C++ Kurs Teil 2

- Zeiger
- Felder
  - zweidimensionale Felder
- Exceptions
  - traditionelle Fehlerbehandlung
  - throw, try und catch
  - Fehlerklassen und -klassenhierarchien
  - "resource allocation is initialization"
- Extreme Programming (XP)
  - Beispiel: Vector3D Klasse

Zeiger

- Direkter Zugriff auf den Speicher des Computers.
- für schnelle maschinennahe Programmierung
- keine Überprüfung auf korrekte Handhabung
- leider teilweise unverzichtbar.
- sehr "gewöhnungsbedürftige" Syntax

& als "Adress-Operator"

```
float a = 42;
std::cout << "Variable a steht im Speicher ab Adresse " << &a << std::endl;
```

Ausgabe

```
:Variable a steht im Speicher ab Adresse 0xbffff588
```

Zeiger

und ihre tiefen Abgründe...

```
int   a = 42;
int* p;
p = &a;

std::cout << "an der Adresse " << p
          << " steht (interpretiert als integer) "
          << *p << std::endl;

float* q;
q = p;
q = reinterpret_cast<float*>( p);

std::cout << "an der Adresse " << q
          << " steht (interpretiert als float) "
          << *q << std::endl;
```

p enthält eine Adresse. Der Speicherinhalt dort soll als **Integer** interpretiert werden

p enthält jetzt die Adresse, wo die Variable a im Speicher liegt.

q enthält eine Adresse. Der Speicherinhalt dort soll als **float** interpretiert werden

hier meckert der Compiler – zurecht!

so muß er es fressen...

```
an der Adresse 0xbffff578 steht (interpretiert als integer) 42
an der Adresse 0xbffff578 steht (interpretiert als float) 5.88545e-44
```

Zeiger

die Bedeutungen von * und & in C++

	*	&

als **Modifier**	`int* p;`
	`void function( int* p);`
	Deklaration eines Zeigers
		`int& a = b;`
		`void function( int& a);`
		Deklaration einer Referenz

als **unärer** Operator	`std::cout << *p;` **Dereferenzierung:** Der Speicherinhalt ab Adresse p wird dem Zeigertyp entsprechend interpretiert und ausgegeben.	`std::cout << &a;` **Adress-Operator:** Gibt die Adresse aus, wo die Variable a im Speicher gespeichert ist.
als **binärer** Operator	`std::cout << a * b;` **Muliplikation** von a und b	`std::cout << (a & b);` **Bitweises logisches "Und"** von a und b

Zeiger

```
int a = 42;
int* b = &a;
*b = 27;
std::cout << a << std::endl;
```

Ergebnis?

```
int a = 42;
int b = 137;
int* c = &a;
int* d = &b;
c = d;
std::cout << a << std::endl;
```

Ergebnis?

```
int a = 42;
int b = 137;
int* c = &a;
int** d = &c;
std::cout << *d << std::endl;
```

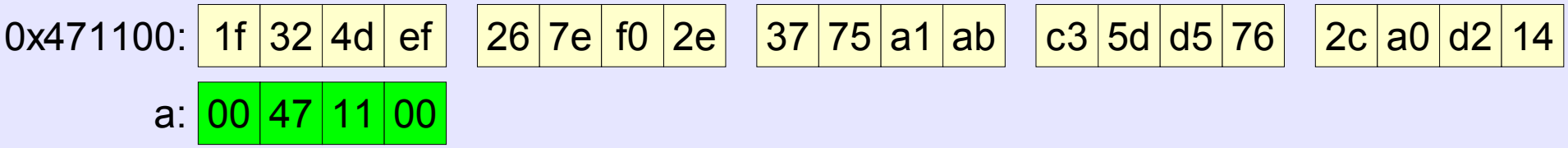
Ergebnis?

Zeiger und Felder

```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;           // schreibe 42 in das erste Element

a[4] = 65535;        // schreibe 65535 in das letzte Element

int* b = a + 1;      // Rechenoperationen bei Zeigern berücksichtigen
                     // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;           // schreibe 1234 in das zweite Element des Feldes

delete[] a;          // gib den Speicherbereich wieder frei.
```

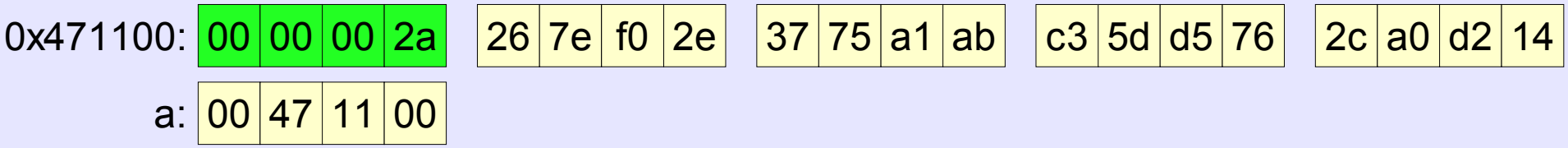
Zeiger und Felder

```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;               // schreibe 42 in das erste Element

a[4] = 65535;            // schreibe 65535 in das letzte Element

int* b = a + 1;          // Rechenoperationen bei Zeigern berücksichtigen
                         // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;               // schreibe 1234 in das zweite Element des Feldes

delete[] a;              // gib den Speicherbereich wieder frei.
```



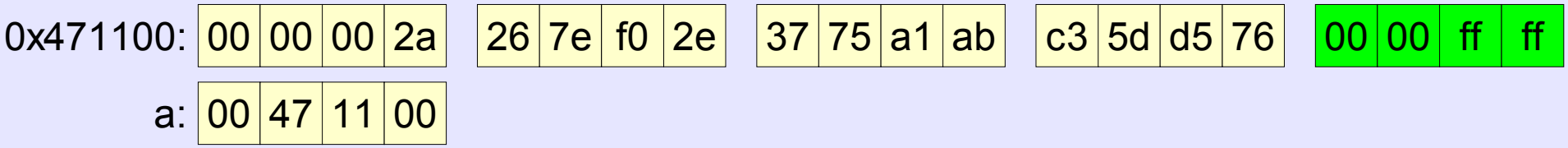
Zeiger und Felder

```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;               // schreibe 42 in das erste Element

a[4] = 65535;            // schreibe 65535 in das letzte Element

int* b = a + 1;          // Rechenoperationen bei Zeigern berücksichtigen
                         // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;               // schreibe 1234 in das zweite Element des Feldes

delete[] a;              // gib den Speicherbereich wieder frei.
```



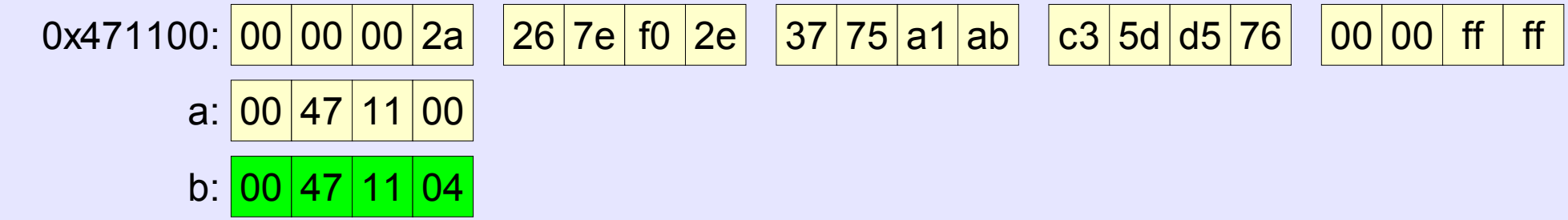
Zeiger und Felder

```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;               // schreibe 42 in das erste Element

a[4] = 65535;            // schreibe 65535 in das letzte Element

int* b = a + 1;          // Rechenoperationen bei Zeigern berücksichtigen
                         // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;               // schreibe 1234 in das zweite Element des Feldes

delete[] a;              // gib den Speicherbereich wieder frei.
```



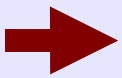
Zeiger und Felder

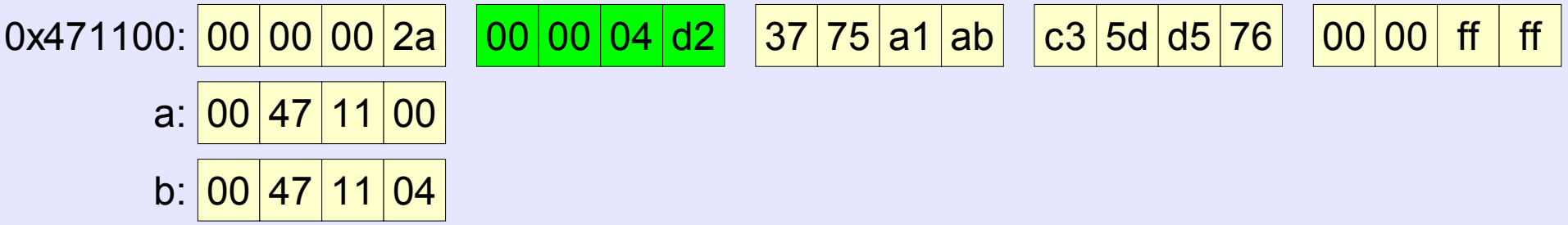
```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;           // schreibe 42 in das erste Element

a[4] = 65535;        // schreibe 65535 in das letzte Element

int* b = a + 1;      // Rechenoperationen bei Zeigern berücksichtigen
                     // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;           // schreibe 1234 in das zweite Element des Feldes

delete[] a;          // gib den Speicherbereich wieder frei.
```



Zeiger und Felder

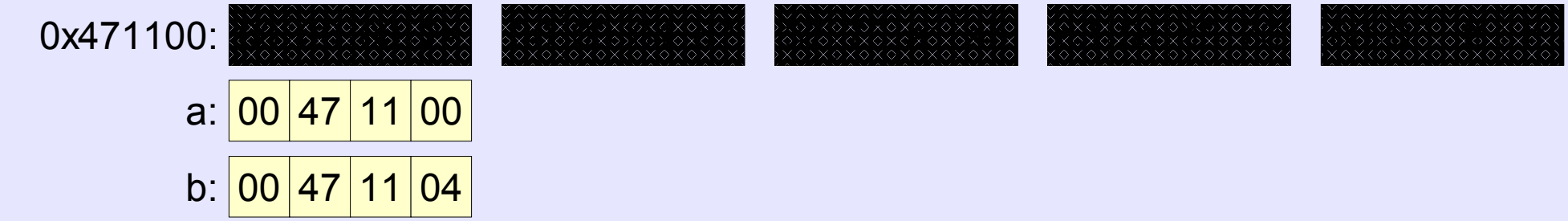
```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;           // schreibe 42 in das erste Element

a[4] = 65535;        // schreibe 65535 in das letzte Element

int* b = a + 1;      // Rechenoperationen bei Zeigern berücksichtigen
                     // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;           // schreibe 1234 in das zweite Element des Feldes

delete[] a;          // gib den Speicherbereich wieder frei.
```





Zeiger und Felder

```
int* a = new int[5];     // reserviere einen Speicherbereich, der
                         // 5 integers aufnehmen kann. In a wird die
                         // Anfangsadresse von diesem Bereich gespeichert

a[0] = 42;           // schreibe 42 in das erste Element

a[4] = 65535;        // schreibe 65535 in das letzte Element

int* b = a + 1;      // Rechenoperationen bei Zeigern berücksichtigen
                     // deren Datentyp! Also zeigt b 4 Bytes hinter a

*b = 1234;           // schreibe 1234 in das zweite Element des Feldes

delete[] a;          // gib den Speicherbereich wieder frei.
```



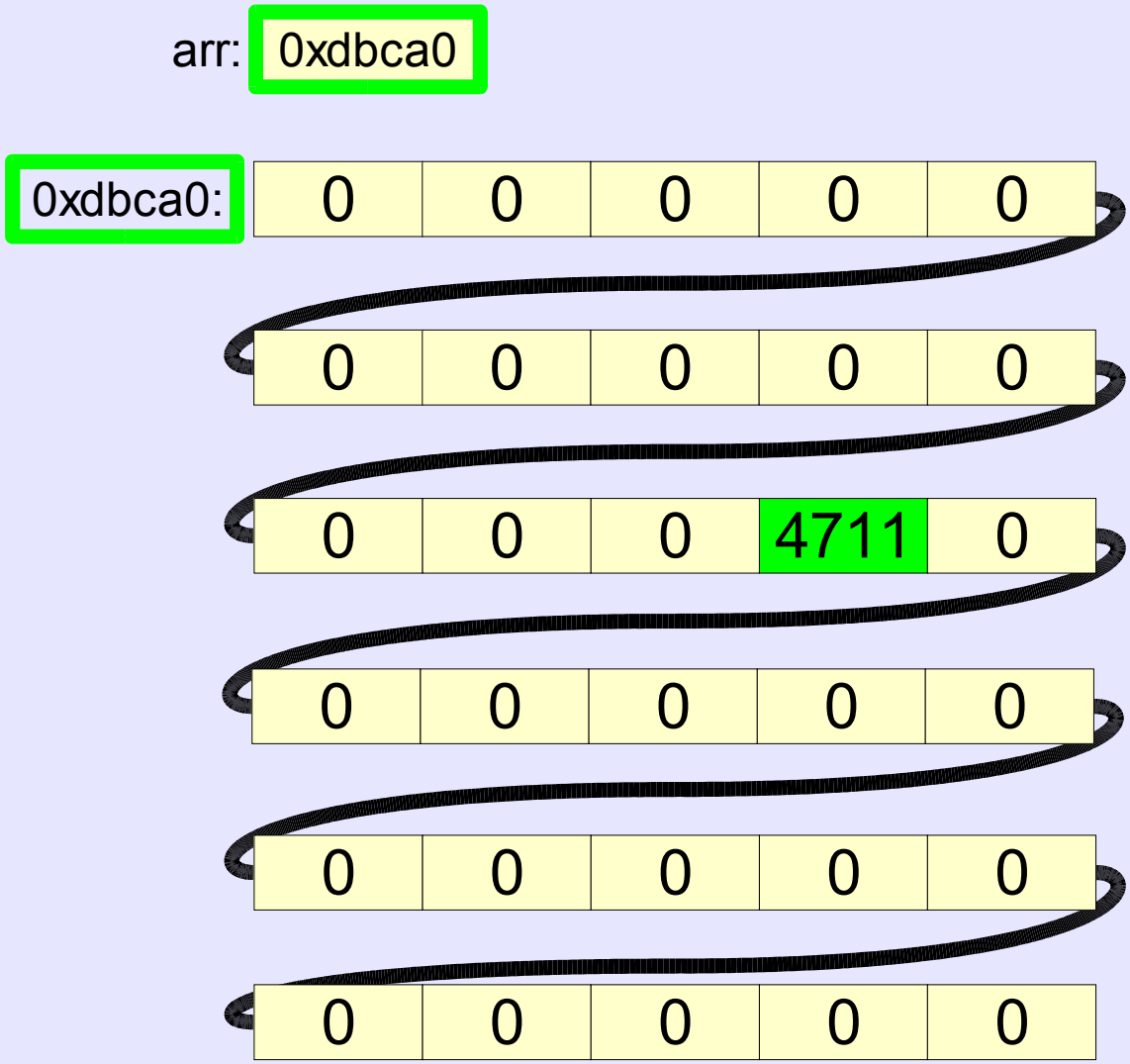
Zeiger und Felder

Ein primitives Feld wird in C / C++ nur durch einen Zeiger auf den Anfang des Speicherbereichs dargestellt. Der Zugriff auf ein Element wird nicht überprüft.

+ Optimale Performance
+ Optimale Speicherplatznutzung
- Der Programmierer muß sich selbst die Größe merken
- Wenn man auf ein illegales Element zugreift, wird einfach dort in den Speicher geschrieben. Wenn man Glück hat, gehört dieser Speicher einem anderen Programm und das eigene Programm wird vom Betriebssystem abgeschossen (**Segmentation Fault**)
+ Philosophie: Auf dieser Basis können beliebig komfortablere aber langsamere Felder (z.B. mit überpüften Zugriffen) entwickelt werden. Umgekehrt wäre das nicht möglich!

Zweidimensionale Felder

Speicherplatzsparende aber langsame Variante

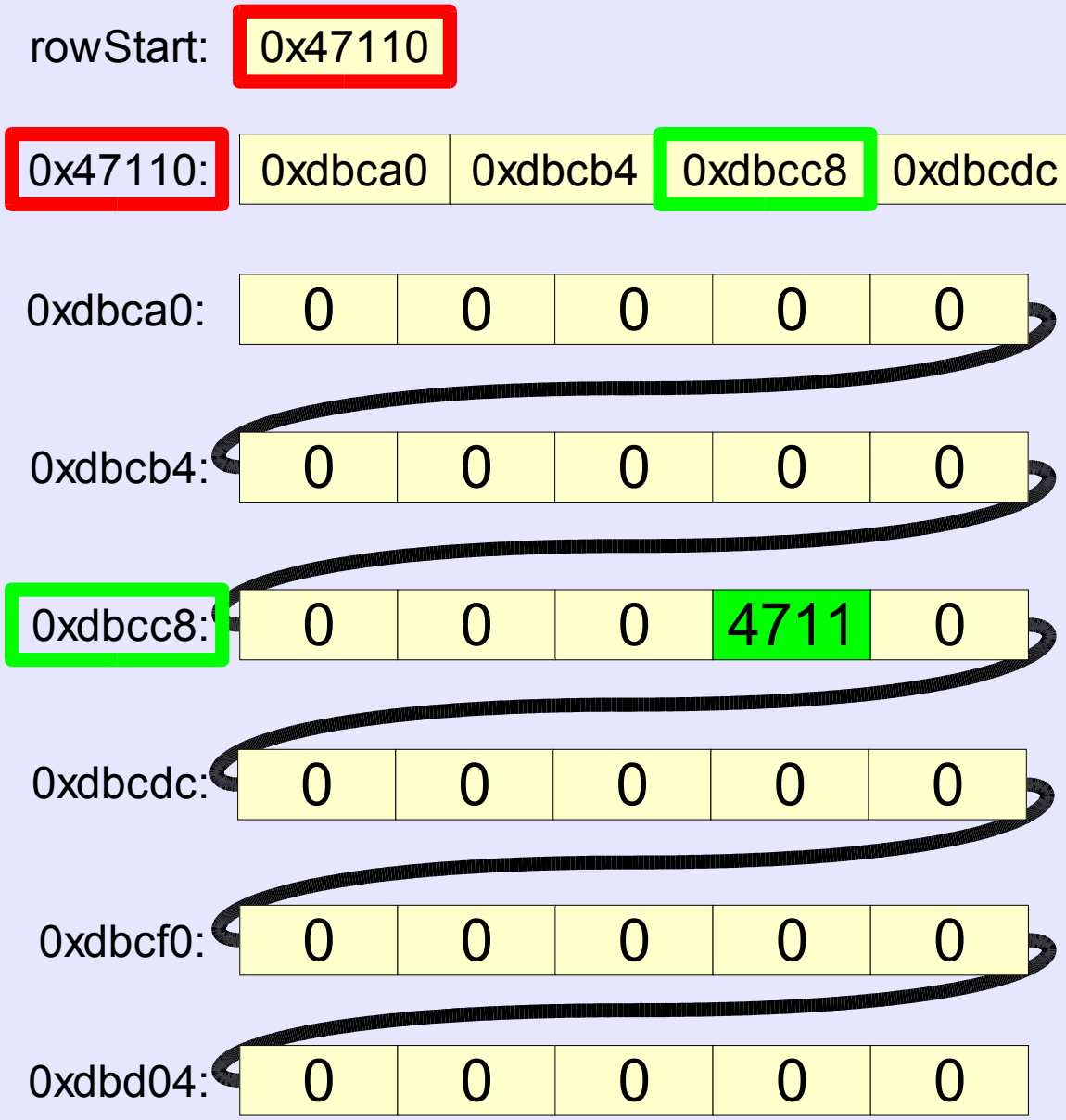



```
/* Speicher für 6x5 Matrix
reservieren */
int* arr = new int[30];


/* Element an Position (2,3)
auf 4711 setzen */
int row = 2;
int col = 3;
arr[row*5 + col] = 4711;
```

solche Feldzugriffe sind teuer, da für jeden Zugriff eine Multiplikation ausgeführt werden muß

Zweidimensionale Felder

schnelle Variante mit etwas Speicher-Overhead




```
/* Speicher für 6x5 Matrix
reservieren */
int* arr = new int[30];
int** rowStart = new int*[6];

// (Hausaufgabe...)

/* Element an Position (2,3)
auf 4711 setzen */
rowStart[2][3] = 4711;
```

statt der Multiplikation wird jetzt in einer Tabelle nachgeschaut.

Exceptions

- Im traditionellen C (und vielen anderen Sprachen) ist es nicht möglich, die Fehlerbehandlung sauber vom eigentlichen Code zu trennen.
- Fehlerbehandlung wird dort typischerweise über verschiedene Rückgabewerte realisiert.

Traditionelle Fehlerbehandlung

```
handle = dc1394_create_handle(0);
if (handle==NULL)
{
  exit( 1);
}
status = dc1394_setup_capture(handle,..., &camera);
if( status != DC1394_SUCCESS)
{
   raw1394_destroy_handle(handle);
   exit(1);
}
status = dc1394_start_iso_transmission
(handle,camera.node);
if( status != DC1394_SUCCESS)
{
   dc1394_release_camera(handle,&camera);
   raw1394_destroy_handle(handle);
   exit(1);
}
status = dc1394_single_capture(handle,&camera);
if( status != DC1394_SUCCESS)
{  ...
```

- Das eigentliche Programm besteht nur aus 4 Zeilen
- Durch die Fehlerbehandlung ist die Struktur des Programmes kaum noch zu erkennen
- Schließen der Firewire-Verbindung erfolgt an etlichen verschiedenen Stellen

Traditionelle Fehlerbehandlung

meistgenutze Alternative: Fehler ignorieren...

```
handle = dc1394_create_handle(0);
dc1394_setup_capture(handle,..., &camera);
dc1394_start_iso_transmission(handle,camera.node);
dc1394_single_capture(handle,&camera);
```

- Da die trad. Fehlerbehandlung zu umständlich ist, wird vielfach der Rückgabewert gar nicht abgefragt. Daraus resultieren dann unerklärliche Abstürze des Programms durch Nutzung von kaputten Datenstrukturen.

Exceptions

- Exceptions erlauben es, das eigentliche Programm und die Fehlerbehandlung sauber zu trennen.
- Wenn in einer Funktion eine exception "geworfen" wird, wird nicht von der Funktion zurückgesprungen, sondern stattdessen ein passendes catch() gesucht.

```
try
{
  handle = dc1394_create_handle(0);
  dc1394_setup_capture(handle,..., &camera);
  dc1394_start_iso_transmission(handle,camera.node);
  dc1394_single_capture(handle,&camera);
}
catch( int errorNumber)
{
  ....
}
```

throw, try und catch

```
void irgendeineFunktion()
{
   ...
   if (a != b) throw 42;
   if (x == y) throw 137;
}
```

- Um eine Exception zu werfen wird der 'throw' Befehl genutzt. Hier wird einfach ein 'int' geworfen
- Die geworfene Exception hangelt sich solange nach oben, bis es ein passendes 'catch' findet. Wenn es keines gibt, wird das Programm abgebrochen.

Fehlerklassen

- throw(xyz) kann als Aufruf der Methode catch() mit dem Parameter xyz aufgefaßt werden, wobei nach Beendigung des catch-Block natürlich nicht zum Punkt des throws zurückgekehrt wird.
- Einen 'int' als Exception zu werfen, ist nur bedingt brauchbar. Besser ist es, die Standard-Fehlerklassen zu benutzen, oder eine eine eigene FehlerKlasse dafür zu erzeugen, die dann auch noch weitere Information vom throw zum catch transportieren kann

Fehlerklassen

Einem try-Block dürfen mehrere catch-Blöcke folgen

```
void myFunction()
{
  ...

  if( file == 0)
  {
    FileOpenError err( fileName);
    throw err;
  }
  ...

  if( format == -1)
  {
    throw WrongFormatError();
  }
  ...
}
```

```
try
{
  ...
  myFunction();
  ...
}
catch( FileOpenError& err)
{
  std::cerr << "Can't open file "
            << err.fileName() << std::endl;
}
catch( WrongFormatError& err)
{
   std::cerr << "File has wrong format\n";
}
catch( ...) // hier stehen wirklich drei Punkte!
{
  std::cerr << "Unknown error occured\n";
}
```

Klassenhierarchien für exceptions

```
// normal errors
class SppError { .... };
class SppErrorDeviceBusy : public SppError {....};
...
// internal errors
class SppInternalError : public SppError {....};
class SppErrorSaneNotInitialized :  public SppInternalError{....};
...
```

- **catch(SppError& err)** fängt alle Fehlerklassen (also auch die abgeleiteten)
- **catch(SppInternalError& err)** fängt alle "internen" Fehler.
- **catch(SppErrorDeviceBusy& err)** fängt nur den "device busy error"
- Bei Fehlerklassen-Hierarchien Immer "**catch by reference**", nie "**catch bei value**"!
  (siehe Thinking in C++ Vol 2: *Programming with exceptions*)

"Resource allocation is initialization"

```
try
{
   handle = dc1394_create_handle(0);
   dc1394_setup_capture(handle,..., &camera);
   dc1394_start_iso_transmission(handle,camera.node);
   dc1394_single_capture(handle,&camera);
}
catch( int errorNumber)
{
   /* Wir wissen nicht, ob raw1394_destroy_handle(handle);
      aufgerufen werden muß */
}
```

- Damit geöffnete Resourcen automatisch wieder geschlossen werden, wenn eine exception auftritt, kann man die Methode "Resource allocation is initialization" anwenden:

"Resource allocation is initialization"

```
try
{
   ...
   ifstream hurz( "abcdeg.txt");
   ...
}  // hier wird das File hurz durch den Destructor automatisch geschlossen
catch( int errorNumber)
{
   /* An dieser Stelle befinden sich die Resourcen
      wieder in einem "sauberen" Zustand */
}
```

Extreme Programming (XP)

- "Write tests first"
  - Statt die Funktionalitäten einer Klasse in UML oder sonstigen Diagrammen zu definieren, werden diese Definitionen direkt in **Test-Programmen** notiert, so daß nachher automatisch die Funktionalität geprüft werden kann.
  - Jegliche interne **Erweiterung oder Umstrukturierung** darf nur eingechekt werden, wenn alle Tests fehlerfrei durchlaufen
  - Wenn später ein **unerwarteter Fehler** in der Klasse auftritt, so wird erst ein Test geschrieben, der diesen Fehler erkennt, und dann die Klasse so korrigiert, daß er nicht mehr auftritt.
- "Pair programming"
  - Einer denkt, einer tippt – kommt später dran...

XP Beispiel: Entwicklung einer Vector3D-Klasse

Schritt 1: Testprogramm schreiben

```
#include <iostream>
#include "Vector3D.hh"

static void testDefaultConstructorAndAccess()
{
  std::cout << "testing default constructor: ";
  Vector3D vec;
  if( vec.x() == 0  && vec.y() == 0  && vec.z() == 0)
  {
    std::cout << "SUCCESS\n";
  }
  else
  {
    std::cout << "FAILED\n";
  }
}

static void testConstructorWithParams()
{
  std::cout << "testing constructor with parameters: ";
  Vector3D vec( 3,7,42);
  if(      vec.x() == 3
     && vec.y() == 7
     && vec.z() == 42)
  {
    std::cout << "SUCCESS\n";
  }
  else
  {
    std::cout << "FAILED\n";
  }
}
```

```
static void testAddition()
{
  std::cout << "testing addition: ";
  Vector3D a( 1,2,4);
  Vector3D b( 16,8,32);
  Vector3D c;

  c = a + b;

  if(      c.x() == 17
     && c.y() == 10
     && c.z() == 36)
  {
    std::cout << "SUCCESS\n";
  }
  else
  {
    std::cout << "FAILED\n";
  }
}

// ...

int main( int argc, char** argv)
{
  testDefaultConstructorAndAccess();
  testConstructorWithParams();
  testAddition();
  return 0;
}
```

XP Beispiel: Entwicklung einer Vector3D-Klasse

Schritt 2: Compilierbar machen

Makefile:

```
CXX = g++-3.2
CXXFLAGS = -Wall

check: testVector3D
↦      ./testVector3D
```

g++ Version 3.2 benutzen

"Warnings: all" – alle Warnungen anzeigen

Um 'check' zu erstellen, muß zuerst 'testVector3D' erstellt werden. Wie das geht, weiß make selber

Dies ist ein Tab!

Wenn die Vorbedingungen erfüllt sind, soll ./testVector3D ausgeführt werden

Erster Test:

```
bienemaja:~/sp02ws/Vorlesungen/tests> make check
g++-3.2 -Wall    testVector3D.cc   -o testVector3D
testVector3D.cc:2:23: Vector3D.hh: Datei oder Verzeichnis nicht gefunden
testVector3D.cc: In function `void testDefaultConstructorAndAccess()':
testVector3D.cc:7: `Vector3D' undeclared (first use this function)
testVector3D.cc:7: (Each undeclared identifier is reported only once for each
   function it appears in.)
....
```

XP Beispiel: Entwicklung einer Vector3D-Klasse
Schritt 2: Compilierbar machen

Erstellen von Vector3D.hh und Vector3D.icc, so daß der Compiler durchläuft.
Nur "Dummy" Funktionen!

Vector3D.hh

```
class Vector3D
{
public:
  Vector3D();

  Vector3D( float x, float y, float z);

  float x() const;
  float y() const;
  float z() const;

  Vector3D operator+( const Vector3D& vec);
};

#include "Vector3D.icc"
```

Vector3D.icc

```
Vector3D::Vector3D()
{}

Vector3D::Vector3D( float x, float y, float z)
{}

float Vector3D::x() const
{
  return -1;
}

float Vector3D::y() const
{
  return -1;
}

float Vector3D::z() const
{
  return -1;
}

Vector3D Vector3D::operator+( const Vector3D& vec)
{
  return Vector3D(-1,-1,-1);
}
```

XP Beispiel: Entwicklung einer Vector3D-Klasse
Schritt 2: Compilierbar machen

Jetzt läuft der Compiler durch und es kommen die Meldungen von den fehlgeschlagenen Tests:

```
bienemaja:~/sp02ws/Vorlesungen/tests> make check
g++-3.2 -Wall    testVector3D.cc    -o testVector3D
./testVector3D
testing default constructor: FAILED
testing constructor with parameters: FAILED
testing addition: FAILED
```

Schritt 3: Funktionalität implementieren

Nun kann begonnen werden, die eigentliche Funktionalität zu implementieren, solange, bis alle Tests mit SUCCESS abgeschlossen werden.

Übungsaufgabe: RGB-Bildklasse