Modernes C++

...für Programmierer

Unit 07: Modularisierung

by

Dr. Günter Kolousek

31. August 2019

Überblick

- Modul
- Übersetzungsmodell
- Headerdateien
- Binden bzw. Linken
- Programm und Executable
- Speicherabbild

Modul

- stellt Funktionalität über Schnittstelle
- Implementierung nicht einsehbar
- verwendet meist weitere Module
- C++ bietet (derzeit) *kein* Konzept an, das Module *direkt* unterstützt
- C++ bietet…
  - Dateien zur physischen Strukturierung:
    - `.h` … Schnittstelle des Moduls
    - `.cpp` … Implementierung
  - sprachliche Hilfsmittel
    - Variablen, Funktionen, Klassen und Namensräume
    - Templates zur generischen Programmierung
    - `inline`, `extern`, `static`, `const`, `constexpr`

Übersetzungsmodell

1. C++ Präprozessor (engl. preprocessor)
   - Aufgabe: Textersetzung
     - `#include`
     - `#define`, `#ifndef`, `#endif`
   - Eingabe: Source-Code; `prog.cpp`
   - Ausgabe: erweiterter Source-Code in temporärer Datei (optional: `g++ -E prog.cpp`)
2. Compiler
   - Aufgabe: Übersetzung von C++ Sourcecode in Assembler (optional)
   - Ausgabe: assembler file; `prog.s` (optioal: `g++ -S prog.cpp`)

Übersetzungsmodell – 2

3. Assembler
   - Aufgabe: Übersetzung von Assemblercode in Objektcode
   - Ausgabe: object file; `prog.o` (optional: `g++ -c prog.cpp`)
4. Linker (auch link editor)
   - Aufgabe: Auflösen der Bezeichner
   - Eingabe: Objektcodedateien (`.o`), Librarydateien (`.a`)
   - Ausgabe: executable; `a.out` oder mit `-o prog`

Headerdateien

- Verwenden von Include-Dateien
  - `#include <iostream>` ... Standardverzeichnisse der C++-Implementierung
    - Konvention der Standardbibliothek: Headerdatei aus der C-Bibliothek, direkt nutzbar in C++ $\rightarrow$ Dateiname beginnt mit kleinem C, z.B. `cstdint`.
  - =#include ”mathutils.h”= ... Verzeichnisse des aktuellen Projektes
- Erstellen von Include-Dateien $\rightarrow$ Guards

  Datei `mathutils.h`:

```
#ifndef MATHUTILS_H
#define MATHUTILS_H
double squared(double val);
#endif
```

Binden C++ Bezeichner an Objekte

- externe Bindung (engl. external linkage): Verwendung eines Bezeichners einer Übersetzungseinheit in einer anderen Übersetzungseinheit
  - Funktionen
  - globale Variable (Deklaration in Headerdatei z.B. `extern double pi;`)
  - ODR (one-definition rule): Zwei Definitionen einer Klasse, eines Template oder einer `inline`- bzw. `constexpr`-Funktion werden als eine betrachtet, wenn diese aus Sicht von C++ gleich sind.
  - Konstanten werden syntaktisch (ab C++17) folgendermaßen in einer Headerdatei angeschrieben: `inline const X x;`

Binden C++ Bezeichner an Objekte - 2

- interne Bindung (engl. internal linkage): nur *innerhalb* einer Übersetzungseinheit
  - globale `static` Variable, z.B. `static int people;`
  - `static` Funktionen, z.B. `static double squared(double);`
  - `const` Variable, z.B. `const double pi{3.1415};`
    - aber extern mittels: `extern const double pi{3.1415};`
    - auch für `constexpr` Variable (hat in der Regel keine Speicheradresse!)

Statisches Linken

- Linken zur Übersetzungszeit
- Einfügen des ausführbaren Code der Funktionen in das Programm
- Erweiterung: `.a` (Windows `.lib`)
- Vorteile
  - leichter zu verteilen
  - auch in eingeschränkten Umgebungen einsetzbar (ohne Installation)
  - startet schneller
  - keine Versionsproblematik/fehlende Shared Objects beim Starten

Dynamisches Linken

- Laden der benötigten Shared Objects (DLLs; wenn noch nicht im Hauptspeicher)
  - beim Starten (load-time dynamic linking)
  - zur Laufzeit (run-time dynamic linking)
- Linken zur Übersetzungszeit durch Loader
- Erweiterung `.so` (Windows `.dll`)
- Vorteile
  - weniger Ressourcenverbrauch (Hauptspeicher, Cache, Festplatte)
    - speziell bei mehreren Prozessen!
  - Plugins nur mittels Shared Objects (und Laden zur Laufzeit: run-time dynamic linking)
  - Verwendung von Libraries, die unter der LGPL stehen
  - Bug fixing…
  - kein neues Linken aller Programme

Statisch vs. dynamisch

```
#include <iostream>  // dynstatic.cpp
using namespace std;
int main() {
    cout << "Hello, World!"s << endl;
}
```

```
g++ -std=c++1y src/dynstatic.cpp -o go
ls -l go|awk '{print $5}'
g++ -std=c++1y -static src/dynstatic.cpp -o go
ls -l go|awk '{print $5}'
```

```
7568
2024000
```

Programm

- Funktion `main`
  - `int main() { /* ... */ }`
  - `int main(int argc, char* argv[]) { /* ... */ }`
- Funktion `exit(int)`
  - Headerdatei `<cstdlib>`
  - Destruktoren von `static` und thread-lokalen Variablen
    - jedoch nicht lokale!
  - geöffnete Dateien werden geschlossen

Format eines ”Executable”

ist in einzelne Sektionen unterteilt. Die wichtigsten sind:

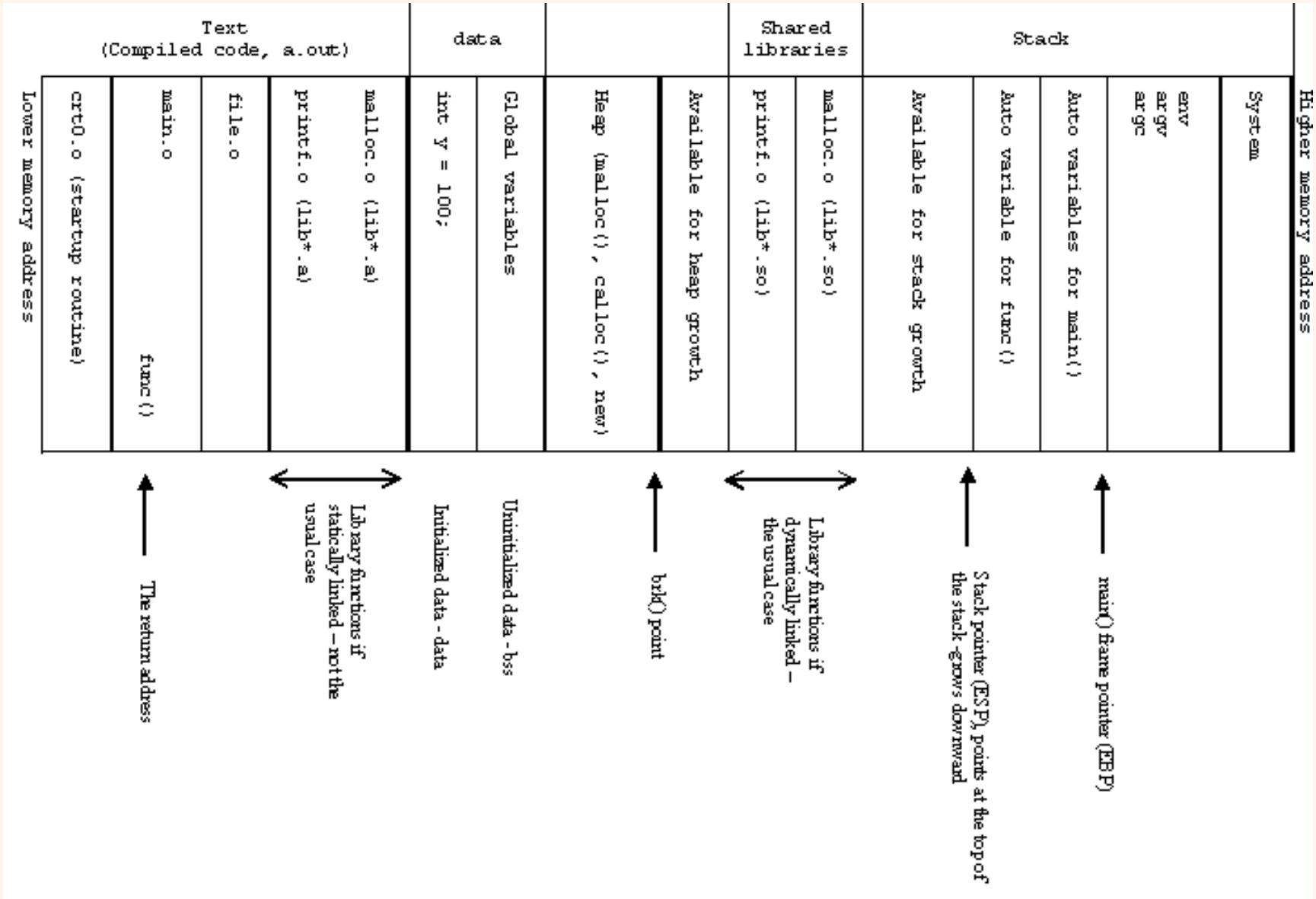
`.text`	ausführbare Anweisungen; read & execute
`.bss`	(block started by symbol) nicht explizit initialisierte globale und statische Variable (); kein Platz in Datei, nur im Prozessimage
`.data`	initialisierte globale und statische Variable; read & write
`.rodata`	nur lesbare Daten: Konstanten und Stringliterale

Format eines ”Executable” – 2

```
#include <iostream>  // rodata.cpp
using namespace std;
int main() {
    char cstr[4]{”abc”};
    cstr[1] = 'x';
    cout << cstr << endl;
    char* cptr{”abc”};  // -Wno-write-strings
    cptr[1] = 'x';
    cout << cptr << endl;
}
```

```
axc
fish: Job 2, 'go' terminated by signal SIGSEGV (Adr
```

Speicherabbild



Speicherabbild – 2

```cpp
#include <iostream>  // stack_heap.cpp
using namespace std;
int main() {
    int i;
    int j;
    int k;
    int* p1 = new int;
    int* p2 = new int;
    int* p3 = new int;
    cout << &i << ' ' << &j << ' ' << &k << endl;
    cout<< &*p1<< ' '<< &*p2<< ' '<< &*p3<< endl;
}
```

```
0x7fff35332834 0x7fff35332838 0x7fff3533283c
0x55a29fe87e70 0x55a29fe87e90 0x55a29fe87eb0
```