hcmake

Technische Dokumentation

Inhalt

1. Definitionen & Konventionen

Definitionen

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. Die h-Datei wird von der c-Datei immer als allererstes mittels einer include-Anweisung eingebunden.

Komponente: Eine Komponente besteht aus einer c-Datei einer gleichnamigen h-Datei und eventuell einem Test-Driver (einer weiteren c-Datei).

Abhängigkeiten (»Dependency«): Eine Komponente A hängt von einer anderen Komponente B ab, wenn eine der Dateien von Komponente A die Zeile
#include "B.h"
enthält.

Include-Graph: Der Include-Graph stellt die Abhängigkeiten zwischen den Komponenten dar. Hierbei repräsentiert ein Knoten eine Komponente und eine Kante eine include-Anweisung.

Bezeichnerwahl

Datei-Namen

Der Name einer Quelltext-Datei hat die folgende Form: *xx_Yyy* oder *xx_Yyy_*

Falls ein Unterstrich angehängt ist, handelt es sich bei der Datei um einen Test-Driver, der die Funktionsfähigkeit dieser Komponente überprüft.

C++ Bezeichner

1. Ein Klassenname hat die Form: *xx_yyy*
   Beispiel: hm_args
2. Der Name einer Funktion oder einer lokalen Variable hat die Form: *Xxx*.
   Beispiel: ExplainOptions
3. Der Name eines Datenmembers hat die Form: *d_Xxx* oder *s_Xxx*. Hierbei kennzeichnet das Präfix s_ ein statisches Datenmember.
   Beispiel: *d_SpecPath*

Sonstiges

Weitere Konventionen

- Freie Variablen und freie Funktionen werden nur innerhalb einer c-Datei definiert und sind als static markiert (d.h. sie sind über diese Übersetzungseinheit hinaus nicht verfügbar).
- Wenn bei einem Funktionsaufruf ein Zeiger übergeben wird, ist ein 0-Zeiger grundsätzlich nicht zulässig, es sei denn es wird im Kommentar explizit angegeben, daß dies erlaubt ist. Ein Default-Wert (= 0) gilt ebenfalls als eine solche explizite Angabe.

Dependencies

Die Abhängigkeiten zwischen den Komponenten bilden einen gerichteten kreisfreien Graphen. Die Vermeidung von Kreisen erhöht die Modularität und führt zu einer verbesserten Testbarkeit der einzelnen Komponenten. Für eine detailliertere Diskussion dieses Themas siehe [1].

2. Übersicht

Hinweis: Es wird davon ausgegangen, daß der Leser dieses Dokuments mit dem Benutzerhandbuch vertraut ist. Die Einzelheiten der Funktionsweise von hcmake werden hier nicht noch einmal wiedergegeben.

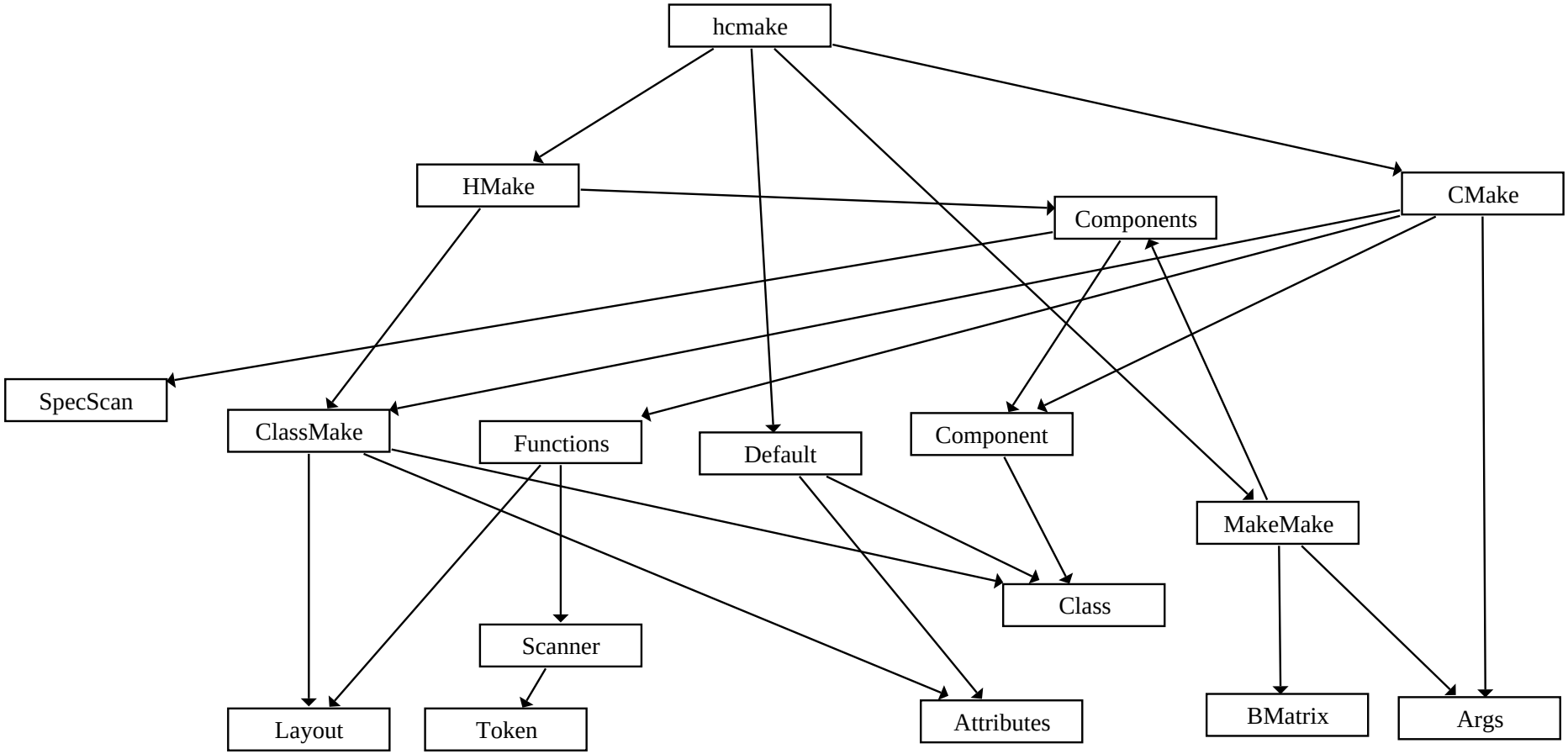
Komponenten

hcmake besteht aus 17 Komponenten:

- hm_Args
- hm_Attribute
- le_BMatrix
- hm_Class
- hm_ClassMake
- hm_CMake
- hm_Component
- hm_Components
- hm_Default
- hm_Functions
- hm_hcmake
- hm_HMake
- hm_Layout
- hm_MakeMake
- ul_Scanner
- hm_SpecScan
- ul_Token

Die main-Funktion wird innerhalb der Komponente hcmake definiert.

Das folgende Diagramm zeigt die Abhängigkeiten zwischen den Komponenten:


hcmake
HMake
CMake
Components
SpecScan
ClassMake
Functions
Default
Component
MakeMake
Class
Scanner
Layout
Token
Attributes
BMatrix
Args

Main

An dieser Stelle folgen die wesentlichen Ausschnitte aus der main-Funktion:

Initialisierung und Erstellung von Komponentendaten

```
// Attributliste initialisieren
list<hm_attribute_protocol *> Attributes;

Attributes.push_back (new hm_default_attributes);
// Zusätzliche Attributklassen sind an dieser Stelle einzufügen

// Spec-File
list<string> LegalAttributes;
for (list<hm_attribute_protocol *>::iterator Iter (Attributes.begin());
      Iter!=Attributes.end(); ++Iter)
      (*Iter)->WriteAttributes (LegalAttributes);

ifstream Spec (Args.GetSpecPath().c_str());
hm_components Components (Spec, LegalAttributes);
if (Components.HadErrors()) return 1;
```

Dies wird in Kapitel 3 beschrieben. Die relevanten Komponenten sind:

- hm_SpecScan
- hm_Components
- hm_Component
- hm_Class
- hm_Attribute
- hm_Default

Erzeugen der h-Datei

```
// Make-Operationen
 if (Args.HasOption (hm_args::HMake))
{
      hm_hmake HMake (Layout);
      if (!HMake.Make (Components.GetComponents(), Attributes))
            return 1;
}
```

Dies wird in Kapitel 4 beschrieben. Die relevanten Komponenten sind:

- hm_HMake
- hm_ClassMake

Erzeugung der c-Datei

```
if (Args.HasOption (hm_args::CMake))
{
    hm_cmake CMake (Layout, Args);
    if (!CMake.Make (Components.GetComponents(), Attributes))
        return 1;
}
```

Dies wird in Kapitel 5 beschrieben. Die relevanten Komponenten sind:

- hm_CMake
- hm_Function
- ul_Token
- ul_Scanner

Erzeugung des Makefiles

```
if (Args.HasOption (hm_args::MakeMake))
{
    hm_makemake MakeMake (Args);
    if (!MakeMake.Make (Components))
        return 1;
}
```

Dies wird in Kapitel 6 beschrieben. Die relevanten Komponenten sind:

- hm_MakeMake
- le_BMatrix

Hilfskomponenten

hm_Layout

Diese Komponente definiert eine Klasse (*hm_layout*), die für das Einlesen der Layout-Datei verantwortlich ist. Das Einlesen erfolgt bereits bei der Konstruktion eines *hm_layout*-Objektes.

Die folgenden Funktionen ermöglichen das Auslesen der einzelnen Labels (siehe hierzu die Beschreibung des Layout-Formats im Benutzerhandbuch):

GetEmptyState, *GetTabMode*, *GetWidth*, *GetFunctionCommentMode*, *HasStrictFunctionComments*, *GetIndent*, *GetIndentWidth*, *GetOrder*, *GetHeading*, *GetIndentStyle*

Die Funktion *GetOrder* gibt einen bereits komplettierten String zurück (d.h. nicht

aufgeführte Abschnitte wurden ergänzt).

Die Funktion *WriteHeading* schreibt eine bereits korrekt formatierte Abschnittsüberschrift in einen Stream.

hm_Host

(keine c-Datei, keine Klassen)

Diese Datei definiert einige Präprozessor-Konstanten, die für die Anpassung an verschiedene Plattformen notwendig sind.

hm_Args

Diese Komponente definiert eine Klasse (*hm_args*), die für das Einlesen der Kommando-Zeilen-Argumente verantwortlich ist.

Hinweis: Die Optionen *PointerAssert* und *PureVirtuals* sind in dieser Version nicht implementiert.

3. Erstellung von Komponentendaten

Aus einer Spec-Datei wird mit Hilfe der hm_SpecScan-Komponente ein hm_components-Objekt generiert. *hm_components*-Objekte enthalten hm_component-Objekte, die wiederum *hm_class*-Objekte enthalten.

Jeder Klasse werden sogenannte Attribute zugeordnet, die beschreiben, welche zusätzlichen Bestandteile (Funktionen, include-Anweisungen usw.) bei der Erzeugung einer h-Datei einzufügen sind. Hierzu wird eine Basisklasse (*hm_attribute_protocol*) definiert, von der zur Erzeugung konkreter Attribute Unterklassen abgeleitet werden müssen (z.Z. nur die Klasse *hm_default_attributes*). Objekte dieser abgeleiteten Klassen werden in eine Liste aufgenommen, die für das Einlesen der Spec-Datei und die Generierung der h-Datei (in der nächsten Phase) verwendet wird.

Components

Diese Komponente ist für das Einlesen und Interpretieren der Spec-Dateien verantwortlich. Hierzu wird die Komponente SpecScan verwendet.

hm_components

Bei der Konstruktion eines *hm_components*-Objekts wird aus einem Stream eine Auflistung von Komponenten gelesen. Das Einlesen geschieht in drei Schritten, von denen jeder in einer eigenen privaten Hilfsfunktion implementiert ist:

1. *Scan*

   Unter Verwendung eines *hm_spec_scanner*-Objekts wird die Spec-Datei (logische) Zeile für Zeile eingelesen.

   Falls der Anteil links des Doppelpunkts eine Datei-Angabe ist (*hm_spec_line::CFile* oder *hm_spec_line::HFile*), wird nach einer bereits bestehenden Komponente mit demselben Dateinamen gesucht und eine solche Komponente angelegt, falls sie noch nicht existiert. Außerdem werden die Argumente (Anteil rechts des Doppelpunkts) in die Komponente eingetragen.

   Falls der Anteil links des Doppelpunkts keine Datei-Angabe ist (*hm_spec_line::NoFile*), muß es sich um eine Klassenangabe handeln. Klassen werden zunächst in eine separate Liste eingetragen, die als lokale Variable innerhalb des Konstruktors deklariert wurde.

2. *IdentifyClasses*

   In diesem Schritt erfolgt die Zuordnung der Klassen zu den Komponenten. Hierzu wird die Komponentenliste durchlaufen. Bei jeder Komponente wird jedes Element der Argumentliste dahingehend überprüft, ob es eine Klasse mit gleichem Namen gibt und ggf. die Klasse in die Klassenliste der Komponente übernommen (dies geschieht in der Funktion *hm_component::MakeClasses*). Wird der Name nicht in der Klassenliste gefunden, muß es sich zwangsläufig um den Namen einer Datei handeln, die mit include eingebunden werden soll (der Name wird in diesem Fall in der Argumentliste des Komponente belassen).

3. *CheckClasses*

   In diesem Schritt wird auf Klassen geprüft, die keiner Komponente zugeordnet werden konnten. Ggf. wird eine Warnung ausgegeben. Das Error-Flag von hm_components wird jedoch nicht gesetzt.

SpecScan

Diese Komponente liest eine Spec-Datei (logische) Zeile für Zeile ein und generiert daraus *hm_spec_line*-Objekte.

hm_spec_line

Dieses Objekt repräsentiert eine logische Zeile. Eine logische Zeile setzt sich zusammen aus einem Anteil links des Doppelpunktes (abzufragen mit *GetFile und GetType*) und einer Auflistung von Argumenten rechts des Doppelpunktes (abzufragen mit *GetArgs*).

hm_spec_scanner

Ein *hm_spec_scanner* wird bei seiner Konstruktion mit einem Stream und einer Auflistung der bekannten Argumentnamen konfiguriert.

Das einlesen logischer Zeile erfolgt mittels der Funktion *Get*. Hierbei wird ggf. ein Fehlerflag gesetzt, das mit *HasErrors* abgefragt werden kann.

Component

hm_component

Eine Komponente besteht aus einem Namen, einer je Auflistung von Klassen für die h- und die c-Datei und je einer Auflistung von Include-Namen für die h- und die c-Datei. Diese können mittels der Funktionen *GetName*, *GetClasses* und *GetIncludes* erfragt werden.

Nach der Konstruktion werden Spec-Datei-Argumente mittels *AddArg* eingetragen. Diese Argumente werden zunächst als Namen von mittels include einzubindender Dateien angesehen.

Anschließend werden von der Funktion *MakeClasses* Klassen in die Klassenlisten aussortiert. Diese Funktion erhält als Argumente eine Liste der noch unbenutzten Klassen und eine Liste der bereits benutzten Klassen. Klassen werden ggf. von der einen Liste in die andere Übertragen (die Liste benutzter Klassen dient zum Identifizieren von mehrfachen Vorkommen einer Klasse in mehreren Komponenten).

Hinweis: Es gibt zwei *MakeClasses*-Funktionen. Die zweite Funktion ist eine private HIlfsfunktion, die von der public-Funktion aufgerufen wird.

Als letztes müssen mittels *CleanIncludes* mehrfach vorkommende Namen aus den include-Listen entfernt werden.

Class

hm_class

Ein *hm_class*-Objekt besteht aus einem Klassennamen und einer Auflistung von Attributen.

Attribute

hm_output_protocol

Diese Klasse fungiert als Interface für *hm_attribute_protocol*-Ausgaben, wobei nach Funktionen und nach zu Funktionen gehörenden Kommentaren sortiert wird. Eine konkrete Ausgabe-Klasse (von hm_output_protocol abgeleitet) wird in der Komponente ClassMake definiert.

hm_attribute_protocol

Dies ist eine Basisklasse für die Definition von Attributen (Default-Implementationen sind leer), wobei eine Subklasse durchaus mehrere Attribute definieren kann. Diese Klasse verfügt über die folgenden Funktionalitäten:

1. Auflistung der bekannten Attributnamen *(WriteAttributes)*.

(alle nun folgenden Funktionen erhalten ein *hm_class*-Objekt als Argument; die Funktion hat zu prüfen, ob diese Klasse über ein bekanntes Attribut verfügt und muß entsprechende Ausgaben durchführen, falls dies der Fall ist)

2. Auflistung von Dateien, die mittels include eingebunden werden müssen *(WriteIncludes)*.
3. Generieren von Funktionen *(WriteFriends*, *WriteNotImp*, *WriteFree*, *WriteInline*, *WritePubCon*, *WriteProCon*, *WritePriCon*, *WritePubObs*, *WriteProObs*, *WritePriObs*, *WritePubMani*, *WriteProMani*, *WritePriMani*, *WritePubStat*, *WriteProStat*, *WritePriStat)*.

Default

hm_default_attributes

Dies ist eine von *hm_attribute_protocol* abgeleitete Klasse, die die Standard-Attribute definiert: *copy*, *nocopy*, *assign*, *noassign*, *compare*, *fullcomapre*, *stream* und *opX* (hierbei steht X für die Operatoren +, -, *, /).

4. Erzeugung der h-Datei

Konzept: Die Komponenten im verwendeten *hm_components*-Objekt werden durchlaufen. Für jede vorhandene Komponente wird die h-Datei anhand eines *hm_components*-Objektes und der Attribut-Liste generiert.

hm_HMake benutzt hm_ClassMake, um Klassen-Definitionen zu erzeugen.

HMake

hm_hmake

hm_make::Make erzeugt aus einer Komponentenliste und einer Attribut-Liste die in der Spec-Datei spezifizierten h-Dateien. Hierzu wird die Komponentenliste durchlaufen und für jede Komponente die Funktion *MakeComponent* aufgerufen.

ClassMake

hm_class_make

Bereits bei der Konstruktion eines *hm_class_make*-Objektes werden die zu schreibenden Funktionen aufgelistet. Dies erfolgt über die private Hilfsfunktion *AddOutput*. Intern verwendet hm_class_make zwei Listen (eine für Member und eine für nicht-Member). Jede dieser Listen enthält wiederum weitere Listen (durch die Hilfsklasse hm_output repräsentiert, siehe c-Datei), von denen jede die Funktionen für einen Abschnitt (siehe Definition des Layout-Formats) enthält. Diese Funktionen werden durch die Hilfsklasse *hm_function2* repräsentiert (siehe c-Datei).

hm_class_make::Write durchläuft beide Listen (die Member- und die Nicht-Member-Liste). Wird ein Abschnitt gefunden, der nicht leer ist, werden der relevante Text in einen Stream geschrieben (ggf. mit einer passenden Überschrift). Dabei wird die Reihenfolge verwendet, die in der Layout-Datei festgelegt wurde. Eventuell wird auch für einen leeren Abschnitt eine passende Überschrift generiert (abhängig von den Einstellungen in der Layout-Datei).

5. Erzeugung der c-Datei

Konzept: Die Komponenten im verwendeten *hm_components*-Objekt wird durchlaufen. Für jede vorhandene Komponente wird eine h-Datei eingelesen. Aus dieser wird dann eine c-Datei erzeugt. Gegebenenfalls werden noch zusätzliche Klassen in die c-Datei eingefügt. Dies geschieht indem der Quelltext sowohl in die c-Datei als auch in einen Buffer geschrieben wird und der Inhalt des Buffers danach wie eine h-Datei ausgelesen und verarbeitet wird.

CMake

hm_cmake

hm_cmake::Make erzeugt unter Verwendung der Komponentenauflistung und der Liste der Attribute aus den h-Dateien die c-Dateien. Hierzu wird die Komponentenliste durchlaufen und für jede Komponente die Funktion *MakeComponent* aufgerufen.

MakeComponent durchläuft die h-Datei zweimal. Beim ersten Mal werden inline definierte Funktionen aufgelistet. Beim zweiten Mal wird dann die c-Datei erzeugt. Der zweite Durchlauf besteht aus folgenden Schritten:

1. Erzeugung der include-Anweisungen
2. Einfügen der ausschließlich in der c-Datei definierten Klassen

   Hierfür werden erneut die Komponentendaten herangezogen. Dieser Text wird sowohl in die c-Datei als auch in einen Buffer geschrieben.
3. Einlesen des im 2. Schritt geschriebenen Textes (aus dem Buffer) und Aufruf der privaten Hilfsfunktion *H2C*
4. Aufruf von *H2C* für die eigentliche h-Datei
5. Ggf. Erzeugung der main-Funktion

Die Funktion H2C liest eine h-Datei ein generiert einen entsprechenden Text für die c-Datei. Hierzu wird ein C++-Scanner verwendet (siehe Scanner- und Token-Komponente).

Für das Einlesen der Funktionen (einschließlich der zugehörigen Kommentare) wird die Klasse *hm_function* (Function-Komponente) verwendet.

H2C ruft sich selbst rekursiv auf, wenn eine Klasse (class oder struct) gefunden wird.

Function

hm_function

Diese Klasse wird zum Einlesen und zur (gemäß Layout-Datei formatierten) Ausgabe von Funktionen verwendet. hm_function verwendet den C++-Scanner (siehe Scanner- und Token-Komponente), um die Funktionen einzulesen.

hm_function hat zwei verschiedene Konstruktoren:

hm_function (const hm_layout&) erzeugt eine "leeres Funktions-Objekt", d.h. eine nicht

vorhandene Funktion.

Der zweite Konstruktor *hm_function (const hm_layout&, ul_scanner&, const map<string, string>, const string&, const list<ul_token>&)* versucht eine Funktion einzulesen (dies kann trotzdem scheitern, falls der Scanner an dieser Stelle nicht die Token einer Funktion liefert).

Ein hm_function-Objekt kann mittels der Funktion *IsEmpty* auf eine leere Funktion geprüft werden (erzeugt entweder durch den Konstruktor für leere Funktions-Objekte oder durch den zweiten Konstruktor, der keine Funktion vorgefunden hat).

Funktionen können mittels *IsEqual* auf Gleichheit überprüft werden *(operator==* und *operator!=* sind ebenfalls definiert), wobei zwei identische Funktionen mit verschiedenen Einrückungen oder Zeilenumbrüchen oder Kommentaren als identisch erkannt werden.

Funktionen können mittels *Write* in einen Stream geschrieben werden *(operator<<* ist ebenfalls definiert), wobei die in dem bei der Konstruktion übergebenen Layout-Objekt festgelegte Formatierung beachtet wird.

Darüber hinaus existieren verschiedene Funktionen, die einige Eigenschaften der Funktion erfragen *(IsStatic, IsConst, IsINMark, IsInline)*.

Token

ul_token

Diese Klasse repräsentiert ein C++-Quelltext-Token.

Scanner

ul_scanner

Diese Klasse ist ein Scanner für C++-Quelltext. Eine Folge von chars aus einem Stream wird in eine Folge von *ul_token*-Objekte umgewandelt. Dies geschieht mit der Funktion *GetNext*. Tokens können mit der Funktion *Putback* zurückgelegt werden, so daß sie beim nächsten Aufruf von *GetNext* wieder erscheinen (die Putback-Kapazität ist unbegrenzt).

Hinweis: Die Operatoren << und >> sind für ul_scanner in solcher Weise überladen, daß sie die Funktionen *GetNext* und *Putback* aufrufen.

Weiterhin verfügt diese Klasse über die Fähigkeit Tokens zu filtern, d.h. bestimmte Tokens nicht durchzulesen. In einem solchen Fall wird das betreffende Token ignoriert und *GetNext* gibt stattdessen das nächste Token zurück. Die Filter können mit den Funktionen *Enable*, *Disable*, *EnablePreprocessor* und *DisablePreprocessor* an und abgeschaltet werden.

6. Erzeugung des Makefiles

Konzept: Die Komponenten im verwendenten *hm_components*-Objekt werden durchlaufen, um die Komponente ausfindig zu machen, die die main-Funktion enthält. Anhand dieser Daten wird dann ein Makefile-Skelett generiert.

MakeMake

Diese Komponente erzeugt Makefiles.

hm_makemake

hm_makemake::Make erzeugt aus einer übergebenen Komponentenauflistung ein Makefile. Dies geschieht in den folgenden Schritten:

1. Makefile-Makros schreiben

   Dies geschieht in der privaten Hilfsfunktion *WriteMacros*. Es werden Makros für die verwendeten Tools und die Toolflags geschrieben. Der erzeugte Text ist unabhängig von der Komponentenauflistung.
2. Include-Matrix aufstellen

   Hier wird anhaben des Include-Graphen eine Matrix erstellt (dies geschieht mit Hilfe der Komponente BMatrix).

   Der Include-Graph wird benötigt, um die Auflistungen von Komponenten/Objekt-Dateien für ein Target festzulegen. Außerdem wird der Include-Graph verwendet, um mehrfaches Aufführen derselben Komponente zu vermeiden. Dies geschieht, indem alle redundanten Kanten des Graphen entfernt werden (siehe Beschreibung von *le_binary_matrix*).
3. Finales Target schreiben

   Falls mehr als ein Target erzeugt werden soll, wird ein zusätzliches Target namens all angelegt, das sämtliche Targets auflistet.
4. Sonstige Targets schreiben

   Die Komponentenliste wird nach Komponenten durchsucht die eine *main*-Funktion definieren. Für jede dieser Komponenten wird mittels des Inlcude-Graphen ein Target erzeugt.
5. Implizite Regeln zur Compilierung schreiben

   Dies geschieht in de privaten Hilfsfunktion *WriteSuffixes*. Der erzeugte Text ist unabhängig von der Komponentenauflistung.

BMatrix

le_binary_matrix

Dies ist eine quadratische Matrix mit binären Einträgen, die zur Darstellung eines gerichteten Graphen verwendet wird (Spalten und Zeilen repräsentieren die Knoten, ein Eintrag in der Matrix repräsentiert eine Kante zwischen den beiden Knoten). Besonders zu erwähnen sind zwei Funktionen:

1. *MakeTransitive*

   Diese Funktion erzeugt den transitiven Abschluß des Graphen.

2. *MakeNonTransitive*

   Diese Funktion entfernt alle redundanten Einträge (eine Kante von beispielsweise Knoten A nach Knoten C heißt dann redundant, wenn es Kanten von Knoten A nach Knoten B und von Knoten B nach Knoten C gibt).

Für eine Erläuterung der verwendeten Algorithmen siehe [2] und [3].

Bibliographie

[1] John Lakos, "Large-Scale C++ Software Design", Addison-Wesley
[2] Aho, Hopcroft, & Ullman, “Data Structures And Algorithms,” Addison-Wesley
[3] Warshall, S. [1962]. “A theorem on Boolean matrices,” Journal of the ACM, 9:1