

Embedded C++

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

3. November 2010

Primitive Datentypen

Komplexe Datentypen

Funktionen

Zeiger und Parameterübergabe

Klassen

Typumwandlung

Namensräume

Tipps und Tricks

Templates

Fokus

Worum geht es hier nicht?

- ▶ Standardbibliothek
- ▶ Speicherverwaltung
- ▶ Ein- und Ausgabe

Warum geht es nicht darum?

- ▶ **Weil wir es hier nicht brauchen!**

C++-Programme: Ausführung auf der **nackten Hardware**

- ▶ kein Laufzeitsystem
- ▶ keine Typinformation
- ▶ Ausnahme: RTTI und Exceptions (**teuer!**)

Primitive Datentypen in C++

- ▶ primitive Datentypen in Java \mapsto exakt spezifiziert
 - ▶ Breite (in Bits) von Ganzzahltypen exakt und absolut angegeben
- ▶ primitive Datentypen in C++ \mapsto **implementierungsabhängig**
 - ▶ Breite von Ganzzahltypen ist relativ angegeben
 - ▶ z.B. $\text{Bits}(\text{int}) \geq \text{Bits}(\text{short})$

Primitive Datentypen - Übersicht

Diese Angaben für C++-Datentypen gelten z.B. auf heutigen x86-Systemen mit dem GNU C Compiler (Version 3.3.5):

Java				C++			
Bezeichner	Größe	min	max	Bezeichner	Größe	min	max
boolean	-	-	-	bool	-	-	-
char	16 Bit	Unicode 0	Unicode $2^{16} - 1$	char	8 Bit	-128	+127
-	-	-	-	signed char	8 Bit	-128	+127
-	-	-	-	unsigned char	8 Bit	0	255
byte	8 Bit	-128	127	-	-	-	-
short	16 Bit	-2^{15}	$2^{15} - 1$	signed short	16 Bit	-2^{15}	$2^{15} - 1$
-	-	-	-	unsigned short	16 Bit	0	$2^{16} - 1$
int	32 Bit	-2^{31}	$2^{31} - 1$	signed int	32 Bit	-2^{31}	$2^{31} - 1$
-	-	-	-	unsigned int	32 Bit	0	$2^{32} - 1$
long	64 Bit	-2^{63}	$2^{63} - 1$	signed long	32 Bit	-2^{31}	$2^{31} - 1$
-	-	-	-	unsigned long	32 Bit	0	$2^{32} - 1$
float	32 Bit	IEEE754	IEEE754	float	32 Bit	-	-
double	64 Bit	IEEE754	IEEE754	double	64 Bit	-	-
-	-	-	-	long double	96 Bit	-	-
void	-	-	-	void	-	-	-

Primitive Datentypen - Übersicht

Diese Angaben für C++-Datentypen gelten z.B. auf einem H8/300 mit dem GNU C Compiler (Version 3.4):

Java				C++			
Bezeichner	Größe	min	max	Bezeichner	Größe	min	max
boolean	-	-	-	bool	-	-	-
char	16 Bit	Unicode 0	Unicode $2^{16} - 1$	char	8 Bit	-128	+127
-	-	-	-	signed char	8 Bit	-128	+127
-	-	-	-	unsigned char	8 Bit	0	255
byte	8 Bit	-128	127	-	-	-	-
short	16 Bit	-2^{15}	$2^{15} - 1$	signed short	16 Bit	-2^{15}	$2^{15} - 1$
-	-	-	-	unsigned short	16 Bit	0	$2^{16} - 1$
int	32 Bit	-2^{31}	$2^{31} - 1$	signed int	16 Bit	-2^{15}	$2^{15} - 1$
-	-	-	-	unsigned int	16 Bit	0	$2^{16} - 1$
long	64 Bit	-2^{63}	$2^{63} - 1$	signed long	32 Bit	-2^{31}	$2^{31} - 1$
-	-	-	-	unsigned long	32 Bit	0	$2^{32} - 1$
float	32 Bit	IEEE754	IEEE754	float	32 Bit	-	-
double	64 Bit	IEEE754	IEEE754	double	32 Bit	-	-
-	-	-	-	long double	32 Bit	-	-
void	-	-	-	void	-	-	-

Wie löst man dieses Problem?

- ▶ Abbildung auf wohldefinierte Typen

Abbildung

```
typedef unsigned char          ezstubs_uint8 ;  
// ..  
typedef long long             ezstubs_int64 ;  
typedef ezstubs_uint32        ezstubs_addrword ;
```

- ▶ ausschließliche Verwendung der wohldefinierten Typen

Verwendung

```
ezstubs_uint32 MyClass::myMethod() {  
    // ...  
}
```

Konstanten in C++:

Präprozessor

```
#define PI 3.14159
```

Problematisch, weil:

- ▶ rein textuelle Ersetzung
- ▶ keine Integration ins C++-Typsystem

Konstanten in C++:

Schlüsselwort `const`

```
const unsigned int konstante = 5;
```

- ▶ **muss initialisiert werden**
- ▶ ins C++-Typsystem integriert
- ▶ **Zusicherung des Programmierers an den Übersetzer!**
- ▶ Unterschied zu C: benötigt nicht unbedingt Speicherplatz

Arrays

Arrays

```
int i[10];  
myClass myClassObjects[CONST_EXPRESSION];
```

- ▶ sind keine Objekte
- ▶ einfache Aneinanderreihung von Objekten gleichen Typs

Aufzählung

Schlüsselwort `enum`

```
enum Wochentage {  
    Montag ,  
    Dienstag ,  
    ...  
    Sonntag  
};
```

- ▶ Zusammenfassung: Menge von Werten \mapsto eigenständiger Typ

Verbund

Schlüsselwort `union`

```
union TCSR {  
    unsigned char val;  
    struct {  
        unsigned char OCLRA : 1;  
        unsigned char OVF   : 1;  
        ...  
        unsigned char ICFA  : 1;  
    } bits;  
};
```

- ▶ `val` und `bits` sollten gleich groß sein
- ▶ derselbe Speicherbereich aber unterschiedliche Interpretationen

Strukturen und Klassen

Schlüsselwort `struct`

```
struct Foo {  
    ...  
};
```

Schlüsselwort `class`

```
class Bar {  
    ...  
};
```

- ▶ Strukturen und Klassen sind in C++ fast äquivalent
- ▶ Unterschied: Sichtbarkeit
 - ▶ Strukturen: standardmäßig **public**
 - ▶ Klassen: standardmäßig **private**
- ▶ zu Klassen später mehr ...

Benutzerdefinierte Typen

Schlüsselwort `typedef`

```
typedef type_A type_B;
```

- ▶ `type_B` kann synonym zu `type_A` verwendet werden
- ▶ **Vorsicht:** *Forward Declarations* und `typedef`

Forward Declaration \neq `typedef`

```
typedef my_Class_A my_Class_B;  
...  
class my_Class_B;
```

obiges Beispiel wird eine Fehlermeldung beim Übersetzen liefern!

Funktionsdefinitionen in C++

Funktionsdefinition im Klassenrumpf

```
class Foo {  
    int Bar(int i) { return i + 5; }  
};
```

Funktionsdefinition außerhalb des Klassenrumpfs

```
class Foo {  
    int Bar(int i);  
};  
int Foo::Bar(int i) { return i + 5; }
```

- ▶ Funktionsdefinition innerhalb oder außerhalb des Klassenrumpfs
- ▶ auch globale Funktionen sind erlaubt

Einbettung von Funktionen

Original

```
int inc(int a) {  
    return a + 1;  
}  
int main() {  
    int a = 2;  
    a = inc(a);  
    a = inc(a);  
    a = inc(a);  
    return 0;  
}
```

Eingebettet

```
int main() {  
    int a = 2;  
    { int a_temp = a;  
      a = a_temp + 1; }  
    { int a_temp = a;  
      a = a_temp + 1; }  
    { int a_temp = a;  
      a = a_temp + 1; }  
    return 0;  
}
```

- ▶ erhält Semantik der Parameterübergabe: hier **call by value**

Einbettung von Funktionen

- ▶ Einbettung \neq textueller Ersetzung
 - ▶ letzteres macht der Präprozessor
- ▶ **Vorteil:** Man spart den Overhead des Funktionsaufrufs.
- ▶ **Gefahr:** Code Bloat durch Code-Duplikation
- ▶ **Achtung:** Die optimale Inlining-Strategie zu finden ist schwer!

Einbettung von Funktionen in Java

Schlüsselwort `final`

```
class Bar {  
    final int Foo() { ... }  
}
```

- ▶ Es gibt kein explizites Inlining in Java!
- ▶ Schlüsselwort `final` \leadsto Methode wird nicht überschrieben!
- 👉 Solche Methoden kann der Compiler einbetten.

Einbettung von Funktionen in C++

- ▶ durch Definition im Klassenrumpf
- ▶ durch entsprechende Deklaration

Schlüsselwort `inline`

```
class Foo {  
    inline int Bar(int i);  
};  
int Foo::Bar(int i) { return i + 5; }
```

- ▶ Definition muss beim Aufruf für den Compiler sichtbar sein!
- ▶ Letztendliche Einbettung hängt vom Compiler und den Compiler-Optionen ab!

Operatoren entsprechen *normalen* Funktionen

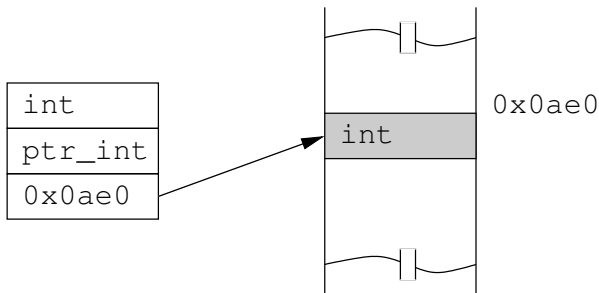
Beispiele

```
class Foo {  
    // Zuweisungsoperator  
    Foo operator=(Foo val);  
    // Typkonvertierung  
    operator Bar ();  
};
```

- ▶ Syntactic Sugar
- ▶ keine *echten* benutzerdefinierten Operatoren
- ▶ Gewisse Operatoren können nicht überladen werden.
- ▶ Die Präzedenz kann nicht beeinflusst werden.

Was sind Zeiger

- ▶ ein Zeiger ist ein Tripel (Typ, Bezeichner, Wert)
- ▶ Wert eines Zeigers: eine Adresse
- ▶ Semantik: an der Adresse ist ein Objekt vom Typ `Typ`
- ▶ Java: alle komplexen Datentypen sind Zeiger
- ▶ C++: wählbar



Verwendung von Zeigern

- ▶ Deklaration:

```
int* i;  
MyClass* foo;
```

- ▶ Zugriff auf den referenzierten Wert:

Operator `*`

```
*i = 5;
```

- ▶ Wo kommen Zeiger her?

Operator `&`

```
int i;  
int* ptr_to_i = &i;
```

Operator `new`

```
Foo* f = new Foo();
```

Referenzen

- ▶ \approx konstanter Zeiger, der automatisch dereferenziert wird
- ▶ **muss initialisiert werden**
- ▶ Verwendung:

```
int i = 6;  
int& c = i;  
Foo bar;  
Foo& foobar = bar;  
  
c = 5; // i == 5!  
foobar.bar_method();
```

▶ Java

- ▶ primitive Datentypen: *call by value*
- ▶ komplexe Datentypen: *call by reference*

▶ C++

- ▶ *call by value*: ein Objekt übergeben
- ▶ *call by reference*: einen Zeiger oder eine Referenz übergeben

```
void by_value(int i , Foo f );  
void by_reference(int* i , Foo& f );
```

```
int i = 5;  
Foo f ;
```

```
int main() {  
    by_value(i , f );           // call by value  
    by_reference(&i , f );     // call by reference  
    return 0;  
}
```


Klassenvariablen - statische Felder

▶ Deklaration

Schlüsselwort `static`

```
class Foo {  
    static int bar;  
    static void foobar();  
};
```

▶ In C++ muss aber **explizit** Speicher belegt werden:

```
int Foo::bar = 0;
```

▶ `sonst` \leadsto `undefined reference ...` beim Binden

▶ **Vorsicht:** `static` für globale Variablen/Funktionen

▶ Das gibt es auch, es hat aber eine andere Bedeutung:

 Sichtbarkeit auf die umgebende Übersetzungseinheit beschränkt

Einfache Vererbung

C++

```
class Foo : public Bar {  
    ...  
};
```

- ▶ Sichtbarkeit von Feldern und Methoden der Basisklasse muss spezifiziert werden.

Sichtbarkeit und Vererbung

Basisklasse	Vererbung	Klasse
public	public	public
	protected	protected
	private	private
protected	public	protected
	protected	protected
	private	private
private	public	private
	protected	private
	private	private

- ▶ **Vorsicht:** protected/private-Vererbung ändert die Schnittstelle
- ▶ Interface- vs. Implementation-Inheritance
- ▶ Standard: `private`

Wozu ist das gut?

- ▶ Definition von Benutzerschnittstellen
- ▶ Beispiel: Unterbrechungssynchronisation

Schnittstelle

```
// not synchronized
class MyClass {
public :
    myMethod ();
};
// synchronized
class MyGuardedClass
    : protected MyClass
{
public :
    myMethod ();
};
```

Verwendung

```
MyGuardedClass myObject ;

// OK
myObject.myMethod ();

// Compiler Fehler
myObject.
    MyClass::myMethod ();
```

Mehrfache Vererbung

Mehrfachvererbung

```
class Foo
  : public Bar1, protected Bar2, ... private Barn
{
  ...
};
```

- ▶ eine Klasse kann mehr als eine Basisklasse haben
- ▶ **Vorsicht:** Konflikte können entstehen
 - ▶ Klasse `Bar1` definiert Methode `void Bar1::FooBar()`
 - ▶ Klasse `Bar2` definiert Methode `void Bar2::FooBar()`in Klasse `Foo`: welche Methode ist gemeint?
- ▶ Konflikte müssen vom Programmierer aufgelöst werden!

Polymorphismus

Auswahl der Methode zur Laufzeit

```
int main() {  
    Oberklasse* o;  
    Unterklasse1* u1 = new Unterklasse1 ();  
    Unterklasse2* u2 = new Unterklasse2 ();  
  
    // Unterklasse1::foo()  
    o = u1;  
    o->foo ();  
    // Unterklasse2::foo()  
    o = u2;  
    o->foo ();  
  
    return 0;  
}
```

Virtuelle Methoden

- ▶ Polymorphismus nur bei virtuellen Methoden
- ▶ in Java sind alle Methoden virtuell
- ▶ in C++ müssen Methoden als virtuell deklariert werden

Schlüsselwort `virtual`

```
class Foo {  
    virtual void bar();  
};
```

- ▶ **Vorsicht:** Virtuelle Methoden sind teuer!

Konstruktoren

- ▶ Reihenfolge:
 1. Konstruktor der Basisklasse
 2. Konstruktor der eigentlichen Klasse
- ▶ **Achtung:** Reihenfolge der Basisklassenkonstruktoren bei Mehrfachvererbung ist nicht immer definiert!
- ▶ Unterschied: **kein** `super`
- ▶ Basisklassenkonstruktor in der Initializer-list des Konstruktors

```
class Foo : public Bar {  
    Foo() : Bar() {  
        ...  
    }  
};
```

- ▶ nicht notwendig bei Default-Konstruktoren

Destruktoren

- ▶ räumt ein Objekt auf, wenn seine Lebenszeit endet
- ▶ Reihenfolge:
 1. Destruktor der eigentlichen Klasse
 2. Destruktor der Basisklasse
- ▶ **Achtung:** Reihenfolge der Basisklassendestruktoren bei Mehrfachvererbung ist nicht immer definiert!
- ▶ Es gibt für jede Klasse **genau einen** Destruktor!

Syntax

```
class Foo {  
    Foo() { .. }    // Konstruktor  
    ~Foo() { .. }  // Destruktor  
};
```

Explizite Erzeugung

- ▶ dynamische Erzeugung von Objekten mit dem Operator `new`

```
MyClass* mc = new MyClass ();
```

- ▶ liefert einen Zeiger auf das erzeugte Objekt
- ▶ benötigt dynamische Speicherverwaltung
- ▶ **Vorsicht:** Speicher mit `delete` wieder freigegeben!

```
delete mc;
```

- ▶ **Keine Garbage Collection!**

Implizite Erzeugung - lokale Variablen

Lokale Variablen

```
int main() {  
    myClass a;           // Default-Konstruktor  
    yourClass b(3);     // benutzerdefinierter Konstruktor  
    ...  
    return 0;  
}
```

- ▶ werden auf dem Stapel angelegt
- ▶ beim Betreten des Sichtbarkeitsbereichs: Konstruktor
- ▶ beim Verlassen des Sichtbarkeitsbereichs: Destruktor

Implizite Erzeugung - Felder

Felder einer Klasse

```
class Foo {  
    myClass a;           // Default-Konstruktor  
    yourClass b;        // benutzerdefinierter Konstruktor  
    ourClass c;         // benutzerdefinierter Konstruktor  
  
    Foo() : b(3), c("Hallo") {...}  
};
```

- ▶ **Kind-Objekte** werden implizit mit dem **Eltern-Objekt** erzeugt
- ▶ Aufruf des Default-Konstruktors durch den Compiler
- ▶ andere Konstruktoren in der Initializer-Liste

Implizite Erzeugung - global

Globale Variablen

```
myClass a;           // Default-Konstruktor  
yourClass b(3);     // benutzerdefinierter Konstruktor
```

- ▶ globaler Sichtbarkeitsbereich wird beim
 - ▶ Start der Anwendung betreten
 - ▶ Beenden der Anwendung verlassen
- ▶ Compiler erzeugt entsprechende Konstruktor/Destruktor-Aufrufe
- ▶ **Vorsicht:** Reihenfolge nicht spezifiziert!
- ▶ Compiler-spezifische Erweiterungen notwendig

Zugriff auf nicht-statische Felder

Zugriff auf nicht-statische Felder bei Objekten

Operator .

```
class Foo {  
public :  
    int bar_var;  
    int bar_method ();  
};
```

```
int main() {  
    int a,b;  
    Foo foo;  
  
    a = foo.bar_var;  
    b = foo.bar_method ();  
  
    return 0;  
}
```

Zugriff auf nicht-statische Felder

Zugriff auf nicht-statische Felder bei Zeigern auf Objekte

Operator ->

```
class Foo {  
  public :  
    int bar_var ;  
    int bar_method () ;  
};  
  
int main () {  
  int a, b ;  
  Foo* foo = new Foo () ;  
  
  a = foo->bar_var ;  
  b = foo->bar_method () ;  
  
  return 0 ;  
}
```

- ▶ `foo->a` ist Kurzform für `(*foo).a`
- ▶ Syntactic Sugar

Zugriff auf statische Felder

Operator :: (scope-Operator)

```
class Foo {  
  public :  
    static int bar_var;  
    static int bar_method();  
};  
  
int main() {  
  int a,b;  
  
  a = Foo::bar_var;  
  b = Foo::bar_method();  
  
  return 0;  
}
```


Typumwandlung in C++

```
type var = const_cast< type >(param);
```

- ▶ entfernt `const`-Qualifizierung
- ▶ besser: `mutable`-Qualifizierung für Instanzvariablen
- ▶ Ausnahme: existierende API, die kein `const` unterstützt

```
void foo(const char* str) {  
    legacy_func(const_cast< char* >(str));  
}
```

- ▶ **Achtung:** `legacy_func` darf `str` nicht verändern!

Typumwandlung in C++

```
type var = static_cast< type >(param);
```

- ▶ Konvertierung zwischen Zahlenformaten (z.B. `int` und `float`)
- ▶ Konvertierung zwischen verwandten Zeiger- und Referenztypen
 - ▶ Typen werden statisch während der Übersetzung aufgelöst
- ▶ **Achtung:** Typdefinition muss bekannt sein
 - ▶ Deklaration ist nicht ausreichend \leadsto Übersetzungsfehler
- ▶ `this`-Zeiger werden bei Bedarf angepasst
 - ▶ z.B. im Falle der Mehrfachvererbung

Typumwandlung in C++

```
type var = dynamic_cast< type >(param);
```

- ▶ Konvertierung zwischen verwandten Zeiger- und Referenztypen
 - ▶ wird dynamisch während der Laufzeit aufgelöst
 - ▶ Typinformation notwendig \leadsto RTTI
- ▶ Fehlerfall
 - ▶ Zeiger: 0 wird zurück gegeben
 - ▶ Referenzen: `bad_cast`-Ausnahme wird geworfen
- ▶ nur mit polymorphen Typen möglich \leadsto virtuelle Methoden
- ▶ Alternative: **dynamic dispatch** durch virtuelle Funktionen

Typumwandlung in C++

```
type var = reinterpret_cast< type >(param);
```

- ▶ erzwungene Reinterpretation einer bestimmten Speicherstelle
- ▶ ähnlich zu Typumwandlungen für Zeiger in C
(≠ Typumwandlung im C-Stil)
- ▶ keine Überprüfung - weder zur Übersetzungs- noch zur Laufzeit
- ▶ in der Regel: **Finger weg!**

Typumwandlung im C-Stil

```
class A { ... };  
class B : public class A { ...};
```

```
A* a = new A(); // OK  
B* b = new B(); // OK  
A* pa = b; // OK  
B* b1 = a; // Fehler  
B* b2 = pa; // Fehler  
B* b3 = (B*)pa; // OK (mit einfacher Vererbung)  
B* b4 = (B*)a; // Fehler (nicht erkennbar)
```

Typumwandlung im C-Stil (considered harmful)

- ▶ \neq Typumwandlung in C
- ▶ *probiert* verschiedene C++-Typumwandlungen:
 1. `const_cast`
 2. `static_cast`
 3. `static_cast`, dann `const_cast`
 4. `reinterpret_cast`
 5. `reinterpret_cast`, dann `const_cast`
- ▶ das Ergebnis unterscheidet sich, je nachdem ob
 - ▶ alle Header eingebunden wurden \mapsto `static_cast` oder
 - ▶ nur eine Deklaration verfügbar ist \mapsto `reinterpret_cast`
- ▶ solange es irgendwie geht: **Finger weg!**
 - ▶ **Problem:** Verwendung von C-Bibliotheken in C++-Programmen

▶ C++: Namensräume

Schlüsselwort `namespace`

```
namespace Bar {  
  
    class Foo { ... };  
}
```

▶ Vollständiger Name der Klasse "Foo": `Bar::Foo`

Präprozessormakros

- ▶ Register entspricht einer Stelle im Speicherbereich
- ▶ Register hat eine Adresse
- ▶ Interpretiere Adresse als Zeiger

```
#define TIER *((volatile unsigned char*)0xff90)
```

```
unsigned char a = TIER;
```

```
TIER = 5;
```

```
TIER |= 0x80;
```

- ▶ `volatile` schützt vor übereifrigem Compiler

Präprozessormakros

- ▶ eleganter: `union` (aber nicht wirklich standardkonform)

```
union tier {  
    volatile unsigned char val;  
    struct {  
        volatile unsigned char ICIAE : 1;  
        ...  
    } bits;  
};
```

```
#define TIER *((union tier*)0xff90)
```

```
unsigned char a = TIER.val;  
TIER.bits.ICIAE = 1;  
TIER.val |= 0x85;
```

Operatoren

- ▶ Überladen des
 - ▶ Zuweisungsoperators
 - ▶ Typumwandlungsoperators (für Referenzen)

```
template < typename TYPE, int ADDR > class MemMap {  
public :  
    TYPE operator=(TYPE val) const {  
        *((volatile TYPE*)ADDR) = val; return val;  
    }  
    operator TYPE () const {  
        return *((volatile TYPE*)ADDR);  
    }  
    operator TYPE& () const {  
        return *((volatile TYPE*)ADDR);  
    }  
};
```

Operatoren

► Verwendung

```
MemMap< unsigned char, 0xff90 > TIER;
```

```
unsigned char a = TIER;
```

```
TIER = 0x80;
```

```
TIER |= 0x40;
```

```
TIER &= ~0x80;
```

Warum Assembler?

- ▶ **Effizienz???** \leadsto in unserem Fall eher nicht
- ☞ Der Compiler unterstützt nicht alle Instruktionen der CPU.
- ☞ Manches lässt sich in einer Hochsprache nicht formulieren.

Assemblerdateien

my_add.s

```
_my_add:  
mov 8(%esp) ,%eax  
mov 12(%esp),%edx  
add %edx,%eax  
ret
```

my_source.c

```
unsigned int my_add(unsigned int ,unsigned int );  
int main() {  
    unsigned int a = 3,b = 5,c = 0;  
    c = my_add(a,b);  
    return 0;  
}
```

Inline-Assembler

- ▶ kleine Assembler-Abschnitte: Aufwand einer eigenen Datei groß

Inline-Assembler

```
int main() {  
    unsigned int a = 3, b = 5, c = 0;  
    __asm__ ( "mov %2,%eax;"  
             "mov %3,%edx;"  
             "add %edx,%eax;"  
             "mov %eax,%1"  
             : "=m" (c)  
             : "m" (a), "m" (b)  
             : "eax", "edx" );  
    return 0;  
}
```

Vorsicht: Ganzzahlarithmetik

Divisionen

```
unsigned int a = 99, b = 100;  
unsigned int c = a / b; // c == 0 !
```

Multiplikationen

```
unsigned short a = 1000, b = 100;  
unsigned short c = a * b; // c != 100000 !  
                        // c == 34464 !
```

Additionen und Subtraktionen

```
unsigned short a = 0xffff;  
a++; // a != 0x10000 !  
     // a == 0 !
```

Gleitkommaarithmetik

- ▶ Viele Prozessoren unterstützen keine Gleitkommazahlen
- ▶ Abbildung auf Ganzzahlarithmetik
- ▶ **Extrem teuer** (Nur mit FPU sinnvoll)
- ▶ Vorsicht mit mathematischen Funktionen!

Keine dynamische Speicherverwaltung!

- ▶ kein `malloc/free`
- ▶ kein `new/delete`
- ▶ keine Speicherlöcher!
- ▶ Speicher wird angefordert:
 - ▶ dynamisch auf dem Stack
 - ▶ statisch im Datensegment (globale Variablen)

Globale Variablen

► Problem:

```
A a;  
B b;  
C c;
```

In welcher Reihenfolge
werden die Konstruktoren
aufgerufen?

► Lösung: `__attribute__((init_priority(x)))`

```
A a __attribute__((init_priority(1000))); // 1.  
B b __attribute__((init_priority(2000))); // 2.  
C c __attribute__((init_priority(3000))); // 3.
```

► GCC only - im Standard nicht spezifiziert

Allgemein

- ▶ Templates ermöglichen generische Programmierung in C++

Generische Programmierung

- ▶ möglichst allgemeine Algorithmen und Datenstrukturen
 - ▶ verzichte auf die Festlegung von Typen
 - ▶ Anforderungen an Typen sind erlaubt
 - ▶ **structural vs. nominative type system**
- ▶ zwei Ausprägungen
 - ▶ Funktionstemplates
 - ▶ Klassentemplates

Funktionstemplates

- ▶ generische Beschreibung von Algorithmen
- ▶ Deklaration/Definition
- ▶ Verwendung

Definition/Deklaration

▶ Deklaration

```
template <class class_name> void myfunc();  
template <type type_name> void myfunc();
```

▶ Definition

```
template <class class_name> void myfunc() {  
    // ...  
}
```

▶ es sind auch **mehrere Parameter** erlaubt

```
template <class A, class B> void myfunc();
```

▶ gültige Parameterarten: **Typen** (auch **Zeiger**) und **Ganzzahlen**

Verwendung

- ▶ Beispiel: generische max()-Funktion

```
template < type T > T max(T a, T b) {  
    return a > b ? a : b;  
}
```

- ▶ Verwendung

```
int a = 5, b = 7, c;  
c = max< int >(a, b);
```

- ▶ manchmal kann der Übersetzer den Typ *erraten*

```
int a = 5, b = 7, c;  
c = max(a, b);
```

Klassentemplates

- ▶ generische Beschreibung von
 - ▶ Datenstrukturen und
 - ▶ zugehöriger Algorithmen
- ▶ Deklaration/Definition
- ▶ Definition von Methoden
- ▶ Verwendung

Deklaration/Definition

▶ Deklaration

```
template < type T, int l > class Tuple;
```

▶ Definition

```
template < type T, int l > class Tuple {  
public:  
    void put(T item, int pos);  
    T    get(int pos);  
    void sort();  
};
```

▶ Parameterlisten: siehe [Funktionstemplates](#)

Definition von Methoden

- ▶ im Klassenrumpf

```
template < type T, int I > class Tuple {  
public:  
    void put(T item, int pos) { /* ... */ }  
    // ...  
};
```

- ▶ außerhalb des Klassenrumpfs

```
template < type T, int I >  
void Tuple< T, I >::put(T item, int pos) {  
    // ...  
}
```

- ▶ **Achtung:** Trennung in Header (.h) und Implementierungseinheit (.cc) existiert für Templates in gängigen Toolchains noch nicht

Verwendung

▶ Beispiel

```
int a = 1, b = 2, c;  
Tuple< int, 5 > my_tuple;  
  
my_tuple.put(a, 0);  
my_tuple.put(b, 1);  
my_tuple.put(4, 2);  
  
c = my_tuple.get(2);
```

▶ Templates ermöglichen **typsichere** Programmierung

```
float d = 0;  
my_tuple.put(d, 3); // compile time error
```

Template-Instanziierung

- ▶ Templates beschreiben noch keinen ausführbaren Code
- ▶ Übersetzer muss erst noch Lücken (Parameter) füllen
- ▶ diesen Vorgang bezeichnet man als *Instanziierung*
- ▶ Beispiel: generische max()-Funktion

Template

```
template < type T >
T max(T a, T b) {
    return a > b ? a : b;
}
```

Instanziierung (T == int)

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

- ▶ Wann findet die Instanziierung statt? - Bei der Verwendung!

```
c = max< int >(a, b);
```

Template-Spezialisierung

- ▶ Unterschiedliche Behandlung bestimmter Template-Parameter
 - ▶ Behandlung von Sonderfälle
- ▶ Beispiel: Sortieren
 - ▶ Tupel der Länge 1 sind trivialerweise sortiert
- ▶ Lösung: man *spezialisiert* das Template

```
template< type T, int L > struct Sorter {  
    void sort(T* array) { ... }  
};
```

```
template< type T > struct Sorter< T,1 > {  
    void sort(T* array) {}  
}
```

vollständige vs. partielle Spezialisierung

- ▶ vollständige Spezialisierung legt alle Parameter fest

```
template<> struct Sorter< int, 1 > {  
    void sort(int* array) {}  
}
```

- ▶ partielle Spezialisierung schränkt den Parameterraum ein

```
template< type T > struct Sorter< T, 1 > {  
    void sort(T* array) {}  
}
```

- ▶ Funktionstemplates sind **nicht** partiell spezialisierbar!

Template Metaprogramming

- ▶ ... C++ Templates sind noch viel mehr als das ...
- ▶ C++ Templates sind eine funktionale Programmiersprache
 - ▶ eingebettet in C++
 - ▶ wird vom Übersetzer *ausgeführt*
 - ▶ geeignet zum schreiben von Metaprogrammen
 - ▶ geeignet für generative Programmierung

Metaprogramme

Ein *Metaprogramm* verarbeitet, modifiziert oder erzeugt ein oder mehrere andere Programme.

Generative Programmierung

Generative Programmierung ist ein Paradigma, dessen charakteristische Eigenschaft es ist, dass Quelltext durch einen Generator erzeugt wird.

Fakultät als Template Metaprogramm

```
volatile int c;
```

```
template< int l > struct Faculty {  
    enum { result = l * Faculty< l - 1 >::result };  
};
```

```
template<> struct Faculty< 1 > {  
    enum { result = 1 };  
};
```

```
int main(void) {  
    c = Faculty< 5 >::result;  
    return 0;  
}
```

Fakultät als Template Metaprogramm

- ▶ die Fakultätsfunktion wird komplett im Übersetzer berechnet

Assembler-Dump der Funktion `main()`

<main >:

```
lea    0x4(%esp),%ecx
and    $0xffffffff0,%esp
push   0xffffffffc(%ecx)
push   %ebp
mov    %esp,%ebp
push   %ecx
mov    $0x78,0x8049640
mov    $0x0,%eax
pop    %ecx
pop    %ebp
lea    0xffffffffc(%ecx),%esp
ret
```




Andrei Alexandrescu.

Modern C++ Design: Generic Programming and Design Patterns Applied.

Addison-Wesley, 2001.



Krzysztof Czarnecki and Ulrich W. Eisenecker.

Generative Programming. Methods, Tools and Applications.

Addison-Wesley, May 2000.



Bruce Eckel.

Thinking in C++.

Prentice Hall PTR, Upper Saddle River, NJ, USA, 1995.



Stefan Kuhlins and Martin Schader.

Die C++-Standardbibliothek.

Springer-Verlag, 1999.



Thomas Strasser.

C++ Programmieren mit Stil.

dpunkt Verlag, 1997.