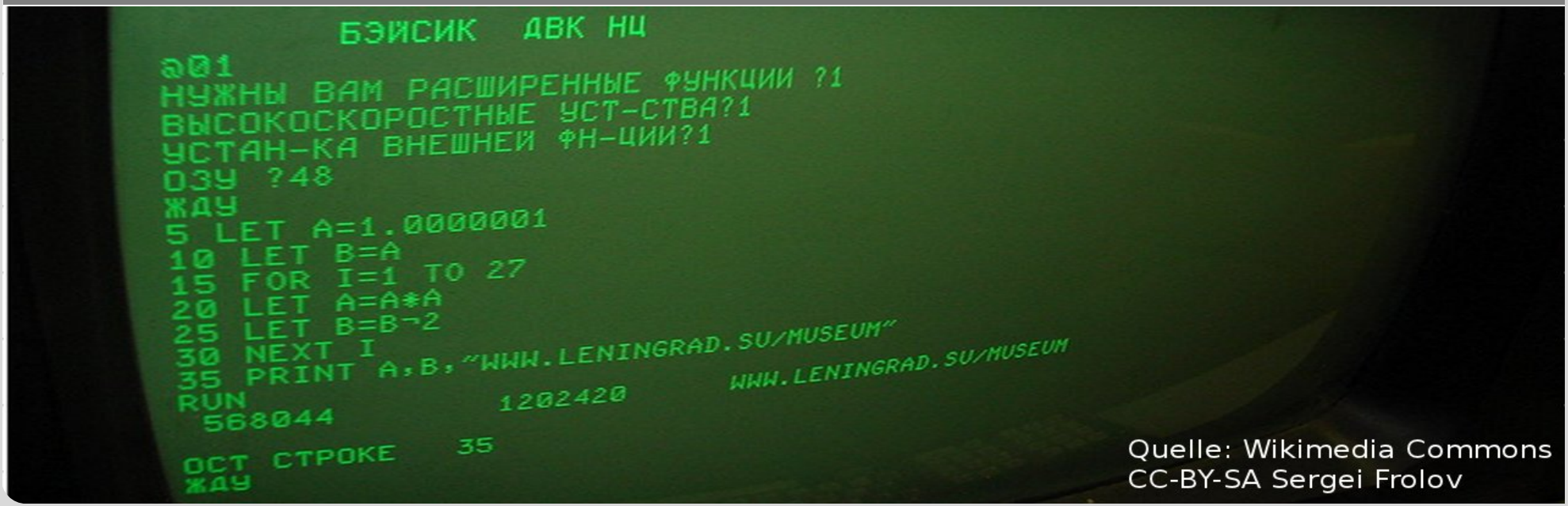
Karlsruher Institut für Technologie

C++ Workshop

7. Block, 15.06.2012

Christian Käser, Robert Schneider | 17. Juni 2012



Gliederung

1. static libraries
   - Wozu?
   - statically linked libraries
2. SDL: Simple Directmedia Layer
   - Was ist das?
   - Grundkonzepte
3. Praxis

1. static libraries
   - Wozu?
   - statically linked libraries

Woher kommt C++?

C++ versucht, ein Hansdampf in allen Gassen zu sein, also:

- Effizient, schnell
- flexibel: für alle Plattformen
- flexibel: für alle Aufgaben
- flexibel: für n+1 Wege, etwas zu tun

Woher kommt C++?

C++ versucht, ein Hansdampf in allen Gassen zu sein, also:

- Effizient, schnell
- flexibel: für alle Plattformen
- flexibel: für alle Aufgaben
- flexibel: für n+1 Wege, etwas zu tun

Resultat: Eine riesige und komplexe Sprache, die auf dem *kleinsten gemeinsamen Nenner* definiert ist.

Was ist C++?

C++ ist eine standardisierte Programmiersprache. Wir arbeiten mit dem (veralteten) ISO/IEC 14882:2003

Der Standard beschreibt grob:

- Erzeugung von Quellcode (Präprozessor, Templates)
- was gültiger Quellcode ist
- das beobachtbare Verhalten eines Programms

Was ist C++?

C++ ist eine standardisierte Programmiersprache. Wir arbeiten mit dem (veralteten) ISO/IEC 14882:2003

Der Standard beschreibt grob:

- Erzeugung von Quellcode (Präprozessor, Templates)
- was gültiger Quellcode ist
- das beobachtbare Verhalten eines Programms

Jede C++ Implementierung (build system) muss Programme erzeugen, deren beobachtbares Verhalten dem vom Standard spezifizierten entsprechen. Alle weiteren Spezifikationen im Standard (z.B. Resultat von Addition) muss nur für beobachtbares Verhalten eingehalten werden $\Longrightarrow$ erlaubt Optimierungen.

Was ist C++?

C++ ist eine standardisierte Programmiersprache. Wir arbeiten mit dem (veralteten) ISO/IEC 14882:2003

Der Standard beschreibt grob:

- Erzeugung von Quellcode (Präprozessor, Templates)
- was gültiger Quellcode ist
- das beobachtbare Verhalten eines Programms

Jede C++ Implementierung (build system) muss Programme erzeugen, deren beobachtbares Verhalten dem vom Standard spezifizierten entsprechen. Alle weiteren Spezifikationen im Standard (z.B. Resultat von Addition) muss nur für beobachtbares Verhalten eingehalten werden $\Longrightarrow$ erlaubt Optimierungen.

Beobachtbares Verhalten

- Speicherzugriff auf als `volatile` markierte Daten
- I/O-Funktionen der (Standard-)Bibliothek

Beispiel

```
1  void unobservable(int& p)
2  {
3    int undetermined = 40;
4    undetermined += 2;
5
6    p = i;
7  }
8
9  void observable(
10   volatile int& p)
11 {
12   p = 42;
13 }
```

Beispiel

```
1  void unobservable(int& p)
2  {
3    int undetermined = 40;
4    undetermined += 2;
5
6    p = i;
7  }
8
9  void observable(
10   volatile int& p)
11 {
12   p = 42;
13 }
```

```
16 int main()
17 {
18   volatile int visible = 21;
19   visible *= 2;
```

Beispiel

```
1  void unobservable(int& p)
2  {
3    int undetermined = 40;
4    undetermined += 2;
5
6    p = i;
7  }
8
9  void observable(
10   volatile int& p)
11 {
12   p = 42;
13 }
```

```
16 int main()
17 {
18   volatile int visible = 21;
19   visible *= 2;
20
21   int invisible = 44;
22   invisible -= 2;
```

Beispiel

```
1  void unobservable(int& p)
2  {
3    int undetermined = 40;
4    undetermined += 2;
5
6    p = i;
7  }
8
9  void observable(
10   volatile int& p)
11 {
12   p = 42;
13 }
```

```
16 int main()
17 {
18   volatile int visible = 21;
19   visible *= 2;
20
21   int invisible = 44;
22   invisible -= 2;
23
24   int not_yet_vis = 84;
25   not_yet_vis /= 2;
```

Beispiel

```
1  void unobservable(int& p)
2  {
3    int undetermined = 40;
4    undetermined += 2;
5
6    p = i;
7  }
8
9  void observable(
10   volatile int& p)
11 {
12   p = 42;
13 }
```

```
16 int main()
17 {
18   volatile int visible = 21;
19   visible *= 2;
20
21   int invisible = 44;
22   invisible -= 2;
23
24   int not_yet_vis = 84;
25   not_yet_vis /= 2;
26
27   observable(visible);
28   unobservable(invisible);
29   unobservable(not_yet_vis);
```

Beispiel

```
1  void unobservable(int& p)
2  {
3    int undetermined = 40;
4    undetermined += 2;
5
6    p = i;
7  }
8
9  void observable(
10   volatile int& p)
11 {
12   p = 42;
13 }
```

```
16 int main()
17 {
18   volatile int visible = 21;
19   visible *= 2;
20
21   int invisible = 44;
22   invisible -= 2;
23
24   int not_yet_vis = 84;
25   not_yet_vis /= 2;
26
27   observable(visible);
28   unobservable(invisible);
29   unobservable(not_yet_vis);
30
31   std::cout << visible;
32   std::cout << not_yet_vis;
33 }
```

Was kann C++?

Speicherzugriff

- Berechnungen, Prüfungen usw.
- Kopien an bestimmte Orte

Was kann C++?

Speicherzugriff

- Berechnungen, Prüfungen usw.
- Kopien an bestimmte Orte

I/O-Funktionen

- Zeichenweise Ein-/Ausgabe (`cin`, `cout`) ABER keine Konsole/Shell
- Zugriff auf Dateien (`ofstream`, `ifstream`) ABER keine Ordnerstruktur

Programm-Bibliotheken

Programm-Bibliotheken enthalten code (Anweisungen) und/oder Programmierhilfen.

Szenarien für Programm-Bibliotheken:

- Modularisierung des eigenen Programms
- gebräuchliche Funktionalitäten (z.B. Ringpuffer)
- Abstraktion / Verstecken von Implementierung (z.B. SDL, qt)
- Verringerung der build-Dauer

Mehr Funktionalität durch Bibliothek

- Der Compiler kennt die target platform (z.B. Linux auf x64)!

Mehr Funktionalität durch Bibliothek

- Der Compiler kennt die target platform (z.B. Linux auf x64)!
- Compiler-spezifische Anweisungen können in Plattform-abhängige Befehle übersetzt werden

Mehr Funktionalität durch Bibliothek

- Der Compiler kennt die target platform (z.B. Linux auf x64)!
- Compiler-spezifische Anweisungen können in Plattform-abhängige Befehle übersetzt werden
- bestimmte Anweisungen (z.B. für CPU) für Kommunikation mit Hardware oder Betriebssystem

Mehr Funktionalität durch Bibliothek

- Der Compiler kennt die target platform (z.B. Linux auf x64)!
- Compiler-spezifische Anweisungen können in Plattform-abhängige Befehle übersetzt werden
- bestimmte Anweisungen (z.B. für CPU) für Kommunikation mit Hardware oder Betriebssystem

Entweder: die Compiler-spezifischen Anweisungen direkt in den Quellcode
Oder:

Mehr Funktionalität durch Bibliothek

- Der Compiler kennt die target platform (z.B. Linux auf x64)!
- Compiler-spezifische Anweisungen können in Plattform-abhängige Befehle übersetzt werden
- bestimmte Anweisungen (z.B. für CPU) für Kommunikation mit Hardware oder Betriebssystem

Entweder: die Compiler-spezifischen Anweisungen direkt in den Quellcode
Oder: diese Anweisungen in Bibliotheken verstecken

Beispiele für verschiedene Bibliotheken

Modularisierung

- Bitmap-Framework als static library
- cuviso device com libs

Beispiele für verschiedene Bibliotheken

Modularisierung

- Bitmap-Framework als static library
- cuviso device com libs

Gebräuchliche Funktionalitäten

- Boost.Math, Boost.Algorithm, Boost.Container (header-only!)
- FFT-Bibliotheken

Beispiele für verschiedene Bibliotheken

Modularisierung

- Bitmap-Framework als static library
- cuviso device com libs

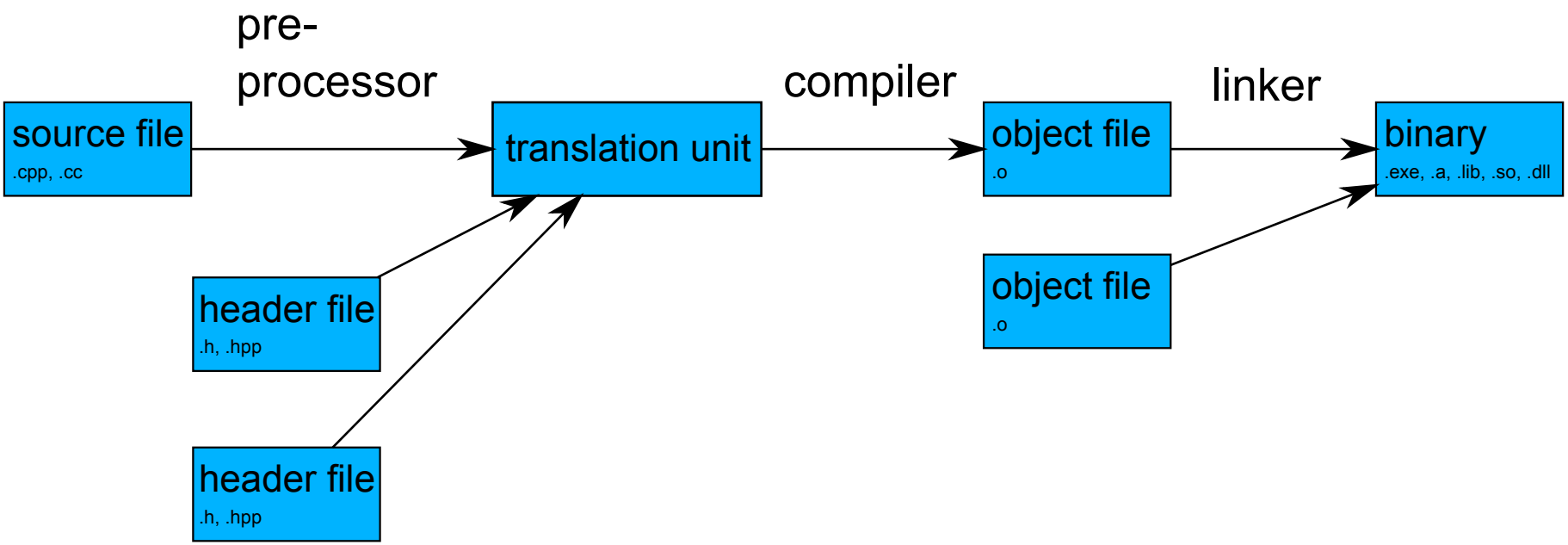
Gebräuchliche Funktionalitäten

- Boost.Math, Boost.Algorithm, Boost.Container (header-only!)
- FFT-Bibliotheken

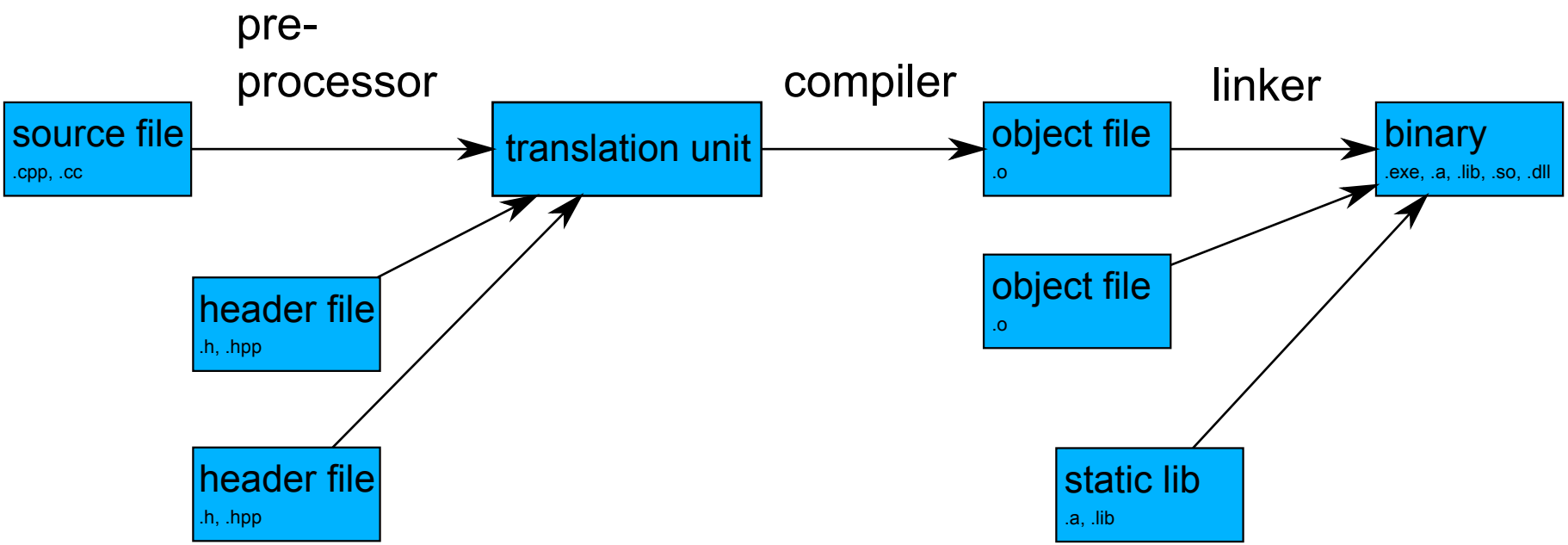
Abstraktion von der Implementierung

- Standard-I/O-Bibliotheken
- pthreads, Boost.Thread
- qt (mehr als nur eine lib)
- Windows SDK (mehr als nur eine lib)

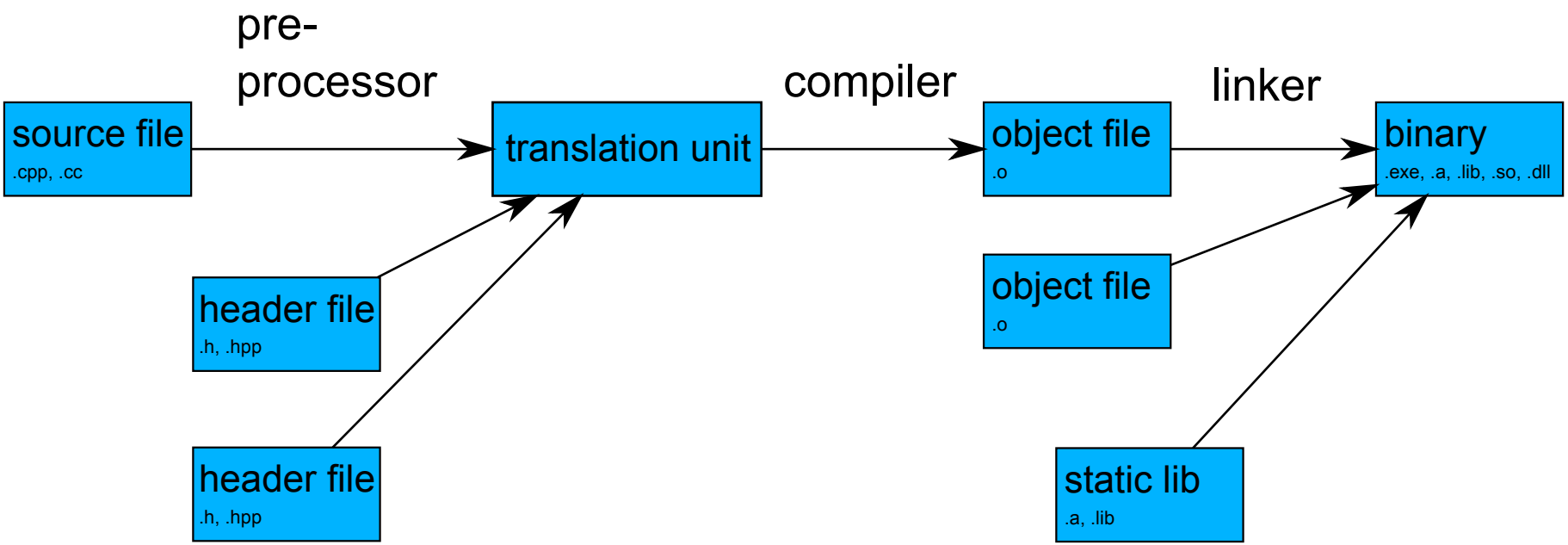
Zusammenspiel mit dem Programm



Zusammenspiel mit dem Programm



Zusammenspiel mit dem Programm



Terminologie

statically linked program library, kurz *static library*

Eine vom Linker während des build-Vorgangs fest eingebaute Bibliothek.

Zusammenspiel mit C++

C++ selbst bietet keinerlei Unterstützung für static libraries

Zusammenspiel mit C++

C++ selbst bietet keinerlei Unterstützung für static libraries

- Wie auch schon das Mitteilen, welche object files der Linker verarbeiten soll, ist die Verwendung von static libraries abhängig vom verwendeten Linker.

Zusammenspiel mit C++

C++ selbst bietet keinerlei Unterstützung für static libraries

- Wie auch schon das Mitteilen, welche object files der Linker verarbeiten soll, ist die Verwendung von static libraries abhängig vom verwendeten Linker.
- Zumeist werden static libraries wie vorkompilierte object files behandelt, es gelten also dieselben linkage-Regeln.

Zusammenspiel mit C++

C++ selbst bietet keinerlei Unterstützung für static libraries

- Wie auch schon das Mitteilen, welche object files der Linker verarbeiten soll, ist die Verwendung von static libraries abhängig vom verwendeten Linker.
- Zumeist werden static libraries wie vorkompilierte object files behandelt, es gelten also dieselben linkage-Regeln.
- Header-Files zu static libraries ermöglichen den Zugriff im Programm auf Namen in der Bibliothek mit external linkage.

Zusammenspiel mit C++

C++ selbst bietet keinerlei Unterstützung für static libraries

- Wie auch schon das Mitteilen, welche object files der Linker verarbeiten soll, ist die Verwendung von static libraries abhängig vom verwendeten Linker.
- Zumeist werden static libraries wie vorkompilierte object files behandelt, es gelten also dieselben linkage-Regeln.
- Header-Files zu static libraries ermöglichen den Zugriff im Programm auf Namen in der Bibliothek mit external linkage.

Eine static library lässt sich

- schreiben wie „ein Programm ohne main“
- nutzen wie ein weiterer Header (compiling) und wie eine weitere object file (linking)

Zusammenspiel mit C++

C++ selbst bietet keinerlei Unterstützung für static libraries

- Wie auch schon das Mitteilen, welche object files der Linker verarbeiten soll, ist die Verwendung von static libraries abhängig vom verwendeten Linker.
- Zumeist werden static libraries wie vorkompilierte object files behandelt, es gelten also dieselben linkage-Regeln.
- Header-Files zu static libraries ermöglichen den Zugriff im Programm auf Namen in der Bibliothek mit external linkage.

Eine static library lässt sich

- schreiben wie „ein Programm ohne main“
- nutzen wie ein weiterer Header (compiling) und wie eine weitere object file (linking)

Aber: Gott tötet kleine Kätzchen wenn man einfach Programm X nimmt, die main weglässt, und das eine Bibliothek nennt.

Verwendung von static libs (cmd-line)

Übliche cmd-line-Parameter

- Kompilieren: `compile program.cpp -o program.o`
- Linken (1): `link program.o libmyLib.a -o program`
- Linken (2): `link program.o -lmyLib -o program`

Verwendung von static libs (cmd-line)

Übliche cmd-line-Parameter

- Kompilieren: `compile program.cpp -o program.o`
- Linken (1): `link program.o libmyLib.a -o program`
- Linken (2): `link program.o -lmyLib -o program`

Vereinfachung: library search paths

- Libraries sind zumeist in sich abgeschlossen, und daher prädestiniert für einen eigenen Ordner.
- Libraries für gebräuchliche Funktionalitäten sind prädestiniert für Projekt-unabhängige Orte/Ordner.

Verwendung von static libs (cmd-line)

Übliche cmd-line-Parameter

- Kompilieren: `compile program.cpp -o program.o`
- Linken (1): `link program.o libmyLib.a -o program`
- Linken (2): `link program.o -lmyLib -o program`

Vereinfachung: library search paths

- Libraries sind zumeist in sich abgeschlossen, und daher prädestiniert für einen eigenen Ordner.
- Libraries für gebräuchliche Funktionalitäten sind prädestiniert für Projekt-unabhängige Orte/Ordner.

include search path: Pfad, von wo aus der Compiler nach `#include <path/file.ext>` sucht
Angabe durch Parameter: `compile -I"include_search_path"`

Verwendung von static libs (cmd-line)

Übliche cmd-line-Parameter

- Kompilieren: `compile program.cpp -o program.o`
- Linken (1): `link program.o libmyLib.a -o program`
- Linken (2): `link program.o -lmyLib -o program`

Vereinfachung: library search paths

- Libraries sind zumeist in sich abgeschlossen, und daher prädestiniert für einen eigenen Ordner.
- Libraries für gebräuchliche Funktionalitäten sind prädestiniert für Projekt-unabhängige Orte/Ordner.

include search path: Pfad, von wo aus der Compiler nach `#include <path/file.ext>` sucht
Angabe durch Parameter: `compile -I"include_search_path"`

library search path: Pfad, wo der Linker eine Bibliothek sucht
Angabe durch Parameter: `link -L"lib_search_path"`

Verwendung von static libs (IDE)

- In IDEs gibt es üblicherweise GUIs zur Konfiguration der cmd-line-Parameter von Compiler und Linker.

Verwendung von static libs (IDE)

- In IDEs gibt es üblicherweise GUIs zur Konfiguration der cmd-line-Parameter von Compiler und Linker.
- Im speziellen: Verwaltung von static libs

Verwendung von static libs (IDE)

- In IDEs gibt es üblicherweise GUIs zur Konfiguration der cmd-line-Parameter von Compiler und Linker.
- Im speziellen: Verwaltung von static libs
- Je nach IDE an unterschiedlicher Stelle und mit unterschiedlicher Mächtigkeit

Verwendung von static libs (IDE)

- In IDEs gibt es üblicherweise GUIs zur Konfiguration der cmd-line-Parameter von Compiler und Linker.
- Im speziellen: Verwaltung von static libs
- Je nach IDE an unterschiedlicher Stelle und mit unterschiedlicher Mächtigkeit
- Manuell die Pfade (include search path, library search path) anzugeben ist eine *sehr* rudimentäre Methode, aufwändig und fehleranfällig.

Verwendung von static libs (IDE)

- In IDEs gibt es üblicherweise GUIs zur Konfiguration der cmd-line-Parameter von Compiler und Linker.
- Im speziellen: Verwaltung von static libs
- Je nach IDE an unterschiedlicher Stelle und mit unterschiedlicher Mächtigkeit
- Manuell die Pfade (include search path, library search path) anzugeben ist eine *sehr* rudimentäre Methode, aufwändig und fehleranfällig.

Fazit: Am Ende sind es cmd-line-Parameter von Compiler und Linker, man kann sie auf unterschiedlichen Wegen setzen.

Hinweise zur Verwendung

- üblicherweise werden static libraries in Ordner gegliedert:
  `mylib/inc` $\rightarrow$ include search path
  `mylib/lib` $\rightarrow$ library search path

Hinweise zur Verwendung

- üblicherweise werden static libraries in Ordner gegliedert:
  `mylib/inc` → include search path
  `mylib/lib` → library search path
- library header (`.h`) → Compiler
  library file (`.a`, `.lib`) → Linker

Hinweise zur Verwendung

- üblicherweise werden static libraries in Ordner gegliedert:
  `mylib/inc` → include search path
  `mylib/lib` → library search path
- library header (`.h`) → Compiler
  library file (`.a`, `.lib`) → Linker
- (precompiled) static libraries sind Compiler-, Linker- und Plattform-abhängig! Auch mglw. von deren Version!

Hinweise zur Verwendung

- üblicherweise werden static libraries in Ordner gegliedert:
  `mylib/inc` → include search path
  `mylib/lib` → library search path
- library header (`.h`) → Compiler
  library file (`.a`, `.lib`) → Linker
- (precompiled) static libraries sind Compiler-, Linker- und Plattform-abhängig! Auch mglw. von deren Version!
- static libraries erhält man vorwiegend aus dem Internet/www, unter Linux auch über die Paketverwaltung

Hinweise zur Verwendung

- üblicherweise werden static libraries in Ordner gegliedert:
  `mylib/inc` $\rightarrow$ include search path
  `mylib/lib` $\rightarrow$ library search path
- library header (`.h`) $\rightarrow$ Compiler
  library file (`.a`, `.lib`) $\rightarrow$ Linker
- (precompiled) static libraries sind Compiler-, Linker- und Plattform-abhängig! Auch mglw. von deren Version!
- static libraries erhält man vorwiegend aus dem Internet/www, unter Linux auch über die Paketverwaltung
- manche static libraries (z.B. SDL) binden intern (versteckt) eine DLL / ein SO ein $\Longrightarrow$ achte auf Abhängigkeiten!

2. SDL: Simple Directmedia Layer
   - Was ist das?
   - Grundkonzepte


SDL
Simple Directmedia Layer

Was ist das?

Eine **in C geschriebene** plattformunabhängige modulare Multimediabibliothek zum Zugriff auf:

- Grafik (sdl, sdl_gfx)
- Audio (sdl_sound, sdl_mixer)
- Eingabegeräte (sdl)
- Netzwerk (sdl_net)
- Und einiges mehr (sdl_image, sdl_rtf, sdl_ttf, ...)

Wir konzentrieren uns nur auf das Hauptmodul (Grafik, Eingabegeräte)

Surfaces

`SDL_Surface` repräsentiert eine Oberfläche, auf die gezeichnet werden kann:

- der Bildschirm
- eine Textur bzw. ein Bild

Außerdem kann von `SDL_Surface` natürlich auch gelesen werden.

Grundlegende Operationen auf Surfaces

- `SDL_CreateRGBSurface`: Erzeuge ein neues Surfaceobjekt.
- `SDL_SetVideoMode`: Öffne ein Fenster und gebe ein Surfaceobjekt zurück, das dessen Hintergrund repräsentiert.
- `SDL_LoadBMP`: Lade ein Surfaceobjekt aus einer .bmp Bilddatei.
- `SDL_FreeSurface`: Lösche ein Surfaceobjekt.
- `SDL_FillRect`: Fülle einen Teil eines Surfaceobjekts mit einer Farbe.
- `SDL_BlitSurface`: Kopiere einen Ausschnitt aus einem Surfaceobjekt in ein anderes.

Events

Ereignisse wie etwa Tastendrücke und Mausbewegungen werden über die Struktur `SDL_Event` repräsentiert. Sie werden von SDL in einem Puffer abgelegt, aus dem man sich die Ereignisse mit `SDL_PollEvent` oder `SDL_WaitEvent` nach und nach abholen kann.
Normalerweise in der Hauptschleife jeweils am Anfang.

Events

```
1  while(1){
2    SDL_Event event;
3    /* Loop until there are no events left on the queue */
4    while(SDL_PollEvent(&event)) {
5      switch(event.type) { /* Process the appropriate event type */
6        case SDL_KEYDOWN:  /* Handle a KEYDOWN event */
7          printf("Oh! Key press\n");
8          break;
9        case SDL_MOUSEMOTION:
10         .
11         .
12         .
13       default: /* Report an unhandled event */
14         printf("I don't know what this event is!\n");
15     }
16   }
17   // do some other stuff here -- draw your app,
18   // play your music, perform other operations...
19 }
```

3 Praxis

Praxis!

- Aufgabe 1: SDL Einstieg
- Aufgabe 2: Schach GUI

`https://github.com/kit-cpp-workshop/workshop-ss12-07`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`