C++ Einführung

Dipl.-Medieninform.(FH) Severin S. Todt, MSc.

Fachgruppe Computergraphik und Multimediasysteme

Universität Siegen – Fachbereich 12

Version : 04.11.2004

Übersicht :

Übersicht (forts.):

Übersicht (forts.):

3.6. Zusammengesetzte Zuweisungen

1. Einleitung

1.1. „Hello World“

```
#include <iostream>

using namespace std;

int main(int argc, char** argv)
{
  cout << “Hello World” << endl;
  return 0;
}
```

→ *Hello World*

1.2. Struktur eines C++ Programms

Präprozessor Direktiven (#)

- Typ- und Variablendeklarationen
- Konstantendeklaration
- Makro Definition

- System- und Lokale Includes (von Headerfiles)

Anweisungsblock

- Anweisungssequenz aus einzelnen Anweisungen (**;**)
- (Funktions-) Block (**{ }**)

Kommentare

- Beliebige Position im Quellcode (***/**…***/*** , ***//***…)

1.3. Erzeugung von Anwendungen

Compileraufruf (hier unter Linux):

```
g++ HelloWorld.cpp
```

Interne Verarbeitung :

1. Preprocessing :
   - Einbinden von Headerfiles
   - Makro Ersetzung
   - bedingtes Kompilieren (Entfernen nicht relevanter Passagen)

   → *lesbarer Sourcecode*

2. Übersetzen in Assemblercode
   - Übersetzen des Quellcodes in Assemblercode

   → *lesbarer Assemblercode*

3. Objektcode Erzeugung
   - Übersetzen des Assemblercodes in Maschinencode

   → *Maschinencode*

4. Linken
   - Verbinden aller Objektfiles (auch zusätzlicher externer)

   →*Ausführbares Programm*

1.4. Literatur

Stroustrup, B.: „The C++ Programming Language“, Addison Wesley, 1997

Stroustrup, B.: „The Design and Evolution of C++”. Addisom Wesley, 1994

Eckel, B.: „Thinking in C++ Volume 1" & „Thinking in C++ Volume 2”, Broschiert, 2000
(auch zum Download http://www.mindview.net/Books)

2.1. Einfache Datentypen

Typ	Speicherbedarf in Byte	Inhalt	Bsp.
char	1	Character-Zeichen	'H', 'e', ' \n'
int	4	Ganze Zahlen	-32767, -231
short [int]	2	Ganze Zahlen	-32767
long [int]	4	Ganze Zahlen	-32767, -231
float	4	Gleitkommazahlen	1.1, -1.56e-32
double	8	Gleitkommazahlen	1.1, -1.56e-32, 5.68e+287
unsigned [int]	4	Natürliche Zahlen	32767, 32769, 231 - 1
long long [int]	8	Ganze Zahlen	-231, 263 - 1
long double	12	Gleitkommazahlen	5.68e+287, 5.68e+420
bool	1	Boolean {true,false}	true, false

2.2. Variablen

2.2.1. Variablendeklaration

[<storage specifier>] <type> identifier;

z.B.:
```
float myFloatVar;
int   myIntVar;
char  myCharVar;
```

- Variablenbezeichner werden in der Regel kleingeschrieben
- Bei zusammengesetzte Bezeichnern beginnt jeder Teil mit einem Großbuchstaben

2.2.2. Keywords

- Die definierten Keywords dürfen nicht als Variablenbezeichner oder Konstantennamen Verwendung finden

asm	do	if	return	typedef	auto
double	inline	short	typeid	bool	int
signed	typename	break	else	long	sizeof
union	case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using	char
export	new	struct	virtual	class	extern
operator	switch	void	const	false	private
template	volatile	const_cast	float	protected ´	this
wchar_t	continue	for	public	throw	while
default	friend	register	true	delete	goto
try	reinterpret_cast			dynamic_cast	

- Zusätzlich sollten einige Bezeichner nicht verwendet werden, die als alternative Notation zugelassen sind :

bitand	and	bitor	or	xor	compl
and_eq	or_eq	xor_eq	not	not_eq	

2.2.3. Initialisierung von Variablen

- Die Inittialisierung von Variablen kann direkt mit der Deklaration erfolgen :

  [<storage specifier>]<type> identifier = initial_value;

  z.B. :
```
float myFloatVar=0.5;
int   myIntVar=6;
char  myCharVar='s';
```

- Alternativ kann der initiale Wert auch in Klammern hinter dem Variablenbezeichner angegeben werden :

  [<storage specifier>]<type> identifier(initial_value);

  z.B. :
```
float myFloatVar(0.5);
int   myIntVar(6);
char  myCharVar('s');
```

2.2.4. Sichtbarkeitsbereiche von Variablen

- Sichtbarkeit von Variablen ist auf den aktuellen Block beschränkt:
  → Erinnerung : Ein Block begrenzt eine Sequenz von Anweisungen durch { und }

```
int main(int argc, char** argv)
{
  float myFloatVar(0.5);
  cout << “myFloatVar hat den Wert ” << myFloatVar << endl;
  return 0;
}
```

→ *myFloatVar hat den Wert 0.5*

- Globale Variablen (außerhalb von Funktionen) sind innerhalb des Files sichtbar

```
float myFloatVar(0.5);
int main(int argc, char** argv)
{
  cout << “myFloatVar hat den Wert ” << myFloatVar << endl;
  return 0;
}
```

→ *myFloatVar hat den Wert 0.5*

- Überschreibt eine lokale Variable eine globale Variable, wird die lokale ausgewertet :

```
float myFloatVar(0.5);
int main(int argc, char** argv)
{
  float myFloatVar(0.6);
  cout << “myFloatVar hat den Wert ” << myFloatVar << endl;
  return 0;
}
```

→ *myFloatVar hat den Wert 0.6*

--

```
float myFloatVar(0.5);
int main(int argc, char** argv)
{
  float myFloatVar(0.6);
  {
     float myFloatVar(0.8);
     cout << “myFloatVar hat den Wert ” << myFloatVar << endl;
  }
  return 0;
}
```

→ *myFloatVar hat den Wert 0.8*

2.2.5. Storage Specifier für Variablen

• Storage Specifier spezifizieren …
  → erweitern den Gültigkeitsbereich von Variablen
  → bestimmen die Lebensdauer von Variablen

[<storage specifier>] <type> identifier = initial_value;

storage specifier ::= {auto, register, static, extern, mutable}

→ auto : sog. Automatik Variablen, nur innerhalb eines Blocks erlaubt, entsprechen der Default Einstellung
→ register : nur für **int** (kompatible) Variablen erlaubt, sorgt als Compiler Hint dafür, dass Variable möglichst im Register Speicher abgelegt wird (Performance)
→ static : Außerhalb jeder Funktion : Variable ist global, aber extern nicht sichtbar.
Innerhalb einer Funktion : Sie überdauert den Funktionsaufruf,
d.h. Werte, Speicher etc. bleibt erhalten
→ extern : Bezugnahme auf ein globales externes Objekt in einem anderen Modul

2.3. Konstanten

2.3.1. Definierte Konstanten (`#define`)

- Eine einfache Art zur Definition von Konstanten erfolgt über die Vergabe von eigenen Namen für Werte jeden Typs
- Mit der **`#define`** Präprozessordirektive wird eine Textsequenz definiert, die beim Prelinken durch den angegebenen Wert ersetzt wird

```
#define PI 3.14159265
```

- Konstanten können auch für ganze Ausdrücke stehen → Makros

```
#define SCHREIB(c) cout << c << endl

int main(int argc, char** argv)
{
  SCHREIB("Hallo Welt");
  return 0;
}
```

→ *Hallo Welt*

(mehrzeilige Makros werden mit Hilfe der Escape-Sequenz „\“ erzeugt)

2.3.2. Deklarierte Konstanten (`const`)

- Deklarierte Konstanten werden als eine Variablendeklaration mit gleichzeitiger Initialisierung unter Verwendung des Keywords **`const`** erstellt.
- Als Typen dieser Konstanten kommen alle internen oder von ihnen abgeleiteten Datentypen in Frage

<type> identifier = initial_value;

```
const int breite = 100;
const float hoehe = 1.5;
```

- Der Präprozessor ersetzt jedes Vorkommen des Bezeichners durch den angegebenen Wert
- Im Gegensatz zu definierten Konstanten ist hier der Typ des Wertes bekannt (Typkonversionen können berücksichtigt werden)

3. Ausdrücke, Operatoren Mathematische Ausdrücke

3.1. Zuweisungsoperator

- Der Zuweisungsoperator wird genutzt, um Variablen eine Wert zuzuweisen

```
int x,y;
x = 0;
y = x;
```

3.2. Arithmetische Operatoren

- Arithmetische Operatoren dienen der Durchführung von arithmetischen Berechnungen wie Addition, Subtraktion, Multiplikation, Division und Modulo
- Es wird unterschieden zwischen unären und binären Operatoren
- Bei arithmetischen Operationen kann es zu impliziten Typkonversionen kommen

Unäre Arithmetische Operatoren :

- Arbeiten nur auf einem Operanden

Operator :	Beschreibung :	Beispiel :
-	Negation	-a

Binäre Arithemtische Operatoren :

- Arbeiten immer auf zwei Operanden

Operator :	Beschreibung :	Beispiel :
+	Addition	b + a
-	Subtraktion	b - a
*	Multiplikation	b * a
/	Division	b / a
%	Rest bei ganzzahliger Division	b % a

3.2.1. Inkrement / Dekrement Operatoren

- Der Inkrement und Dekrement Operator sind spezielle Implementierungen zur Erhöhung oder Verringerung eines ganzzahligen Variablenwertes um 1
- Bei der Verwendung ist zwischen Postifix und Prefix Notation zu unterscheiden

Prefix Notation

- Bei der Prefix Notation wird die Inkrement/Dekrement Operation ausgeführt, bevor die Variable weiter ausgewertet wird

```
int temp=0;
cout << "Die Variable hat den Wert : " << ++temp << endl;
```

→ *Die Variabel hat den Wert : 1*

Postfix Notation

- Bei der Postfix Notation wird die Operation nach der Auswertung der Variable ausgeführt

```
int temp=0;
 cout << "Die Variable hat den Wert : " << temp++ << endl;
```

→ *Die Variabel hat den Wert : 0*

3.3. Vergleichsoperator

- Vergleichsoperatoren ermöglichen den Vergleich zweier Ausdrücke
- Bei Vergleichsoperatoren handelt es sich ausschließlich um binäre Operatoren
- Das Ergebnis einer Vergleichsoperationen ist stets ein boolscher Wert, der zu **`true`** oder **`false`** ausgewertet werden kann

Operator :	Beschreibung :	Beispiel :
`>`	größer	`b > a`
`>=`	größer oder gleich	`b >= 3.14`
`<`	kleiner	`b < a`
`<=`	kleiner oder gleich	`b <= a/3`
`==`	gleich (Vorsicht bei Gleitkommazahlen)	`b == a` `(b – a) <= DBL_EPSILON`
`!=`	ungleich (Vorsicht bei Gleitkommazahlen)	`b != a` `(b –a )>= DBL_EPSILON`

3.4. Logische Operatoren

- Logische Operatoren ermöglichen Boolsche Operationen auf Ausdrücken
- Es wird unterschieden zwischen unären und binären Operatoren

Unäre logische Operatoren :

Operator :	Beschreibung :	Beispiel :	
!	logische Negation	! (3>4)	→ true

Binäre logische Operatoren :

Operator :	Beschreibung :	Beispiel :	
&&	logisches UND	(3>4)&&(3<4)	→ false
| |	logisches ODER	(3>4) | | (3<4)	→ true

3.5. Bitorientierte Operatoren

- Bitoperatoren arbeiten auf der Bitrepräsentation von Ausdrücken
- Als Operanden treten alle möglichen einfachen Basistypen auf
- Es ist zwischen unären und binären Operatoren zu unterscheiden

Unäre bitorientierte Operatoren :

Operator :	Beschreibung :	Beispiel :
~	Binärkomplement, bitweise Negation	~1 (~00000001 = 11111110)

Binäre bitorientierte Operatoren :

Operator :	Beschreibung :	Beispiel :	
&	bitweises UND	1 & 1	(00000001&00000001=00000001)
|	bitweises ODER	1 | 2	(00000001| 00000010=00000011)
^	bitw. Exklusiv ODER	1 ^ 3	(00000001^00000011=00000010)
<<	Linksverscheibung der Bits	1 << 3	(00000001<<3=00001000)
>>	logisches ODER	8 >> 1	(00001000>>1=00000100)

3.6. Zusammengesetzte Zuweisungen

- Zuweisungen und arithmetische sowie bitweise Operationen können in einer Anweisung zusammengefasst werden
- Die Variable auf die zugewiesen wird tritt dabei als linksseitiger Operand auf

```
int i,j,w;
float x,y;

i += j;
w>>= 1;
x *=y;
```