ILV Datenstrukturen und Algorithmen

01: Keyword const, copy constructor and assignment operator

Stefan Huber

FH Salzburg, Studiengang MMT / 2012

Kapitel

Einführung und Organisation

Was ist ein Algorithmus?

- Algorithmen sind die Lösungen zu einem Berechnungsproblem der Informatik.
- Ein Algorithmus ist eine Folge von Programmanweisungen (bzw. Rechenschritten), welche aus einer Eingabe eine Ausgabe *berechnet*.

Beispiel

Gegeben sind 10^6 natürliche Zahlen in aufsteigender Reihenfolge. Kommt die Zahl n darunter vor?

- Eingabe: Sortierte Folge S von 10^6 Zahlen aus $\mathbb{N}$ und die Zahl $n \in \mathbb{N}$.
- Ausgabe: `true` falls n in S vorkommt, `false` falls nicht.

- Wie lautet ein Algorithmus zur Lösung des Problems?
- Wieviele Rechenschritte braucht dieser Algorithmus?
  - Kann ich durch das Betrachten von nur 20 dieser Zahlen eine korrekte Antwort liefern?
- Wieviel Speicher braucht dieser Algorithmus?
- Wie schnell keine eine algorithmische Lösung bestenfalls sein?

Was ist eine Datenstruktur?

- Eine Datenstruktur speichert Daten nach einem bestimmten System ab.
- Je nach Datenstrukturen kann der Zugriff, das Hinzufügen, das Löschen, usw. von Daten unterschiedlich effizient sein.
- Beispiel: Ein array ist eine Datenstruktur.

Beispiel

Kann ich Zahlen clever abspeichern (in einer Datenstruktur), sodass, wenn ich 10^6 Zahlen in beliebiger Reihenfolge einfüge, ich effizient (in 20 Schritten) feststellen kann, ob die Zahl n vorkommt? Wie „teuer“ ist das Einfügen einer Zahl?

Ja: Ausgeglichene Binärbäume!

Inhaltsübersicht

- Teil I (Einführung Programmierung):
  - Keyword const, copy constructor, deep and shallow copy
  - Vererbung vs. Komposition, assert vs. exception
  - Polymorphismus
  - Statische Klassenfunktione und -attribute, function/operator overloading
  - Templates
- Teil II (Algorithmen & Datenstrukturen):
  - Verkette Listen
  - Queue, Stack
  - Binärbäume
  - Rekursion, Komplexitätstheorie
  - Sortieren
  - Hashtables
  - Standard Template Library
  - Graphen 1
  - Graphen 2

Organisatorisches

- Aufbau
  - 3 Einheiten Vorlesung (Stefan Huber)
  - 2 Einheiten Übungen (Gerhard Mitterlechner)
- Beurteilung durch zwei gleichgewichtete Prüfungsteile:
  1. 4 Tests zu 30–40 min.
  2. Übungsleistung:
     - Hausübungen
     - Präsentationen

  Beide Prüfungsteile müssen positiv absolviert werden!
- Hausübungen:
  - Wöchentlich 2–3 Aufgaben.
  - Abgabe per git.
  - Deadlines sind strikt!
  - Code muss mit gcc-4.x.x übersetzt werden können!

Details zur Organisation finden sie im ▸ Wiki.

Kapitel

Konstante Variablen, Parameter, Pointer und Klassenfunktionen

Konstante Variablen

- Mit dem Schlüsselwort `const` erlaubt C++ die Definition von Konstanten, d.h. konstanten Variablen.
- Konstanten können nicht verändert werden.
- Der Wert der Variable muss bei der Initialisierung angegeben werden.

```
1 const int size = 10;
2 int array[size];    // Array sizes need to be const
3 const double pi = 3.1415926535897324;
4 size = 4;           // Error!
```

Konstante Variablen

- Auch Klassenvariablen können als konstant deklariert werden.
  - Die Initialisierung findet im Konstruktor durch *Initialisierungslisten* statt.
- Klassenvariablen von konstanten Instanzen dieser Klasse werden selbst als konstant angesehen.

```
1  class ClassA
2  {
3    const int member;
4    int member2;
5
6    ClassA(int param) :
7      member(param)
8    {
9    }
10 };
11
12 void test()
13 {
14   ClassA o1(0);
15   const ClassA o2(0);
16   o2.member2 = 2;          // Error, o2 is const!
17 }
```

Konstante Parameter

- Konstante Parameter sind wie konstante Variablen.
- Konstante Objekte können nicht per Referenz oder per Pointer an Funktionen gereicht werden.
  - Sie können als konstante Referenz, konstante Pointer oder per Kopie übergeben werden.
  - Sonst könnte das konstante Objekt in der Funktion verändert werden.

```
1  struct Vector
2  {
3    double x, y;
4  };
5  double len(const Vector& v)
6  {
7    return sqrt(v.x*v.x + v.y*v.y);
8  }
9  void test()
10 {
11   const Vector v;
12   cout << "Length: " << len(v) << endl; // must take const ref!
13 }
```

Konstante Pointer

- Mit dem Schlüsselwort `const` kann sowohl ein Pointer an sich (d.h. die Adresse), als auch der Wert, auf den gezeigt wird, als konstant deklariert werden.

```
1  int i = 5;
2  int j = 6;
3
4  const int* p1 = &i;              // Pointer to a const int
5  *p1 = 0;                         // Error
6  p1 = &j;                         // OK
7
8  int* const p2 = &i;              // Const pointer to an int
9  *p2 = 0;                         // OK
10 p2 = &j;                         // Error
11
12 const int* const p3 = &i;        // Const pointer to a const int
13 *p3 = 0;                         // Error
14 p3 = &j;                         // Error
```

Konstante Klassenfunktionen

- Von einer konstanten Instanz der Klasse A können nur konstante Klassenfunktionen von A aufgerufen werden.
- In konstanten Klassenfunktionen von A können wieder nur konstante Klassenfunktionen von A aufgerufen werden.

```
1  struct Vector
2  {
3    double x, y;
4    double len() const;
5    double lensq() const;
6  };
7  double Vector::len() const
8  {
9    return sqrt(lensq());  // lensq() must be const!
10 }
11 double Vector::lensq() const
12 {
13   return x*x + y*y;
14 }
15 void test(const Vector& v)  // v cannot be altered
16 {
17   cout << "Length: " << v.len() << endl;  // len() must be const!
18 }
```

Allgemeines zu const

- Konstante Variablen müssen bei der Erzeugung mit ihrem Wert initialisiert werden.
- Konstantes bleibt unter sich:
  - Konstante Instanzen einer Klasse und konstante Klassenfunktionen einer Klasse rufen nur konstante Klassenfunktionen dieser Klasse auf.
  - Konstante Variablen können nur per konstante Referenz, konstante Pointer oder Kopie an Funktionen übergeben werden.

Wozu const verwenden?

- Anstatt eine konkrete Zahl immer und immer wieder anzugeben ist es deutlich besser eine Konstante zu definieren.
  - Lesbarer, wartbarer, weniger fehleranfällig.
  - Konstanten müssen nur an einer Stelle verändert werden, konkrete Zahlen an vielen Stellen.

```
1 int array[100];
2 for(i=0; i<100; i++)
3   aray[i] = 0;
4 for(i=0; i<100; i++)
5   array[i] += 1;
```

- Das Deklarieren von Variablen als `const` schützt einem vor einem versehentlichen Ändern der Variable.
  - Besonders nützlich bei Pointer!
- Wenn Variablen, die nicht geändert werden, als konstant deklariert werden, dann kann der Compiler Optimierungen zur compile time durchführen.
- Das gezielte verwenden von `const` zwingt einem zu einem bessern Klassen-Design. („Welche Methoden sollen konstant sein?“)

Kapitel

Copy constructor & assignment operator

Copy constructor

- Der copy constructor einer Klasse `A` ist ein Konstruktor, welcher ein Argument vom Typ `const A &` übernimmt.

```
1  class Vector
2  {
3    public:
4      double x, y;
5      Vector();                          // Default constructor
6      Vector(const Vector& cpy);         // Copy constructor
7  };
8
9  Vector::Vector(const Vector& cpy) :
10   x(cpy.x),
11   y(cpy.y)
12 {
13 }
14
15 void test()
16 {
17   Vector a;
18   Vector b(a);  // Create a copy b of a
19 }
```

Impliziter Aufruf des copy constructors

- Der copy constructor wird vom Compiler automatisch immer dann aufgerufen, wenn ein Objekt als Kopie eines anderen Objekts erstellt wird:
  - Call per value bei Funktionsaufruf
  - Instantiierung mit Zuweisung.

```
1  void func(Vector w)
2  {
3    // We got a copy w of u!
4  }
5
6  void test()
7  {
8    Vector u;
9    Vector v = u;   // creation of v by copying u: copy constructor
10   func(u);        // call by value: copy constructor
11 }
```

Notwendigkeit eines copy constructors

- Ist kein copy constructor definiert, so verwendet der C++ compiler einen *impliziten copy constructor*:
  - Dieser kopiert einfach alle member variablen.
  - Meistens ist das ausreichend, manchmal aber fatal!

```
1  class IntArray
2  {
3    private:
4      int size;
5      int* mem;
6    public:
7      IntArray(int size);      // Calls mem = new int[size]
8      ~IntArray();             // Calls delete[] mem
9  };
10
11 void test()
12 {
13   IntArray a(10);
14   IntArray b = a;            // Calls implicit copy constructor
15 } // Error: double free of a.mem when a and b are destructed
```

Assignment operator

- Der Zuweisungsoperator = kann speziell definiert („überladen“) werden:

```
1  class Vector
2  {
3    public:
4      double x, y;
5    public:
6      Vector& operator=(const Vector& cpy);
7  };
8  Vector& Vector::operator=(const Vector& cpy)
9  {
10   x = cpy.x;
11   y = cpy.y;
12   return *this;
13 }
14 void test()
15 {
16   Vector x;
17   Vector y = x;   // copy constructor
18   Vector z;
19   z = x;          // assignment operator
20 }
```

Notwendigkeit eines assignment operators

- Ist kein assignment operator definiert, so verwendet der C++ compiler einen *impliziten assignment operator*:
  - Dieser kopiert einfach alle member variablen.
  - Meistens ist das ausreichend, manchmal aber fatal!

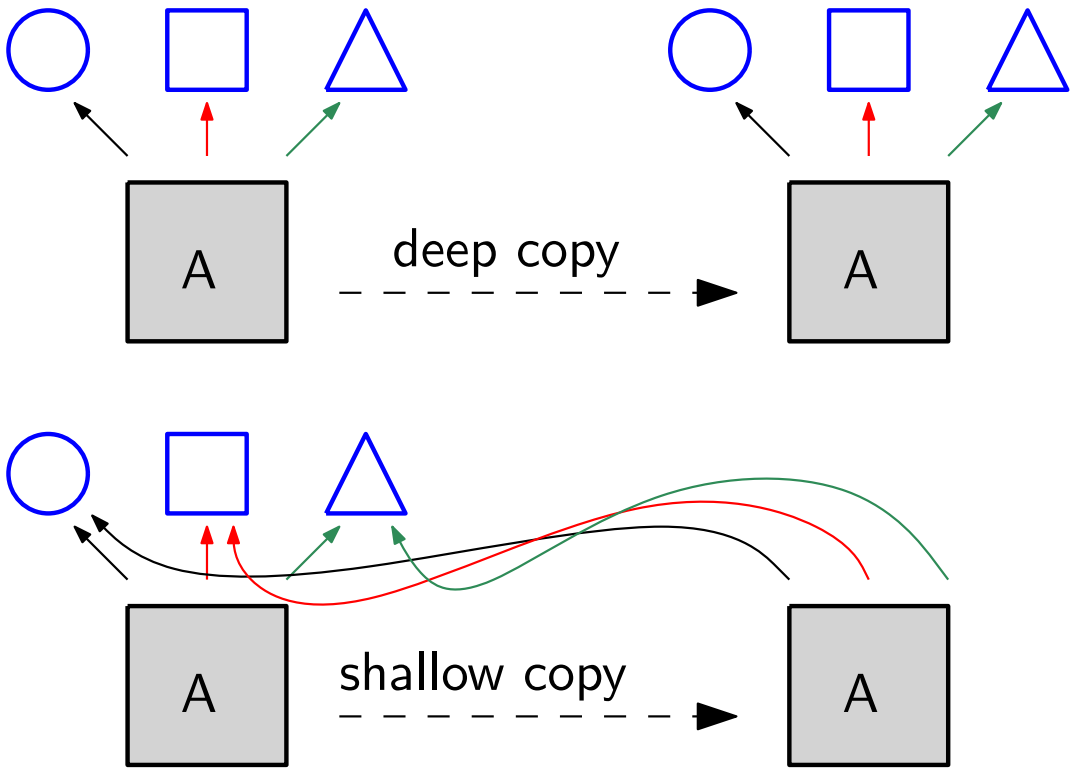
```
1  class IntArray
2  {
3    private:
4      int size;
5      int* mem;
6    public:
7      IntArray(int size);      // Calls mem = new int[size]
8      ~IntArray();             // Calls delete[] mem
9  };
10
11 void test()
12 {
13   IntArray a(10);
14   IntArray b;
15   b = a;              // Calls implicit assignment operator
16 } // Error: double free of a.mem when a and b are destructed
```

Zusammenfassung

- Impliziter copy constructor und impliziter assignment operator werden vom C++ compiler automatisch zur Verfügung gestellt.
- Aber: verwaltet eine Klasse dynamischen Speicher (`new`, `delete`) so **muss** der copy constructor und der assignment operator unbedingt definiert werden!

Deep copy vs. shallow copy

- Eine Instanz einer Klasse `A` verweise auf verschiedene Objekte.
- Wir wollen von dieser Instanz eine Kopie anfertigen. Kopieren wir die Objekte, auf die verwiesen wird, ebenfalls?
  1. **deep copy**: ja
  2. **shallow copy**: nein



Beispiel: shallow copy

```
1  class Manufacturer
2  {
3    std::string name;
4  };
5  class Car
6  {
7    Manufacturer* manufacturer;
8  };
9  void test()
10 {
11   Car c1;
12   Car c2 = c1; // Implicit copy constructor: shallow copy
13   Car c3;
14   c3 = c1;     // Implicit assignment operator: shallow copy
15 }
```

- Copy constructor und assignment operator kopieren einfach den Pointer.
- Wenn ein `Car` dupliziert wird, dann wird `Manufacturer` nicht dupliziert: `c1`, `c2`, `c3` zeigen alle auf den gleichen `Manufacturer`.
- Der Destruktor von `Car` darf `manufacturer` **nicht** freigeben!

Beispiel: deep copy

```
1  class IntArray
2  {
3      const unsigned size;
4      int* mem;
5    public:
6      IntArray(unsigned size);
7      IntArray(const IntArray& cpy);
8      IntArray& operator=(const IntArray& cpy);
9  };
10 IntArray& IntArray::operator=(const IntArray& cpy)
11 {
12   if( mem != 0 )
13     delete[] mem;
14   size = cpy.size;
15   mem = new int[size];
16   for(unsigned i=0; i<size; i++)
17     mem[i] = cpy.mem[i];
18   return *this;
19 }
20 IntArray::IntArray(const IntArray& cpy) :
21   mem(0)
22 {
23   *this = cpy;   // Call assignment operator
24 }
```

Beispiel: deep copy

- Copy constructor und assignment operator kopieren des gesamte Array.

```
1
2  IntArray a1(2);
3  a1.mem[0] = 1;
4  a1.mem[1] = 8;
5
6  IntArray a2 = a1;
7  a2.mem[0] = 2;
8
9  IntArray a3(2);
10 a3 = a1;
11 a3.mem[0] = 3;
12
13 // a1: 1 8
14 // a2: 2 8
15 // a3: 3 8
```