Grundlagen C++

Themen dieser Vorlesung

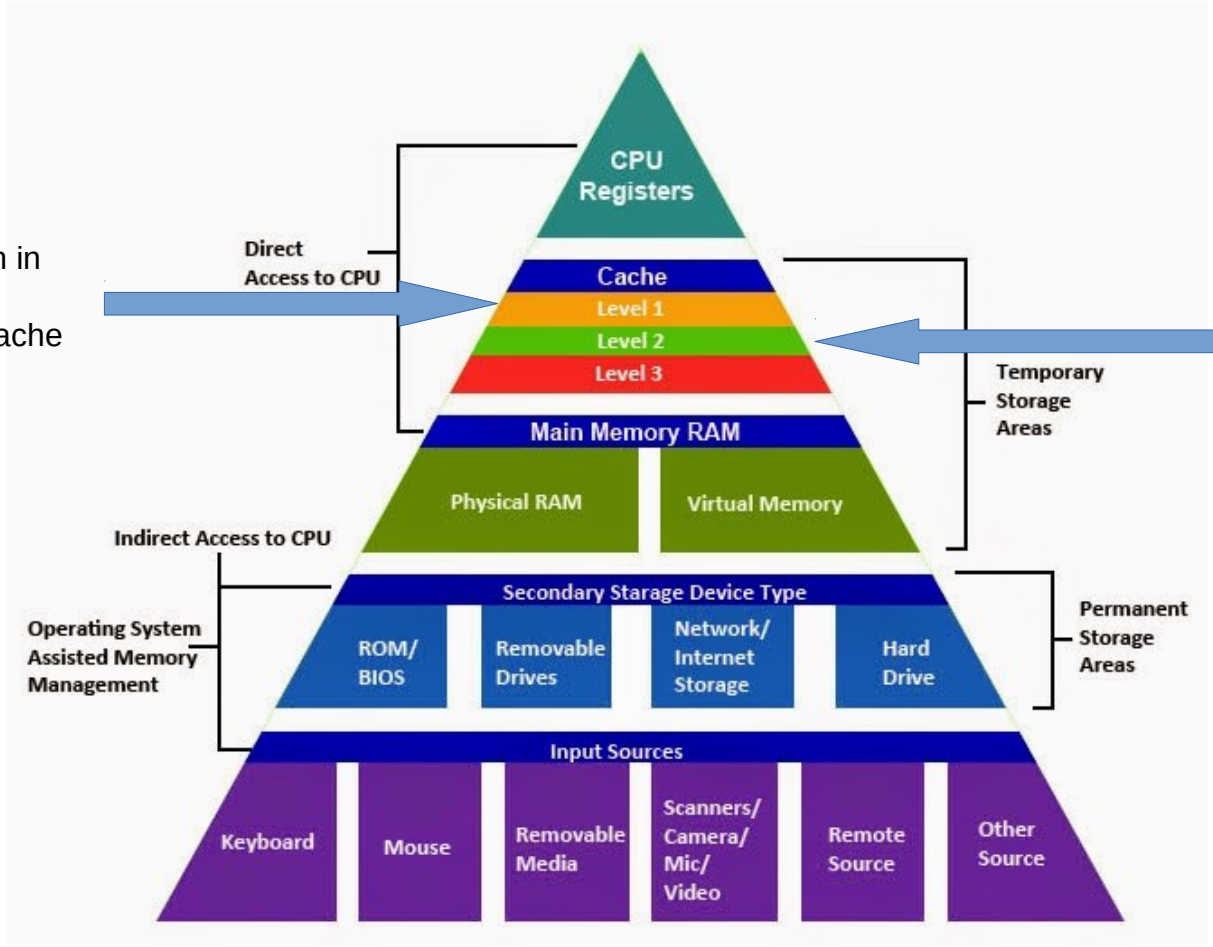
- Hardware Grundbegriffe
- Compiler Grundbegriffe
- Einführung in die C++ Syntax

© M. Sc. Tobias Gerteis

Hardware Grundbegriffe

Hardware Grundbegriffe

Unterteilt sich in Data und Instruction Cache



Ab hier gibt es Probleme beim Multithreading

Quelle: http://liveforge.org/wp-content/uploads/2017/04/Memory-Hierarchy.jpg

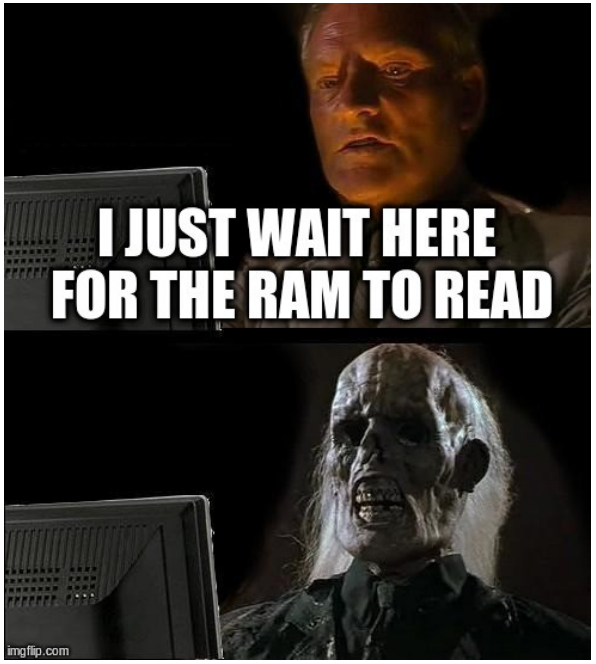
Hardware Grundbegriffe

Intel Haswell

L1 Cache 2x32 KB

L2 Cache 256 KB

RAM



CPU

Hardware Grundbegriffe

One cycle on a 3 GHz processor	1 ns
L1 cache reference	0.5
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	25
Main memory reference	100
Compress 1K bytes with Snappy	3,000
Send 1K bytes over 1 Gbps network	30,000
Read 4K randomly from SSD	150,000
Read 1 MB sequentially from memory	250,000
Round trip within same datacenter	500,000
Read 1 MB sequentially from SSD	1,000,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	20,000,000

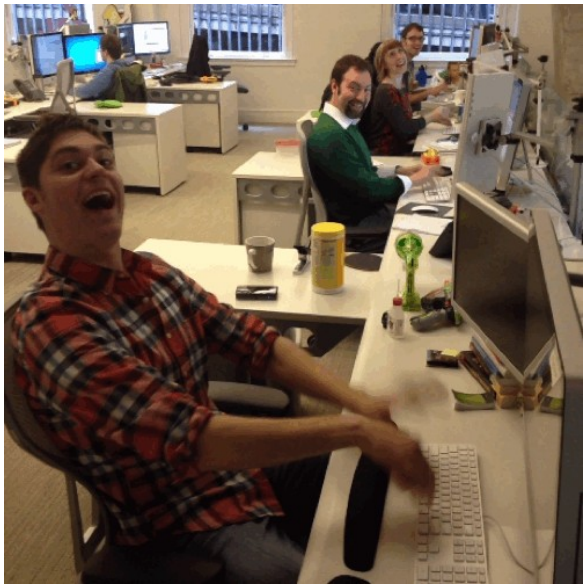
Hardware Grundbegriffe

- Wenn Daten im RAM sind, kann es schon zu lange dauern!
- Keep it **small** and simple → bessere Lokalität im Programmspeicher
- Datenanordnung (alignment) ist essentiell

Compiler Grundbegriffe

Compiler Grundbegriffe

- Wir programmieren für den Compiler!!



Compiler Grundbegriffe

- Compiler kennt bereits viele Optimierungen
- Keep it **small** and **simple**
  - Small → Speicher für Optimierungen ist begrenzt
  - Simple → Compiler kann Code den er versteht besser optimieren
- Versuche nicht schlauer als der Compiler zu sein

Compiler Grundbegriffe

- Flags (gcc, clang)
  - -O2 → Optimierungen ohne Tradeoffs
  - -O3 → Optimierungen auf Geschwindikeit auf Kosten der Größe
  - -Os → Optimierungen auf Größe auf Kosten der Geschwindikeit
  - -Ofast → Volle Optimierung auf Geschwindikeit, möglicherweise nicht ISO-konform
  - gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Compiler Grundbegriffe

- Beispiel Retrun-Value Optimisation (RVO)
- https://godbolt.org/g/UJ7DCj
  - Ermöglicht dem Compiler Funktionen zu „inlinen“
  - Nicht immer möglich
  - C / Java etc. Können das nicht!!!

Einführung in die C++ Syntax

Einführung in die Syntax

- Was ist gleich wie bei C
  - Programm beginnt bei Main
  - Trennung von Deklaration (Header) und Implementierung (Source)
  - Datentypen von C: char, short, int, long, float, double, pointer, array, struct, union, ...
  - Operatoren und deren Wertigkeit
  - Scopes und lifetimes
  - Programmablauf mit Schleifen (for, while, do-while), Bedingungen (if, switch) und Sprungbefehle (continue, break, return)
  - Präprozessor (#define, #ifdef, #pragma, …)
  - Standartfunktionen: sizeof, printf, etc.

Einführung in die Syntax

- Klassen und Objekte
- Referenzen
- Strings und Standardausgaben
- Wichtige Neuerungen in C++ 11 (auto, for_each, smart-pointers)
- Funktionsobjekte / Lambda
- Container
- Templates

Klassen und Objekte

- Klassenbeschreibung ähnlich wie bei Java
  - Konstruktor wird mit Klassennamen angegeben
  - Private, Protected und Public
- Destruktor wird mit ~ angeführt
- Funktionsimplementierung in der Source-Datei wird mit *Klassenname::Funktionsname* angegeben
- Initial Werte für Membervariablen werden in einer Initialisierungsliste im Konstruktor angegeben, oder seit C++11 auch in der Deklaration

Klassen und Objekte

Header:

```
class Example
{
public:
    Example();          // default Konstruktor
    Example(int value); // optionaler Konstruktor
    int getValue() const; //getter

private:
    int myValue = 0;    //nur ab C++11
}
```

Source:

```
Example::Exmaple() : myValue(0)
{
}

Example::Exmaple(int value) : myValue(value)
{
}

Example::~Example()
{
}

int Example::getValue()
{
   return myValue;
}
```

Klassen und Objekte

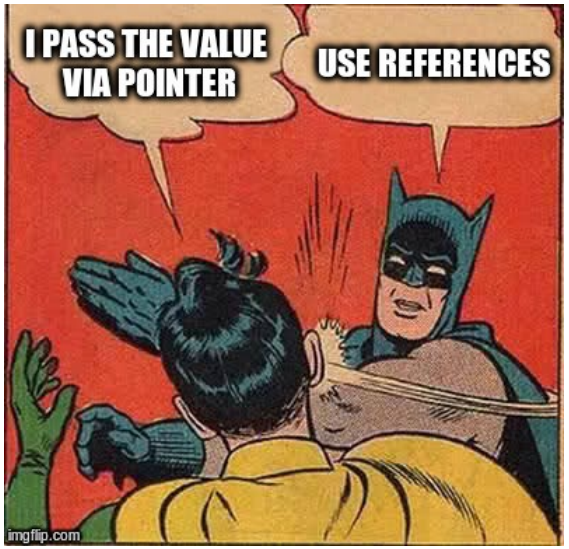
- Objekte werden durch den Konstruktoraufruf initialisiert
  - Mit *new* wird das Objekt auf dem Heap erzeugt und ein Pointer wird zurückgegeben
  - Ohne *new* wird das Objekt auf dem Stack erzeugt
- Mittels *delete* wird das Objekt gelöscht und der Speicher freigegeben
  - Ist nur notwendig, wenn das Objekt auf dem Heap ist
- Memberfunktionen und Variablen werden mittels „.“ (auf dem Stack) und → auf dem Heap aufgerufen

Klassen und Objekte

```
int main()
{
  Example e1 = Example();
  Example* e2 = new Example(5);
  int i1 = e1.getValue();
  int i2 = e2->getValue();
  delete e2;
}
```

Referenzen

- Pointer sind hilfreich aber gefährlich
  - Pointerarithmetik
  - Notwendig bei Erzeugung neuer Objekte
  - Bei Funktionsübergabe, keine Kontrolle ob der Zeiger verändert wird
- Referenzen sind Pointer ohne Arithmetik
  - Werden mit *&* angegeben
  - Membervariablen werden mit . aufgerufen
  - Referenzen können nicht *NULL* sein
  - Vor allem bei „Call by Reference“ extrem hilfreich



Referenzen

```
int b = 1;   // eine andere Variable
int* p = &b; // Ein Pointer auf die Adresse von b

int  a = 1;  // eine Variable
int &r = a;  // Referenz auf die Variable a
```

```
void swap(int &wert1, int &wert2)
{
    int tmp;
    tmp   = wert1;
    wert1 = wert2;
    wert2 = tmp;
}
```

Strings und Standardausgaben

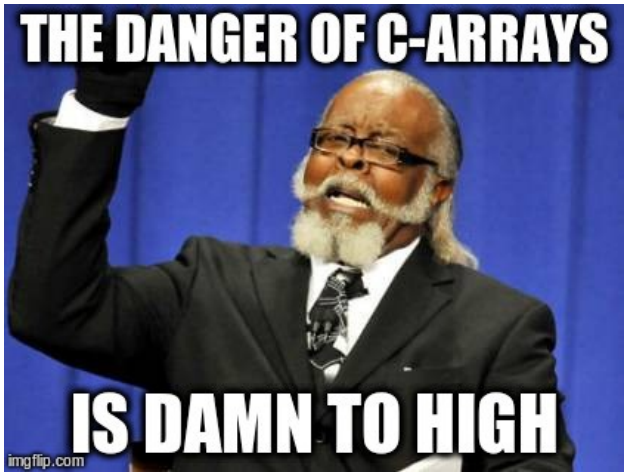
- *std::string* als Ersatz für *char[]*
  - Typensicher
  - Bound-checking
  - Zusatzfunktionen wie *substring, compare*
  - Konkatenation mit +
- *std::cout* als Ersatz für *printf*
  - Verwendbar mit char[ ],strings, integer, short etc.
  - Werte können mit << aneinandergehängt werden
  - Std::endl für Ausgabe des Puffers
  - Aber im Gegensatz zu printf nicht Echtzeitfähig
- std::cin für Leseoperationen

Strings und Standardausgaben

```
char c[10] = "abcdefghi"; //c-style string
std::string s1 = "test";  //c++ string mit zuweisung
std::string s2(test);     //c++ string mit Konstruktor

s1 += s2;   //Konkatenation

std::cin >> s2;  //eingabe lesen, wie scanf

//seltsames Beispiel, zeigt aber wie es funktioniert
std::cout << s1 << " " << c << " " << 5 << std::endl;
```

Container

- Ähnlich wie Collections in Java
  - Map, Set, List, Queue
  - Vector: Array-Implementierung der sich automatisch vergrößert
  - C++11: array, unordered_map, unordered_set
- Zugriff über Iteratoren
  - Haben Bound Checking
  - Compiler-Optimiert (besser als Pointer)
- Fertige Algorithmen
  - fill, accumulate, find, uvm.
  - Compiler optimiert (meist besser als selber schreiben)



Container

```
std::vector<int> v = std::vector<int>(10); //vector mit 10 Elementen

//Schleife mit Indizes
for(int i = 0; i < v.size(); ++i)
{
   v.at(i) = 0;
}

//Schleife mit Iterator
std::vector<int>::iterator itr = t.begin()
for(itr; itr != t.end(); ++itr)
{
    *itr = 0;
}

//Algorithmus
std::fill(t.begin(), t.end(),0);
```

Wichtige Neuerungen in C++ 11

- Automatische Typenerkennung mithilfe von *auto*
  - Erkennt Datentyp zur Compilezeit, nicht zur Laufzeit
  - In manchen Sonderfällen irreführend (fixes zum Teil in C++17)
  - Referenzen müssen mit *auto&* angegeben werden
  - Syntax-Zucker
- For-Each Schleifen, um über den ganzen Container zu iterieren
  - Vermeidet Leichtsinnsfehler bei Schleifen
  - Ebenfalls Syntax-Zucker
- Smart-Pointers um Speicherlecks zu vermeiden
  - Zerstören automatisch das Objekt, wenn der Scope aufgelöst wird
  - Std::unique_ptr ist nützlich, wenn nur ein Pointer auf das Objekt verweisen darf
  - Std::shared_ptr ist für Mehrfachreferenzierung. Objekt wird zerstört, wenn nichts mehr auf das Objekt zeigt
  - Fast immer verwenden!!!

Wichtige Neuerungen in C++ 11

```
auto i = 10;                        //wird zum 32 Bit Integer
auto v = std::vector<int>(i,i); //vector mit 10 Elementen die alle 10 sind

//addiere alle Elemente von v
//funkioniert mit std::accumulate
for(auto& a : v)
{
   i += a;
}
```


```
TestObject* t = new TestObject(123); // C++03
delete t;


//hier brauchen wir kein delete
std::unique_ptr<TestObject> t1(new TestObject(123)); // C++11


std::unique_ptr<TestObject> t2 = std::make_unique<TestObject>(123); // C++14
//oder
auto t3 = std::make_unique<TestObject>(123);   // mit auto
```

Funktionsobjekte / Lambda

- Funktionsobjekte sind Klassen, die nur aus einer Funktion bestehen
  - Eingeführt um Funktionen an Algorithmen zu übergeben
  - An sich praktisch, aber müssen weit außerhalb des Kontextes implementiert werden
- Ab C++11: Lambda
  - Einfacher zu schreiben, müssen nicht außerhalb des Kontextes angelegt werden
  - Syntax eigenartig
    - [] capture: Gibt an, welche Variablen an das Lambda übergeben werden
    - () Parameter: Wie bei der Funktion die Übergabeparameter
    - - > RückgabeTyp: Gibt an was das Lambda zurückgibt, optional
    - {} Funktionsrumpf: Hier kommt der Inhalt der Funktion
  - Details in späterer Vorlesung

Funktionsobjekte / Lambda

```
// Eine Funktionsklasse
class FunctionClass
{
  void operator()(int& val)
  {
     val = 0;
  }
};

//Aufruf irgendwo im Code
std::for_each(t.begin(), t.end(),FunctionClass());

//kleinstes sinnloses Lambda
auto l0 = [](){};
//Capture by Reference, val muss vorher schon definiert sein
auto l1 = [&](){val = 0;};
//Val ist nun ein Uebergabeparameter
auto l2 = [](auto& val){val = 0;};
//Ein Lambda das 0 zurueck gibt
auto l3 = []()-> int {return 0;};

//das selbe Beispiel wie oben
std::for_each(t.begin(), t.end(),[](auto& val){val = 0;});
```

Templates

- Vergleichbar mit Generics in Java
- Übergabe von Compile-Zeit Argumenten
  - Keine Laufzeitpolymorphie
- Verfügbar für Klassen, Funktionen und Variablen
- Mal wieder etwas eigenartige Syntax
- Relativ mächtig durch sogenannte Spezialisierungen

Templates

```
template<typename T>     //Unser Template Datentyp heist jetzt T
T add(T val1, T val2)
{
//T muss + Operator so unterstuetzen dass auch wieder ein T zurueckgegeben wird
//ansonsten Compilefehler
   return val1 + val2;
}

//Aufruf
auto v0 = add(1,2);     //Automatische Datentyp erkennung
auto v1 = add(1,2.1);   //Compile Error
auto v2 = add<unsigned int>(1,2); //Explizite Template angabe


template<typename T>     //Unser Template Datentyp heist jetzt T
class Example
{
public:
   Example(T val);       //als Konstruktor Argument
   T getVal();
private:
   T myValue;            //als Member Variable
}
```


© M. Sc. Tobias Gerteis

Nächstes Mal: Datentypen „In Depth“