Fachhochschule
Münster University of
Applied Sciences

# Embedded Software

# Der C Präprozessor

Sichere objektbasierte Entwicklung dank C Präprozessor

Prof. Dr. Nikolaus Wulff

# Shapes kanonische Form

- Die Aufteilung in Methoden und Attribute erfolgt in zwei Structs XXX_Instance und XXX_Class:

```
typedef struct Shape_Instance_struct* Shape;
typedef struct Shape_Class_struct {
    void (*draw)   (Shape aShape);
    void (*erase)  (Shape aShape);
    void (*moveTo) (Shape aShape, int x ,int y);

} ShapeClass;

typedef struct Shape_Instance_struct {
    ShapeClass *class;
    int x;
    int y;
} ShapInstance;
```

# Rect kanonische Form

- Die Rect Strukturen sind fast identische Kopien der Shape Strukturen:

```
typedef struct Rect_Instance_struct* Rect;
typedef struct Rect_Class_struct {
    void (*draw)   (Shape aShape);
    void (*erase)  (Shape aShape);
    void (*moveTo) (Shape aShape, int x ,int y);
    void (*resize) (Shape aShape, int w ,int h);
} RectClass;

typedef struct Rect_Instance_struct {
    RectClass *class;
    int x;
    int y;
    int w;
    int h;
} RectInstance;
```

# Präprozessoroperatoren

- Der Präprozessor kann helfen solche generischen Strukturen, die fast identische Kopien sind und sich nur durch Namen entscheiden, zu generieren.
- Es lassen sich Macros definieren, die als Vorlagen ähnlich den C++ Templates dienen und den Quellcode durch Textersetzung erzeugen.
- Eine wichtige Rolle spielen hierbei die beiden Präprozessoroperatoren # und ##:
  - # setzt den Parameter in Hochkomma als String
  - ## konkateniert zwei Symbole zu Einem

# Zeichen Macros

```
#define makeStr(x) #x
#define concat(x,y) x ## y
```

- Der Ausdruck `makeStr(hello)` wird dann zum String `"hello"` expandiert.
- Der Ausdruck `concat(Shape, Class)` ergibt das Symbol `ShapeClass` (nicht als String!).
- Mit diesen beiden Operatoren lassen sich Quelltexte generieren.

# C Klassen per Makros

- Die kanonische Aufteilung in Instanz und Class Strukturen lässt sich per „Coding by Convention“ durch Makros erzwingen und generieren.
- `BEG_DEFINE_CLASS(T)` generiert die Class Struktur zum Typen T.
- `BEG_DEFINE_INSTANCE(T)` generiert die Instanz Struktur zum Typen T und beinhaltet auch den Zeiger auf die Class Struktur.
- Ähnlich wie bei C++ Templates oder Java Generics ist T ein Platzhalter für den Typen, z.B. Shape.

# Shapes per Macro generieren

```
#define BEG_DEFINE_CLASS(T) \
typedef struct T##_Instance_struct* T; \
typedef struct T##_Class_struct {

#define END_DEFINE_CLASS(T) \
} T##Class;

#define BEG_DEFINE_INSTANCE(T) \
typedef struct T##_Instance_struct { \
    T##Class *clazz;

#define END_DEFINE_INSTANCE(T) \
} T##Instance; \
extern T new##T(int* args);
```

# Shape Attribute + Methoden

```
#define SHAPE_METHODES \
    void (*draw)   (Shape aShape); \
    void (*erase)  (Shape aShape); \
    void (*moveTo) (Shape aShape, int x ,int y);


#define SHAPE_ATTRIBUTES \
    int x; \
    int y;
```

# Cleveres malloc

- Mit dem Präprozessor lässt sich auch der malloc Aufruf vereinfachen und typsicher machen:

```
#define alloc(T) (T) malloc( \
                      sizeof(T ## Instance))
```

- Mit diesem Macro wird immer Code zum passenden ADT erzeugt und die Verwendung von malloc vereinfacht sich von:

```
Rect obj = (Rect) malloc(
             sizeof(struct Rect_Instance_struct));
```

- zu:

```
Rect obj = alloc(Rect);
```

# Shape.h mit Macros

```
BEG_DEFINE_CLASS(Shape)
     SHAPE_METHODES
END_DEFINE_CLASS(Shape)

BEG_DEFINE_INSTANCE(Shape)
     SHAPE_ATTRIBUTES
END_DEFINE_INSTANCE(Shape)
```

- So einfach und aufgeräumt kann eine *.h Datei sein
- Allerdings sind die Strukturen schwieriger zu verstehen, da die Macros vieles verstecken.
- Rect.h wird mit genau den selben Macros erzeugt.

# Rect.h mit Macros

```
#ifndef __RECT_H_
#define __RECT_H_
#include "Shape.h"

BEG_DEFINE_CLASS(Rect)
    SHAPE_METHODES
    void (*resize) (Rect r, int w, int h);
END_DEFINE_CLASS(Rect)

BEG_DEFINE_INSTANCE(Rect)
    SHAPE_ATTRIBUTES
    int w;
    int h;
END_DEFINE_INSTANCE(Rect)
#define size(R,x,y) R->clazz->resize(R, (x), (y))
#endif /* __RECT_H_ */
```

# Rect kanonische Form

- Diese einfachen Macro Templates generieren zur Compile Zeit die komplette Rect Header Datei:

```
typedef struct Rect_Instance_struct* Rect;
typedef struct Rect_Class_struct {
    void (*draw)   (Shape aShape);
    void (*erase)  (Shape aShape);
    void (*moveTo) (Shape aShape, int x ,int y);
    void (*resize) (Shape aShape, int w ,int h);
} RectClass;

typedef struct Rect_Instance_struct {
    RectClass *class;
    int x;
    int y;
    int w;
    int h;
} RectInstance;
```

# Rect.c

```
static void resize(Rect obj, int h, int w) {
    obj->h = h;
    obj->w = w;
}
static RectClass rectClass =
          {draw, erase, moveTo, resize };

Shape newRect(int args[]) {
     Rect obj = alloc(Rect);
     obj->clazz = &rectClass;
     obj->x = args[0];
     obj->y = args[1];
     obj->h = args[2];
     obj->w = args[3];
     return (Shape) obj;
}
```

# Vererbung automatisieren

- Eine Vererbungshierarchie lässt sich durch zwei weitere Makros `EXTENDS` und `INSTANCE_OF` deklarativ implementieren:

```
BEG_DEFINE_CLASS(Circle) EXTENDS(Shape)
    METHODS(Circle)
END_DEFINE_CLASS(Circle)


BEG_DEFINE_INSTANCE(Circle) INSTANCE_OF(Shape)
    ATTRIBUTES(Circle)
END_DEFINE_INSTANCE(Circle)
```

- Wie sind `EXTENDS` und `INSTANCE_OF` definiert?

# Kürzer geht es nicht...

- Wem selbst dies noch zu viel ist, der kann noch das Makro `DEFINE_BASE_TYPE` für eine Wurzelklasse wie z.B. Shape

```
DEFINE_BASE_TYPE(Shape)
```

- und `DEFINE_EXTENDED_TYPE` für eine Kindklasse wie z.B. Rect definieren:

```
DEFINE_EXTENDED_TYPE(Rect,Shape)
```

- Debugen wird schwieriger mit gcc –save-temps kann der generierte Code gespeichert werden...

# Debug Nachrichten

- Der Präprozessor eignet sich hervorragend um Debugnachrichten auf der Console auszugeben.
- Den Nachrichten lässt sich eine unterschiedliche Priorität zuordnen und sie werden im Releasemode ganz abschalten.
- Debugnachrichten lassen sich in Log-Dateien schreiben und später auswerten.
- Dies hilft, wenn die Anwendung bereits beim Kunden ausgeliefert ist...

# Vordefinierte Namen in C

- Es gibt in C neben den geschützten Worten der Sprache einige vordefinierte Namen die zur Übersetzungszeit vom Compiler expandiert werden:

  `__LINE__` : Die Nummer der aktuellen Quellzeile

  `__FILE__` : Der Name der aktuellen Datei

  `__DATE__` : Das Übersetzungsdatum

  `__TIME__` : Die Übersetzungszeit

  `__STDC__` : Konstante für ANSI C

- Diese vordefinierten Namen lassen sich gut für Debug Macros verwenden...

# Debug Macro

```
#define DEBUG 1

#ifdef DEBUG
#define debug(x) \
  printf("DEBUG %s %d %s\n",__FILE__, __LINE__,x);
#else
#define debug(x)
#endif


static void erase(Shape shape) {
   debug("erase shape");
   free(shape);
}
```

# Fehlerbehandlungen

- Mit dem Präprozessor lassen sich Strategien für die Fehlerbehandlung erstellen.
- Für Entwicklungszwecke dient das *assert* Macro zum Lokalisieren von Fehlern:

```
assert(<conditon>);
```

- Wenn die Bedingung 0 == false ist (C kennt keinen boolean) wird eine Exception geworfen und eine Fehlermeldung mit der Zeilennummer in der Quelldatei in stderr geschrieben.

# Auszug aus dem Assert Macro

```
#define assert(expr) \
  (__ASSERT_VOID_CAST ((expr) ? 0 :  \
  (__assert_fail (__STRING(expr), __FILE__, __LINE__,\
     _ASSERT_FUNCTION), 0)))
```

- Dies Macro liest sich so:
  - Wenn `expr!=0` mach nichts: `(void) 0`
  - Falls `expr==0` rufe die Funktion __assert_fail mit vier Argumenten:
    - Verwandle `expr` in einen String (das # Macro hilf da...)
    - Füge Dateinamen und Zeilennummer hinzu
    - Verwende ein Macro, das den aktuellen Funktionennamen expandiert...

# Design-By-Contract

- Ende der achtziger Jahre wurde das Programmier Paradigma „Design by Contract“ entwickelt.
- Es bassiert auf den drei Begriffen:
  - Vorbedingung (PreCondition)
  - Invariante (Invariant)
  - Nachbedingung( PostCondition)
- Um einen Dienst auszuführen müssen alle Vorbedingungen von Seiten des Aufrufes erfüllt werden.
- Wenn dem so ist, sichert der Dienst die Nachbedingungen zu.

# Pre- & PostCondition

```
#include <assert.h>

#define preCondition(cond, msg) \
if (!(cond)) { \
   printf("PreCondition: %s \n", msg); \
   assert(cond); \
}

#define postCondition(cond, msg) \
if (!(cond)) { \
   printf("PostCondition: %s \n", msg); \
   assert(cond); \
}
```

- Anstatt des assert Macros kann natürlich eine intelligentere Fehlerausgabe erfolgen.

# Pre/PostCondition Varianten

- Anstatt assert im Makro zu rufen können die relevanten Informationen direkt gewonnen werden:

```
#define postCondition(cond, msg) \
if (!(cond)) { \
   printf("PostCondition: %s %s %d %s \n", \
             #cond,__FILE__,__LINE__, msg); \
   abort(); \
}
```

- Interessant ist hier die Anwendung des # Operators.
- Dies gestattet die Bedingung direkt auf der Console als Zeichenkette auszugeben.

# Anwendung einer PreCondition

```
/**
 * @param x int has to be >= 0
 */
int isqrt(int x) {
    preCondition(x >=0 ," x <0 !");
    return (int) sqrt(x);
}
```

- Das Codefragment zeigt die Anwendung der preCondition.
- Das Argument muss größer 0 sein ansonsten wird eine Fehlermeldung auf der Console ausgegeben.

# Try-Catch in C

- C kennt kein echtes Exception Handling wie C++ oder Java.
- Dennoch lassen sich mit dem Präprozessor Macros definieren die eine Programmierung mit try-catch Blöcken gestatten.
- Im Internet sind zahlreiche C Macro Bibliotheken zu diesem Thema zu finden.
- Die Funktionalität und der Umfang sind allerdings recht unterschiedlich.

# Try Catch Semantik

- Mit Hilfe von try-catch-finally Blöcken und Exceptions lassen sich Programmflüsse mit Fehlerbedingungen leicht programmieren.
- Eine Anweisung besteht aus einem try-Block, in dem möglicherweise Fehler passieren können.
- Im anschließenden catch-Block erfolgt eine entsprechende Fehlerbehandlung.
- Der optionale finally-Block wird ausgeführt unabhängig davon ob ein Fehler passierte oder nicht.
- Wird try-catch verwendet macht es Sinn auch die pre- und postConditions mit Exceptions zu versehen.

# Try-Catch Verwendung

```
try {
   if (x == 0)
      throw(NULLPOINTER);
   printf("x nicht null\n");
} catch(NULLPOINTER) {
   printf("x ist null \n");
}
printf("nach dem Try Block\n");
```

- Obiges Beispiel zeigt die Anwendung eines einfachen try-catch Blockes.
- Innerhalb des try Blocks können Fehler geworfen werden, die dann im catch Block gefangen werden.

# try-catch Beispiel in Java

```
private void executeSQL(String sql) {
        Connection con = null;
        Statement stm = null;
        try {
            con = db.getConnection();
            stm = con.createStatement();
            stm.executeUpdate(sql);
        } catch (SQLException e) {
            handleError(e);
        } finally {
            releaseResources(con, stm, null);
        }
    }
```

- Zu sehen ist ein JDBC Zugriff auf eine Datenbank.
- JDBC definiert SQLExceptions die bei allen Db Operationen passieren können...

# Simple try catch Nachahmung

```
#define try
#define throw(x) goto x
#define catch(x) while(0) x:
```

- Diese Definitionen sind so ziemlich die einfachste Methode um try-catch Semantik mit C zu emulieren.
- Diese try-catch Variante funktioniert nur innerhalb einer Methode und nicht übergreifend.
- Es ist nicht möglich „RuntimeExceptions“ zu fangen wie es in C++ mit catch(...) möglich ist.
- Diese Pseudo Implementierung ist sicherlich nicht der Weisheit letzter Schluss ...

# Weitere try-catch Macros

- Im Internet sind zahlreiche Macro Bibliotheken zu finden, die in ihrem Leistungsvermögen an die echten try catch Blöcke der C++ Semantik herankommen:

  http://home.rochester.rr.com/bigbyofrocny/GEF/

  http://cexcept.sourceforge.net/

- Es lohnt sich einen Blick in die cexcept Beispiele zu werfen. Es müssen lediglich die dortigen Macros verwendet werden, um try-catch Semantik Methoden übergreifend verwenden zu können.

# PreCondition mit Exceptions

- Mit Hilfe von try-catch und Exceptions kann auch das Design by Contract Konzept viel sauberer implementiert werden:

```
#define preCondition(cond, msg) \
if (!(cond)) { \
   printf("PostCondition: %s %s %d %s \n", \
            #cond,__FILE__,__LINE__, msg); \
   throw Exception(#cond, msg); \
}
```