# 3. C++ STANDARD LIBRARY I

# Inhalt

- Einführung und Überblick
- Erste nützliche Komponenten
  - Smart Pointers
  - Vector
  - String
  - Ein- und Ausgabe

# Einführung

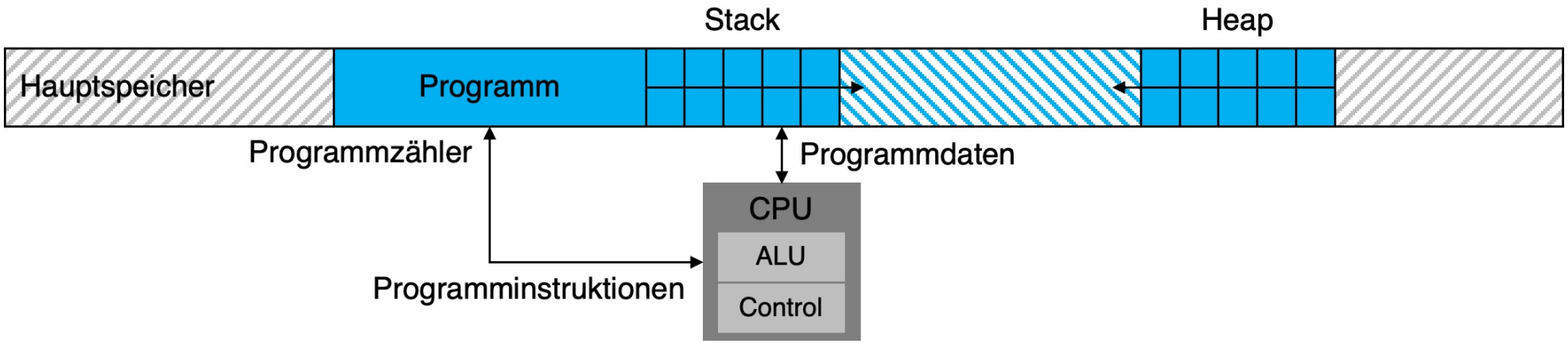
- In Woche 1 haben wir gelernt, dass der C++-Standard zwei Hauptteile umfasst
  - Konzepte des **Sprachkerns**
  - Komponenten der **Standardbibliothek**
- In Woche 2 haben wir vor allem über den Sprachkern geredet
- Heute werfen wir einen ersten Blick auf die Standardbibliothek, damit wir etwas spannendere Programme schreiben können
  - Aufbau und Umfang der Standardbibliothek
  - Erste nützliche Komponenten
- Später reden wir noch detaillierter über die Standardbibliothek

# Die C++-Standardbibliothek

- **GTK** Kein echtes C++-Programme ist nur in der Kernsprache geschrieben, sondern benutzt oft einige **Programmbibliotheken**
- Die **C++-Standardbibliothek** ist eine Sammlung von Komponenten, die für die meisten Programme nützlich sind
  - Datenstrukturen: Vektoren, Mengen, Zuordnungstabellen etc.
  - Algorithmen: iterieren, suchen, sortieren, kopieren etc.
  - Mathematik: Wurzel, Potenz, komplexe Zahlen etc.
  - Datenströme: Ein- und Ausgabe auf Konsole, Dateien etc.
  - Speicherverwaltung: Smart Pointers
  - Zeichenketten: Strings, String Views, Regular Expressions etc.
  - Zeitmessung: Zeit, Datum, Zeitzonen etc.
  - Parallelisierung: Threads
  - C-Standardbibliothek: Rückwärtskompatibilität mit reinem C
- Diese Liste ist nicht vollständig und wird mit jedem neuen **C++-Standard** überarbeitet und erweitert!

# Stack und Heap

Die Daten, mit denen wir bisher gearbeitet haben, wurden im **Stack** gespeichert. Es ist aber auch möglich, Daten im **Heap** zu speichern.




- **Stack**
  - **Sichtbarkeitsbereich** des Namens bestimmt Lebensdauer
  - Speicher wird **automatisch** reserviert und wieder freigegeben
- **Heap**
  - Diese Werte haben **keinen** Namen und sind deshalb an keinen Sichtbarkeitsbereich gebunden
  - Speicher muss **manuell** reserviert und wieder freigegeben werden

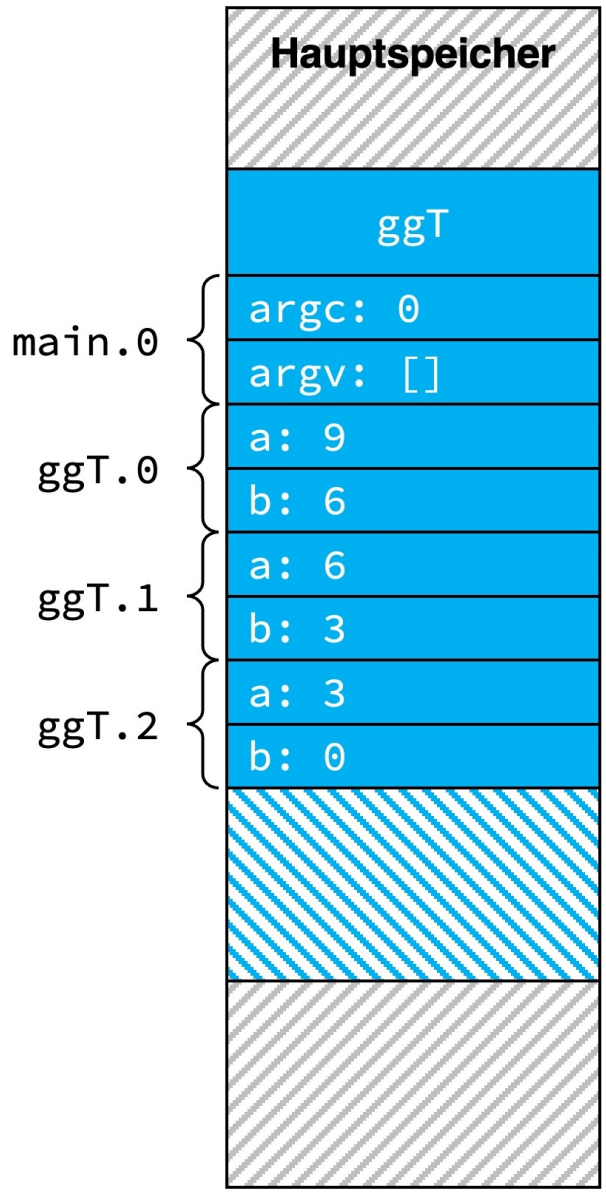
# Stack-basierte Speicherverwaltung

```
#include <iostream>

using namespace std;

int ggT(int a, int b) {
  if (a < b) {
    return ggT(b, a);
  }
  return b == 0 ? a : ggT(b, a % b);
}

int main(int argc, char* argv[]) {
  cout << ggT(9, 6) << endl;
  return EXIT_SUCCESS;
}
```



# Heap-basierte Speicherverwaltung

- Speicherverwaltung im Stack erfolgt **implizit** bzw. automatisch
- Um Daten im Heap zu speichern, muss **explizit** Platz reserviert und dann auch wieder freigegeben werden
  - `new` legt ein **neues** Datenobjekt im Heap-Speicher an
  - `delete` **löscht** ein Datenobjekt und gibt dessen Heap-Speicher frei
- GTK Heap-basierte Speicherverwaltung nennt man auch **dynamische Speicherverwaltung**
- GTK **Regel**: zu jedem `new` muss es auch ein `delete` geben!
  - Die Einhaltung dieser Regel muss **manuell** sichergestellt werden
  - Andenfalls kann es zu **Speicherlecks** („memory leaks“) kommen
  - Dann wächst der **Speicherhunger** des Programm ins Unermessliche
- Manuelle Speicherverwaltung ist **mühsam** und **fehleranfällig**
  - Mit solch unlustigen Sachen wollen wir uns nicht herumschlagen!
  - Dank Abstraktionen der Standardbibliothek müssen wir das auch nicht

# Zeiger

- Wir haben gelernt, dass eine Variable eine Abstraktion für den **Wert** ist, der an einer gewissen Speicheradresse steht
  - Variablen sind **wertebasiert**, d.h. bei einer Zuweisung werden Daten von einer Speicheradresse an eine andere Speicheradresse kopiert
  - Ihre Lebensdauer wird von einem Sichtbarkeitsbereich bestimmt
- Manchmal ist es aber besser, **referenzbasiert** mit Daten im Speicher zu arbeiten
  - Große Datenobjekte im Speicher hin und her zu kopieren ist teuer
  - Ein Datenobjekt soll von mehreren Orten im Programm zugreifbar sein
- Der Zeiger („pointer“) ist eine Abstraktion für die **Speicheradresse**, an der ein gewisser Wert steht
  - Zeiger werden deshalb auch als **Referenzvariablen** bezeichnet
  - Um auf den Wert einer Referenzvariable zuzugreifen, muss sie **dereferenziert** ( `*` ) werden
- Wie mit Strings kann man in C++ auch mit Zeigern auf zwei unterschiedlichen Abstraktionsebenen arbeiten

# Smart Pointers

- Übersicht über die verschiedenen Arten von Zeigern in C++
  - Tiefe Abstraktionsebene: **Raw Pointers** (nächste Woche)
  - Hohe Abstraktionsebene: **Smart Pointers** (diese Woche)
- Smart Pointers kümmern sich im Gegensatz zu raw pointers **automatisch** um Ressourcen- bzw. Speicherverwaltung
  - `std::unique_ptr` repräsentiert exklusiven Besitz von Daten
  - `std::shared_ptr` repräsentiert geteilten Besitz von Daten
  - `std::weak_ptr` unterbricht Zyklen in zirkulären Datenstrukturen
- Wir spezifizieren explizite **Besitzverhältnisse von Ressourcen** anstatt explizite Speicherverwaltung mit `new` und `delete`
  - Wer eine Ressource besitzt, hat die Verantwortung fürs Aufräumen
  - Für jede Art von Smart Pointer gibt es Regeln, wann der entsprechende Speicher wieder freigegeben wird
  - Das macht uns das Leben einfacher und verhindert Speicherlecks!

# Smart Pointers: `std::unique_ptr`

- `std::unique_ptr` ist der **exklusive** Besitzer einer Resource
  - `std::make_unique` erstellt den Zeiger und reserviert Speicher
  - Wird der Zeiger gelöscht, wird auch der Speicher wieder freigegeben
  - Kann nicht kopiert sondern nur verschoben ( `std::move` ) werden
- Typische Anwendungsfälle
  - Übergabe des Besitz einer Ressource an eine Funktion
  - Rückgabe von dynamisch reserviertem Speicher aus einer Funktion
  - Speicherung von Zeigern in Datenstrukturen der Standardbibliothek
- **GTK** Typischerweise führt ein `std::unique_ptr` zu keinem Speicher-Overhead oder Performance-Einbußen im Vergleich zu einem Raw Pointer

# Smart Pointers: `std::shared_ptr`

- `std::shared_ptr` **teilt** sich den Besitz einer Ressource
  - `std::make_shared` erstellt den Zeiger, legt einen Referenzzähler an und reserviert Speicher
  - Der Referenzzähler wird inkrementiert, wenn der Zeiger kopiert wird, und dekrementiert, wenn eine Kopie des Zeigers gelöscht wird
  - Geht der Referenzzähler auf `0`, wird der Speicher wieder freigegeben („Der Letzte macht das Licht aus!“)
- **GTK** Wenn immer möglich, verwenden wir `std::unique_ptr`!
  - `std::shared_ptr` bringt Speicher-Overhead (Referenzzähler) und Performance-Einbußen (Verwaltung des Referenzzählers) mit sich
  - `std::unique_ptr` gewährleistet eine eindeutige und vorhersehbare Lebensdauer von Resourcen
- **GTK** `std::weak_ptr` ist eine spezielle Variante des `std::shared_ptr`, auf die wir hier nicht weiter eingehen

# Beispiele

- Beispiel für die Verwendung von `std::unique_ptr`

```
auto iptr = make_unique<int>();
*iptr = 42;
cout << *iptr << endl;
auto iptr_too = iptr; // Fehler!
cout << *iptr << ", " << *iptr_too << endl;
```

- Beispiel für die Verwendung von `std::shared_ptr`

```
auto iptr = make_shared<int>();
*iptr = 42;
cout << iptr.use_count() << endl; // Referenzzaehler
auto iptr_too = iptr;
cout << iptr.use_count() << endl;
cout << *iptr << ", " << *iptr_too << endl;
*iptr_too = 27;
cout << *iptr << ", " << *iptr_too << endl;
```

# Container: `std::vector`

- Ein **Container** dient zur Speicherung und Verwaltung einer Sammlung von Werten
  - Die Wahl des richtigen Containers mit den passenden Operationen ist ein wesentlicher Schritt in der Konstruktion eines C++-Programms
  - Die C++-Standardbibliothek bietet eine Reihe nützlicher Container an
- Einer der nützlichsten dieser Container ist `std::vector`
  - Ein `std::vector` ist eine Sequenz von Werten eines einzigen Typs
  - Die Elemente eines `std::vector` sind zusammenhängend im Speicher abgelegt und können per Index zugegriffen werden
  - Modifikation kann dazu führen, dass Elemente verschoben werden!
- Wichtige Operationen
  - Zugriff: `[]`, `at`, `front`, `back`
  - Größe und Kapazität: `empty`, `size`, `reserve`, `shrink_to_fit`
  - Einfügen und Entfernen: `push_back`, `emplace`, `insert`, `erase`, `clear`

# Beispiel

- Beispiel für die Verwendung von `std::vector`

```
// Initialisierung mit drei Elementen
vector<int> v = { 1, 2, 3 };
// Zwei weitere Elemente einfuegen
v.push_back(5);
v.push_back(5);
v[3] = 4; // Ueberschreibe das Element an Position 3
for (int i = 0; i < v.size(); i++) {
    cout << n << "␣";
}
cout << endl;
```

- Was gibt das Programm aus?
- GTK Die `for`-Schleife kann man noch eleganter schreiben!

# Zeichenketten

- Den Datentyp `std::string` haben wir bereits kennengelernt.
- Für diesen Datentyp sind viele nützliche Funktionen definiert.
  - Lexikographische Vergleiche: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - Größe: `size` (Anzahl `chars`), `empty`
  - Kapazität: `reserve`, `resize`, `shrink_to_fit`
  - Zugriff: `[]`, `at`, `front`, `back`, `push_back`, `copy`
  - Numerische Konversionen: `stoi`, `stol`, `stof`, `stod`
  - Einfügen, Löschen und Ersetzen: `append`, `insert`, `erase`, `replace`
  - Suchen: `find`, `rfind`, `find_first_of`, `find_last_of`
  - Substrings: `substr`
- GTK Die **C-Standardbibliothek** bietet über `<cctype>` nützliche Funktionen an, um mit Zeichen und Zeichenketten zu arbeiten
  - `isalpha`/`isalnum`/`isdigit`: alphanumerische Zeichen prüfen
  - `islower`/`isupper`: Groß- und Kleinbuchstaben prüfen
  - `tolower`/`toupper`: Groß- und Kleinbuchstaben umwandeln

# Beispiel

- Beispiel für die Verwendung von `std::string`

```
string bs {"Bjarne␣Stroustrup"};      // Konstruktor
string first = bs.substr(0, 6);
string last {bs, 7, string::npos};    // Konstruktor
string full = first + "␣" + last;
cout << full << endl;
full[6] = '%';
cout << full << endl;
cout << bs.insert(7, "Knut␣");
string num {"1234.567"};
float f = stof(num);                  // String zu Float
cout << f << endl;
```

# Programmieren!

- Schreibe ein C++-Programm `reverse.cpp`, das
  - einen String als Argument übergeben bekommt,
  - diesen String umdreht und
  - den umgedrehten String auf der Konsole ausgibt.
- Beispielaufruf

```
> g++ -o reverse reverse.cpp
> ./reverse leahciM
Michael
```

# Ein- und Ausgabe

- Wir haben bereits gesehen, wie man Werte als formatierten Text auf die **Konsole** ausgibt
- Die C++-Standardbibliothek definiert sogenannte **Streams**, in die mit dem Einfügeoperator << geschrieben werden kann
  - `cout` ist der Stream **standard output**
  - `cerr` ist der Stream **standard error**
- Das Einlesen von Werten als formatierten Text von der Konsole funktioniert ganz analog
  - `cin` ist der Steam **standard input**
  - >> ist der Extraktionsoperator

```
string name;
int alter;
cin >> name;
cin >> alter;
cout << name << "␣ist␣" << alter << "␣Jahre␣alt" << endl;
```

- Das können wir gleich in unser letztes Programm einbauen!

# Ein- und Ausgabe

- Mit Streams kann man auch **Dateien** lesen und schreiben
  - `ofstream` repräsentiert einen **output file stream**
  - `ifstream` repräsentiert einen **input file stream**
- Dateien können in verschiedenen Modi geöffnet werden
  - `ios::app` : Ausgabe ans Ende der Datei anhängen („append“)
  - `ios::trunc` : Datei zurücksetzen („truncate“)
  - `ios::binary` : Datei im Binärmodus behandeln
- Ein- und Ausgabe werden wieder mit Einfügeoperator `<<` und Extraktionsoperator `>>` gemacht
- **GTK** Mit der globalen Funktion `getline` aus `<string>` kann zeilenweise aus einem Inputstream gelesen werden

# Ein- und Ausgabe

- Folgende Muster sind oft nützlich, wenn aus einer Datei gelesen werden soll, bis das **Dateiende** („end-of-file“) erreicht ist
  - Zeile für Zeile

```
string line;
while (getline(cin, line)) {
  cout << "Zeile gelesen: " << line << endl;
}
```

  - Zeichen für Zeichen

```
cin >> noskipws;  // Leerraum nicht ueberspringen
char c;
while (cin >> c) {
  cout << "Zeichen gelesen: " << int(c) << endl;
}
```

  - Zahl für Zahl

```
int i;
while (cin >> i) {
  cout << "Zahl gelesen: " << i << endl;
}
```

# Ein- und Ausgabe

- Nach jeder Ein-/Ausgabe-Operation kann man abfragen, ob diese **erfolgreich oder nicht** war
  - `eof()` Ende des Streams erreicht
  - `fail()` Operation fehlgeschlagen (z.B. falsches Format bei Zahlen)
  - `bad()` Stream ist kaputt
- **Beispiel**

```
if (cin.eof()) {
  cerr << "Dateiende erreicht." << endl;
} else if (cin.fail()) {
  cerr << "Letzte operation ist fehlgeschlagen." << endl;
} else if (cin.bad()) {
  cerr << "Alles kaputt!" << endl;
} else {
  cout << "Alles gut!" << endl;
}
```

# Programmieren!

- Schreibe ein C++-Programm `transpose.cpp`, das
  - zwei Dateinamen als Kommandozeilenparameter nimmt,
  - aus der ersten Datei eine $3 \times 3$ Matrix einliest,
  - diese Matrix transponiert und
  - die transponierte Matrix in die zweite Datei schreibt.
- Beispielaufruf

```
> g++ -o transpose transpose.cpp
> cat matrix
1 2 3
4 5 6
7 8 9
> ./transpose matrix matrixT
> cat matrixT
1 4 7
2 5 8
3 6 9
```

# Version des C++-Standards

- Im Programmierkurs beziehen wir uns auf die **Version 17** des C++-Standards (C++17)
  - C++20 ist abgeschlossen, aber von den meisten Compilern noch nicht vollständig unterstützt
  - C++23 ist derzeit in Arbeit und einzelne Teile werden von gewissen Compilern bereits unterstützt
- GTK Manchmal muss man deshalb die Version des C++Standards beim kompilieren explizit festlegen
  - Bei `g++` kann dies über die Option `-std` gemacht werden
  - ```
    > g++ -std=c++17 -o helloWorld helloWorld.cpp
    ```
    ```