1. Vom Sourcecode zum Programm

1.1 Programmablauf

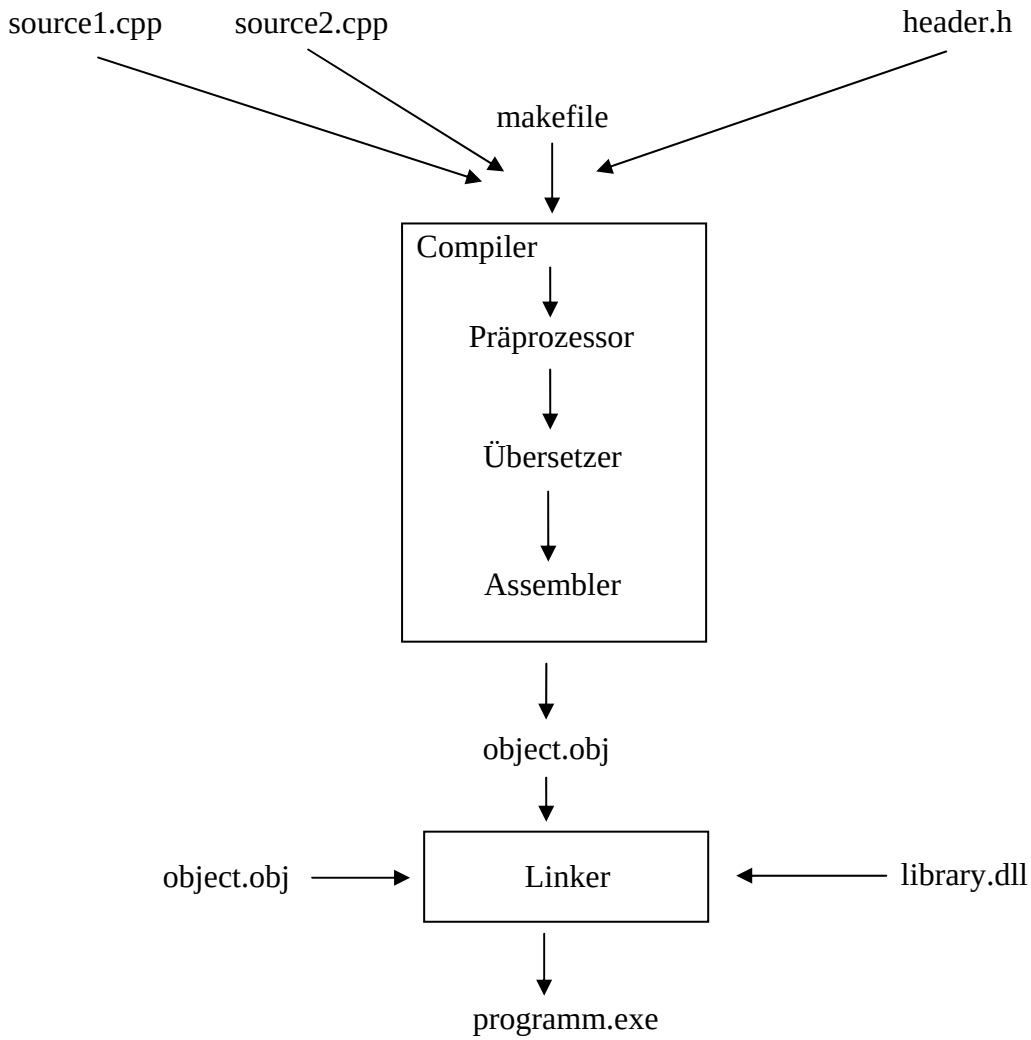


Abbildung: Erstellung eines ausführbaren Programms

Werkzeug	
Editor	… zum Schreiben von Text. … ist in einer Entwicklungsumgebung enthalten
Compiler	
	Präprozessor … stellt den vollständigen Quellcode auf Textebene zusammen. … verarbeitet Anweisungen, die mit einem # beginnen. … sucht und ersetzt Text.
	Übersetzer (Compiler) … erzeugt Assemblercode, der von Menschen gelesen werden kann.
	Assembler … erzeugt aus den Assemblercode Objektcode (Maschinencode). Der Code besteht aus 0 und 1. Maschinencode kann mit einem Hex-Editor gelesen werden.
Linker	… verbindet alle Objektdateien und Bibliotheken zu einer ausführbaren Datei.

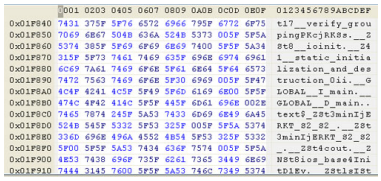
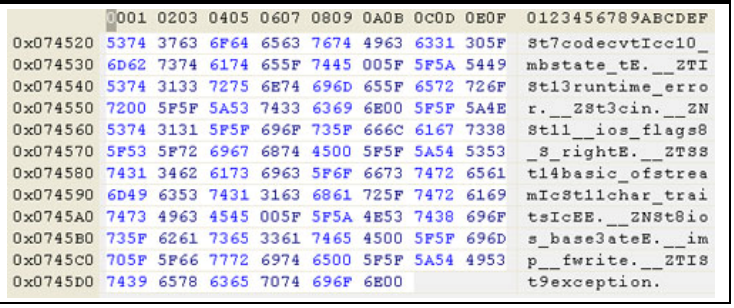
Werkzeug	*erstellt*		

Editor	Quelldatei (.cpp)	Source.
		Ein Programm, welches mit einer bestimmten Sprache (hier C++) geschrieben ist.
			`#include <iostream>`
			`using namespace std;`
			`int main (){`
			`  cout << "Hello World \n";`
			`  return 0;`
			`}`

	Header-Datei (.h)	Enthält Deklarationen von Funktionen und Variablen, die in verschiedenen Quelldateien genutzt werden.	`//@(`
			`extern istream cin;`
			`extern ostream cout;`
			`extern ostream cerr;`
			`extern ostream clog;`
			`#ifdef _GLIBCXX_USE_WCHAR_T`
			`extern wistream wcin;`
			`extern wostream wcout;`
			`extern wostream wcerr;`
			`extern wostream wclog;`
			`#endif`
			`//@)`

Compiler:			

Präprozessor		`# 63 "/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/iostream"`
		`extern istream cin;`
		`extern ostream cout;`
		`extern ostream cerr;`
		`extern ostream clog;`
		`# 77 "/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/iostream"`
		`static ios_base::Init __ioinit;`
		`}`
		`# 2 "myMain.cpp" 2`
		`using namespace std;`
		`int main (){`
		`cout << "Hello World \n";`
		`return 0;`
		`}`	

<table>
<tr><td>Werkzeug</td><td>erstellt</td><td colspan="2"></td></tr>
<tr><td>Übersetzer</td><td></td><td colspan="2">
<pre>
LCO:
	.ascii "Hello World \12\0"
	.text
	.align 2
	.p2align 4,,15
.globl _main
	.def	_main;	.scl	2;	.type	32;	.endef
_main:
	pushl %ebp
	movl	$16, %eax
	movl	%esp, %ebp
	subl	$8, %esp
	andl	$-16, %esp
	call	__alloca
	call	___main
	movl	$__ZSt4cout, (%esp)
	movl	$LCO, %eax
	movl	%eax, 4(%esp)
	call	__ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
	leave
</pre>
</td></tr>
<tr><td>Assembler</td><td>Objektdatei (.obj)</td><td>… besteht aus Maschinencode.
… ist abhängig von der genutzten Programmiersprache, Compiler und dem Rechner, auf dem das Programm übersetzt wurde.</td><td>  </td></tr>
<tr><td>Linker</td><td>Ausführbares Programm (.exe)</td><td colspan="2"> 
</td></tr>
</table>

1.2 makefile

Der Befehl make und die Datei makefile stammen aus der Unix-Welt und werden in der Software-Entwicklung eingesetzt.
Die Datei makefile beschreibt mit einer Skript-ähnlichen Sprache Abhängigkeiten zwischen den einzelnen Dateien eines Projekts. Dateien eines Projekts werden mit Hilfe eines makefiles überwacht.
Mit Hilfe des Befehls make wird die Datei makefile ausgeführt.
Ein einfaches makefile kann folgendermaßen aussehen:

```
myMain.o : myMain.cpp
	g++ -c myMain.cpp
```

Der Befehl make weiß aufgrund dieser Zeilen, dass er nach einer Datei mit dem Namen „myMain.cpp“ schauen muss. Aus dieser Datei wird mit Hilfe des Kommandos `g++ -c myMain.cpp` die Datei „myMain.o“ generiert.
Das makefile enthält eine Regel, die eine Aussage über die Erstellung einer Objektdatei aus einer Quelldatei trifft.

Eine Regel in einem makefile hat folgendes Aussehen:

```
Ziel : Abhängigkeit
     Kommando
```

Ziel ist ein Synonym für die Datei, die erstellt werden soll. In derselben Zeile stehen, durch einen Doppelpunkt getrennt, die Abhängigkeiten. Die Abhängigkeiten bezeichnen alle Dateien, die benötigt werden, um das Ziel zu generieren. Das Kommando beschreibt eine Aktion, die die Abhängigkeiten verarbeitet und das Ziel erstellt. Das Kommando wird mit Hilfe des Tabulators und nicht mit Leerzeichen eingerückt!
Die Erstellung eines Ziels kann mehr als eine Abhängigkeit haben. Die einzelnen Dateien, die benötigt werden, sind durch Leerzeichen getrennt.

```
myProgramm : myMain.o myAusgabe.o
     g++ myMain.o myAusgabe.0 –o myProgramm
```

„myProgramm“ ist von den Objektdateien „myMain“ und „myAusgabe“ abhängig. Abhängigkeiten können in einem makefile auch mehrstufig sein.

```
myProgramm : myMain.o myAusgabe.o
     g++ myMain.o myAusgabe.0 –o myProgramm

myAusgabe.o : myAusgabe.c
     g++ -c myAusgabe.c

myMain.o : myMain.c
     g++ -c myMain.c
```

Wenn der Befehl `make myProgramm` aufgerufen wird, werden zuerst die Objektdateien aus den Quelldateien generiert. Anschließend wird aus den Objektdateien „myProgramm“ erzeugt.
Wenn das Ziel ein jüngeres Datum hat als ihre Abhängigkeiten, wird make nicht gestartet. Beispiel: Das Ziel „myProgramm“ wurde letztmalig am 4.10.2005 generiert. Der Programmier verändert die Datei „myMain.cpp“ am 1.11.2005 und speichert diese ab. Nach der Änderung der Quelldatei wird make gestartet und ausgeführt, da das Ziel älter ist als die Abhängigkeit.

Mit Hilfe des Befehls make können auch Dateien gelöscht werden.

```
clean:
     rm *.obj core
```

`rm` ist ein Befehl zum Löschen von Dateien. Hier werden alle Objektdateien und Dateien, die bei einem Programmabsturz erzeugt werden, gelöscht.
Das Ziel clean wird auch als Pseudo-Ziel bezeichnet. Es besitzt keine Abhängigkeiten und wird nie erzeugt.

Um ein Projekt vollständig zu kompilieren, kann folgende Regel genutzt werden.

```
all : myMain.o
     g++ -o myMain.exe myMain.o
```

Kommentare beginnen in einem makefile mit einem Hash.

```
# Compilieren des Programms Hello World
all : myMain.o
      g++ -o myMain.exe myMain.o

myMain.o : myMain.cpp
      g++ -c myMain.cpp
```

Weitere Informationen können Sie unter
http://wsd.iitb.fhg.de/~hiwigeg1/selflinux/pdf/make.pdf
finden.

1.3 Header-Dateien

Header-Dateien sind Sammlungen von Variablen, Datenstrukturen und Deklarationen von Funktionen zu einem bestimmten Thema. Zum Beispiel die Header-Datei <iostream> enthält Deklarationen von Funktionen und globale Variablen für die Ein- und Ausgabe.

Header-Dateien in C++ haben immer die Dateiendung „.h“. Folgende Beispiele sind gültige Namen für Header-Dateien:

- kreis.h
- berechnung_flaeche.
- myProgramm.revision_1.h

Header-Dateien enthalten alle

- … Definitionen von Variablen, Konstanten, Datentypen und –strukturen,
- … Deklarationen von Funktionen und Klassen und
- … alle Programmteile, die der Präprozessor wie Text ersetzen kann.

Die Definition und Deklarationen in einer Header-Datei werden von vielen verschiedenen Quelldateien benötigt.

HeaderDateien enthalten

- keinen Code, der kompiliert werden muss.
- keine Variablen, Konstanten, Datentypen und –strukturen, die nicht global genutzt werden.

Header-Dateien haben einen bestimmten Aufbau:

Header-Datei „myHeader.h“

```
#ifndef myHeader
#define myHeader

// Kommentar: Dateiname und Aufgabe
// Erstellungsdatum
// Version
// Revision

// Compilerschalter
#define WINDOWS
#undef MSDOS

// Include-Dateien
#include <iostream>
#include "myAusgabe.h"

// Symbolische Konstanten
#define TRUE 1

// Makros
#define MIN(zahlA, zahlB) ((a) < (b) ? (a) : (b))

// Globale Konstanten
const double max = 100;

// Datenstrukturen und Typvereinbarungen
typedef  float Flaeche;

// Globale Variablen
extern int mass;

// Deklarationen von Funktion
double flaeche (float breite, float hoehe);

#endif
```

Mit Hilfe des Codes

```
#ifndef myHeader
#define myHeader
…
#endif
```

wird überprüft, ob der Präprozessor die Header-Datei schon eingebunden hat oder nicht. Wenn die Header-Datei das erste Mal aufgerufen wird, wird die Konstante `myHeader` definiert und alle nachfolgenden Zeilen ausgeführt. Wenn die Header-Datei nochmals aufgerufen wird, wird die Einbindung durch die Anweisung `ifndef` (falls nicht definiert) verhindert.

Beispiel für Klassenmodule:

rect.h

```
// Vermeidung von
// rekursiven Aufrufen
#ifndef RECHTECK_H
#define RECHTECK_H
// Ein Byte ist immer ein Char
typedef unsigned char BYTE;

// RGB-Farben als Struktur
struct Color{
      BYTE red;
      BYTE green;
      BYTE blue;
};

// Definition der Klasse Rechteck
class Rect{
 private:
    short xPos, yPos;
    short breite, hoehe;
    Color rectFarbe;

 public:
  void Init(short x, short y,
       short width, short height);
  void Move(short x, short y);
  void Resize(short width,
       short heigth);
  void SetColor(BYTE r, BYTE g,
       BYTE b);
  void DrawRect();
};
#endif
```

rect.cpp

```
using namespace std;

#include <iostream>
// Klassendefinition einbinden
#include "rect.h"

void Rect::Init(short x, short y,
         short width, short height){
      xPos = x;
      yPos = y;
      hoehe = height;
      breite = width;

      rectFarbe.red = 0x00;
      rectFarbe.green = 0x00;
      rectFarbe.blue = 0x00;
};

void Rect::Move(short x, short y){
      xPos = x;
      yPos = y;
};

void Rect::Resize(short width,
              short heigth){

};

void Rect::SetColor(BYTE r, BYTE g,
     BYTE b){

};

void Rect::DrawRect(){

};
```

Globale Variablen in Header-Dateien verletzen das Prinzip „information hiding“ und sollten vermieden werden. Falls globale Variablen nötig sein sollten, ist in der Header-Datei nur ein Verweis auf die Variable vorhanden. Die globale Variable wird aber in der dazu gehörigen Quelldatei definiert.

globalDaten.h

```
// Vermeidung von
// rekursiven Aufrufen
#ifndef GLOBALDATEN_H
#define GLOBALDATEN_H

 extern int nummer[50];
 extern int feldzeiger:
#endif
```

globalDaten.cpp

```
int nummer[50] = {0};
int feldzeiger = 0;
```