Packing Structs

Optimizing the memory layout of C++ data structures

Volker Krause
vkrause@kde.org
@VolkerKrause

Memory Layout

- Data members are laid out sequentially in declaration order
- Each data member occupies `sizeof(T)` bytes
- Each data member is aligned to `alignof(T)`
- Alignment of a composite type is the maximum alignment of its data members
- Data members of derived classes follow the base class data members
- Virtual inheritance is nasty

Example

```
struct S {
    bool m1;
    int m2;
    bool m3;
};
```

Example

```
struct S {
  bool m1; // size: 1, alignment: 1
  // 3 bytes padding
  int m2; // size: 4, alignment: 4
  bool m3; // size: 1, alignment: 1
  // 3 bytes padding
}; // size: 12, alignment: 4
```

Example

```
struct S {
  int m2; // size: 4, alignment: 4
  bool m1; // size: 1, alignment: 1
  bool m3; // size: 1, alignment: 1
  // 2 bytes padding
}; // size: 8, alignment: 4
```

Tools

- GCC/-Wpadded
  - too noisy
- dwarves/pahole
  - fails on C++ code
- elf-dissector/elf-packcheck (kde:elf-dissector)
  - fails on virtual inheritance
- sizeof/alignof and static_assert

Avoid Padding

- Rule of thumb: order members by alignment
- Keep alignment of base class in mind
  - `sizeof(QSharedData) == 4`
- When optimizing the memory layout, consider:
  - 32bit vs. 64bit architectures
  - compile-time conditionals

Tricky Cases

```
template <typename Key, typename T>
struct QHashNode {
  uint hash;
  Key key;
  T value;
};
```

- Use enable_if to swap order for `alignof(T) <= 4`

Byte Layout

- Reduces memory waste
- Increases cache utilization
- Minimal impact on maintainability, apart from tricky template cases.
- Can we do more?

Information Theory

- How much “information” is actually in the data we store?
- Example: bool
  - holds 1 bit of information
  - needs 8 bit storage
- Example: QObject* on 64 bit architecture
  - holds 61 bit of information (due to 8 byte alignment)
  - needs 64 bit of storage

Bit Layout

- Bit fields: `struct{ uint a:31; bool b:1; }`
- Manual bit twiddling
- `std::vector<bool>`, `QBitArray`, `QBitField`, …
- `enum class E : uint8_t { … };`

- Incurs some extra CPU cost
- Pointers/references don't work on a sub-byte level!
- elf-packcheck can measure sub-byte utilization

Dirty Tricks

- Bypass alignment rules
  - #pragma pack, __attribute__((__packed__))
  - incurs performance penalty
  - SIGBUS on non-x86
- Use the pointer alignment gap
  - log2(`alignof(T)`) bits available
  - Hard to maintain manually, breaks type-safety checks
  - See `QFlagPointer`, `QBiPointer`

The Dark Side

- ABI == memory layout
- Memory layout can impact:
  - CPU cost
  - MT cache conflicts
  - portability
  - maintainability
  - extensibility

Conclusion

- Avoid unnecessary padding
- Think about what information content you need to store
- Consider tweaking the sub-byte layout for high-volume classes
- No replacement for allocating less instances where possible

Questions?

References

- Slides: http://www.kdab.com/~volker/akademy/2015/
- Code: git.kde.org:elf-dissector.git