constructor
  - Prefer QStringView
- `QStringLiteral("string")` never allocates
  - Since Qt 5.9, this is true on all supported platforms
  - Data is stored UTF-16 encoded in the readonly data segment
- Use it to store Unicode strings

QLatin1String

- A literal type that wraps a `const char *` and a size
  - It doesn't manage anything
- Mostly used in overloads when there's a fast-path implementation possible for Latin-1 strings, and they come from string literals:
  - E.g. substring search:

```
int QString::startsWith(const QString &substring);
int QString::startsWith(QLatin1String substring);

QString str = "...";
if (str.startsWith(QString("foo")))  // allocates a temp. QString
    doSomething();
if (str.startsWith(QLatin1String("foo"))) // does not allocate + uses
    doSomething();                         // optimized implementation
```

Qt string classes

- There hasn't been much development around QString / QByteArray in the last few years
- The only important change that happened is that since Qt 5.9 QStringLiteral / QByteArrayLiteral never allocate memory
- Fore more information on the existing Qt string classes, refer to the MeetingC++ 2015 version of this talk

QStringView

- New in Qt 5.10
- A non-owning view over a UTF-16 encoded string:
  - `QString`
  - `QStringView`
  - `std::u16string`
  - Array and `std::basic_string` of `QChar`, `ushort`, `char16_t`, `wchar_t` (on Windows)
- Literal type; akin to C++17's `std::u16string_view`
- Offers the majority of the const QString APIs, without the need of constructing a QString first
  - More APIs, QStringBuilder support etc. expected in 5.11

QStringView as an interface type

- The primary use case for QStringView is for functions parameters
- If a function needs a Unicode string, and it doesn't store it, use QStringView

QStringView as an interface type

- Consider

```
class Document {
    iterator find(StringType substring);
};
```

- What type should `StringType` be?
- QString
  - Forces either a compile-time string (via QStringLiteral), or a dynamically allocated string (maybe allocated *just* for this function call)
- QByteArray
  - Not Unicode safe; same problems as QString
- QLatin1String
  - Not Unicode safe
  - But it makes sense as an additional overload if we can implement a fast path for Latin-1

QStringView as an interface type

- Solution:

```
class Document {
    iterator find(QStringView substring);
};
```

- QStringView
  - Unicode safe
  - **Never** allocates
  - Can be built from a wide variety of sources

```
Document d;
d.find(u"compile time");              // compile time literal

QString allocatedString = "...";
d.find(allocatedString);              // dynamically allocated

d.find(QStringView(bigString, 40)); // substring of big string
```

QStringView as an alloc-free tokenizer

- To extract substrings, without allocating memory
- Example: `QRegularExpressionMatch::capturedView()`:

```
QString str = "...";
QRegularExpression re("reg(.*)ex");
QRegularExpressionMatch match = re.match(str);

if (match.hasMatch()) {
    QStringView cap = match.capturedView(1); // no allocations
    // ...
}
```

QStringView: a game changer

- A huge number of APIs in Qt take a QString and don't need to store it
- In Qt 6, they should all be changed to take a QStringView instead
  - ... any volunteers?
- In Qt 5 there's also QStringRef as a non-owning string view
  - However, it's an API mistake: creating a QStringRef always require a QString (and not just *any* sequence of UTF-16 code units)
  - Use it if you “can't wait”
  - QStringRef will get deprecated as soon as QStringView reaches API parity

Questions?

Thank you!

giuseppe.dangelo@kdab.com

Bonus slides

Does POD mean relocatable?

- Relocability is independent from being POD
- Relocatable types may have non-trivial constructors/destructors
  - E.g. Qt pimpl'd value classes
- A trivial type may not be relocatable
  - E.g. if the address of an object is its identity
  - All C data types are trivial, but non necessarily relocatable

7. Dont' use deprecated Qt APIs.

Deprecated APIs in Qt

- As new APIs get introduced in Qt, older ones may get deprecated
- Due to the API compatibility promise, Qt cannot just remove them
  - Deprecated APIs still work, pass tests, etc.
- Qt 6 will remove most of the deprecated APIs, so start porting away from them!

Deprecated APIs in Qt

- A deprecatation mechanism exists in Qt, but it's opt-in
- Deprecated APIs are tagged in Qt's source code with a deprecation warning and a deprecation version:

```
#if QT_DEPRECATED_SINCE(5, 2)
template <typename RandomAccessIterator>
QT_DEPRECATED_X("Use std::sort")
inline void qSort(RandomAccessIterator start,
                  RandomAccessIterator end) { … }
#endif
```

How to disable deprecated APIs

- Always define `QT_DEPRECATED_WARNINGS`
  - Makes the compiler emit warnings if using deprecated APIs
- Define `QT_DISABLE_DEPRECATED_BEFORE` to the version of Qt you develop against
  - Turns usage of deprecated APIs into hard errors, iff they have been deprecated in that Qt version or in a earlier one
  - No “new” errors if you upgrade Qt
  - E.g. in qmake: `DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x050900`

8. Never use QList for your own code.

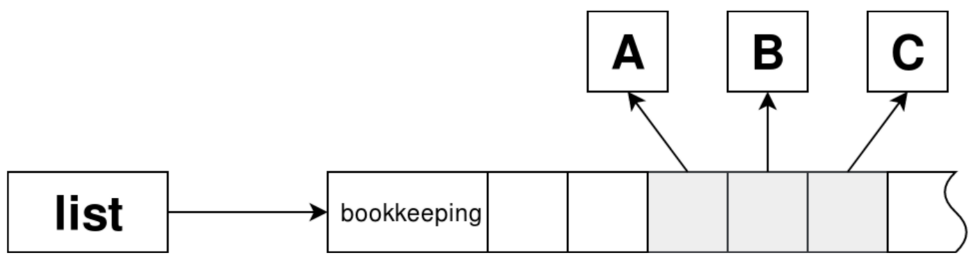
QList

- An “array-backed” list
- QList always manages an array of `void *`
- QList is a strange hybrid: depending on the type held, the array holds
  - pointers to the elements (individually allocated on the heap)
  - the elements themselves
- The array has some room at the front
  - Optimization for prepend

QList: array of pointers mode

- Given a type T for which either
  - `sizeof(T) > sizeof(void *)`; or
  - T is not relocatable (default for user types)

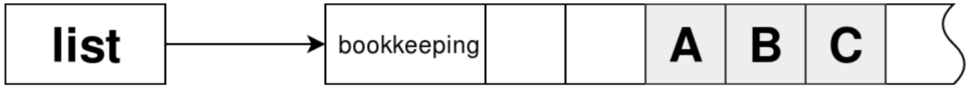
  then QList<T> stores pointers to objects of type T; the objects are allocated on the heap (via `operator new`):



QList: vector mode

- Given a type T for which both hold:
  - `sizeof(T) <= sizeof(void *)`; and
  - T is relocatable (requires type traits)

then QList<T> stores objects of type T directly in its backing array:



QList

- QList design tries to minimize code expansion over speed
  - Type-unaware management of the backing array: always an array of `void *`
- Doing an allocation per element is definitely a huge pessimization
- The double nature of QList is surprising
  - Difficult to say what behavior a given datatype triggers
  - Changes across 32/64 bits
  - Waste of space for small, relocatable datatypes (e.g. int on a 64 bit platform)
- Unfortunately QList is the most common container exposed by Qt APIs

QList or QVector?

- Use QVector
  - Unless the purpose is calling Qt APIs taking QList, of course
- These days QVector expands to less code than QList
- QVector is faster in almost any operation, except for:
  - Frequent insertions in the front
  - Reallocation with really big objects
- QVector does not maintain integrity of references after reallocation
  - If you really need this, use a vector of pointers, to express intent!