1 Kompilierungsprozess

- **Präprozessor**: Führt Direktiven wie `#include` und `#define` aus. Erzeugt eine erweiterte Quellcodedatei.
- **Compiler**: Übersetzt den C++ Quellcode (`.cpp`) in Assembly-Code (`.asm`).
- **Assembler**: Folgt auf den Compiler und übersetzt den Assembly-Code in Maschinencode (Binär - `.obj`).
- **Linker**: Kombiniert verschiedene Objektdateien und Bibliotheken (`.lib`) zu einem einzigen, ausführbaren Programm.

2 Typumwandlung

- **Implicit Casting**: Automatische Typumwandlung durch den Compiler.
  - `int a = 5.4;` $\Longrightarrow$ a wird zu einem int (5)
  - `float b = 7/2;` $\Longrightarrow$ Ganzzahlige Division, Ergebnis 3 wird zu double (3.0)
  - `float c = 7/2.0;` $\Longrightarrow$ Einer der Werte ist float, Ergebnis 3.5
  - `double d = 'A'- 12;` $\Longrightarrow$ char wird zu int (65), dann - 12 (53), dann zu double (53.0)
  - `int e = true + 3;` $\Longrightarrow$ bool wird zu int (1) + 3 (4), dann zu int (4)
  - Allgemein: *Der kleinere Typ wird in den größeren umgewandelt*
- **Explicit Casting**: Manuelle Typumwandlung durch den Programmierer.
  - `int x = (int)3.7;` $\Longrightarrow$ Klassischer Cast: Ergebnis ist 3
  - `int y = static_cast<int>(3.7);` $\Longrightarrow$ Moderner Cast mit `static_cast`: Ergebnis ist ebenfalls 3

3 Hierarchie von Operatoren

Priorität	Operator	Beschreibung
Hoch	`! * &`	**Unär:** Log. NICHT, Deref., Adresse
↓	`* /`	**Binär:** Multiplikation, Division
↓	`+ -`	**Binär:** Addition, Subtraktion
↓	`« »`	**Binär:** Bit-Shift Links/Rechts
↓	`&`	**Binär:** Bitweises UND
↓	`|`	**Binär:** Bitweises ODER
↓	`&&`	**Binär:** Logisches UND
Niedrig	`||`	**Binär:** Logisches ODER

4 Wertebereiche von Datentypen

Datentyp	Bytes	Wertebereich
`bool`	1	`true` oder `false`
`char`	1	-128 bis 127
`unsigned char`	1	0 bis 255
`short`	2	-32.768 bis 32.767
`unsigned short`	2	0 bis 65.535
`int`	4	-2.147.483.648 bis 2.147.483.647
`unsigned int`	4	0 bis 4.294.967.295
`long long`	8	ca. -9,2 x 10^{18} bis 9,2 x 10^{18}
`float`	4	ca. ±3.4 x 10^{38} (7 Dezimalstellen)
`double`	8	ca. ±1.8 x 10^{308} (15 Dezimalstellen)

5 Overflow von Zahlen

Overflow = Zugewiesene oder berechnete Zahl liegt außerhalb des darstellbaren Bereichs eines Datentyps.

- **Ganzzahlen**: Undefiniertes Verhalten. z.B. zu hohe Bits werden abgeschnitten oder es wird auf den Minimalwert zurückgesetzt.
- **Gleitkommazahlen**: Im IEEE 754 Standard wird bei Overflow der Wert `inf` (unendlich) zugewiesen.

6 Definition und Deklaration

- **Definition**: Reserviert Speicherplatz für eine Variable oder Funktion und kann optional initialisiert werden.
  - Beispiel Variable: `int x = 5;`
- **Deklaration**: Informiert den Compiler über den Typ und Namen einer Variable oder Funktion, reserviert aber keinen Speicherplatz.
  - Beispiel Variable: `int x;`
  - Beispiel Funktion: `void foo();`
- **Prototyp**: Funktionsdeklaration ohne Funktionskörper.
  - Beispiel: `double sum(double[]);`
- **Wichtig**: *Jede Definition ist auch eine Deklaration!*

7 Functional und Lambda

Benötigt `#include <functional>`

`std::function<T>` ist nützlich um Funktionen als Objekt zu deklarieren, speichern und übergeben zu können. **Beispiel**:

```
std::function<int(int,int)> sum = [](int a, int b){ return a + b; };
```

Lambda Funktionen

Lambda Funktionen sind anonyme Funktionen, die direkt im Code definiert werden können. Sie haben die folgende Syntax:

```
[capture](parameters)-> return_type { body }
```

- **Capture**: Bestimmt, welche Variablen aus dem umgebenden Kontext verwendet werden können.
  - `[]`: Keine Variablen werden erfasst.
  - `[=]`: Alle Variablen werden per Wert erfasst.
  - `[&]`: Alle Variablen werden per Referenz erfasst.
  - `[x, &y]`: Variable `x` per Wert `y` per Referenz.

8 Iteratoren

Benötigt `#include <vector>`. Iteratoren sind Objekte, die verwendet werden, um über die Elemente eines Containers (wie `std::vector`, `std::list`, etc.) zu iterieren.

- `auto it = vec.begin();`: Iterator auf Anfang
- `*it`: Zugriff auf Element
- `++it`, `--it`: Vorwärts/Rückwärts bewegen
- `it != vec.end()`: Vergleich mit Ende
- `std::advance(it, n)`: Iterator um n Positionen bewegen
- `std::distance(it1, it2)`: Abstand zwischen zwei Iteratoren

9 String und Vector API

Typ	Methode	Kurzbeschreibung
`string/vector`	`.size()`	Anzahl Elemente
`string/vector`	`.empty()`	Leer?
`string/vector`	`.clear()`	Inhalt löschen
`vector`	`.push_back(val)`	Am Ende anhängen
`vector`	`.pop_back()`	Letztes entfernen
`string/vector`	`.at(idx)`	Element an Position
`string/vector`	`.front()/back()`	Erstes/letztes Element
`string/vector`	`.begin()/end()`	Iteratoren
`string`	`.substr(st,len)`	Teilstring `st`
`string`	`.find(str)`	Sucht `str`

10 Nützliche std:: Funktionen

Benötigt `#include <algorithm>` und `#include <functional>`

Methode	Beschreibung
`std::sort(b, e)`	`void` Sortieren
`std::find(b, e, v)`	`Iterator` Sucht einen Wert
`std::reverse(b, e)`	`void` Umkehren der Reihenfolge
`std::max(a, b)`	`T` Gibt das größere von zwei Werten zurück
`std::find_if(b, e, p)`	`Iterator` Sucht das erste Element, das das Prädikat erfüllt
`std::count_if(b, e, p)`	`int` Zählt Elemente, die das Prädikat erfüllen
`std::all_of(b, e, p)`	`bool` Prüft, ob alle Elemente das Prädikat erfüllen
`std::any_of(b, e, p)`	`bool` Prüft, ob mindestens ein Element das Prädikat erfüllt
`std::max_element(b, e)`	`Iterator` auf das größte Element
`std::for_each(b, e, f)`	`void` Wendet Funktion `f` auf jedes Element im Bereich an

- `b` = `begin()`, `e` = `end()`
- `p` = Prädikat (Funktion, die bool zurückgibt) z.B. `[](int x){return x>5;}`
- `v` = Wert, der gesucht wird
- `d` = Zieliterator (z.B. Anfang eines anderen Containers)
- `f` = Funktion, die auf jedes Element angewendet wird (z.B. `[](int x){return x*2;}`)

11 Konventionen

- **Zugriffsmodifikatoren**: Reihenfolge: `public:`, `protected:`, `private:`
- **Konstruktoren**: Immer Explicit angeben
- **Destruktoren**: Immer Virtual angeben, wenn die Klasse vererbt wird
- Membervariablen: Immer mit `m_` oder `_m` kennzeichnen. Keine gleichen Namen wie Parameter im Konstruktor verwenden.
- **Funktionen / Methoden**: Nicht komplett inline definieren: `int add(int a, int b){return a+b}`
- **Void als Parameter**: Nie `void` als Parameter verwenden: `int foo(void);`

12 Objektorientierung

- **Konstruktor / Destruktor**: Konstruktoren werden in verschachtelten Klassen von der innersten zur äußersten Klasse aufgerufen. Destruktoren in umgekehrter Reihenfolge.
- **Virtual / Overrite**: Virtuelle Funktionen werden in der Basisklasse mit `virtual` deklariert und in der abgeleiteten Klasse mit `override` überschrieben.
  - Wenn eine Methode als `virtual` deklariert ist, wird zur Laufzeit die passende Methode der abgeleiteten Klasse aufgerufen, auch wenn der Zeiger oder die Referenz den Typ der Basisklasse hat.
  - Wenn eine Methode nicht als `virtual` deklariert ist, wird die Methode abhängig vom Typ des Zeigers oder der Referenz aufgerufen (statischer Bindung).
- **Final**: Mit `final` kann verhindert werden, dass eine Klasse weiter vererbt wird oder eine Methode überschrieben wird.

13 Smart Pointer

Smart Pointer sind Klassen, die die Verwaltung von dynamisch allozierten Objekten übernehmen und automatisch den Speicher freigeben, wenn der Pointer nicht mehr benötigt wird.

- `std::unique_ptr<T>`: Besitzt ein Objekt exklusiv. Kann nicht kopiert, nur verschoben werden. Nutzt `std::move()` zum Übertragen des Besitzes.
- `std::shared_ptr<T>`: Teilt den Besitz eines Objekts mit anderen `shared_ptr`s. Verwendet Referenzzählung, um zu wissen, wann das Objekt gelöscht werden kann.

make_shared / make_unique

Empfohlene Methode zur Erstellung von Smart Pointern, da sie effizienter und sicherer ist als die direkte Verwendung von `new`.

- `auto ptr = std::make_unique<T>();`: Erstellt einen `unique_ptr` zu einem neuen Objekt vom Typ `T`.
- `auto ptr = std::make_shared<T>();`: Erstellt einen `shared_ptr` zu einem neuen Objekt vom Typ `T`.

std::move

`std::move` markiert ein Objekt als "bewegbar", sodass Ressourcen effizient übernommen werden, statt kopiert zu werden. Das Quellobjekt bleibt gültig, aber sein Zustand ist nicht definiert.

```
1 std::unique_ptr<int> a = std::make_unique<int>(5);
2 std::unique_ptr<int> b = std::move(a);
```

14 Speicherbereiche

Bereich	Beschreibung
Stack	Alle Rücksprungadressen und lokalen Variablen
Heap	Dynamisch alloziierte Objekte mit `new` oder `malloc`
Data Segment	Globale Variablen oder mit `static`; zum Programmstart im Speicher und initialisiert
BSS Segment	Globale Variablen oder mit `static`; zum Programmstart im Speicher, aber nicht initialisiert (werden auf 0 gesetzt)

Zugriffszeiten verschiedener Speicherarten: Register des Prozessors > Cache Speicher > Hauptspeicher (RAM) > SSD/HDD (Von schnell nach langsam)

15 Hashing

- **Hashmap**: Datenstruktur, die Schlüssel-Wert-Paare speichert und schnellen Zugriff auf Werte über ihre Schlüssel ermöglicht.
  - Vorteile: Schneller Zugriff, Einfügen und Löschen in durchschnittlich O(1) Zeit.
- **Hashfunktion**: Berechnet die Position eines Objektes in einer Tabelle (Array). z.B.:
  - $h(k) = k \mod m$, wobei k der Schlüssel und m die Größe des Arrays ist.
- **Kollisionsbehandlung**: Methoden zur Behandlung von Kollisionen:
  - **Verkettung (Chaining)**: Jedes Array-Element enthält eine Liste von Einträgen, die auf diese Position abgebildet werden.
  - **Sondieren (Open Addressing)**: Lineares, quadratisches Sondieren oder doppeltes Hashing.

Beispiele

- $h(k)$: Primäre Hashfunktion (z.B. $k \mod m$)
- i: Anzahl der Versuche (0, 1, 2, ...)
- m: Größe der Hash-Tabelle

- **Lineares Sondieren**: $h_i(k) = (h(k) + i) \mod m$
- **Quadratisches Sondieren**: $h_i(k) = (h(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$
  - c_1, c_2: Konstanten (z.B. 0.5)
- **Doppeltes Hashing**: $h_i(k) = (h(k) + i \cdot h'(k)) \mod m$
  - $h'(k)$: Sekundäre Hashfunktion (z.B. $1 + (k \mod (m'))$ wobei m' eine Primzahl kleiner als m ist)

C++ Hashmap

Benötigt `#include <unordered_map>`. `std::unordered_map<Key, Value> map;`. `std::pair<Key, Value> pair(key, value);`

Methode	Beschreibung
`map[key]`	Zugriff/Ändern eines Werts
`map.insert(pair)`	Einfügen, falls Schlüssel neu
`map.find(key)`	Sucht Schlüssel, Iterator oder `end()`
`map.erase(key)`	Löscht Schlüssel
`map.size()`	Anzahl Elemente

16 Vector vs. List

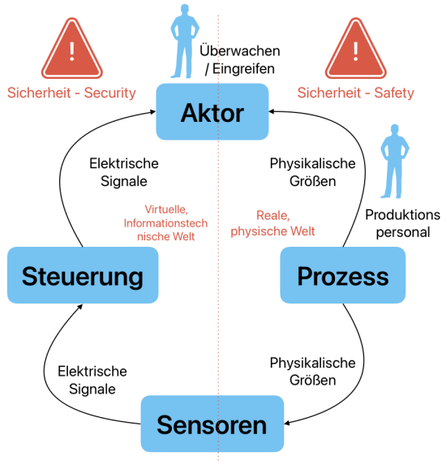
Aspekt	Beschreibung
Speicherstruktur	**Vector**: Kontinuierlicher Speicherblock. **List**: Verkettete Knoten, die nicht zusammenhängend im Speicher liegen.
Zugriffszeit	**Vector**: O(1) für direkten Zugriff. **List**: O(n) für direkten Zugriff.
Einfügen/Löschen	**Vector**: O(n) im Durchschnitt, da Elemente verschoben werden müssen. **List**: O(1), wenn der Iterator bekannt ist.
Speicherverbrauch	**Vector**: Weniger Overhead, da nur ein Speicherblock. **List**: Mehr Overhead durch Zeiger in jedem Knoten.

17 Von Neumann Zyklus

- **Fetch**: Der Prozessor holt den nächsten Befehl aus dem Speicher (RAM) und lädt ihn in das Befehlsregister.
- **Decode**: Der Prozessor dekodiert den Befehl, um zu verstehen, welche Operation ausgeführt werden soll und welche Operanden benötigt werden.
- **Fetch Operands**: Der Prozessor holt die benötigten Operanden aus dem Speicher oder den Registern.
- **Execute**: Der Prozessor führt die dekodierte Operation aus, indem er die erforderlichen Berechnungen durchführt oder Daten verarbeitet.
- **Write Back**: Das Ergebnis der Operation wird zurück in den Speicher oder die Register geschrieben.

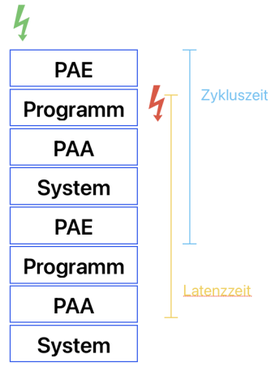
Harvard-Architektur: Getrennte Speicher für Daten und Befehle, paralleler Zugriff möglich.

18 Automatisierungstechnik

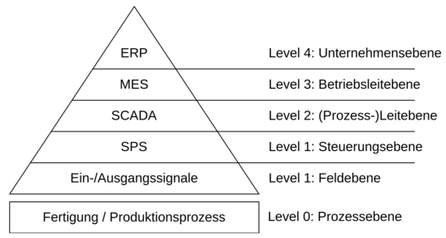


Speicherprogrammierbare Steuerung (SPS): Industrieller Computer zur Steuerung von Maschinen und Prozessen. SPS-Zyklus:

- **Eingabe lesen**: Alle Eingänge (Sensoren, Taster) werden eingelesen (Vom PAE).
- **Programm ausführen**: Das Steuerungsprogramm wird basierend auf den Eingaben ausgeführt.
- **Ausgabe schreiben**: Alle Ausgänge (Aktoren, Lampen) werden entsprechend dem Programmzustand gesetzt (Vom PAA).



Worst-Case: Doppelte der Zykluszeit (Eingabe lesen + Programm ausführen + Ausgabe schreiben).



19 Prozessklassifizierung

- **Kontinuierliche Prozesse**: Ständige Veränderung der Prozessgrößen (z.B. Temperaturregelung).
- **Stück-Prozesse**: Verarbeitung einzelner Einheiten (z.B. Montage von Autos).
- **Batch-Prozesse**: Verarbeitung in Chargen (z.B. Chemische Produktion).

20 IEC 61131-3

Deklaration von Variablen:

```
TYPE Ampel :
STRUCT
    AKTIV          : BOOL;
END_STRUCT
END_TYPE
TYPE AmpelArray : ARRAY[1..3] OF Ampel;
   END_TYPE
VAR
    Hauptstr_Ampel : Ampel;
    Zykluszeit     : TIME := T#5s;
    AlleAmpeln     : AmpelArray;
END_VAR
```

Hinweis: *Variablen können auch in `VAR_GLOBAL ... END_VAR` Blöcken deklariert werden, um sie in mehreren Programmen verfügbar zu machen.*

- **Kontaktplan**: Grafische Programmiersprache, die elektrische Schaltpläne nachbildet.
  - `-]  [-`: Öffner
  - `-]\[-`: Schließer
  - `-()-`: Spule (Aktor)
- **Funktionsbausteinsprache (FBS)**: Logik-Gatter, Flip-Flops, TON, TOF werden als Bausteine dargestellt und verbunden.
- **Continuous Function Chart (CFC)**: Erweiterung der FBS mit freier Anordnung der Bausteine. Keine strikte Abarbeitung von links nach rechts.
- **Strukturierter Text (ST)**: Hochsprachliche Programmiersprache ähnlich zu Pascal/C. Syntax:
  - Variablen mit `VAR ... END_VAR` deklarieren
  - Anweisungen mit `:=` für Zuweisung
  - Kontrollstrukturen: `IF ... THEN ... ELSE ... END_IF;`, `FOR ... TO ... DO ... END_FOR;`
  - Funktionen und Funktionsbausteine mit `FUNCTION ... END_FUNCTION` bzw. `FUNCTION_BLOCK ... END_FUNCTION_BLOCK`

```
FOR i := 1 TO 3 BY 1 DO
    AlleAmpeln[i].AKTIV := FALSE;;

END_FOR;

IF Hauptstr_Ampel.AKTIV THEN
    Nebenstr_Ampel.L_ROT := TRUE;
ELSIF NOT Hauptstr_Ampel.AKTIV THEN
    Nebenstr_Ampel.L_GRUEN := TRUE;
END_IF;
```

- **Ablaufsprache (AS)**: Grafische Sprache für Zustände und Übergänge; ideal für Ablaufsteuerungen. **Aktionen**:

BZ	Beschreibung
N	1 im aktuellen Zustand
R	Reset (auf 0 setzen)
S	Set (auf 1 setzen)
P	1 nach einem Übergang von 0 zu 1
L	1 bis Zeit abgelaufen
D	1 nach Zeitverzögerung

Alternativ vs. Parallel

- **Alternativ**: Nur ein Pfad wird ausgeführt. Übergänge mit Bedingungen.
- **Parallel**: Mehrere Pfade werden gleichzeitig ausgeführt. Synchronisation durch spezielle Übergänge.

21 Objektorientierung (AT)

- **Datenkapselung**: mit GET und SET Methoden.
- **Vererbung**: Mit `EXTENDS` Schlüsselwort.
- **Interfaces**: Mit `INTERFACE ... END_INTERFACE` Blöcken.
- **Funktionsblöcke**: Mit `FUNCTION_BLOCK ... END_FUNCTION_BLOCK` Blöcken.

```
FUNCTION_BLOCK Fun
VAR_INPUT in : INT; END_VAR
VAR_OUTPUT out : INT; END_VAR
out = in * 2;
END_FUNCTION_BLOCK

Fun(in := 5, out => result);
```

22 Automatisierungsarchitekturen

- **Zentral**: Eine SPS steuert alles.
- **Dezentral**: Mehrere SPS teilen Aufgaben (Hohe verlässigkeit $\implies$ Kein Gesamtausfall).
- **Verteilt**: Steuerung über Netzwerk verteilt.
- **Hierarchisch**: Steuerung auf mehreren Ebenen.

23 Redundanz und Fehler

- **Redundanz**: Mehrfache Ausführung von kritischen Komponenten.
  - **Hardware-Redundanz**: Mehrere SPS, Sensoren, Aktoren.
  - **Software-Redundanz**: Mehrere Programme oder Algorithmen für dieselbe Aufgabe.
  - **Zeit-Redundanz**: mehrfache Abfrage des gleichen Messwertes in bestimmten Zeitabständen.

24 Sicherheits-Integritätslevel

Das SIL gibt die erforderliche Risikominderung für sicherheitsbezogene Systeme an.

SIL	Beschreibung
SIL 1	Niedrigste Stufe. Relativ hohe Fehlerwahrscheinlichkeit akzeptiert. Für geringe Risiken, z.B. einfache Alarmsysteme.
SIL 2	Mittlere Stufe. Erhöhte Zuverlässigkeit nötig. Für Anwendungen mit moderatem Gefahrenpotenzial, z.B. Not-Aus-Schalter.
SIL 3	Hohe Stufe. Sehr geringe Fehlerwahrscheinlichkeit. Für schwere Verletzungsgefahr, z.B. Sicherheitsbarrieren, Zugsteuerungen.
SIL 4	Höchste Stufe. Extrem geringe Fehlerwahrscheinlichkeit. Für extrem kritische Systeme, z.B. Kernkraftwerke, Flugzeugsteuerungen.

Das SIL ergibt sich aus einer Risikoanalyse: Schwere (S), Häufigkeit (F), Wahrscheinlichkeit (W), Vermeidbarkeit (P). **Safty Inegrated Function**: Besteht aus: Sensorik (Fehler erkennen), Logik (SPS - Fehler bewerten) und Aktorik (Sicheren Zustand einleiten).

25 MooN Architektur

Es müssen mindestens M von insgesamt N Komponenten korrekt funktionieren, damit das System eine sicherheitsrelevante Aktion ausführt.

Architektur	Sicherheit / Verfügbarkeit	Typische Anwendung
1oo1	Niedrig / Hoch	Einfache Steuerungen
1oo2	Mittel / Hoch	Prozessüberwachung mit Alarm
1oo2D	Hoch / Hoch	SIL 2-3 Anwendungen
2oo2	Sehr Hoch / Niedrig	Not-Aus, Reaktorschutz
2oo3	Hoch / Hoch	Kritische Systeme mit Voting
3oo3	Extrem Hoch / Sehr niedrig	SIL-4 Anwendungen