const References
Classes ++

Informatik für Mathematiker und Physiker

Martin Maag

December 21, 2010

CSElab
Computational Science & Engineering Laboratory
http://www.cse-lab.ethz.ch

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Übersicht

- Referenzen
  - Pointer
  - const Referenzen
- Klassen
  - Konstruktor
  - Destruktor
  - Überladung

Const Referenzen

- int a = 5;
  const int& x = a;
  x = 21; // geht nicht, x ist const!!
  a = 21; // geht, a ist nicht const!!

- Sie sind Referenzen, deren Wert man **nicht** durch die Referenz verändern kann
  Wert kann jedoch durch andere Referenz, die nicht const ist, verändert werden

- Initialisierung: L-Werte und R-Werte

Const Referenzen

- T function(const T parameter)
  {}
- T function(const& T parameter)
- Unterschied zwischen const T und const &T als Funktionsargument?
  - const T: Das ganze Objekt wird kopiert
  - const &T: Adresse vom Objekt wird kopiert

Typedef

- typedef Typ Synonym;
- typedef:
  - Mit typedef erzeugt man einen Synonym für einen Typ
    typedef float real;
  - macht das Programm lesbarer:
    typedef std::pair<double, double> doublepair;
  - Implementation einfach zu ändern
    typedef ~~float~~ double real;
    // alle Variablen vom Typ real sind jetzt genauer

Klassen

- Bisher: Prozedurales Programmieren
  - Datenstrukturen
  - Funktionen, die auf den Datenstrukturen arbeiten
    - setzt Wissen voraus, wie die Datenstruktur aussieht

```
struct abc
{
    int a;
    int b;
    int c;
};

void printabc(abc in)
{
    std::cout<<in.a<<" "<<in.b<<" "<<in.c<<"\n";
}
```

Klassen

- Neu: Objekt-Orientiertes Programmieren
  - Datenstrukturen inkl. Funktionen
    - Funktionen arbeiten auf den Datenstrukturen, in denen sie definiert sind

```
class ABC
{
    int a;
    int b;
    int c;
public:
    void print()
    {
        std::cout<<a<<" "<<b<<" "<<c<<"\n";
    }
}
```

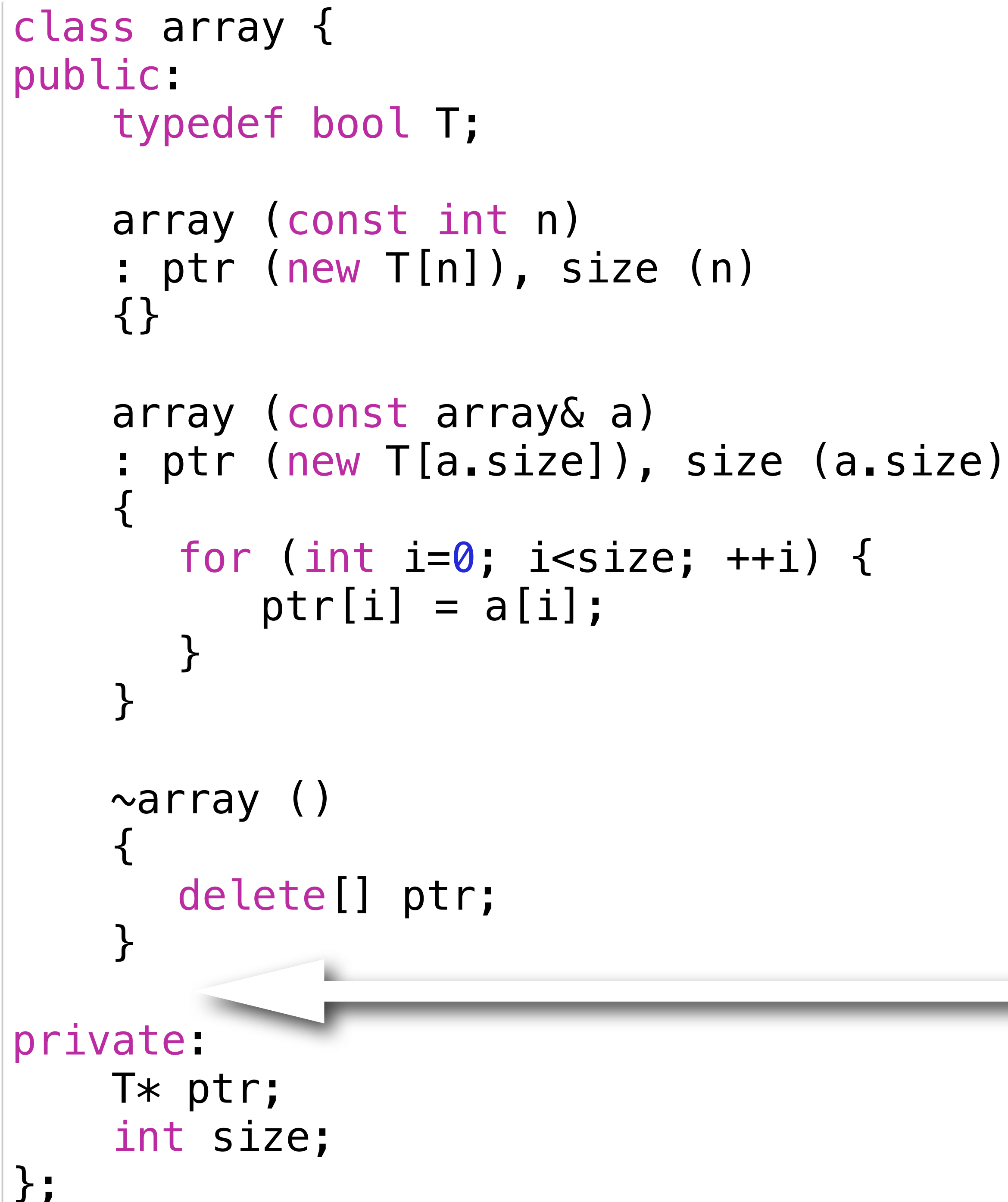
Wozu das ganze?

- Verkapselung
  - Datenstrukturen werden zusammen mit den dazugehörigen Funktionen verpackt und vertrieben
    - Änderungen in der Datenstruktur müssen keine Änderung der dazugehörigen Funktionssignaturen bedeuten
- Data Hiding
  - Nur nötigste Informationen werden dem Benutzer mitgeteilt
    - Fehler durch fehlerhafte Zugriffe auf Daten können so verhindert werden

Details

- C++
  - “struct” und “class” sind eigentlich dasselbe
  - einziger Unterschied:
    - struct: ohne Keyword sind alle Felder “public”
    - class: ohne Keyword sind alle Felder “private”

Klassen



```
class array {
public:
    typedef bool T;

    array (const int n)
    : ptr (new T[n]), size (n)
    {}

    array (const array& a)
    : ptr (new T[a.size]), size (a.size)
    {
       for (int i=0; i<size; ++i) {
          ptr[i] = a[i];
       }
    }

    ~array ()
    {
       delete[] ptr;
    }

private:
    T* ptr;
    int size;
};
```

```
const T& operator[] (const int i) const
{
    return ptr[i];
}

T& operator[] (const int i) const
{
    return ptr[i];
}

array& operator= (const array& a)
{
    // avoid self-assignments
    if (this != &a) {
       // free old memory, and get new memory
       // of appropriate size
       delete[] ptr;
       ptr = new T[size = a.size];
       // copy the elements of a into *this
       for (int i = 0; i < size; ++i)
          ptr[i] = a[i];
    }
    return *this;
}
```

Konstruktor

- Spezielle Mitgliedsfunktionen für Initialisierung
  - Name = Name der Klasse, kein Rückgabetyp

```
rational (const int numerator, const int denominator)
  : n (numerator), d (denominator)
{
  assert (d != 0);
}
```

  - Initialisierungs-Listen: gleicher Effekt wie wenn man Member-Variablen bei Deklaration zuweisen würde
    z.B.: “int n = 1;” wird zu “n(1)” und “int &r = n;” zu “r(n)”
  - Verwendung: “rational x(1,2);” oder “rational x = rational(1,2);”
  - Default Konstruktor bei Aufruf “rational x;”

```
rational ()
  : n (0), d (1)
{}
```

Destruktor

- Wenn eine Klasse ein dynamisches Feld enthält...
- array k(n); //array mit Länge n
- delete [] k; //geht nicht...
- Destruktor: automatisch aufgerufene Funktion, wenn das Objekt "**out of scope**" geht (Speicherdauer endet)
    - Explizite delete-Operationen nicht notwendig

Operator Überladung

- Es kann zwei Überladungen des gleichen Operators geben.
- Die Deklaration von diesen Überladungen müssen unterschiedlich sein:

```
// --- Index operator (const) -------------------------
// PRE: the array has length at least i
// POST: return value is i-th element (as a const ref.)
const T& operator[] (const int i) const {
    return ptr[i];
}

// --- Index operator (non-const) ---------------------
// PRE: the array has length at least i
// POST: return value is i-th element (as a reference)
T& operator[] (const int i) {
   return ptr[i];
}
```

Wieso 2mal?

```
array a(10);
a[1] = 1;
const array b(a);
std::cout << a[1]
          << ", "
          << b[1]
          << '\n';
```

Copy constructor

- shallow copy Problem:
  - “array a = b;” oder “array a(b);”
    - Reminder: array enthält dyn. Speicher “T* ptr”
  - Nur der Zeiger wird kopiert, nicht die Elemente!

- Copy Konstruktor: automatisch aufgerufene Funktion, wenn ein Objekt durch ein anderes Objekt **initialisiert** wird

```
class T {
    T(const T& other)
    {
        ...
    }
};
```

Zuweisungsoperator

- **= Operator** wird überladen -> “a = b”
- ähnlich wie Copy Konstruktor
- zwei Unterschiede:
  - Freigabe des Speichers für den alten Wert
  - Vermeidung von Selbstzuweisungen

```
class T {
    ...
    T& operator=(const T& other)
    {
       ...
       return *this;
    }
};
```

Auf einem Blick

```
struct Foo {
    // Standard constructor
    Foo() {
        std::cout << "FOO\n";
    }
    // Convert int to Foo
    Foo(int i) {
        std::cout << "iii" << i << "\n";
    }
    // Copy constructor
    Foo(const Foo& other) {
        std::cout << "FOO copy\n";
    }
    // Destructor
    ~Foo() {
        std::cout << "FOO bar\n";
    }
    // Assignment operator
    Foo& operator= (const Foo& other) {
        std::cout << "FOO assign\n";
        return *this;
    }
};
```

```
struct Bar {
    Foo f;
    Bar(): f(0) { }
    Bar(int i) {
        f = i;
    }
};

int main() {
    Bar b;
    Bar bi(1);

    return 0;
}
```

Compiles?

Output?

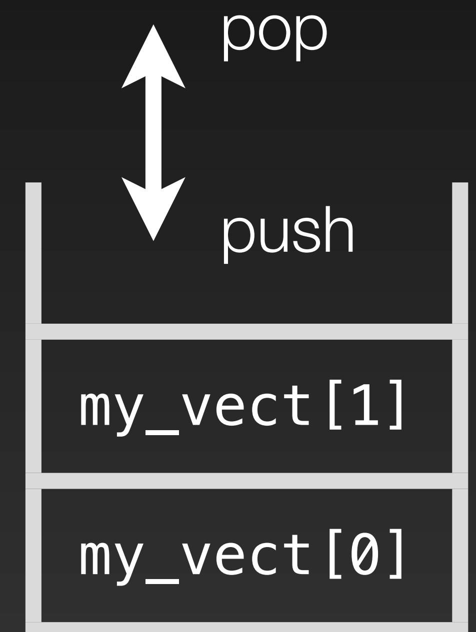
Faustregel:
Wenn 1 von den 3 da,
dann alle 3 da...

Vektoren

- Felder mit dyn. Grösse (Standardbibliothek)
- Header: #include <vector>
- Benutzen:

```
std::vector<double> my_vect;
my_vect.push_back(5.4);
my_vect[0]--;
double a = my_vect.back();
my_vect.pop_back();
```

Stack (Stapel):
push auf Stapel
pop vom Stapel



- Dyn. vergrössert bei “push_back”, nicht (!) bei “my_vect[182] = 0”

Beispiel - Klasse Matrix

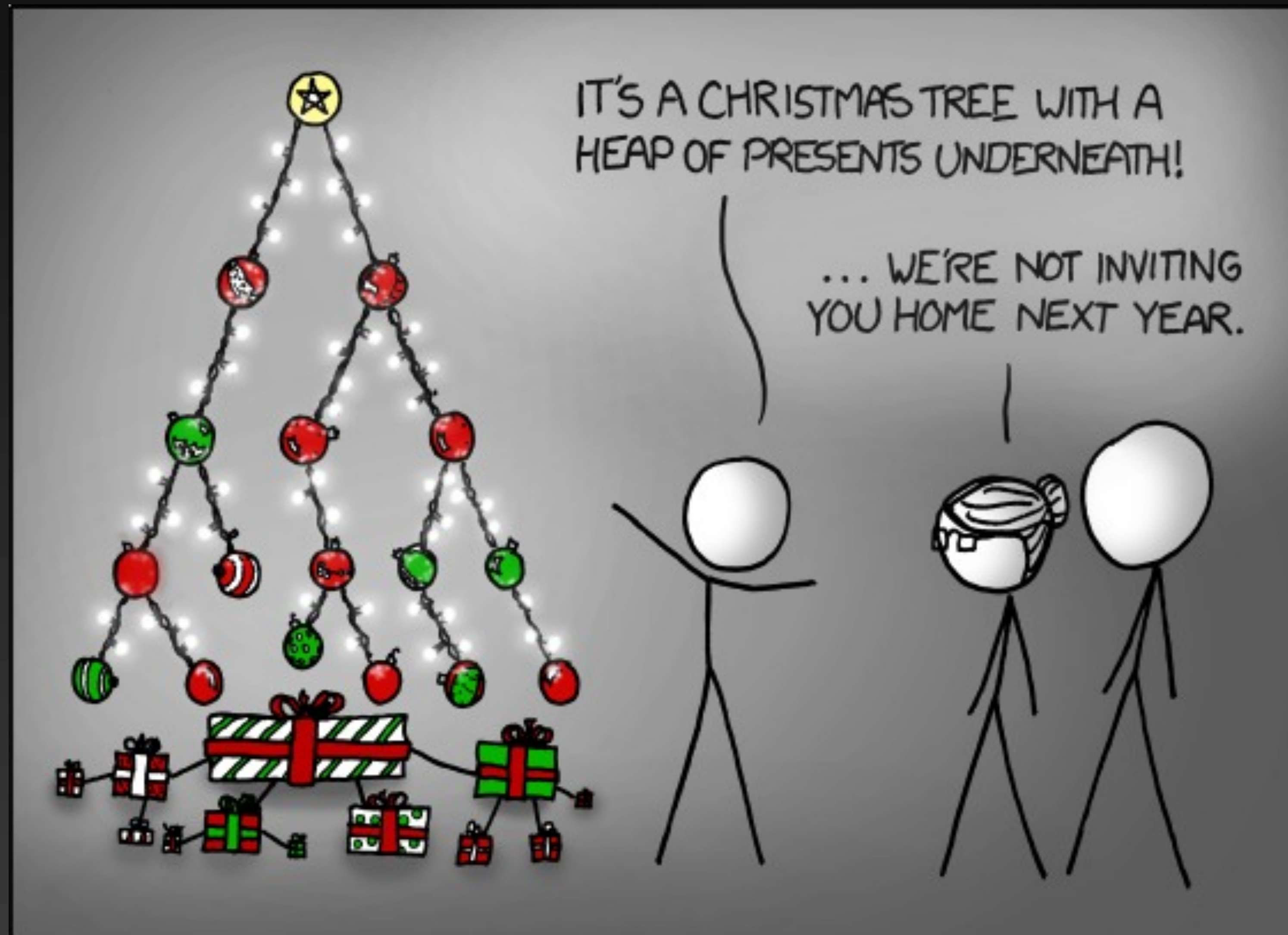
- Die Klasse soll Matrizen variabler Länge speichern

```
// initialize
Matrix2D m(10,20);
const Matrix2D mc(x);
// assign
m = mc;

// get sizes
unsigned int nx = xc.getSizeX();
unsigned int ny = xc.getSizeY();

// access elements
for (int j = 0; j < ny; ++j) {
    for (int i = 0; i < nx; ++i) {
        // typedef for element type
        Matrix2D::ElementType elem;
        // const access
        elem = xc(i, j);
        // non-const assign
        x(i, j) = elem;
    }
}
```

Requirements?

- Constructor “Matrix2D(nx, ny)”
- Copy constructor
- Assignment operator
- Destructor
- “unsigned int getSizeX() const”
- “unsigned int getSizeY() const”
- “typedef ... ElementType”
- “ElementType operator() (i,j) const”
- “ElementType& operator() (i,j)”

Das wars!



http://xkcd.com/835/