Informatik I

Jules Authier
Februar 2018

Grundlagen

Deklaration / Definition

Deklaration: Name wird eingeführt
Definition: Kreiert auch Entität,die für den Namen steht

Expression/ Statement

Jede auswertbare Kombination von Variablen Konstanten und Operatoren.

Speichervariabeln

int a=5;
Der Wert bleibt gleich auch nach Funktionsaufruf erhalten.

Typen

char, short, int, long, unsigned int, double, float, long, bool

Bool

Hat Werte « true » or « false », false→0, true→1, 0->false, alle andere int→true

int / unsigned int

Der Wertebereich von int ist { -2^{B-1},...,0,...,$2^{B-1}-1$ }.
Der Wertebereich von unsigned int ist { 0,...,2B-1 }.
Gemischte Ausdrücke sind immer vom Typ unsigned int.
Konversion:

int		unsigned int
x (>=0)	->	x
x (<0)	->	x + 2^B
x	<-	x (im Wertebereich von int)
implementierungsabhängig	<-	x (nicht im Wertebereich von int

ACHTUNG: eine Variabel definiert in einer Umgebung existiert nur in dieser Umgebung!

Operationen

Symbol	Stelligkeit	Präzedenz	Assosiativ.
+,- ,!(unär)	1	16	rechts
*, / , %	2	14	links
+,- (binär)	2	13	links
++a, --a	1	17	links
a++, a--	1	16	rechts
[ ]	2	17	links
*	1	16	rechts
&&	2	6	links
||	2	5	links
<,>,<=,>=	2	11	links
==, !=	2	10	links

Kontrollanweisungen

If-else

```
if(condition)
{
        satement 1}
else
{
        statement 2 }
```

For loop

```
for(init statement; condition; expression satement)
{
        statement
}
```

While Loop

```
while(condition)
{
        statement
}
```

(the programm enter in the loop only if the condition is true)

Do...while loop

```
do
{
        statement
} while(condition)
```

(the programm will at least execute the statement one time)

Switch Statement

```
switch(summe%7)
{
    case 0: std::cout<< "monday"<<std::endl;
    break;
    case 1: std::cout<< "tuesday"<<std::endl;
    break;
    case 2: std::cout<< "wednesday"<<std::endl;
    break;
    case 3: std::cout<< "thursday"<<std::endl;
    break;
    case 4: std::cout<< "friday"<<std::endl;
    break;
    case 5: std::cout<< "saturday"<<std::endl;
    break;
    case 6: std::cout<< "sunday"<<std::endl;
    break;
}
```

ein default statement kann noch benutzt werden, es wird ausgeführt falls alle andere Falle nicht passen.

Anweisungen

Continue; Rest des Rumpfes der Iterationsanweisung wird übersprungen, Iteration wird aber nicht abgebrochen.
Break; Umschliessende Iteration wird sofort beendet.

Funktionen

Funktion definition:

```
//PRE: -
//POST: switch the values of x and y
void swap (int& x, int& y)
{
    int t;
    t=x;
    x=y;
    y=t;
}
```

Funktions call

```
{
    swap(array[i],array[i-1]);
}
```

Die Funktionen haben immer einen Typ und ein “return” statement, wenn der Typ nicht “void” ist.

Die Lokale Variabeln der Funktion sind ausserhalb der Funktion nicht sichtbar und sind bei jedem Aufruf der Funktion neu angelegt.
In der Funktions definition können normale Variabeln, References und Pointers benutzt werden.

Referenztypen

`T&` Gelesen als „*T*-Referenz"

Zugrundeliegender Typ

- *T&* hat den gleichen Wertebereich und gleiche Funktionalität wie *T*, ...
- nur Initialisierung und Zuweisung funktionieren anders.

Eine Referenz ist ein Alias einer Variabel, eine Zuweisung an die Referenz erfolgt an das Objekt hinter dem Alias.

Intern wird ein Wert vom Typ *T&* durch die Adresse eines Objekts vom Typ *T* repräsentiert.

```
int& j;  // Fehler: j muss Alias von irgendetwas sein

int& k = 5;  // Fehler: Das Literal 5 hat keine Adresse
```

Der Rückgabe Typ einer Funktion kann eine Referenz sein, (return by reference), in diesem Fall ist der Funktionsaufruf selbst ein L-Wert

Referenz-Richtlinie :

Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist so lange "leben" wie die Referenz selbst.

Konstanten

Konstanten sind Variablen mit unveränderbaren Wert Es gibt ein Compiler Fehler, wenn man probiert, den Wert einer const Variabel zu verändern.

Const-Referenzen

Können auch mit R-Werten initialisiert werden

`const T& r = lvalue;`
r wird mit der Adresse von *lvalue* initialisiert (effizient)

`const T& r = rvalue;`
r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (flexibel)

Regel

Argumenttyp `const T &` (call by *read-only* reference) wird aus Effizienzgründen anstatt *T* (call by value) benutzt, wenn der Typ *T* grossen Speicherbedarf hat. Für fundamentale Typen (`int`, `double`,...) lohnt es sich aber nicht.

Const-Objekten können nur const-Referenzen zugewiesen werden.

Arrays

`type name[size];`	//Grösse ist konstant und muss zur Kompilierzeit bekannt sein
`int a[2] = {1,1};`	//Ok
`int a[]= {1,1};`	//Auch Ok

Falls nicht initialisiert, haben die Elemente zufällige Werte, aber:

`int a[500] = {1, 2, 3};`	//Alle andern Elemente sind so 0

Die Indizes beginnen bei 0.
Ein Array kennt seine Länge nicht.
Vergleiche zwischen Arrays sind nicht möglich.

Mehrdimensionale Arrays

`int a[2][3];`	//2x3-Matrix
`int b[][3]={{2,4,6},{1,3,5}}`	//Erste Dimension wird vom Compiler deduziert

Arrays sind primitiv

- Man kann auf Elemente ausserhalb der Array zugriffen (undef. Verhalten)
- Man kann Arrays nicht wie bei anderen Typen initialisieren und zuweisen
- Arrays bieten kein Luxus wie eingebautes Initialisieren und kopieren

Strings als Array

Strings sind als Char-Arrays darstellbar:

`char a[]="Hallo";`	//ok
`std::string b=a;`	//ok

oder:

`char* a ="Hallo";`	//ok
`std:string b=a;`	//ok

Aber Strings können nicht zur Char-Arrays konvertiert werden:

`std::string b="Hallo";`	//ok
`char a[]=b;`	//Fehler !

Strings sind nicht primitiv (aber auch keine wirklichen Vektoren)

`std::string a(9,'d');`	//a="ddddddddd"
`int b= a.length;`	//b=9
`if(string1==string2);`	//Vergleich möglich

easy way to implement an array with a function

```
//PRE: an array of size 3
//POST: implement the array v[3]
void vectorimpl (int (&v)[3])
{
    for (int i=0; i<3; ++i)
    std::cin>>v[i];
}
```

Vektoren

Vektoren sind nicht primitiv, sie kennen ihre Grosse.

`#include <vector>;`	
`std::vector<int> a(n, 0);`	//intVektor aus0 der Länge n
`a[5]= 1;`	//Zugriff wie bei Arrays
`int b = a.size();`	//Kennt seine Länge
`std::vector<std::vector<int>>`	//Zweidimensionaler
`a (n,std::vector<int>(m));`	//Vektor (n x m)

Vektor Iteratoren

`a.begin();`	//Erstes Element von a
`a.end();`	//Letztes Element von a

`typedef std::string::iterator Iterator;` allow us to iterate over vectors, with this function for ex.

```
bool lexicographic_compare(Iterator first1, Iterator last1, Iterator first2, Iterator last2)
{
  do
  {
      if (*first1<*first2)
        return true;
      else if(*first1>*first2)
        return false;
      else
      {
          ++first1;
          ++first2;
      }
  }while(first1!=last1);

if (lexicographic_compare (name1.begin(), name1.end(), name2.begin(), name2.end()))
« functions call »
```

Type Def

alias können verwendet werden um Daten etwas kürzer zu schreiben

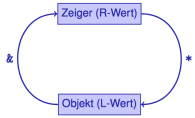
`typedef std::vector<int>::const_iterator Cit;`

Assert

`assert (expression);`

Expression muss nach bool konvertierbar sein. Wird überprüft, falls false, dann wird das Programm hart abgebrochen (mit Fehlermeldung), falls true, dann kein Effekt.

Pointers




```
int a = 10;
int* b = &a;
int c=*b;
```

T* ist ein Pointer auf T
&a liefert die Adresse von a
*b liefert den Wert von b

Man zeigt nicht mit einem *double auf einem int !

Pointersvergleiche

Pointers vergleiche liefern `true`, wenn sie auf das selbe Objekt zeigen, nicht wenn ihre Objekte den selben Wert haben.

Pointers & Arrays

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- a+5 ist ein Zeiger direkt hinter das Ende des Feldes (past-the-end), der nicht dereferenziert werden darf.
- Zeigervergleich (p < a+5) bezieht sich auf die Reihenfolge der beiden Adressen im Speicher.

„a“ ist die adresse des erstes Elements des Arrays

Nullpointer

kann nicht dereferenziert werden, ist benutzt um undefinierten Verhalten zu vermeiden.

Pointers Substraktion

Die Pointers Differenz = Anzahl Elemente zwischen die zwei Zeiger

Const

```
const int a <-> int const a
const int*a <-> int const*a

int const a;       //a ist ein konstantes int
int const*a;       //a ist ein pointer auf einem const int
int* const a;      //a ist ein const Pointer auf einem int
int const* const a;  //a ist ein const Pointer auf
                       einem const int
```

Rekursion & EBNF

Ein EBNF ist ein gegebenes Alphabet, das man benutzen soll, um einen Programm zu schreiben, ex:

```
Tree = '(' Branches ')'.
Branches = Branch { Branch }.
Branch = Length | Length Tree.
Length = unsigned int.
```

Mit der Rekursion in Informatik, brauchen wir:

1. Ein rekursiver Aufruf
2. Ein Fortschritt (damit das Programm keine Endlosschleife wird)

Structs

« A data structure is a group of data elements grouped together under one name. These data elements, known as members, can have different types and different lengths. »

```
//the new type complex
struct complex{
    double r;
    double i;
};
```

here z.B. ein Stuct „complex“ mit 2 Membervariablen

Operator Überladung

es ist sehr nützlich und oft benutzt mit Structs, die Operator Überladung ist es zugelassen, einen neuen Datentype ganz normal zu benutzen, z.B:

```
//PRE: two complex number
//POST: return the result of the addition of a and b
complex operator+(complex a, const complex b)
{
    a.r+=b.r;
    a.i+=b.i;
    return a;
}
```

mit diesen zwei Funktionen kann man ganz normal Komplex addieren et subtrahieren (Reference sind in diesem Fall egal)

```
//PRE: two complex number
//POST: return the result of the substraction of a and b
complex& operator-(complex& a, const complex& b)
{
    a.r-=b.r;
    a.i-=b.i;
    return a;
}
```

oder noch boolien operator zu benutzen

```
//PRE: two complex number
//return true if a and b are the same complex numbers
bool operator==(const complex a, const complex b)
{
    if (a.r==b.r && a.i==b.i)
        return true;
    else
        return false;
}
```

noch schöner, eingabe und ausgabe Operatoren können auch überladen werden

```
// PRE: in starts with a complex number of the form "<r,i>"
// POST: the complex number r has been read from in
std::istream& operator>> (std::istream& in, complex& r)
{
    char c;
    return in >> c >> r.r >> c >> r.i>> c;
}
```

```
//PRE: a complex number r
// POST: r has been written to out
std::ostream& operator<< (std::ostream& out, complex r)
{
    return out << "["<< r.r << ","<< r.i<<"]";
}
```

aber in diesem Fall muss man Reference benutzen.

Klassen

Größte Unterschied mit einem Struct;
In eimen Struct ist standardmäßig nichts versteckt, in einer Klasse ist alles versteckt.
Im Fact benutzen wir in einer Klasse, zwei Abschnitte „private“ wo die Membervariablen sind, und „public“ wo die Memberfunktionen sind.

Konstruktoren

„A class constructor is a special member function of a class that is executed whenever we create new objects of that class.
A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables“.

Getters & Setters

Getters & Setters sind Memberfunktionen, womit wir Variablen einer Klasse initialisieren oder modifizieren(setters) oder bekommen(getters).

Beispiel:

```
class Human
{
  public:
  //constructors:
    Human();
    Human(unsigned int _size,unsigned int _age,char _color);

  //memberfunctions
    unsigned int getsize()const;  //"getter"
    unsigned int setsize(unsigned int _size); //"setter"
    unsigned int getage()const;
    unsigned int setage(unsigned int _age);
    char getcolor() const;
    char setcolor(char _color);

  private:
    unsigned int size;
    unsigned int age;
    char color;
};
```

erst wurde eine Klasse „Human“ gemacht, mit constructors, memberfunctions and private Variablen

```
//constructors
Human::Human():size(0),age(0),color(0) {}

Human::Human(unsigned int _size,unsigned int _age,char _color)
:size(_size),age(_age),color(_color){}
```

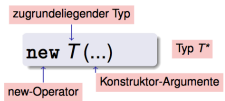
```
//functions
unsigned int Human::getsize()const
{
    return size;
}
unsigned int Human::setsize(unsigned int _size)
{
    size=_size;
}
unsigned int Human::getage()const
{
    return age;
}
unsigned int Human::setage(unsigned int _age)
{
    age=_age;
}
char Human::getcolor() const
{
    return color;
}
char Human::setcolor(char _color)
{
    color=_color;
}
```

da sind die Funktionen geschrieben, alle sind setters oder getters

```
int main()
{
    Human Marc(182,25,'g');

    std::cout<<Marc.getsize()<<'\n';
    std::cout<<Marc.getcolor()<<'\n';
    std::cout<<Marc.getage()<<'\n';

    Human Jules;

    Jules.setsize(173);
    std::cout<<Jules.getsize()<<'\n';
    Jules.setcolor('b');
    std::cout<<Jules.getcolor()<<'\n';
    Jules.setage(18);
    std::cout<<Jules.getage()<<'\n';
}
```

In der main Funktion sind zwei Human herstellt, ein mit einem Konstruktor, der andere mit setters.

New

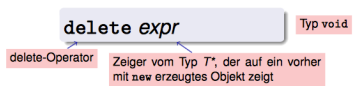


- **Effekt:** Neues Objekt vom Typ *T* wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
int* myInt1 = new int;          // allocate memory for an integer
int* myInt2 = new int(2);       // stores 2 at new address
int* myIntRange = new int[5];   // allocates memory for 5
                                // consecutive integers
```

Delete

Objekte, die mit `new` erzeugt worden sind, haben *dynamische Speicherdauer:* sie “leben”, bis sie explizit *gelöscht* werden.



- **Effekt:** Objekt wird gelöscht, Speicher wird wieder freigegeben

```
delete myInt1;          // deconstruct dynamic variable
myInt1 = 0;             // set pointer to point 'nowhere'
delete[] myIntRange;    // deconstructs dynamic range
myIntRange = 0;
```

Achtung ! Ein Pointer darf nicht auf einem deleted memory space zeigen, deshalb MUSS man es gleich 0 setzen.

Kopierkonstruktor

Der Kopierkonstruktor einer Klasse T ist der eindeutige Konstruktor mit der Deklaration T(const T& x);
Er wird automatisch aufgerufen, wenn Werte vom Typ T mit Werten vom Typ T initialisiert werden.
T x = t; //t vom Typ t T x (t);
Falls kein Kopierkonstruktor deklariert ist, so wird der automatisch erzeugt und initialisiert mitgliedsweise, was zu Problemen führen kann.

```
//POST: new queue containing copy of queue other
Queue::Queue(const Queue& other) : first(0), last(0)
{
    Node* p = other.first;
    while(p!=0)
    {
        int val=p->value;
        enqueue(val);
        p=p->next;
    }
}
```

Destruktor

Wenn ein Objekt nicht mehr gebraucht wird, soll es gelöscht werden (delete). Dafür ist der Destruktor zuständig. Er ist eine Mitgliedsfunktion und wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjektes endet. Wird kein Destruktor deklariert, so wird er automatisch erzeugt und ruft die Destruktoren für die Datenmitglieder auf.

```
Queue::~Queue()
{
 if(!is_empty())
    first->clear();
}
```