

3. Datentypen und Variablen II

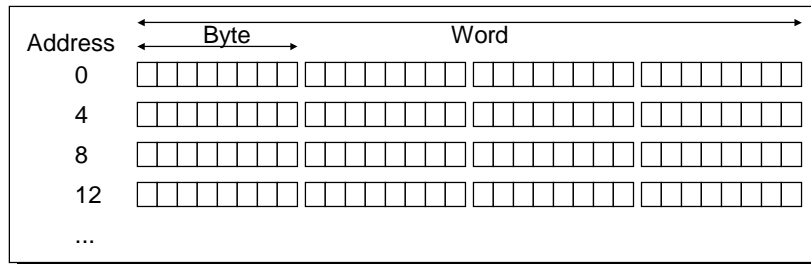
Prof. Dr. Markus Gross
 Informatik I für D-ITET (WS 03/04)

- Basistypen der Sprache C++
- Namenskonventionen
- Definitionsbereiche von Typen
- `const`-Variablen und deren Verwendung
- Arithmetische Operatoren
- Typenkonversion: automatisch und explizit



Der Hauptspeicher

- Die Daten sind im Speicher des Computers als Bitfolge abgelegt:
 - ◆ *Bit* (Binary unit), 2 (2^1) mögliche Zustände 0/1
 - ◆ *Byte* = 8 Bits, 256 (2^8) mögliche Zustände 00000000 - 11111111
 - ◆ *Words* (Wörter) sind architekturabhängig

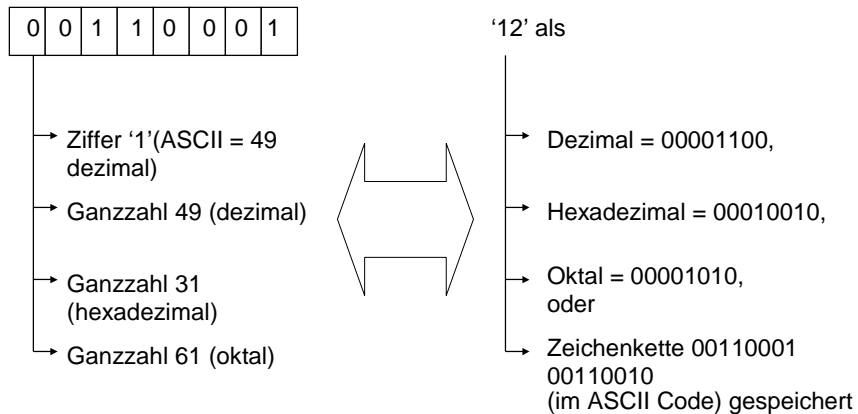


Der Hauptspeicher

- Die *Wortgrösse* eines Prozessors ist durch die Grösse der Prozessorregister vorgegeben
- Beispiele sind 32 bit (Pentium III) oder 64 bit SPARC
 - ◆ Words definieren den *Adressraum*
 - 2 Bytes, 2^{16} (64 KB)
 - 4 Bytes, 2^{32} (4 GB)
 - 8 Bytes, 2^{64} (4048 PB)

Den Speicher lesen

- Typen erlauben eine sinnvolle Interpretation des Speicherinhalts.



Typen in C++

- C++ unterscheidet zwischen Basistypen und abgeleiteten Typen
- Basistypen, wie `int`, `double`, `float`, `long`, `bool` sind in die Sprache eingebaut
- Abgeleitete Typen müssen explizit deklariert und definiert werden
- Beispiele sind Arrays oder Structs
- Abgeleitete Typen sind Vorläufer von *Klassen*
- Dieses Kapitel behandelt Basistypen

Namensgebung von Variablen

- In C++ dürfen Buchstaben, Zahlen und `_` zur Namensgebung verwendet werden
- Das führende Zeichen darf keine Zahl sein
 - ◆ Nicht möglich: `int 2_fel;`
- Grossbuchstaben werden von Kleinbuchstaben unterschieden
- Keine Begrenzung in der Anzahl der Zeichen
 - ◆ Alle verwendeten Zeichen sind signifikant
- Elemente der Sprache (*keywords*) können nicht verwendet werden
 - ◆ Nicht möglich: `int if = 1;`

Namensgebung von Variablen

- Namen dürfen nicht mit zwei `_` beginnen
- Namen dürfen auch nicht mit `_`Grossbuchstabe beginnen
 - ◆ `__time; _Donut; // nicht möglich`
- Derartige Definitionen führen zu unvorhersehbarem Verhalten
- Beispiele:


```
int P00;          // möglich
Int Poo;         // nicht möglich
int my-dream;    // nicht möglich
int my_dream;    // möglich
```

Namensgebung von Variablen

- Konventionen für Namen, die aus mehreren Worten bestehen:
 - ◆ C-Veteranen benutzen `_`(underscore)
 - ◆ `int markus_gross; // C-Style`
 - ◆ C++-Programmierer verwenden oft die Grossbuchstaben für den ersten Buchstaben ab dem zweiten Wort
 - ◆ `int markusGross; // C++-Style`
 - ◆ `int myLittleProg; // C++-Style`
- Frage des Programmierstils
- Wichtig ist eine *konsistente* Regelung zur Namensgebung



Integer Typen

- Der Datentyp Integer dient zur Speicherung ganzer, (vorzeichenbehafteter) Zahlen
 - ◆ 2, 46767, -23245, 0
- Da der Speicher nur endlich gross ist, können keine beliebig grossen Zahlen gespeichert werden
- C++ unterstützt 4 verschiedene Integer Basistypen
 - ◆ `char`
 - ◆ `short`
 - ◆ `int`
 - ◆ `long`
- Diese Typen besitzen unterschiedliche Grössen
- Alle Integer Typen können *signed* und *unsigned* definiert werden

Integer Typen

- Beispiel
 - ◆ `unsigned short number;`
 - ◆ Wenn Wertebereich von `short` z.B. -32768...32767, dann Wertebereich von `unsigned short` 0...65535
- Die Grösse der Integer-Typen ist implementationsabhängig
- Kann wie folgt ermittelt werden:
 - 1. `sizeof`-Operator
 - ◆ `sizeof(int);`
 - 2. Konstanten im `<climits>` header file
 - ◆ `MAX_INT;`
- Oft ist die Grösse von `int` an die Registerlänge der Architektur angelehnt (z.B. 32 bit)

Beispiel_1: Grösse von Int-Typen

```
// limits.cpp -- some integer limits
#include <iostream>
using namespace std;
#include <climits>

// use limits.h for older systems
int main()
{
    int n_int = INT_MAX;
    // initialize n_int to max int value
    short n_short = SHRT_MAX;
    // symbols defined in limits.h file
    long n_long = LONG_MAX;

    // sizeof operator yields size of type or of variable
    cout << "int is " << sizeof (int) << " bytes.\n";
    cout << "short is " << sizeof n_short << " bytes.\n";
    cout << "long is " << sizeof n_long << " bytes.\n\n";

    cout << "Maximum values:\n";
    cout << "int: " << n_int << "\n";
    cout << "short: " << n_short << "\n";
    cout << "long: " << n_long << "\n\n";

    cout << "Minimum int value = " << INT_MIN << "\n";
    return 0;
}
```

Header file mit symbolischen Konstanten

Definition und Initialisierung von Variablen

sizeof-Operator

Symbolische Konstanten climits

Symbolic Constant	Represents
CHAR_BIT	Number of bits in a char
CHAR_MAX	Maximum char value
CHAR_MIN	Minimum char value
SCHAR_MAX	Maximum signed char value
SCHAR_MIN	Minimum signed char value
UCHAR_MAX	Maximum unsigned char value
SHRT_MAX	Maximum short value
SHRT_MIN	Minimum short value
USHRT_MAX	Maximum unsigned short value

Symbolische Konstanten climits

Symbolic Constant	Represents
<code>INT_MAX</code>	Maximum <code>int</code> value
<code>INT_MIN</code>	Minimum <code>int</code> value
<code>UINT_MAX</code>	Maximum <code>unsigned int</code> value
<code>LONG_MAX</code>	Maximum <code>long</code> value
<code>LONG_MIN</code>	Minimum <code>long</code> value
<code>ULONG_MAX</code>	Maximum <code>unsigned long</code> value

Beispiel_2: Ueberlauf

```
// exceed.cpp -- exceeding some integer limits
#include <iostream>
using namespace std;
#define ZERO 0 // makes ZERO symbol for 0 value
#include <climits> // defines INT_MAX as largest int value
int main()
{
    short sam = SHRT_MAX; // initialize a variable to max value
    unsigned short sue = sam; // okay if variable sam already defined

    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nAdd $1 to each account.\nNow ";
    sam = sam + 1;
    sue = sue + 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nPoor Sam!\n";
    sam = ZERO;
    sue = ZERO;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\n";
    cout << "Take $1 from each account.\nNow ";
    sam = sam - 1;
    sue = sue - 1;
    cout << "Sam has " << sam << " dollars and Sue has " << sue;
    cout << " dollars deposited.\nLucky Sue!\n";
    return 0;
}
```

- Variablen zeigen *zyklisches* Verhalten
- Variablenüberlauf ist eine gefährliche Fehlerquelle

Copyright: M. Gross, ETHZ, 2003 15 **ETH**
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Deklaration, Definition, und Initialisierung

```

char index;           // Deklaration und Definition
index = 'A';         // Initialisierung
    
```

Deklaration
Definition
Initialisierung

index
index
index

Copyright: M. Gross, ETHZ, 2003 16 **ETH**
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Initialisierung

- Variablen innerhalb von Funktionen müssen *explizit initialisiert* werden
- Ansonsten undefiniertes Verhalten des Programms
- Initialisierung von Integer-Typen mit Konstanten kann in 3 verschiedenen Zahlensystemen erfolgen
 - ◆ Dezimal
 - ◆ `int chest = 42;`
 - ◆ Hexadezimal
 - ◆ `int waist = 0x42;`
 - ◆ Oktal
 - ◆ `int foot = 042;`
- Typen können explizit durch Suffixe vorgegeben werden
 - ◆ `22022UL`

Char-Variablen

- Typ `char` ist nur 1 Byte= 8 bit gross
- Dient zur Speicherung eines einzelnen *alphanumerischen* Zeichens oder kleiner Zahlen
- Bekanntester Zeichensatz in der Computerwelt ist der *ASCII* Zeichensatz
- Char-Variablen werden wie folgt initialisiert
 - ◆ `char c = 'M';`
- In C++ (sowie in C) werden Characters als Integers abgespeichert
 - ◆ `'M'` wird beispielsweise als 77 abgelegt
- Somit können arithmetische Operatoren angewandt werden
- `char` ist per default weder signed noch unsigned (implementationsabhängig)
 - ◆ `signed char number; // explizit vorgeben!!`
 - ◆ `unsigned char id;`

Beispiel_3: Char und Int

```
// morechar.cpp -- the char type and int type contrasted
#include <iostream>
using namespace std;

int main()
{
    char c = 'M';          // assign ASCII code for M to c
    int i = c;            // store same code in an int
    cout << "The ASCII code for " << c << " is " << i << "\n";

    cout << "Add one to the character code:\n";
    c = c + 1;
    i = c;
    cout << "The ASCII code for " << c << " is " << i << '\n';

    // using the cout.put() member function to display a char
    cout << "Displaying char c using cout.put(c): ";
    cout.put(c);

    // using cout.put() to display a char constant
    cout.put('!');

    cout << "\nDone\n";
    return 0;
}
```

Inkrement einer char-Variablen

Aufruf einer Member-Funktion von cout

Objekte und Membership

- Ein Hauch von Objektorientierung
- `cout.put` ist eine *Membership-Funktion* des Objektes `cout`
- Die Funktion `cout.put` gehört also zum Objekt `cout`
- Der Operator `.` heisst *Membership-Operator*
- Er muss zum Aufruf dieser speziellen Funktion verwendet werden
- Mehr dazu später !

Escape-Sequenzen

- Manche Zeichen haben eine besondere Bedeutung
- Sie können zur Steuerung der Ausgabe oder zu sonstigen Zwecken dienen
- *Escape-Sequenzen* beginnen alle mit einem `\` (*backslash*)
- Beispiele:
- `\n` newline char
- `\a` alert char
- Kann beliebig in die Ausgabe eingebaut werden

Copyright: M. Gross, ETHZ, 2003 21 **ETH**
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Beispiel_4: Escape-Sequenzen

```

// bondini.cpp -- using escape sequences
#include <iostream>
using namespace std;
int main()
{
    cout << "\aOperation \"HyperHype\" is now activated!\n";
    cout << "Enter your agent code:_____\\b\\b\\b\\b\\b\\b\\b";
    long code;
    cin >> code;
    cout << "\aYou entered " << code << "...\\n";
    cout << "\aCode verified! Proceed with Plan 'Z3!\\n";
    return 0;
}
    
```

Alert zum akustischen Signal

Newline zur neuen Zeile

Backspace um den Cursor zurückzufahren

Copyright: M. Gross, ETHZ, 2003 22 **ETH**
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Escape-Sequenzen

Character Name	ASCII Symbol	C++ Code	ASCII Decimal Code	ASCII Hex Code
Newline	NL (LF)	<code>\n</code>	10	0xA
Horizontal tab	HT	<code>\t</code>	9	0x9
Vertical tab	VT	<code>\v</code>	11	0xB
Backspace	BS	<code>\b</code>	8	0x8
Carriage return	CR	<code>\r</code>	13	0xD
Alert	BEL	<code>\a</code>	7	0x7
Backslash	<code>\</code>	<code>\\</code>	92	0x5C
Question mark	<code>?</code>	<code>\?</code>	63	0x3F
Single quote	<code>'</code>	<code>\'</code>	39	0x27
Double quote	<code>"</code>	<code>\"</code>	34	0x22

Boolean und Konstanten

- In C++ gibt es neu einen Typ `bool`
- Er kann nur die binären Werte `true` und `false` annehmen
 - ◆ `bool neumann = true;`
- Numerische Werte ungleich 0 werden nach `true` konvertiert
 - ◆ `bool neumann = -100;`
 - ◆ `bool neumann = 0;`
- 0 wird nach `false` konvertiert
- Konstanten werden durch einen zusätzlichen *Qualifier* definiert
 - ◆ `const int MONTH = 13;`
- Konstanten werden oft in Grossbuchstaben definiert

Konstanten

- Zuweisungen von neuen Werten innerhalb des Programmflusses ergeben Compilerfehler
- Konstanten müssen während der Deklaration initialisiert werden
 - ◆ `const int fat = 10;`
- In C wurden Konstanten durch den Präprozessor behandelt
 - ◆ `#define MONTH 12;`
- `const` Qualifier ist wesentlich mächtiger
 - ◆ Explizite Typenangabe
 - ◆ *Gültigkeitsbereich (scope)* einschränkbar
 - ◆ Kann in Verbindung mit komplexen Datentypen verwendet werden

Gleitkommazahlen

- Gleitkommazahlen, auch *floating point numbers* genannt, werden im Computer in einem besonderen Format abgespeichert
- Es besteht aus Vorzeichen, Mantisse und Exponent
 - ◆ `-12.345*10+13;`
- Vorzeichen beziehen sich auf Zahl und Exponent
- C++ stellt 3 Typen für Floating Point Zahlen zur Verfügung
- `float`, `double`, `long double`
- Grösse ist implementationsabhängig
- Kann mit `climits` ermittelt werden

Gleitkommazahlen

- Eingabe von Gleitkommazahlen auf zweierlei Art möglich
 - ◆ `x = 1.0;`
 - ◆ `float x = 1.0E-4;`
- Zusätzliche Suffixe `f`, `F`, `l`, `L` zur Spezifizierung
- Die Gleitkommadarstellung besitzt nur begrenzte Präzision
- Der Wertebereich der zu repräsentierenden Grösse sollte gut bekannt sein
- Für numerische Operationen sollte `double`, besser `long double` verwendet werden

Beispiel_5: Genauigkeit

```
//fltadd.cpp -- precision problems with float
#include <iostream>
using namespace std;
int main()
{
    float a = 2.34E+22f;
    float b = a + 1.0f;

    cout << "a = " << a << "\n";
    cout << "b - a = " << b - a << "\n";
    return 0;
}
```

Grosse Gleitkommazahl

Differenz ist ungenau

Typenkonversion

- Typen werden in C++ in folgenden Fällen automatisch konvertiert:
 - ◆ Zuweisung verschiedener Typen
 - ◆ Ausdrücke mit verschiedenen Typen
 - ◆ Funktionsargumente
- Konversionen können zu Genauigkeitsverlust oder zu nichtkontrollierbaren Rundungsfehlern führen
 - ◆ `long so_long = 1000000000;`
 - ◆ `short thirty = 30;`
 - ◆ `so_long = thirty; // O.K.`
- Insbesondere wird bei der Konversion von `float` nach `int` abgeschnitten
- Bei Konversion folgt Compiler einer Checkliste



Explizite Typenkonversion

- Typenkonversion kann durch einen *type cast* erzwungen werden
- Gewünschter Typ wird dazu vor die Variable gestellt
- Syntax lässt zwei Notationen zu:
 - ◆ `(long) thorn;`
 - ◆ `long (thorn);`
- Allgemein
 - ◆ `(typename) value;`
 - ◆ `typename (value);`
- Zweite Variante neu von C++ und erinnert an einen Funktionsaufruf
- Im Zweifelsfall dienen explizite Typecasts der Lesbarkeit des Codes

Beispiel_6: Genauigkeit

```
// assign.cpp -- type changes on assignment
#include <iostream>
using namespace std;
int main()
{
    float tree = 3;           // int converted to float
    int guess = 3.9832;      // float converted to int
    int debt = 3.0E12;       // result not defined in C++
    cout << "tree = " << tree << "\n";
    cout << "guess = " << guess << "\n";
    cout << "debt = " << debt << "\n";
    return 0;
}
```

Abschneiden der
Nachkommastellen

Arithmetische Operatoren

- *Operatoren* dienen zur Manipulation von Variablen (oder ganzen Objekten)
- Zusammen mit den Operanden bilden sie einen *Ausdruck (expression)*
- C++ kennt 5 Basisoperatoren für arithmetische Operationen:
 - ◆ +, -, *, /, %
- / Operator liefert bei Integer-Typen nur den ganzzahligen Anteil, also $17 / 3 = 5$
- %-Operator errechnet den *Modulus*, also den Rest
- Also $17 \% 3 = 2$
- % ist nur auf Integer-Typen anwendbar

Prioritäten und Assoziativität

- Wenn mehrere Operatoren auf den gleichen Operanden angewandt werden, kann es Konflikte geben
 - ◆ `int flies = 3 + 4 * 5; // 35 oder 23 ?`
- Prioritätsregeln lösen diese Konflikte (*precedence*)
- Bei gleicher Priorität unterscheidet man zwischen *linksassoziativ* und *rechtsassoziativ*
 - ◆ `int flies = 120 / 4 * 5; // 150 oder 6 ?`
- Regeln dazu sind in entsprechenden Tabellen zu finden
- ()-Klammer besitzt hohe Priorität und kann im Zweifelsfall immer verwendet werden
 - ◆ `int flies = (3 + 4) * 5; // 35`

Priorität und Assoziativität

Operator level	Associativity
::	left to right
() [] -> .	left to right
! ~ ++ -- + (unary) - (unary) * (deference) & (address of) new delete (data type)	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
> <= > >=	left to right
== !=	left to right
&	left to right
	left to right
&&	left to right
	left to right
= += -= *= /= %=	right to left

abnehmende Priorität

Komplette Tabelle:
siehe Prata,
Anhang D