C++ Kurs

Teil 6

Stefan Westerfeld <stefan@space.twc.de>

Eine Primzahltabelle

- Für einige Algorithmen will man
  - prüfen, ob eine Zahl eine Primzahl ist
  - eine Primzahl in der Nähe einer vorgegebenen Zahl finden

  => um das effizient zu gestalten, kann es sinnvoll sein, eine Tabelle im benötigten Bereich vorrätig zu halten
- Dazu implementieren wir eine Klasse:

```
class PrimeTable;
```

mit der Aufgabe, die Primzahlen im Bereich [0..maxIndex] vorrätig zu halten

Öffentliche Schnittstelle

- Wir definieren zunächst, wie die Klasse von aussen zu benutzen sein wird:

```
class PrimeTable
{
public:
    PrimeTable (unsigned int maxIndex);

    bool isPrime (unsigned int p) const;
    unsigned int findPrimeNear (unsigned int i) const;
    unsigned int maxIndex() const;
};
```

Was bedeutet „const“ ?

- Alle Methoden, die keine Daten des Objekts ändern, markiert man mit **const**

```
class PrimeTable
{
public:
    PrimeTable (unsigned int maxIndex);

    bool isPrime (unsigned int p) const;
    unsigned int findPrimeNear (unsigned int i) const;
    unsigned int maxIndex() const;
};
```

Konstruktor / Destruktor

- Ein Konstruktor dient der Initialisierung des Objekts; er trägt den Namen der Klasse

```
class PrimeTable
{
public:
    PrimeTable (unsigned int maxIndex);

    bool isPrime (unsigned int p) const;
    unsigned int findPrimeNear (unsigned int i) const;
    unsigned int maxIndex() const;
};
```

- Der Destruktor zur Freigabe des Objekts müsste in unserem Fall **~PrimeTable** heissen, aber der von C++ automatisch erzeugte Destruktor ist „gut genug“

maxIndex

- Um den maximal erlaubten Wert für Anfragen zu speichern, legen wir eine Membervariable an:

```
class PrimeTable
{
private:
    unsigned int m_maxIndex;
public:
    PrimeTable (unsigned int maxIndex);

    bool isPrime (unsigned int p) const;
    unsigned int findPrimeNear (unsigned int i) const;
    unsigned int maxIndex() const;
};
```

- Als Konvention tragen Membervariablen den Prefix **m_** am Anfang des Namens

maxIndex

- Im Konstruktor initialisieren wir m_maxIndex mit dem übergebenen Wert

```
PrimeTable::PrimeTable (unsigned int maxIndex) :
    m_maxIndex (maxIndex)
{
    assert (maxIndex >= 2);
}
```

- Diese Initialisierung bedeutet (ungefähr)

```
PrimeTable::PrimeTable (unsigned int maxIndex)
{
    m_maxIndex = maxIndex;
    assert (maxIndex >= 2);
}
```

maxIndex

- Die Methode zum Abfragen des Wertes sieht dann so aus

```
class PrimeTable
{
private:
    unsigned int m_maxIndex;
public:
    PrimeTable (unsigned int maxIndex);

    // ...
    unsigned int maxIndex() const;
};
```

```
unsigned int PrimeTable::maxIndex() const
{
    return m_maxIndex;
}
```

Die eigentliche Tabelle

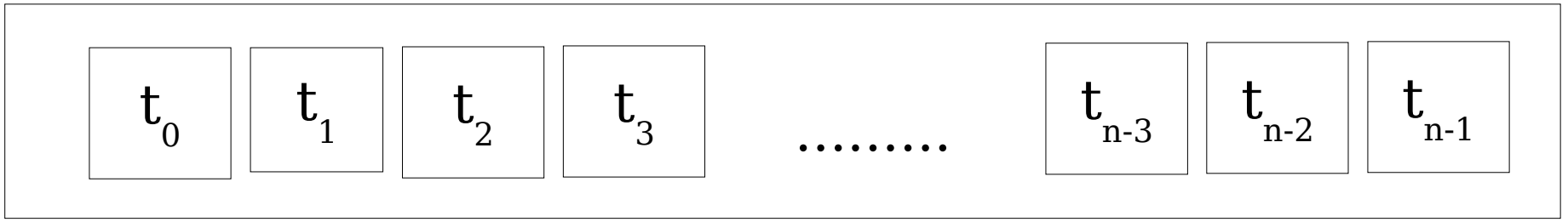
- Wir legen eine zweite Membervariable an, um für jede Zahl zu hinterlegen, ob es sich um eine Primzahl handelt

```
class PrimeTable
{
private:
    std::vector<bool> m_primes;
    unsigned int      m_maxIndex;

public:
    PrimeTable (unsigned int maxIndex);

    bool isPrime (unsigned int p) const;
    unsigned int findPrimeNear (unsigned int i) const;
    unsigned int maxIndex() const;
};
```

Was ist ein std::vector<T>?



- Enthält n Objekte vom Typ T, mit den Indizes 0..n-1:

```
#include <vector>
...
std::vector<int> iv (3); // Vektor der Länge 3 anlegen

iv[1] = 42;     // setze mittleres Element auf 42
printf („Vektor hat %d Elemente\n“, iv.size());
iv.push_back (12);  // jetzt 4 Elemente: 0 42 0 12
iv.resize (6);  // jetzt 6 Elemente: 0 42 0 12 0 0
for (unsigned int i = 0; i < iv.size(); i++)
    printf („iv[%d] = %d\n“, i, iv[i]);
```

Die eigentliche Tabelle

- Im Konstruktor initialisieren wir m_primes zunächst so, daß wir annehmen, alle Zahlen außer 0 und 1 seinen Primzahlen

```
PrimeTable::PrimeTable (unsigned int maxIndex) :
    m_primes (maxIndex + 1, true),
    m_maxIndex (maxIndex)
{
    assert (maxIndex >= 2);


    m_primes[0] = false;
    m_primes[1] = false;


    // Sieve of Eratosthens
    [...]
}
```

Die eigentliche Tabelle

- Durch das „Sieb des Eratosthenes“ wird der Rest initialisiert

```
// Sieve of Eratosthens
for (unsigned int p = 2; p <= maxIndex; p++)
{
    if (m_primes[p])        // p is a prime number
    {
        for (unsigned int no_p = p * 2;
             no_p <= maxIndex; no_p += p)
        {
            // no_p can be divided by p
            // no_p => is not a prime number
            m_primes[no_p] = false;
        }
    }
}
```

Die „isPrime“ Methode

- Die Methode zum Abfragen des Wertes sieht dann so aus

```
class PrimeTable
{
private:
    std::vector<bool> m_primes;
    unsigned int      m_maxIndex;
public:
    [...]
    bool isPrime (unsigned int p) const;
};
```

```
bool PrimeTable::isPrime (unsigned int p) const
{
    assert (p <= m_maxIndex);

    return m_primes[p];
}
```

Wozu assert()?

- Mit assert kann der Programmierer an einer bestimmten Stelle sagen: „hier garantiere ich (erwarte ich), daß eine bestimmte Bedingung erfüllt ist“

```
#include <assert.h>
[...]
bool PrimeTable::isPrime (unsigned int p) const
{
    assert (p <= m_maxIndex);
    return m_primes[p];
}
```

- Wenn das Programm läuft, werden solche Zusicherungen dann geprüft, und wenn sie fehlschlagen, wird das Programm beendet.
- Assertions können verwendet werden um Fehlern vorzubeugen, in Form von defensivem Programmieren oder Programmieren nach einem Vertragsmodell

Die „findPrimeNear“ Methode

- Methode zur Suche einer nahegelegenen Primzahl:

```
unsigned int PrimeTable::findPrimeNear (unsigned int i) const
{
    assert (i <= m_maxIndex);

    if (i < 2)
        i = 2;

    while (!m_primes[i])
        i--;

    return i;
}
```

Ein main()

```
int main()
{
    PrimeTable primeTable (100);
    srand (time (NULL));

    for (unsigned int i = 0; i <= primeTable.maxIndex(); i++)
    {
        if (primeTable.isPrime (i))
            printf ("%d\n", i);
    }
    for (unsigned int i = 0; i < 10; i++)
    {
        unsigned int z = rand() % (primeTable.maxIndex() + 1);
        printf ("Eine Primzahl nahe %d ist %d\n", z,
                primeTable.findPrimeNear (z));
    }
    return 0;
}
```

Aufgabe 1 (Verbesserungen an PrimeTable)

(a) Es gilt: sei n keine Primzahl, dann gibt es eine Faktorisierung

$$n = a * b$$

daraus folgt, dass man bei Primzahltests nur Faktoren <= Wurzel n berücksichtigen muss. Verwende dieses Wissen, um den Sieb-Algorithmus zu optimieren

(b) PrimeTable::findPrimeNear() findet unter Umständen Primzahlen, die weiter von der Vorgabe entfernt sind, als sie es sein müssten, da nur nach unten gesucht wird – behebe dies

Aufgabe 2 (Reimplementation von PrimeTable)

(a) Schreibe eine neue Version von PrimeTable, die, statt für jede Zahl zu speichern, ob es eine Primzahl ist, einen `std::vector<unsigned int>` verwendet um zu speichern welche Primzehlen es (im Bereich [0..maxIndex]) gibt

Dabei soll die öffentliche Schnittstelle erhalten bleiben, sodaß keine Änderungen an main() nötig sind, um die neue Version zu verwenden.

(b) Welche der beiden verschiedenen Implementationen hat welche Vor- und Nachteile?

Aufgabe 3 (dynamisches Wachstum)

(a) Schreibe eine der beiden Klassen so um, daß – wenn es nötig wird - neue Primzahlen nachberechnet werden. Damit entfällt `maxIndex`. Einige **`const`** müssen entfernt werden.

Beispiel:

```
DynamicPrimeTable primeTable;

if (primeTable.isPrime (10))
{
    // primeTable weiss hier: [2, 3, 5, 7] prim
    printf („10 is prime\n“);
}
if (primeTable.isPrime (11))
{
    // primeTable weiss hier: [2, 3, 5, 7, 11] prim
    printf („11 is prime\n“);
}
```