Objektorientierte Programmierung
(Winter 2010/2011)

Kapitel 18: Der Linker: Getrennte Übersetzung

- Programme aus mehreren Modulen
- Deklaration vs. Definition (Wiederholung)
- Linker, Objektdateien, Dynamisches Linken
- Make (Wiederholung und Erweiterung)

Inhalt

Motivation (1)

- Die meisten Programme bestehen aus mehreren Modulen (C++-Quelldateien), die getrennt übersetzt werden. Gründe dafür sind:
  - ◇ Ein großes Programm ist besser zu verstehen, wenn es in natürliche Stücke unterteilt ist.
  - ◇ Mehrere Programmierer arbeiten zusammen.

    Verschiedene Personen können nicht die gleiche Datei zur gleichen Zeit editieren (nur mit Spezialsoftware für "Collaborative Work").
  - ◇ Das Programm ist groß.

    Die Übersetzung von einer Million Zeilen dauert etwas.

Motivation (2)

- Gründe für Aufspaltung eines Programms in mehrere Quelldateien (Forts.):
  - ◇ Wiederverwendbare Codebausteine.

    Eine C++-Datei kann Bestandteil mehrerer Programme sein.
  - ◇ Mehrere Varianten einzelner Module.

    Z.B. eine für UNIX/Linux und eine für Windows.
  - ◇ Module können in verschiedenen Programmiersprachen entwickelt sein.

    Z.B. besonders Performance-kritische Teile, oder Teile mit direktem Zugriff auf die Hardware in Assembler (in C++ fast nie nötig). Oder existierender Programmcode (in anderen Sprachen geschrieben) muss in das neue Programm integriert werden.

Motivation (3)

- Module vs. Klassen:
  - ◇ Ein Modul enthält meist eine Klasse oder eine kleine Anzahl zusammengehöriger Klassen.

    Z.B. eine Klasse für Listen (Anker u.s.w.) und eine Klasse für Listenknoten zusammen in einem Modul.
  - ◇ Module gab es schon in C.

    In gewisser Weise nahmen Module auch Aufgaben war, für die man heute eher Klassen verwendet. Ein Modul enthielt typischerweise einen Datentyp zusammen mit Funktionen für seine Operationen. Auch heute werden Module in vielen Programmiersprachen verwendet, die nicht objektorientiert sind (z.B. Prolog).
  - ◇ Modul ist die nächstgrößere Strukturierungseinheit nach der Klasse.

Motivation (4)

- Funktionen, Klasse, Module haben eine Abschottungsfunktion: Die Sichtbarkeit/Zugreifbarkeit von Bezeichnern (z.B. für Variablen) wird eingeschränkt:
  - ◇ Namenskonflikte werden vermieden.
  - ◇ Zugriff nur über eine definierte Schnittstelle.

    Andere Dinge sind vor direktem Zugriff geschützt.
  - ◇ Um die jeweilige Einheit benutzen zu können, braucht man nicht interne Details zu verstehen.
  - ◇ Zusammengehörigkeit der Teile im Innern wird betont (macht Programmstruktur klarer).

Motivation (5)

- Die Klassen/Funktionen eines Moduls sollen
  - ◇ nach innen einen starken Zusammenhalt haben,

    Also sich gegenseitig aufrufen, auf den gleichen Daten arbeiten, die gleiche Art von Aufgaben erfüllen, oder in anderer Weise sehr eng zusammengehören.
  - ◇ nach außen eine kleine / einfache Schnittstelle haben.
- Man sollte die Strukturierung in Module so wählen, dass man an Übersicht gewinnt.

  Einzelne Module können auch sehr klein sein, aber wenn man sehr viele sehr kleine Module hat, nutzt man diesen Mechanismus nicht optimal. Mir scheinen ca. 50–500 Zeilen pro Modul normal.

Getrennte Übersetzung (1)

- Ein C++-Programm definiert fast nie alle Funktionen, die es aufruft.
- Schon das “Hello, World” Beispiel hat zur Ausgabe << benutzt. Das ist nur eine andere Notation für den Aufruf der Funktion/Methode `operator<<`.
- Diese Funktion ist in der Include-Datei `iostream`
  - ◇ deklariert (Kopf mit Parametern: Anzahl und Datentypen, sowie Ergebnis-/Rückgabe-Typ), aber
  - ◇ nicht definiert (kein Rumpf/Implementierung).

Getrennte Übersetzung (2)

- Die Funktion `operator<<` ist auch in C++ geschrieben (z.B. von den Entwicklern des Compilers).

  Genauer gibt es mehrere solche Funktionen für unterschiedliche Datentypen.

- Die C++-Datei mit der Definition von `operator<<` wurde übersetzt, und das Ergebnis (Objektdatei mit Maschinencode) in die Standard-Bibliothek eingefügt.

  Eine Bibliothek besteht aus mehreren Objektdateien. Natürlich gibt es neben der Standard-Bibliothek auch noch andere Bibliotheken, man kann auch selbst welche entwickeln.

Getrennte Übersetzung (3)

- Nachdem die Datei "`hello.cpp`" (mit dem "Hello, World" Beispiel) übersetzt war (auch in eine Objektdatei), wurde der Linker aufgerufen, der
  - ◇ die nötigen Objektdateien (u.a. mit der Implementierung von `operator<<`) aus der Bibliothek zum Ergebnisprogramm hinzubindet.

    Er muß also erreichen, dass der Prozeduraufruf in Ihrem Programm den Maschinencode aus der Bibliothek ausführt.

- So enthält das ausführbare Programm am Ende auch die Implementierung von `operator<<`.

  Bei dynamischem Linken ist es etwas komplizierter (s.u.).

Getrennte Übersetzung (4)

- Wir machen also schon die ganze Zeit über Gebrauch von getrennter Übersetzung und mehreren Modulen.
- Es ist nur nicht so auffällig, weil
  - ◇ der Compiler bisher automatisch den Linker aufruft,

    und die Objektdatei, die zwischendurch entsteht, wieder löscht,

  - ◇ wir die Standard-Bibliothek nicht selbst geschrieben haben.

    Am Anfang nimmt man sie als gegeben an: Jedes Programm verwendet sie.

Beispiel (1)

- Folgende Quelldatei `a.cpp` verwendet eine Funktion `f`, die in ihr nicht definiert ist:

```
#include <iostream>
using namespace std;

int f(int n); // Deklaration, nicht Definition

int main()
{
    int i = f(1);
    cout << "Das Ergebnis ist: " << i << "\n";
    return 0;
};
```

Beispiel (2)

- Die Funktion `f` kann nun in einer anderen Quelldatei `b.cpp` definiert werden:

```
int f(int n) // Deklaration und Definition
{
    int s = 0;
    for(int i = 1; i <= n; i++)
        s = s + i;
    return s;
};
```

- Man kann auch jetzt mit einem Befehl compilieren und linken: `g++ a.cpp b.cpp`

Beispiel (3)

- Man möchte aber die Dateien getrennt übersetzen.
- Der Aufruf `g++ a.cpp` ergibt:

```
/tmp/cc5s6OQL.o: In function ‘main’:
a.cpp:(.text+0x11): undefined reference to ‘f(int)’
collect2: ld returned 1 exit status
```

- Die Fehlermeldung kommt hier vom Linker “`ld`”.

  Der Name ist eher historisch. Der Linker (deutsch “Binder”) ist aus dem “loader” (deutsch “Lader”) entstanden, der Programme zur Ausführung in den Hauptspeicher geladen hat.

  Die Compilierung in die Objektdatei (`.o`) verlief dagegen fehlerfrei. Man sieht hier auch, dass implizit immer eine Objektdatei erzeugt wird, aber im Verzeichnis für temporäre Dateien (hinterher gelöscht).

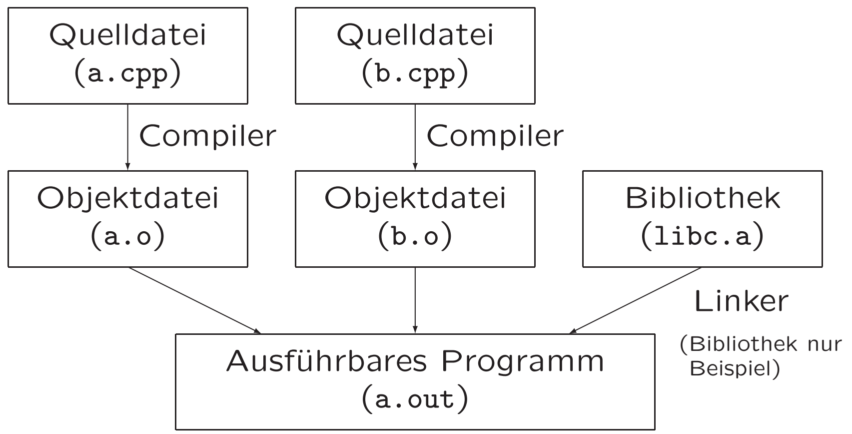
Beispiel (4)

- Wenn man nur den Compiler aufrufen will, und nicht den Linker, muß man beim `g++` die Option `-c` angeben: `g++ -c a.cpp`
- In diesem Fall wird die Objektdatei `a.o` im gleichen Verzeichnis angelegt und natürlich nicht gelöscht.
- Entsprechend erzeugt man `b.o` (aus `b.cpp`).
- Dann kann man den Linker aufrufen mit

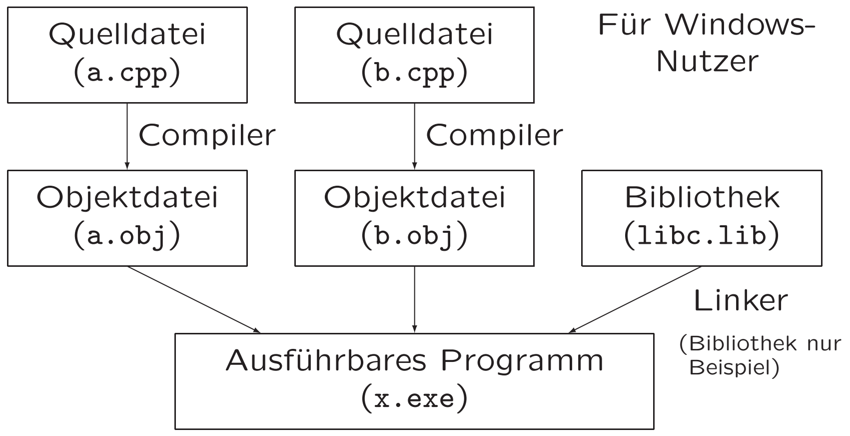
  `g++ a.o b.o`

  Dies ruft `ld` auf und fügt automatisch die Standard-C++ Bibliothek hinzu. Der Linker `ld` selbst ist nicht spezifisch für C++.

Beispiel (5)



Beispiel (6)



Beispiel (7)

- Tatsächlich werden mehrere Bibliotheken und Objektdateien hinzugebunden.

  Mit der Option `-v` ("verbose") gibt `g++` mehr Informationen. Objektdateien: `crt1.o`, `crti.o`, `crtbegin.o`, `crtend.o`, `crtn.o` (crt steht für "C run time": Diverse Initialisierungen und Abschlußarbeiten). Bibliotheken: `libstdc++`, `libm` (mathematische Funktionen), `libgcc_s`, `libgcc`, `libc` (Standard C-Bibliothek).
- Mit `-o prog` ("output") kann man den Namen des ausführbaren Programms setzen.
- Wenn man `a.cpp` verändert, muß man nur diese Datei neu compilieren und anschließend den Linker aufrufen (man spart den Compilerlauf für `b.cpp`).

Inhalt

Include-Dateien (1)

- Natürlich muß die Funktion `f` in beiden Modulen gleich deklariert sein. Angenommen, man deklariert `f` in `a` mit Rückgabetyp `float`:

```
float f(int n);
```

- Man bekommt dann beim Compilieren und Linken keinerlei Fehlermeldung:
  - ◇ Der Compiler sieht nicht beide Dateien gleichzeitig, kann die Inkonsistenz also nicht bemerken.
  - ◇ Der Linker versteht nichts von C++.

Include-Dateien (2)

- Man bekommt aber, wenn man `f(1)` aufruft, ein völlig falsches Ergebnis: -2147483648.
  - ◇ Die aufgerufene Funktion `f` liefert den `int`-Wert 1.
  - ◇ Der Aufrufer `main` interpretiert dieses Bitmuster als `float`.
- Früher (bei C) wären auch falsche Parametertypen und Parameteranzahlen nicht aufgefallen.
- Da C++ aber Überladen erlaubt, codiert der Compiler die Typen in den Symbolen in der Objektdatei.

  Daher liefert der Linker einen Fehler, wenn man den Parametertyp von `int` in `float` ändert. Die Funktionen heißen für ihn unterschiedlich.

Include-Dateien (3)

- Um dieses Risiko zu vermeiden, ist es üblich, die von dem Modul `b.cpp` exportierten Funktionen in einer Datei `b.h` zu deklarieren.
- Die Datei enthält also nur die Funktionsdeklaration:

```
int f(int n);
```

- Mit einer Include-Anweisung in `a.cpp` und in `b.cpp` fordert man den Compiler auf, diese Datei beim Compilieren zu lesen:

```
#include "b.h"
```


Include-Dateien (4)

b.h:

```
int f(int n);
```

b.cpp:

```
#include "b.h"

int f(int n)
{
    int s = 0;
    for(int i = 1; i <= n; i++)
        s = s + i;
    return s;
};
```


Include-Dateien (5)

- Die Datei `a.cpp` sieht jetzt also so aus (`include`-Anweisung statt expliziter Deklaration von `f`):

a.cpp:

```
#include <iostream>
using namespace std;
#include "b.h"
int main()
    ...
```

- Feinheit der `include`-Syntax:
  - ◇ <X>: Sucht X nur in Systemverzeichnissen.
  - ◇ "X": Sucht X zuerst im aktuellen Verzeichnis.

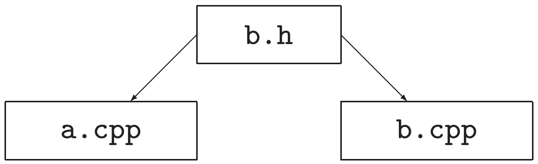
Include-Dateien (6)

- `b.cpp` enthält die gleiche `include`-Anweisung.
- Dadurch sieht der Compiler erst die Deklaration aus `b.h`, dann die Definition aus `b.cpp`.
- So kann der Compiler die Konsistenz prüfen. Er gibt eine Fehlermeldung aus, wenn sich die Rückgabetypen unterscheiden.

  Er gibt keine Fehlermeldung aus, wenn sich die Parametertypen unterscheiden. In diesem Fall nimmt er an, dass es sich um eine überladene Funktion handelt, dass es also zwei verschiedene Funktionen `f` gibt. Aber dieser Fehler wird ja bereits vom Linker gefunden.

Include-Dateien (7)

- Der entscheidene Punkt ist also, dass die Funktion `f` nur einmal zentral deklariert wird (in der Include-Datei `b.h`), und dann sowohl in
  - ◇ der Datei mit der Definition (Implementierung) der Funktion (`b.cpp`) included wird, als auch in
  - ◇ allen Quelldateien, die diese Funktion benutzen (aufrufen), im Beispiel nur `a.cpp`.



Include-Dateien (8)

- Include-Dateien können beliebigen C++-Code enthalten.
- Weil sie vom Compiler meist mehrfach gelesen werden (bei der Übersetzung verschiedener Module), enthalten sie üblicherweise nur Deklarationen von
  - ◇ Funktionen
  - ◇ Datentypen
  - ◇ Konstanten
  - ◇ Klassen

Include-Dateien (9)

- Die Include-Datei `X.h` enthält also die Schnittstelle des Moduls `X`, d.h. das,
  - ◇ was nach außen sichtbar sein soll,
  - ◇ was der Benutzer des Moduls wissen muss, bzw.
  - ◇ was der Compiler braucht, um Aufrufe der Funktionen des Moduls zu übersetzen.
- Die zugegehörige Datei `X.cpp` die Implementierung.

  Manchmal stehen Teile der Implementierung schon in der `.h`-Datei, weil der Compiler sie zur Übersetzung braucht. Z.B. sind die Attribute einer Klasse `private`, und für den Benutzer der Klasse nicht interessant, dennoch braucht der Compiler sie, um Objekte der Klasse erzeugen zu können (zur Bestimmung der Speichergröße).

Include und Klassen (1)

- Bei objektorientierter Programmierung enthalten die Include-Dateien natürlich eher Klassen als einzelne Funktions- und Typ-Deklarationen.
- Falls die Rümpfe aller Methoden schon in der Klassendeklaration angegeben sind (in der `.h`-Datei), braucht man keine `.cpp`-Datei mehr.

  Für Klassenattribute (`static`) braucht man allerdings eine Definition außerhalb der Klassendeklaration, und nur eine (auch wenn die Klasse in mehreren Modulen verwendet wird). Diese Definition kommt dann in die `.cpp`-Datei.

Include und Klassen (2)

- Es ist üblich, mindestens größere Funktionsrümpfe in die `.cpp`-Datei zu schreiben.
  - ◇ Es erhöht die Übersichtlichkeit.

    Die Klassendeklaration selbst wird kleiner (die müßte der Benutzer der Klasse lesen).
  - ◇ Es spart Speicherplatz.

    Funktionen, die in der Klasse (mit Rumpf) definiert sind, sind automatisch `inline`: Der Compiler fügt normalerweise den Rumpf der Funktion an allen Stellen ein, an denen die Funktion aufgerufen wird. Der Maschinencode ist dann mehrfach enthalten.
  - ◇ Es kostet etwas Laufzeit.

    Wenn der Rumpf der Funktion anstelle des Aufrufs eingefügt wird, spart das die Befehle für den expliziten Funktionsaufruf.

External/Internal Linkage (1)

- Eine "Translation Unit" ("Übersetzungseinheit") ist das, was in einem Compilerlauf übersetzt wird.

  Im obigen Beispiel gibt es also zwei "Translation Units": (1) `a.cpp` zusammen mit `b.h` und (2) `b.cpp` zusammen mit `b.h`.
- Dinge, die nur während eines Compilerlaufes (innerhalb einer Translation Unit) bekannt sind, haben "Internal Linkage".
- Dinge, die in den Objektdateien abgelegt werden, und damit über die "Translation Unit" hinaus bekannt sind, haben "External Linkage".

External/Internal Linkage (2)

- Definitionen globaler Variablen haben z.B. "External Linkage":
  - ◇ Wenn zwei Module eine globale Variable `int n` definieren, kommt es beim Linken zu einem Fehler (Namenskonflikt):

```
b.o:(.bss+0x0): multiple definition of ‘n’
a.o:(.bss+0x0): first defined here
```

    `bss` ist ein Segment im Objektprogramm für globale Variablen, die nicht explizit initialisiert sind, und beim Start des Programms mit Null-Bits gefüllt werden. Nach dem C/C++-Standard sind globale Variablen automatisch mit 0 initialisiert, wenn nichts anderes angegeben ist. (bss steht historisch für "block started by symbol", heute passt besser "blank static storage".)

External/Internal Linkage (3)

- Man kann bei globalen Variablen auch nicht verhindern, dass auf sie von außen zugegriffen wird (sie sind eben "global").
  - ◇ Angenommen, `int n` ist in `b.cpp` definiert.
  - ◇ Dann kann `a.cpp` eine Deklaration der Variablen enthalten, die keine Definition ist (also keinen Speicherplatz reserviert). Dies geschieht mit:

    ```
    extern int n;
    ```
  - ◇ Nun hat der Programmcode in `a.cpp` Zugriff auf die Variable `n`, die in `b.cpp` definiert wurde.

External/Internal Linkage (4)

- Die Definition einer Variablen (mit Reservierung von Speicherplatz) in einer Include-Datei führt zum Fehler beim Linken ("doubly defined symbol").

  Wenn die Include-Datei von mehreren Modulen verwendet wird.
- Lösung: Include-Datei enthält nur die Deklaration:

  ```
  extern int n;
  ```
- Genau eine `.cpp`-Datei muß die Variable dann auch definieren, ggf. mit expliziter Initialisierung:

  ```
  int n = 3;
  ```


External/Internal Linkage (5)

- Will man für eine globale Variable "Internal Linkage" haben, so muß man "`static`" voranstellen:

  ```
  static int n = 3;
  ```

  Das geht auch für Funktionen, die außerhalb von Klassen deklariert sind. Innerhalb von Klassen hat "`static`" eine andere Bedeutung.
- Das Symbol `n` wird dann nicht in der Objektdatei abgelegt (Speicherplatz wird natürlich reserviert).

  Andere Module können auf diese Variable dann nicht zugreifen. Sie können sich natürlich ein eigene Variable "`n`" anlegen.
- Diese Verwendung von `static` gilt aber als veraltet.

  Sie wird nur noch aus Kompatibilitätsgründen unterstützt.

External/Internal Linkage (6)

- Es wird empfohlen, für globale Variablen und für Funktionen, die von keinem anderen Modul aus zugreifbar sein sollen, einen "unnamed namespace" zu verwenden:

  ```
  namespace {
      int n = 3;
      int f() { ...}
  }
  ```

  Die Symbole in dem Namespace können bei diesem Übersetzerlauf verwendet werden (wie mit einer impliziten `using` Deklaration), aber ein Zugriff von außen ist nicht möglich (da der Namespace keinen Namen hat, kann man ihn nicht explizit ansprechen).

External/Internal Linkage (7)

- Konstanten wie z.B.

```
const int n = 3;
```

  haben "Internal Linkage", d.h. sie sind nur beim aktuellen Compilerlauf bekannt, und werden normalerweise nicht in der Objektdatei abgelegt.

- Das ist praktisch, weil sie oft in Include-Dateien stehen (so kein "doubly defined symbol" Fehler).

  Der Compiler möchte ihren Wert möglichst auch gleich überall einsetzen, und nicht explizit Speicherplatz für sie reservieren. Das geht nicht immer (es ist z.B. legal, ihre Adresse zu bestimmen), aber wenn sie in der Objektdatei stehen müssten, würde es überhaupt nicht gehen.

External/Internal Linkage (8)

- Es wäre legal (aber verwirrend), dass `a.cpp`

```
const int n = 3;
```

  enthält, und `b.cpp`

```
const int n = 4;
```

  Aufgrund des "Internal Linkage" hat jedes Modul seine eigene Konstante `n`. Sie stehen nicht in der Objektdatei (zumindest nicht einfach mit Namen `n`), es kommt daher nicht zu einem Fehler beim Linken.

- Konstanten, die man in mehreren Modulen verwenden will, schreibt man in Include-Dateien.

  Damit sind sie zentral definiert und haben überall den gleichen Wert.

External/Internal Linkage (9)

- Für konstante Zeichenketten (C-Stil) muß man folgendes schreiben:

```
const char* const LOGFILE = "/fwsoft/logfile";
```

  Das erste `const` bedeutet, dass die Elemente des Arrays, auf das `LOGFILE` zeigt, nicht verändert werden dürfen. Das zweite `const` bedeutet, dass die Variable `LOGFILE` selbst konstant ist (Ein Zeiger wird durch `*const` konstant. Beim ersten `const` ist es dagegen egal, ob es vor oder hinter `char` steht.). Läßt man das zweite `const` weg, handelt es sich um eine globale Variable, und dann bekommt man natürlich den "doubly defined symbol" Fehler, wenn man diese Anweisung mittels `#include` in mehrere Module einbindet. Natürlich kann man sich auch erst mit "`typedef const char *str_t;`" einen String-Typ (Zeiger auf konstanten Speicherbereich) deklarieren, und dann mit "`const str_t LOGFILE = ...;`" die Konstante.

External/Internal Linkage (10)

- Konstanten können auch "External Linkage" haben. Dazu muss man "`extern`" voranstellen:

```
extern const int n = 3;
```

- Nun kann man von anderen Modulen aus sich auf diese Konstante beziehen mit

```
extern const int n;
```

- Ohne Initialisierung ist es eine Deklaration, mit Initialisierung eine Definition.

  Es ist jetzt einfach eine globale Variable, die nach der Initialisierung nicht mehr geändert werden kann.

Geschachteltes Include (1)

- Oft benötigt eine Klasse andere Klassen, Konstanten oder Datentypen.
- Angenommen, die Include-Datei `b.h` kann nur übersetzt werden, wenn zuvor `c.h` übersetzt wurde.
- Dann sollte man oben in die Datei `b.h` den Befehl

  ```
  #include "c.h"
  ```

  schreiben.
- Nun muß der Benutzer der in `b.h` definierten Klassen/Typen/Funktionen nicht mehr daran denken, zuvor `c.h` einzulesen — das geht automatisch.

Geschachteltes Include (2)

- Es kann jetzt aber passieren, dass die Datei `c.h` bei einem Compilerlauf mehrfach verarbeitet wird:
  - ◇ Eventuell hat der Benutzer von `b.h` doch explizit `c.h` included (weil er die Konstanten etc. auch direkt braucht), oder
  - ◇ er hat eine Datei `d.h` included, die ihrerseits auch `c.h` benötigt und selbst einbindet.
- Bei reinen Deklarationen schadet es nichts, wenn sie mehrfach gelesen werden, aber bei Definitionen von Klassen oder Konstanten ist es ein Fehler.

Geschachteltes Include (3)

- Um die doppelte Verarbeitung einer Include-Datei zu vermeiden, verwendet man meist einen Rahmen aus Präprozessor-Befehlen, die den eigentlichen Inhalt der Datei nur einmal durchlassen:
  - ◇ Oben in der Datei `b.h`:

    ```
    #ifndef B_INCLUDED
    #define B_INCLUDED
    ```

    D.h.: Falls `B_INCLUDED` noch nicht definiert ist, dann definiere es, und lies den Rest der Datei. Sonst wird alles bis zum `#endif` übersprungen. Natürlich kann man das Symbol `B_INCLUDED` auch anders nennen (z.B. `B_H`), jede Datei muss nur ihr eigenes Symbol haben.
  - ◇ Ganz unten: `#endif`

Include-Optimierung (1)

- Wenn eine Klasse `X` nur Zeiger auf Objekte einer Klasse `Y` enthält, muß die Datei `X.h` nicht unbedingt die Datei `Y.h` includen.
- Es reicht in diesem Fall eine Deklaration der Klasse in `X.h`:

  ```
  class Y;
  ```
- Für Zeiger ist die Größe der Objekte der Klasse `Y` nicht wichtig.

  Natürlich kann man dann auch nicht direkt in der Klassendeklaration von `X` in `X.h` Methodenrümpfe angeben, die Methoden von `Y` aufrufen. Größere/Komplexere Methodenrümpfe sollten aber ohnehin eher in `X.cpp` stehen.

Include-Optimierung (2)

- Die Methoden der Klasse `X` werden natürlich Methoden der Klasse `Y` aufrufen, daher braucht man die Include-Anweisung dann spätestens in `X.cpp`.
- Für die Anwender der Klasse `X` (die `Y` selbst nicht brauchen), geht das Compilieren etwas schneller (weil nur `X.h`, aber nicht `Y.h` eingelesen wird).

  `Y` könnte dann ja auch seinerseits wieder andere Klassen benötigen, so dass man schnell sehr viele bzw. sehr große Include-Dateien einlesen muss, wenn man die Kette nicht, wie hier gezeigt, unterbricht.

Inhalt

make: Motivation (1)

- Angenommen, das Programm besteht aus zwei Modulen `a.cpp` und `b.cpp`, die beide `b.h` includen.
  - ◇ Wird `a.cpp` geändert, muß nur diese Datei neu compiliert werden.

    Die Objektdatei `b.o`, die das Ergebnis der Übersetzung von `b.cpp` enthält, ist von der Änderung nicht betroffen.
  - ◇ Wird `b.cpp` geändert: Nur diese compilieren.
  - ◇ Wird dagegen `b.h` geändert, müssen beide Dateien neu compiliert werden.
- Anschließend immer den Linker aufrufen.

make: Motivation (2)

- Wenn Include-Dateien selbst andere Dateien includen, kann das sehr unübersichtlich werden.
- Im Beispiel könnte man einfach immer alle Module neu übersetzen, bei größeren Programmen mit vielen Modulen würde das aber zu lange dauern.

  Selbst bei mittelgroßen Programmen muß man eine Viertelstunde oder länger warten, bis sie vollständig neu aus den Quellen übersetzt sind. Während der Programmentwicklung hält das zu lange auf.
- Vergißt man eine Datei neu zu übersetzen, kann das zu schwierig zu findenen Fehlern führen.

  Das Verhalten des Programms entspricht nicht dem Quellcode.

make: Motivation (3)

- Das Programm `make` automatisiert die Schritte zur Erstellung des ausführbaren Programms, indem es Datum und Uhrzeit der letzten Änderung von Quelldateien, Objektdateien u.s.w. vergleicht.

  Wurde `a.cpp` nach `a.o` geändert, muß der Compiler neu aufgerufen werden. Falls `a.cpp` die Datei `b.h` (direkt oder indirekt) included, muss der Compiler auch aufgerufen werden, wenn `b.h` nach `a.o` geändert wurde. Außerdem muß der Compiler natürlich aufgerufen werden, wenn es `a.o` noch gar nicht gibt (es wird aus `a.cpp` und `b.h` erzeugt).

- Es basiert auf einer Datei (`Makefile` oder `makefile`), die die Abhängigkeiten zwischen den Dateien und die Befehle zur Erzeugung von Dateien enthält.

make: Motivation (4)

- Die Verwendung von `make` kann selbst in ganz kleinen Projekten mit nur einem Modul sinnvoll sein, weil man die Übersetzungsbefehle (mit eventuell vielen Optionen) nur einmal aufschreiben muß.

  Vermutlich lohnt es sich nicht, `make` nur für solche Projekte zu erlernen, aber wenn man es erst einmal beherrscht, will man es auch bei Kleinstprojekten nicht missen.

- `make` ist nicht auf die Übersetzung von Programmen beschränkt, sondern kann immer verwendet werden, wenn Dateien aus anderen erzeugt werden.

make: Problem, Ausblick

- `make` funktioniert nur, wenn die Regeln im `Makefile` korrekt sind, also z.B. alle `include`-Beziehungen widerspiegeln.
- Bei kleinen Projekten sollte das kein Problem sein, bei etwas größeren Projekten muß man diesen Abschnitt im `Makefile` automatisch generieren.

  Der `g++` hat mehrere Optionen, um Abhängigkeiten zu erzeugen, z.B. `-M` (inklusive System-Include-Dateien) und `-MM` (ohne diese, das ist meist übersichtlicher). Mit Shellskripten kann man das `Makefile` automatisch ändern, so dass die erzeugten Abhängigkeiten dort eingefügt werden. Für noch höhere Anforderungen, z.B. die Unterstützung unterschiedlicher Plattformen und Konfigurationen, schaue man sich `CMake`, `qmake`, `SCons` und die GNU Autotools an.

Makefile (1)

- Ein `Makefile` bestehet aus einer Liste von Abhängigkeitsregeln und Makro-Definitionen.
- Eine Regel besteht aus
  - ◇ Ziel (normalerweise Datei) A,
  - ◇ Dateien $B_1, \ldots, B_n$, von denen A abhängt,
  - ◇ Kommandos $C_1, \ldots, C_k$ zur Erstellung von A.

$$
\begin{array}{ll}
A\!: & B_1\ B_2\ \ldots\ B_n \\
 & C_1 \\
 & \vdots \\
 & C_k
\end{array}
$$

Makefile (2)

- Um das Ziel A zu erstellen, erstellt `make` zunächst rekursiv die Ziele $B_1, \ldots, B_n$.
- Dann prüft `make`, ob es die Datei A gibt:
  - ◇ Falls nein: Die Kommandos $C_1, \ldots, C_k$ werden ausgeführt, um A zu erstellen.
  - ◇ Falls ja, werden Datum und Uhrzeit der letzten Änderung von $B_1, \ldots, B_n$ mit dem entsprechenden Zeitstempel von A verglichen. Wurde ein B_i nach A geändert, werden $C_1, \ldots, C_k$ ausgeführt.

    Sonst (A existiert und ist jünger als alle B_i) ist A also aktuell. Die Kommandos $C_1, \ldots, C_k$ werden dann nicht ausgeführt.

Makefile (3)

Beispiel für Makefile:

- Angenommen, das Programm `myprog` besteht aus den beiden Modulen `main.cpp` und `list.cpp`.
- Beide schließen `list.h` per `#include` ein.
- Das `Makefile` besteht aus drei Regeln:

```
myprog: main.o list.o
        g++ main.o list.o -o myprog
main.o: main.cpp list.h
        g++ -c main.cpp
list.o: list.cpp list.h
        g++ -c list.cpp
```


Makefile (4)

Syntaktische Feinheiten:

- Das Ziel muss in der ersten Spalte beginnen. Es ist keine Leerzeichen davor erlaubt.
- Die Liste der Dateien, von denen das Ziel abhängig ist, erstreckt sich nur bis zum Zeilenende.

  Wenn man sie in der nächsten Zeile fortsetzen will, muß man die aktuelle Zeile mit \ beenden. Bei Bedarf kann man die Liste der Abhängigkeiten so auch mehrfach fortsetzen.

- Die Kommandos müssen mit ein oder mehreren Tabulator-Zeichen eingerückt werden.

  Alternativ: Kennzeichnung der Kommandos durch vorangestelltes "`;`".

Makefile (5)

Default-Ziel:

- Wenn man `make` ohne Argumente aufruft, versucht es, das erste Ziel im `Makefile` zu erstellen.

  Dies ist das "Default-Goal" oder "Default-Target".

- Insofern ist wichtig, was die erste Regel in der Datei ist. Die Reihenfolge der übrigen Regeln ist egal.

  Auf Folie 18-58 wird erklärt, wie man ein Ziel eigens als Default-Ziel einführt. Der Linker-Aufruf muß dann nicht mehr oben stehen.

- Man kann beim Aufruf von `make` auch explizit ein Ziel angeben:

```
make main.o
```

Makefile (6)

Ziele und Dateien:

- Ziele müssen nicht unbedingt Dateien entsprechen.

  Beliebige Namen sind möglich. Da es das Ziel dann nicht gibt, werden die Kommandos immer ausgeführt, wenn das Ziel gemacht werden soll. Man so also weitere Kommandofolgen im `Makefile` hinterlegen. Solche Ziele werden "phony targets" genannt.

- Beispiel-Regel (meist unten in des `Makefile`):

```
clean:
        rm -f main.o list.o myprog
```

- Mit dem Befehl "`make clean`" werden die Objektdateien und das ausführbare Programm gelöscht.

  Anschließend übersetzt "`make`" das Programm dann vollständig neu aus den Quellen.

Makefile (7)

Extra Default-Ziel:

- Wenn man den Befehl zum Linken in der logischen Reihenfolge nach den Compiler-Befehlen schreiben will, kann man oben einen beliebigen Bezeichner wie "`all`" als Default-Ziel einführen und den vom Programm abhängig machen.

```
all: myprog
```

  Auch hier versucht `make` dann rekursiv die bei den Abhängigkeiten genannten Ziele herzustellen, in diesem Fall also das Programm. Da es in dieser Regel keine Kommandos gibt, wird das Default-Ziel damit nur weitergegeben. Natürlich funktioniert dies auch, wenn man mehrere Programme mit einem Makefile erstellen will: Diese Programm können dann alle als Dateien, von denen `all` abhängig ist, aufgelistet werden.

Makefile (8)

Ausgabe von Kommandos:

- Normalerweise gibt `make` alle Kommandos aus, die es ausführt. Das kann u.U. unübersichtlich wirken.

  Z.B. bei langem Befehl zum Linken (mit vielen Bibliotheken).

- Stellt man Befehlen ein "`@`" voran, werden sie nicht ausgegeben.

- Mit dem UNIX-Befehl `echo` kann man die gewünschte Ausgabe erreichen:

```
myprog: main.o list.o
        @echo -n "Linking ... "
        @g++ -o myprog main.o list.o
        @echo "Done."
```


Makefile (9)

Ausführung von Kommandos:

- Wenn eine Regel mehrere Kommandozeilen enthält, wird jede Zeile mit einem eigenen Aufruf des Kommandointerpreters (shell) ausgeführt.

  Z.B. wirkt sich ein Wechsel des aktuellen Verzeichnisses in einer Zeile nicht auf folgende Zeilen aus. Man kann aber durch "\" vor dem Zeilenende ein Kommando in der nächsten Zeile fortsetzen.

- Die Ausführung wird gestoppt, wenn ein Kommando einen Fehler meldet.

  D.h. der Wert beim Aufruf von `exit` bzw. der von `main` zurückgelieferte Wert ist verschieden von `0`. Will man ggf. trotz Fehlercode weitermachen, kann man dem Kommando ein "-" voranstellen.

Makefile (10)

Mehrere Regeln für ein Ziel:

- Es kann mehrere Regeln für das gleiche Ziel geben, aber nur eine davon darf Kommandos enthalten.
- Mit den anderen Regeln (ohne Kommandos) werden nur weitere Abhängigkeiten zu den bestehenden hinzugefügt.

  Man kann so lange Abhängigkeitslisten übersichtlicher aufschreiben. Auch günstig für automatische Erzeugung von Abhängigkeiten.
- Falls diese Abhängigkeiten zeigen, dass das Ziel nicht mehr aktuell ist, werden die Kommandos der einen Regel mit Kommandos ausgeführt.

Makefile (11)

Kommentare:

- Kommentare werden in einem `Makefile` durch ein vorangestelltes "`#`"-Zeichen gekennzeichnet.
- Sie erstrecken sich dann bis zum Ende der Zeile.
- Im Kommando-Teil werden Kommentare nicht von `make` selbst ausgewertet, sondern an die Shell (den UNIX Kommando-Interpreter) weitergegeben.

  Dort werden Kommentare auch mit "`#`" gekennzeichnet, es bleibt also ein Kommentar. Es wird aber mit ausgegeben (ggf. "`@`" voranstellen).
- Natürlich kann man Leerzeilen zur Strukturierung im `Makefile` benutzen.

Makefile (12)

Zusammenfassung (bis hierher):

- Einfaches `Makefile` für Programm aus 2 Modulen:

```
# Dies ist ein Kommentar.
all: myprog
main.o: main.cpp list.h
        g++ -c main.cpp
list.o: list.cpp list.h
        g++ -c list.cpp
myprog: main.o list.o
        g++ -o myprog main.o list.o
clean:
        rm -f myprog main.o list.o
```


Makefile: Makros (1)

- Im obigen Beispiel - Makefile werden die Objektdateien an drei Stellen aufgelistet:
  - ◇ Bei den Abhängigkeiten der Programmdatei.
  - ◇ Beim Aufruf des Linkers.
  - ◇ Bei der Regel für "`make clean`".
- Wie sonst auch in der Programmierung, möchte man die mehrfache Angabe der gleichen Information vermeiden, da
  - ◇ das unnötige Arbeit macht,
  - ◇ das Risiko der Inkonsistenz besteht.

Makefile: Makros (2)

- Make bietet nun die Möglichkeit, benannte Zeichenketten zu definieren, die an mehreren Stellen eingesetzt werden können (Makros).
- Beispiel:

```
OBJECTS = main.o list.o
```

Der Name des Maros ist "`OBJECTS`", der Ersetzungstext "`main.o list.o`" (alles bis zum Zeilenende).

Der Ersetzungstext enthält ggf. auch Leerzeichen vor dem Zeilenende.

- Wenn man anschließend `$(OBJECTS)` schreibt, wird dafür automatisch "`main.o list.o`" eingesetzt.

(...) um Makro-Namen ist nötig wenn Name mehr als ein Zeichen.

Makefile: Makros (3)

```
# Beispiel fuer Makefile mit Makros.
PROG    = myprog
OBJECTS = main.o list.o

all: $(PROG)
main.o: main.cpp list.h
        g++ -c main.cpp
list.o: list.cpp list.h
        g++ -c list.cpp
$(PROG): $(OBJECTS)
        g++ -o $(PROG) $(OBJECTS)
clean:
        rm -f $(PROG) $(OBJECTS)
```


Makefile: Suffix-Regeln (1)

- Wenn man viele Module hat, ist es unpraktisch, das Kommando zum Compilieren einzeln aufschreiben zu müssen (selbst mit Makro).
- Man kann dann eine allgemeine Regel definieren, wie man aus einer `.cpp`-Datei eine `.o`-Datei macht:

```
# Beispiel fuer Suffix-Regel (mit Makros).
CXX      = g++
CXXFLAGS = -Wall -Wextra
.SUFFIXES: .o .cpp
.cpp.o:
        $(CXX) -c $(CXXFLAGS) -o $@ $<
```


Makefile: Suffix-Regeln (2)

- Mit `.SUFFIXES` werden die Datei-Endungen festgelegt, für die es solche Regeln gibt.

Dateien weiter vorne in der Liste können aus Dateien weiter hinten in der Liste erzeugt werden. Mit dem speziellen Ziel "`.SUFFIXES`" werden Suffixe hinten an die Liste angehängt.

- Die Regel, wie man von `.cpp` nach `.o` kommt, ist

```
.cpp.o:
        $(CXX) -c $(CXXFLAGS) -o $@ $<
```

- Spezielle Makros (siehe nächste Folie):
  - ◇ `$@` ist der Name des Ziels (`.o`-Datei).
  - ◇ `$<` ist die Abhängigkeit der Regel (`.cpp`-Datei).

Makefile: Suffix-Regeln (3)

- Spezielle Makros:
  - ◇ `$@`: Dateiname des aktuellen Ziels.
  - ◇ `$*`: Dateiname des aktuellen Ziels ohne Suffix.
  - ◇ `$?`: Dateien, von denen das Ziel abhängig ist, und seit der letzten Erstellung des Ziels geändert.
  - ◇ `$^`: Alle Dateien, von denen das Ziel abhängig ist (ohne Duplikate).
  - ◇ `$+`: Alle Dateien, von denen Ziel abhängig (in Reihenfolge des Vorkommens, mit Duplikaten).
  - ◇ `$<`: Nur die erste Datei, von der das Ziel abhängig ist (in Suffix-Regeln: Voraussetzung der Regel).

Makefile: Suffix-Regeln (4)

- Für ein vollständiges `Makefile` braucht man dann nur noch
  - ◇ Die Regel für den Aufruf des Linkers (zur Erstellung des Programms aus den Objektdateien).
  - ◇ Regeln (ohne Kommandos) für die Abhängigkeiten von Includedateien, z.B.

```
main.o: list.h
```

Aufgrund der Suffix-Regel weiß make, dass `main.o` von `main.cpp` abhängig ist, und welches Kommando ggf. ausgeführt werden muss. Weitere Abhängigkeiten müssen explizit angegeben werden.

Inhalt

Objektdateien (1)

- Eine Objektdatei enthält im wesentlichen:
  - ◇ Maschinencode für die definierten Funktionen,

    Mit "Löchern" (z.B. Null-Bytes) für die Adressen aufgerufener, noch unbekannter Funktionen.
  - ◇ eine Tabelle definierter Symbole mit Wert,

    Definierte Symbole sind Funktionsnamen und globale Variablen, Wert ist die zugehörige Adresse,
  - ◇ eine Liste benutzer Symbole zusammen mit den Stellen, an denen ihr Wert anzutragen ist,

    D.h. Position der "Löcher", in die die Startadresse der Funktion einzutragen ist.
  - ◇ Informationen zur Code-Verschiebung (s.u.).

Objektdateien (2)

- Außer Programmcode für Funktionen und Methoden enthalten Objektdateien auch
  - ◇ Daten, z.B. Zeichenketten im Programm, oder Werte initialisierter globaler/statischer Variablen (insbesondere Arrays),
  - ◇ Angaben, wie viel Speicherplatz für nicht initialisierte globale/statische Variablen benötigt wird,

    Diese Variablen werden dann automatisch auf 0 initialisiert.
  - ◇ Ggf. Daten für einen Debugger.

Objektdateien (3)

- Objektdateien (und ausführbares Programm) bestehen normalerweise aus mehreren Abschnitten:
  - ◇ Das "text" Segment enthält Maschinenbefehle.
  - ◇ Das "data" Segment enthält Variablen mit expliziter Initialisierung.
  - ◇ Das "bss" Segment enthält Platz für Variablen, die implizit auf 0 initialisiert werden.

    Während das ausführbare Programm ein Hauptspeicher-Abbild aller initialisierten globalen/statischen Variablen (im "data" Segment) enthält, braucht es für das "bss" Segment ("block started by symbol") nur die Größe (Anzahl Bytes) anzugeben. Wenn das Programm in den Hauptspeicher geladen wird, reserviert das Betriebssystem einen entsprechenden Hauptspeicher-Bereich.

Objektdateien (4)

- Tatsächlich enthalten Objektdateien häufig noch viele weitere Abschnitte.

  Z.B. "`rodata`" für "read-only Daten", also z.B. String-Konstanten, die das Betriebssystem vor schreibendem Zugriff schützen sollte/könnte. Manchmal wird unterschieden zwischen "segments" (Daten, die später im ausführbaren Programm benötigt werden) und "sections" (Daten, die nur zum Linken nötig sind).
- Unter UNIX ist ELF ein verbreitetes Format für Objektdateien (und ausführbare Programmdateien).

  Es ist ganz typisch, dass es ein gemeinsames Format für Objektdateien und Programmdateien gibt, so dass Eingabe und Ausgabe des Linkers im Prinzip das gleiche Format haben. ELF steht für "Executable and Linkable Format" bzw. früher "Extensible Linking Format".

Objektdateien (5)

- Es gibt verschiedene Programme zum Anzeigen von Inhalten von Objektdateien.

  Unter UNIX zeigt das Programm `nm` die Symboltabelle einer Objektdatei an. Das GNU-Programm `objdump` kann benutzt werden, um den ganzen Inhalt einer Objektdatei anzuzeigen. Das Programm `readelf` leistet Ähnliches speziell für ELF-Dateien. Visual C++ kommt mit einem Programm `dumpbin` zur Anzeige von Objektdateien. Beispielaufruf: `dumpbin /all /disasm /out:a.x a.obj` (schreibt Ergebnis in `a.x`).
- Das Programm "`strip`" dient zur Entfernung von Symboltabellen.

  Das ausführbare Programm braucht keine Symbole mehr zu enthalten (zum Debuggen wären sie allerdings nützlich). Gerade bei kommerzieller Software möchte der Hersteller das Disassemblieren bzw. Reverse Engineering erschweren.

Arbeitsweise des Linkers (1)

- Im wesentlichen fügt der Linker einfach die Segmente gleichen Typs aus den verschiedenen Objektdateien zu jeweils einem Segment zusammen.

  Z.B. alle Text-Segmente mit Maschinencode.

- Der Compiler übersetzt jede `.cpp`-Datei so, dass die erzeugten Maschinenbefehle an einer festen Adresse starten.
- Wenn der Linker den Maschinencode aus den Objektdateien zusammenfügt, muss er die Codestücke im Speicher hintereinander ablegen.

Arbeitsweise des Linkers (2)

- Daher ist die erste Aufgabe des Linkers, Programmcode zu verschieben ("Code Relocation").
  - ◇ Weil der Maschinencode Sprungbefehle enthält, müssen die Zieladressen angepasst werden.

    Das gleiche gilt für den Zugriff auf Daten (globale/statische Variablen): Auch hier verschieben sich die Adressen während des Linkens.

  - ◇ Zu diesem Zweck enthalten die Objektdateien auch Information darüber, an welchen Stellen im Programm Adressen stehen, die angepasst werden müssen.

Arbeitsweise des Linkers (3)

- Die zweite Aufgabe des Linkers ist es, Referenzen zwischen Modulen aufzulösen:
  - ◇ Wenn ein Modul A die Funktion f verwendet (aufruft), und Modul B die Funktion f definiert,
  - ◇ muss die Startadresse von f (aus dem Text-Segment von B nach Verschiebung) an den entsprechenden Stellen im Programmcode von A nachgetragen werden.

    Zu diesem Zweck enthalten die Objektdateien entsprechende Tabellen benutzender und definierender Vorkommen von Symbolen (s.o.).

Bibliotheken (1)

- Eine weitere Aufgabe des Linkers ist es, Programmcode aus Bibliotheken hinzuzubinden.
- Bibliotheken sind Sammlungen von Objektdateien.

  Unter UNIX werden Bibliotheken mit dem Archiver-Programm `ar` aus Objektdateien erstellt. Z.B. zeigt "`ar t /usr/lib/libc.a`" die Liste der Objektdateien in der Standard-C Bibliothek. Eine Symboltabelle für schnelleren Zugriff kann man mit `ranlib` erstellen. Bei Microsoft Visual C++ gibt es das Programm "`lib`" zur Verwaltung von Bibliotheken.

- Da immer ganze Objektdateien aus Bibliotheken hinzugebunden werden, enthalten Bibliotheken sehr viele kleine Objektdateien (oft nur eine Funktion).

Bibliotheken (2)

- Der Linker stellt fest, welche Symbole in den gegebenen Objektdateien benutzt werden, die in keiner dieser Dateien definiert sind.
- Dann durchsucht er die Bibliotheken nach passenden Objektdateien und bindet diese zum Ergebnisprogramm.
- Das geschieht genau wie die explizit angegebenen Objektdateien. Der einzige Unterschied bei den Bibliotheken ist, dass der Linker hier entscheidet, welche Objektdateien nötig sind.

Bibliotheken (3)

- Wenn eine Objektdatei aus der Bibliothek selbst Symbole verwendet, die sie nicht definiert, müssen dann weitere Objektdateien (aus dieser oder anderen Bibliotheken) hinzugenommen werden.
- Der Linker durchsucht die Bibliotheken in der Reihenfolge, in der sie in der Kommandozeile angegeben werden.

  Wenn eine Bibliothek A eine Bibliothek B verwendet, muss man also A vor B angeben. Sonst können eventuell nicht definierte Symbole aus A gemeldet werden, weil diese Symbole noch nicht benötigt wurden, als der Linker B verarbeitet hat. In seltenen Fällen (zyklische Referenzen) muss man die gleiche Bibliothek mehrfach angeben.

Linker: Optionen

- Mit der Option `-L` werden Verzeichnisse angegeben, in denen der Linker nach Bibliotheken sucht.

  Standardverzeichnisse sind `/lib` und `/usr/lib`.
- Mit der Option `-l`X wird die Bibliothek `lib`X`.*` hinzugenommen.

  D.h. die Bibliothek wird an dieser Stelle nach Funktionen durchsucht, die aufgerufen wurden, aber noch nicht definiert sind. Daher werden Bibliotheken im Kommando am Ende (nach den Objektdateien) genannt. Benutzt Bibliothek A Bibliothek B, so muss `-l`A vor `-l`B angegeben werden (s.o.).
- Mit `-o` P gibt man die Ausgabedatei des Linkens an (ausführbares Programm).

Inhalt

Statisches Linken (1)

- Oben wurde das klassische Link-Verfahren, das sogenannte “statische Linken”, erläutert:
  - ◇ Bei diesem Verfahren wird der benötigte Programmcode zum Zeitpunkt des Linkens aus der Bibliothek in das ausführbare Programm kopiert.
  - ◇ Der Programmcode für häufig benutzte Bibliotheksfunktionen ist dann Bestandteil vieler Programmdateien.
  - ◇ Die Verbindung zwischen dem Programm und der Implementierung der Bibliotheksfunktion ist anschließend fest.

Statisches Linken (2)

- Das statische Linken hat einige Nachteile:
  - ◇ Verschwendeter Plattenplatz: Wesentliche Teile häufig verwendeter Bibliotheken sind in jedem ausführbaren Programm enthalten (in Kopie).

    D.h. das gleiche Stück Maschinencode (z.B. für Ein-/Ausgabe) kann sehr häufig auf der Platte stehen.
  - ◇ Verschwendeter Hauptspeicher: Werden mehrere Programme gleichzeitig ausgeführt, die die gleiche Bibliotheksfunktion verwenden, stehen mehrere Kopien dieser Funktion im Hauptspeicher.

Statisches Linken (3)

- Nachteile statischen Linkens, Forts.:
  - ◇ Weniger Flexibilität: Wenn eine neue Version der Bibliothek veröffentlicht wird, müssen Anwendungen, die mit der alten Version gelinkt wurden, mit der neuen Version neu gelinkt werden.

    Der Benutzer muss sich also ein Update der Anwendung besorgen, nicht nur ein Update der Bibliothek. Wenn z.B. eine Sicherheitslücke in einer bestimmten Version einer Bibliothek gefunden wird, ist auch nicht sofort klar, welche Programme diese Bibliothek beinhalten. Ohne Zugriff auf Quellcode oder zumindest Objektdateien der Anwendung kann man das Problem auch nicht selbst lösen (mit vertretbarem Aufwand). Man benötigt ein (ggf. kostenpflichtiges) Update durch den Anbieter.

Statisches Linken (4)

- Nachteile statischen Linkens, Forts.:
  - ◇ Aufwendigere Konfiguration: Wenn es mehrere Versionen eines Programms gibt, die sich nur in wenigen Modulen unterscheiden, muss der Anbieter dennoch das vollständige ausführbare Programm für jede Version ausliefern.
  - ◇ Keine Erweiterung durch den Benutzer: Der Benutzer kann das Programm nicht mit selbst entwickelten Funktionen erweitern.

    Kommerzielle Anbieter liefern nicht gerne Objektdateien, weil es mehr über die innere Funktionsweise des Programms verrät.

Dynamisches Linken (1)

- Bei dynamischem Linken ist der Programmcode aus der Bibliothek nicht Bestandteil des ausführbaren Programms (es ist unvollständig).
- Der Compiler übersetzt den Aufruf einer Bibliotheksfunktion dann in einen indirekten Sprung über eine "Import Tabelle".
- Das ausführbare Programm enthält die Namen der nötigen Bibliotheken und die Namen/Nummern der verwendeten Funktionen, zusammen mit dem zugehörigen Index in der Import-Tabelle.

Dynamisches Linken (2)

- Wenn das Programm ausgeführt werden soll, tut das Betriebssystem (bzw. dyn. Linker) Folgendes:
  - ◇ es sucht die verwendeten Bibliotheken,
  - ◇ läd sie in den Hauptspeicher, falls sie nicht bereits dort sind,
  - ◇ gibt dem Programm Zugriff auf den Bibliothekscode (geteilter Hauptspeicherbereich),

    Modifikationen sind natürlich verboten. Auch Problem: Gleiche Adresse in allen Programmen oder relative Adressierung.
  - ◇ füllt die Import-Tabelle mit den Adressen der aufgerufenen Funktionen.

Dynamisches Linken (3)

- Aus Sicht des Programmierers ist die Verwendung einer "Dynamic Link Library (DLL)" oder "Shared Library" nicht so viel anders als die Verwendung einer klassischen Bibliothek:
  - ◇ Beim Linken wird überprüft, dass es die verwendeten Funktionen in den Bibliotheken gibt.

    Fehler (undefinierte Funktionen) werden also gefunden.
  - ◇ Der Programmcode dieser Funktionen wird aber nicht ins ausführbare Programm kopiert.
  - ◇ Das Hinzufügen des Programmcodes geschieht erst zur Laufzeit (zweiter Teil des Linkens).

Dynamisches Linken (4)

- Man kann eine Dynamic Link Library etwas wie eine Erweiterung des Betriebssystems sehen:
  - ◇ Im Betriebssystem gibt es auch viele Funktionen, die normale Benutzerprogramme aufrufen können, wobei der Programmcode nicht Teil der Benutzerprogramme ist.
  - ◇ Ein Betriebssystem-Aufruf ist aber relativ teuer, da damit auch ein Kontextwechsel und höhere Privilegien verbunden sind.
  - ◇ Ein Ziel ist auch, den Betriebssystem-Kern klein zu halten.

Dynamisches Linken (5)

- Dynamisches Linken hat auch Nachteile:
  - ◇ Bei der Installation auf einem anderen Rechner reicht es nicht, nur das Programm auszuliefern, sondern es ist auch sicherzustellen, dass die verwendeten Bibliotheken vorhanden sind.
  - ◇ Das Betriebssystem muss die Bibliotheken auch finden, wenn das Programm ausgeführt wird.

    Bei Linux/UNIX muss der Benutzer ggf. die Umgebungsvariable `LD_LIBRARY_PATH` passend setzen.
    Windows durchsucht: (1) das Verzeichnis mit dem ausführbaren Programm, (2) das aktuelle Verzeichnis, (3) `C:\WINDOWS\SYSTEM`, (4) `C:\WINDOWS`, (5) die Verzeichnisse der Umgebungsvariable `PATH`.

Dynamisches Linken (6)

- Nachteile dynamischen Links, Forts.:
  - ◇ Es gibt das Risiko von Versionskonflikten.

    Es werden gelegentlich neuere Versionen von dynamischen Bibliotheken installiert, mit denen das Programm nicht getestet wurde. Natürlich versuchen die Programmierer der DLLs, inkompatible Änderungen zu vermeiden. Falls aber vorher undokumentierte Features ausgenutzt werden, kann es sein, dass das Programm mit der neuen Version nicht mehr funktioniert (während andere Programme unbedingt die neue Version brauchen).
  - ◇ Bei der Deinstallation von Programmen ist u.U. nicht klar, welche dynamischen Bibliotheken mit gelöscht werden können.

Dynamisches Linken (7)

- Es gibt noch eine andere Art dynamischen Linkens (“Explicit Linking” unter Windows), bei dem
  - ◇ das Programm zur Laufzeit das Betriebssystem bittet, eine Bibliothek zu laden,
  - ◇ und dann die Adressen von Prozeduren in der Bibliothek abfragt.
- Auf diese Art können die geladenen Bibliotheken von Benutzereingaben abhängen, was z.B. für Plugins wichtig ist.