Fachhochschule Nordwestschweiz
Hochschule für Technik

Programmieren in C++

Christian Lang (Lac)

25. Oktober 2019

Spezielle Memory Konzepte

Inhalt

- Bit-Zugriff
- Bit-Felder
- Memory-Alignment

Bit-Zugriff

in C mittels maskieren und schieben

```
1 uint8_t reg = 37;
2 bool bit2 = (reg >> 2) & 0x01;
3 reg = static_cast<uint8_t>(reg & ~(0x01 << 2));
```

in C++ mittels `std::bitset`

```
1 uint8_t reg = 37;
2 std::bitset<8> access = reg;
3 bool bit2 = access[2];
4 access[2] = false;
5 reg = static_cast<uint8_t>(access.to_ulong());
```

Einschränkungen

- Kopieren nötig
- keine Felder mit mehr als einem Bit möglich

Bit-Felder

- Felder in `struct` können auf Anzahl Bits definiert werden
- Padding kann eingeschränkt kontrolliert werden

```
1  struct BitField {  // üblicherweise 2 Bytes gross
2    // 3 bits: value of b1
3    // 5 bits: unused
4    // 6 bits: value of b2
5    // 2 bits: value of b3
6    uint8_t b1 : 3;
7    uint8_t :0; // start a new byte
8    uint8_t b2 : 6;
9    uint8_t b3 : 2;
10 };
```

Einschränkungen

- Memory-Layout nicht im Standard
- keine Pointer/Referenzen möglich/sinnvoll

Bit-Felder

Implementations-abhängig:

- Packing
- Overflow-Handling

Andere Fakten:

- Feld Typ soll immer gleich sein
- Referenzen auf Feld erzeugt Temporary mit Kopie des Wertes
- Es können mehr Bits definiert werden als der Typ hat. Wertebereich ist aber trotzdem durch Typ definiert. Restliche Bits sind Padding.

Memory-Alignment

- Instanzen werden immer auf optimale Adressen platziert
- Memory-Layout von Klassen wird mittels Padding optimiert
- hängt beides von Alignment-Grösse ab
- diese hängt von Hardware (CPU-Architektur) ab
- genaues Handling ist Implementations-abhängig

```
1  int main() {
2    uint8_t a;
3    uint32_t b;
4    uint64_t c;
5  }
```

Adressen erfüllen:

Alignment: 1 Byte	Alignment: 4 Bytes
`&a % 1 == 0`	`&a % 4 == 0`
`&b % 4 == 0`	`&b % 4 == 0`
`&c % 8 == 0`	`&c % 8 == 0`

Memory-Alignment in Klassen

- Klassen werden mit Padding optimiert
- Optimierung für Zugriff (in Arrays) wichtig

```
1  struct Data {
2    uint8_t  a;   // 1 Bytes + 3 padding Bytes
3    uint32_t b;   // 4 Bytes
4    uint16_t c;   // 2 Bytes + 2 padding Bytes
5  };
```

übliche Regeln

1. Das nächste Feld ist "aligned" mit seiner eigenen Grösse.
   → Das vorherige Feld wird "gepadded".
2. Die ganze Klasse ist so gross wie ein Vielfaches seines grössten Feldes.
   → Das letzte Feld wird "gepadded".

Memory-Alignment kontrollieren

- Alignment kann mittels `alignas` kontrolliert werden
- Alignment kann mittels `alignof` überprüft werden
- Padding von Klassen kann mittels `#pragma pack` kontrolliert werden

```
1  #pragma pack(2) // Padding nur auf 2 Bytes
2  struct C {
3    uint8_t  a;   // 1 Bytes + 1 padding Bytes
4    uint32_t b;   // 4 Bytes
5    uint16_t c;   // 2 Bytes
6  };
7  #pragma pack()  // auf Default zurück stellen
```

Verwendung

- üblicherweise nicht beeinflussen
- kann bei Netzwerk-Traffic interessant sein

Placement `new`

- "normales" `new` alloziert Speicher und initialisiert Instanz
- Placement `new` kann verwendet werden, wenn Speicher bereits vorhanden
- nur noch Initialisierung → expliziter Aufruf des Konstruktors

```
1  struct SomeClass {
2    SomeClass() : a(1), b(42) {}
3    int a;
4    float b;
5  };
6
7  void* memory = malloc(sizeof(SomeClass));          // alloziert uninit. Speicher
8  assert(memory != nullptr);
9  const auto* instance = new (memory) SomeClass();   // ruft ctor auf
10 // instance->a == 1
11 // instance->b == 42
```

Best-Practice

Selten verwendet → z.B: für Container-Entwickler