hcmake

Benutzerhandbuch

Inhalt

Inhalt **1**

Übersicht **2**

Aufruf 3

Anwendung 3

Behandlung von Kommentaren 4

Limitationen 5

Layout **6**

Format der Layout-Datei 6

Labels 6

Specs **12**

Format der Spec-Datei 12

Angaben zu einer Datei 12

Angaben zu einer Klasse 13

Attribute 13

Übersicht

hcmake ist ein Tool, das die Entwicklung von c++-Programmen unterstützt, indem bestimmte, immer wiederkehrende Aufgaben automatisiert werden:

a. Erzeugung einer h-Datei:
   Anhand einer Textdatei (Spec-File) werden Klassengerüste erzeugt, die bereits einige Standardfunktionen enthalten können. In diese Gerüste können manuell weitere Funktionen eingefügt werden.
b. Erzeugung einer c-Datei aus der h-Datei
c. Erzeugung eines Makefile-Gerüsts

Weiterhin bietet hcmake die Möglichkeit in einer Textdatei das Layout der zu erzeugenden h- und c-Dateien festzulegen. Auf diese Weise kann hcmake an die individuellen Anforderungen des jeweiligen Programmierers angepaßt werden.

Begriffserläuterung

c-Datei: Eine c-Datei ist eine C++-Implementationsdatei in der Form name.c++, name.cpp oder ähnliches. Unabhängig von der Endung wird diese Datei im Folgenden immer nur c-Datei genannt.

h-Datei: Eine h-Datei ist eine C++-Headerdatei. Sie hat den gleichen Namen wie die c-Datei, aber mit der Endung .h.

Beschränkungen

Durch die Layout-Datei ist der Benutzer in der Lage hcmake an seinen eigenen Code-Stil anzupassen. Es wurde versucht, dem Benutzer hierbei größtmögliche Freiheit zu lassen. Allerdings forciert hcmake einige Eigenschaften des Code-Layouts, die der Autor als elementar ansieht (bei der Beschreibung der jeweiligen Funktionalitäten von hcmake aufgeführt).

Systemvoraussetzungen

Es sind Versionen verfügbar für Linux und RISC OS. hcmake wurde mit dem gcc Compiler getestet. Für die Linux-Variante ist gcc 3.3 notwendig, für RISC OS genügt 2.95.

hcmake wurde unter SuSeE Linux 7.1 und RISC OS 4 getestet.

Aufruf

hcmake wird von der Kommandozeile aus aufgerufen.

Syntax: hcmake [options] spec-file
options:

-h	*create h-files*
-c	*create c-files*
-m	*create make-file*
-e<DotExt>	*set dot extension for c-file*
-l<Layout>	*specify layout-file*

(der obige Hilfstext wird beim Aufruf von hcmake ohne Argumente ausgegeben)

Falls die -e Option nicht angegeben wird, wird unter Unix die Endung cpp verwendet, während unter RISC OS die Dateien in das Verzeichnis c++ gelegt werden

Falls die -l Option nicht angegeben wird, wird unter Unix die Datei .layout verwendet. Unter RISC OS muß die -l Option angegeben werden.

Beispiele:
hcmake -h -m Spec
hcmake -c -ecxx Spec

Anwendung

Die Anwendung von hcmake untergliedert sich in die folgenden Schritte:

1. **Schritt: Layout**
   Bevor hcmake zum ersten Mal verwendet wird, muß eine Layout-Datei erstellt werden (diese Distribution enthält eine Beispieldatei, die als Ausgangsbasis verwendet werden kann).
   In der Regel sollte eine Layout-Datei pro Benutzer genügen. Es kann aber auch sinnvoll sein für jedes Projekt eine eigene Layout-Datei zu erstellen.

2. **Schritt: Schreiben der Spec-Datei**
   In der Spec-Datei müssen h- und c-Dateien, sowie Klassen aufgelistet werden (siehe hierzu das Kapitel Specs).
   Es ist möglich für ein bereits bestehendes Projekt eine weitere Spec-Datei zu schreiben, um nachträglich h- und c-Dateien mit neuen Klassen zu erzeugen (das Einfügen von neuen Klassen in bereits bestehende h- und c-Dateien ist nicht möglich). In diesem Fall müssen die Schritte 2-4 wiederholt werden. Ein automatisches Einfügen der zusätzlichen

Dateien in das bereits bestehende Makefile (Schritt 5) ist nicht möglich.

3. **Schritt: Erzeugen der h-Dateien**

   Im nächsten Schritt wird hcmake verwendet, um die h-Dateien zu erzeugen. Dies geschieht zum Beispiel mit dem folgenden Aufruf:

   hcmake -h -lLayout Spec

   Hiermit werden die h-Dateien mit den angegebenen Klassen und bestimmten in der Spec-Datei festgelegten Standard-Funktionen erzeugt.

4. **Schritt: Erzeugen der c-Dateien**

   Vor dem nächsten Aufruf von hcmake besteht die Möglichkeit manuell weitere Funktionen in die h-Dateien einzutragen. Anschließend erfolgt der nächste Aufruf zur Erzeugung der c-Datei:

   hcmake -c -lLayout Spec

   Hinweis: Die Spec-Datei muß auch bei diesem Aufruf von hcmake mit angegeben werden. Sie wird an dieser Stelle primär dazu benötigt Klassen zu erzeugen die vollständig innerhalb der c-Datei defininiert werden. Die Angabe der Spec-Datei kann jedoch auch dann **nicht** weggelassen werden, wenn keine solchen Klassen existieren.

   Falls es nicht erforderlich ist manuell zusätzliche Funktionen einzutragen, können der 3. und der 4. Schritt auch mit einem Aufruf absolviert werden, z.B:

   hcmake -h -c -lLayout Spec

5. **Schritt: Erzeugen des Makefiles**

   Zuletzt wird aus der Spec-Datei ein Makefile-Gerüst erzeugt:

   hcmake -m -lLayout Spec

   Dieser Schritt kann mit dem 3. oder dem 4. Schritt kombiniert werden:

   hcmake -h -m -lLayout Spec

   hcmake -c -m -lLayout Spec

Behandlung von Kommentaren

Beim Übergang von der h-Datei zur c-Datei wird versucht Kommentare so gut wie möglich den jeweiligen Funktionen zuzuordnen. Kommentare, bei denen dies nicht gelingt (entweder weil der Erkennungsmechanismus versagt hat bzw. in der Layout-Datei fehlerhaft konfiguriert wurde oder aber weil der Kommentar tatsächlich nicht zu einer Funktion gehört) werden ohne weitere Rückmeldung an den Benutzer entfernt. Sonstige Kommentare bleiben (soweit in den folenden Kapiteln nicht anders erwähnt) werden ebenfalls entfernt.

Darüber hinaus bietet die Layout-Datei auch die Möglichkeit an bestimmten Stellen zusätzliche Kommentare einzufügen.

Limitationen

hcmake geht in allen Schritten davon aus syntaktisch korrekten C++-Input zu erhalten. Ist dies nicht der Fall, ist das Resultat undefiniert.

Layout

Die Layout-Datei legt fest wie der zu erzeugende Quelltext zu formatieren ist und enthält darüber hinaus Angaben wie manuelle Quelltext-Eingaben zu interpretieren sind. Die hcmake-Distribution enthält eine Layout-Beispiel-Datei mit Namen layout_ im Verzeichnis Test.

Format der Layout-Datei

Das Format der Layout-Datei ist zeilenorientiert, d.h. eine Layout-Datei besteht aus einer Folge von Zeilen, von denen jede das folgende Format hat:

<Label>:<Argumente>

Argumente dürfen NICHT in Anführungszeichen gesetzt werden! Dem Argument kann zur besseren Lesbarkeit ein Leerzeichen vorangestellt werden (nach dem Doppelpunkt!). Falls die Argumente allerdings textueller Natur sind, wird dieses Leerzeichen als Teil des Arguments übernommen.

Bei textuellen Argumente können folgende Escape-Codes verwendet werden:
\\: \
n: Neue Zeile
c: Name der momentanen Klasse
d: Aktuelles Datum

Labels

Hinweis: Falls die Label con, obs, mani, free, friend, notimp, stat oder class nicht in der Layout-Datei aufgelistet werden, werden die entsprechenden Kommentare beim Erzeugen der h-Datei weggelassen. Abgesehen von dem Label free beziehen sich die oben aufgeführten Label ausschließlich auf Member-Funktionen.

Als Bespiele sind innerhalb des Verzeichnisses *Beispiele* einige Layout-Dateien aufgeführt (einschließlich Spec-Dateien und resultierender h- und c-Dateien).

Folgende Labels sind zulässig:

class

Dieses Label erzeugt einen Kommentar, der vor dem Beginn einer Klasse eingefügt wird.

Beispiel:

class:Klasse: \c

con

Dieses Label erzeugt einen Kommentar, der den Beginn des Abschnitts anzeigt, in dem Konstuktoren und Destruktoren stehen. Dieser Kommentar wird an der Stelle, an der die jeweiligen Member-Funktionen definiert werden (üblicherweie in der c-Datei, außer bei inline-Funktionen) nicht dupliziert.

Beispiel:

con:Konstruktoren

empty

Dieses Label legt fest, ob auch leere Abschnitte (siehe die Beschreibungen der Label con, obs, mani und stat) einen Überschrifts-Kommentar vorangestellt bekommen. Hierbei wird zwischen public-, protected- und private-Bereichen unterschieden. Diejenigen Bereiche, in denen auch leere Abschnitte kommentiert werden sollen, müssen durch Leerzeichen getrennt aufgeführt werden. Falls dieser Label nicht aufgeführt wird, werden keine Kommentare für leere Abschnitte erzeugt.

Beispiel:

empty:public protected

fcom

Dieses Label legt die Position der zu Funktionen gehörenden Kommentare fest (die Identifizierung der zu einer Funktion gehörenden Kommentare ist neben Textformattierungs-Aspekten vor allem notwendig, um die Kommentare von nicht aus der h-Datei übernommenen Funktionen entfernen zu können). Dies betrifft in dieser Version von hcmake ausschließlich Kommentare, die vom Benutzer eingefügt werden.

Mögliche Werte sind: b, B, a, A

Außerdem kann ein s angehängt werden, um eine strikte Überprüfung der Einrückung zu erzwingen, d.h. Kommentare, die anders eingerückt sind als die benachbarte Funktion, werden nicht als zugehörig zu dieser Funktion angesehen.

Falls dieser Label nicht angegeben wird, wird *fcom:bs* verwendet.

B-Option (Kommentar vor Funktion):

// Kommentar
void foo();

oder:

// Kommentar

void foo();

b-Option (Kommentar vor Funktion, keine Leerzeile):

// Kommentar
void foo();

A-Option (Kommentar nach Funktion):
void foo();
// Kommentar

oder:
void foo();

// Kommentar

a-Option (Kommentar nach Funktion, keine Leerzeile):
void foo();
// Kommentar

Wenn eine Leerzeile zulässig ist, wird bei automatisch generierten Funktionen immer eine Leerzeile eingefügt.
Die hier beschriebene Positionierung von Kommentaren betrifft nur die h-Datei (und Klasen die statt in der h-Datei in der c-Datei definiert werden). Bei der Umwandlung der h-Datei in die c-Datei wird der Kommentar einer Funktion immer vorangestellt.

file

Dieses Label setzt einen Kommentar an den Anfang der h-Datei. Dieser Kommentar wird bleibt beim Erzeugen der c-Datei erhalten.
Beispiel:
file:Project: hcmake\nDatum:\d\n

free

Dieses Label erzeugt einen Kommentar, der den Beginn des Abschnitts anzeigt, in dem freie Funktionen stehen. Dieser Kommentar wird an der Stelle, an der die jeweiligen Funktionen definiert werden (üblicherweie in der c-Datei, außer bei inline-Funktionen) nicht dupliziert.
Beispiel:
free:Freie Funktionen

friend

Dieses Label erzeugt einen Kommentar, der den Beginn des Abschnitts anzeigt, in dem friend-Funktionen stehen.
Beispiel:
friend:Friends

indent

Dieses Label legt die Einrücktiefen fest. Hierzu müssen vier positive Integer-Werte angegeben werden (durch Leerzeichen getrennt aufgeführt):

1. Einrückung der Schlüsselworte public, private, protected

2. con-, obs-, mani-, stat- und noimp-Markierungen
3. Member-Funktionen und zu Member-Funktionen gehörende Kommentare (innerhalb der Klassendefinition, bei der Generierung der c-Datei aus der h-Datei wird diese Einrückung nicht angewandt)
4. Implementation im Einrückmodes 2

Falls dieser Label nicht angegeben ist, wird *ident: 4 4 8 4* verwendet

istyle

Dieses Label legt den Einrückstil fest (bei der Übertagung von Funktioen aus der h-Datei in die c-Datei). Dies wird im folgenden am Beispiel einer main-Funktion erläutert:

Stil 0:

```
int main()
{

}
```

Stil 1:

```
 int main() {

}
```

Stil 2:

```
int main()
      {

      }
```

Falls dieser Label nicht angegeben wird, wird Stil 0 verwendet.

Beispiel:

istyle: 1

mani

Dieses Label erzeugt einen Kommentar, der den Beginn des Abschnitts anzeigt, in dem nicht-konstante Funktionen stehen. Dieser Kommentar wird an der Stelle, an der die jeweilige Member-Funktion definiert wird (üblicherweise in der c-Datei, außer bei inline-Funktionen) nicht dupliziert.

Beispiel:

mani:Manipulatoren

notimp

Dieser Kommentar zeigt einen Abschnitt an, der nicht implementierte Funktionen enthält.

Beispiel:

notimp:Nicht implementiert

Anmerkungen zu nicht implementierten Funktionen: Der C++-Compiler generiert innerhalb einer Klasse bestimmte Funktionen automatisch, falls diese nicht expliziet angegeben werden (z.B. der Copy-.Konstruktor). Diese automatisch erzeugten Funktionen sind nicht immer für die Klasse geeignet. Um diese automatisch generierten Funktionen zu unterdrücken, ohne sie selbst implementieren zu müssen, ist ein gängiges Verfahren diese Funktionen private zu deklarieren und nicht zu implementieren.

obs

Dieses Label erzeugt einen Kommentar, der den Beginn des Abschnitts anzeigt, in dem konstante Funktionen stehen. Dieser Kommentar wird an der Stelle, an der die jeweilige Member-Funktion definiert wird (üblicherweise in der c-Datei, außer bei inline-Funktionen) nicht dupliziert.

Beispiel:

obs:Observatoren

order

Dieses Label gibt die Reihenfolge der verschiedenen Abschnitte an (con, obs, mani, stat, friend, public, protected, private). Dies geschieht in Form einer Zeichenkette, wobei jedes Zeichen für einen Abschnitt steht.

n nicht implementierte Funktionen (private)

c Konstruktoren/Destruktoren (private)

d Konstruktoren/Destruktoren (protected)

o Konstante Funktionen (private)

p Konstante Funktionen (protected)

l Nicht konstante Funktionen (private)

m Nicht konstante Funktionen (protected)

s Statische Funktionen (private)

t Statische Funktionen (protected)

C Konstruktoren/Destruktoren (public)

O Konstante Funktionen (public)

M Nicht konstante Funktionen (public)

S Statische Funktionen (public)

F Friend

f Freie Funktion (außerhalb der Klasse)

Nicht aufgelistete Abschnitte werden am Ende angehängt. Ausgenommen hiervon sind die Abschnitte d, p, l und t. Diese werden, falls nicht vorhanden, hinter c, o, m und s eingeordnet.

Beispiel:

order:ncomCOM

stat

Dieses Label erzeugt einen Kommentar, der den Beginn des Abschnitts anzeigt, in dem statische Funktionen stehen. Dieser Kommentar wird an der Stelle, an der die jeweilige Member-Funktion definiert wird (üblicherweise in der c-Datei, außer bei inline-Funktionen) nicht dupliziert.

Beispiel:

stat:Statische

tab

Dies gibt an wie vielen Leerzeichen ein Tabulator entspricht. Dies wird beim Erzeugen der Einrückungen benötigt. Falls dieses Label nicht angegeben wird, wird der Wert 8 verwendet.

Beispiel:

tab: 4

Specs

Die Spec-Datei legt fest, welche Datein und welche Klassen-Gerüste erzeugt werden sollen.

Eine einzige Spec-Datei kann Angaben zu mehreren verschiedenen Dateien erhalten.

Die hcmake-Distribution enthält eine Spec-Beispiel-Datei mit Namen Spec im Verzeichnis Test.

Format der Spec-Datei

Die Spec-Datei hat ein zeilenorientiertes Format, d.h. sie besteht aus einer Folge von Spezifikationszeilen. Kommentare werden mit einem # eingeleitet und enden am Ende der Zeile. Zwei physikalische Zeilen können durch Anfügen eines / am Ende der ersten Zeile zu einer logischen Zeile zusammengefügt werden.

Beispiel:
zeile 1 /
zeile 2

Dies wäre dasselbe wie:
zeile 1 zeile 2

Eine Spezifikationszeile kann entweder Angaben zu einer Datei oder zu einer Klasse machen (zwischen diesen beiden Formen einer Spezifikationszeile ist strikt zu unterscheiden).

Angaben zu einer Datei

Eine Angabe zu einer Datei hat die Form:
<dateiname>.<endung>:<argumente>

Zulässige Endungen sind:

h: kennzeichnet die h-Datei

c, cc, c++, cpp: kennzeichnet eine c-Datei

Die beim Aufruf von hcmake angegebene Endung hat hierbei keine Bedeutung.

Im folgenden werden die zulässigen Argumente aufgeführt:

Name einer Datei: Die Datei wird durch eine include-Anweisung in die Übersetzungseinheit eingefügt. Diese Information wird außerdem für die Erzeugung des Makefiles genutzt.

Achtung: Die Datei darf keine Punkt-Endung haben!

Name einer Klasse: Gerüst für angegebene Klasse in dieser Datei erzeugen (das Gerüst kann entweder in der h-Datei oder in der c-Datei erzeugt werden)

Jede Klasse darf nur einmal auf diese Weise aufgeführt werden. Jedes Argument, für das die Spec-Datei keine Klassen-Zeile enthält, wird automatisch als Dateiname angesehen.

main: Dies kennzeichnet die c-Datei als diejenige Datei, die die main-Funktion erhält (erzeugt main-Gerüst und ist außerdem relevant für Makefile-Erzeugung).

Mehrere Datei-Angaben mit main-Argument innerhalb einer Spec-Datei sind zulässig!

Angaben zu einer Klasse

Dies hat die Form:

<klassenname>:<attributliste>

Eine Attributliste ist eine durch Leerzeichen getrennte Auflistung von Attributen. Die Attribute bestimmen, welche Funktionen automatisch erzeugt werden.

Die zulässigen Attribute werden im folgenden Abschnitt aufgeführt.

inline-Funktionen

Einige Attribute erzeugen Funktionen die gleich inline definiert werden. Diese Implementierung der jeweiligen Funktion wird immer strikt von ihrer Deklaration getrennt. Dies bedeutet insbesondere, daß manche Funktionen zweimal aufgeführt werden (einmal ohne inline Keyword und ohne Funktionsrumpf und einmal mit inline Keyword und mit Funktionsrumpf). Dies ist kein Bug, sondern eine bewußte Design-Entscheidung des Autors, um Interface und Implementation möglichst klar zu trennen.

Attribute

Zulässige Attribute sind:

assign

Zuweisungsoperator erzeugen

compare

Operatoren == und != erzeugen

operator!= wird inline implementiert als:

inline int operator!= (const T& Left, const &T Right) { return !(Left==Right); }

copy

Copy-Konstruktor erzeugen

fullcompare

wie compare; zusätzlich werden die Operatoren >, >=, < und <= erzeugt

op+, op-, op*, op/

Freier + Operator, sowie += Member-Operator erzeugen. Freier Operator wird inline implementiert als:

inline T operator+ (const T& Left, const &T Right) { return T (Left) += Right; }

(analog für -, * und /)

noassign

Implizite Erzeugung eines Zuweisungsoperator durch private Deklaration verbieten (siehe hierzu auch die Erläuterungen zum Label notimp im vorangegangenen Kapitel)

nocopy

Implizite Erzeugung eines Copy-Konstruktur durch private Deklaration verbieten (siehe hierzu auch die Erläuterungen zum Label notimp im vorangegangenen Kapitel)

stream

Operatoren << und >> für ostream bzw. istream

Außerdem wird <*iostream*> “included”.

Die Operatoren werden nicht automatisch als friend deklariert. Dies muß bei Bedarf manuell geschehen.