Part 3: Function Pointers, Linked Lists and nD Arrays

Prof. Dr. Ulrik Schroeder

C++ - Einführung ins Programmieren

WS 03/04

http://learntech.rwth-aachen.de/lehre/ws04/c++/folien/index.html

Pointer to functions

```
double volume(double x, double y=1, double z=1 ) {
        return x * y * z;
} // volume( )
double (*f) ( double, double, double ); // pointer to function
f = &volume; // defines this function; & operator is optional


double x = f( 2, 3, 4 ); // evaluates to 24.0
x = f( 2, 3  );              // ERROR: too few arguments
x = volume( 2, 3 );          // OK: evaluates to 6.0
```

Applying varying functions to data

```
double sine( double x )          { return sin(x); }
double cosine( double x )        { return cos(x); }
double square( double x )        { return x*x; }
double logarithm( double x )     { return log(x); }

double (*f[ ])(double)   = { sine, cosine, square, logarithm };
...
```

array of generic functions:
double → double

defines 4
functions within
this array

Applying varying functions to data

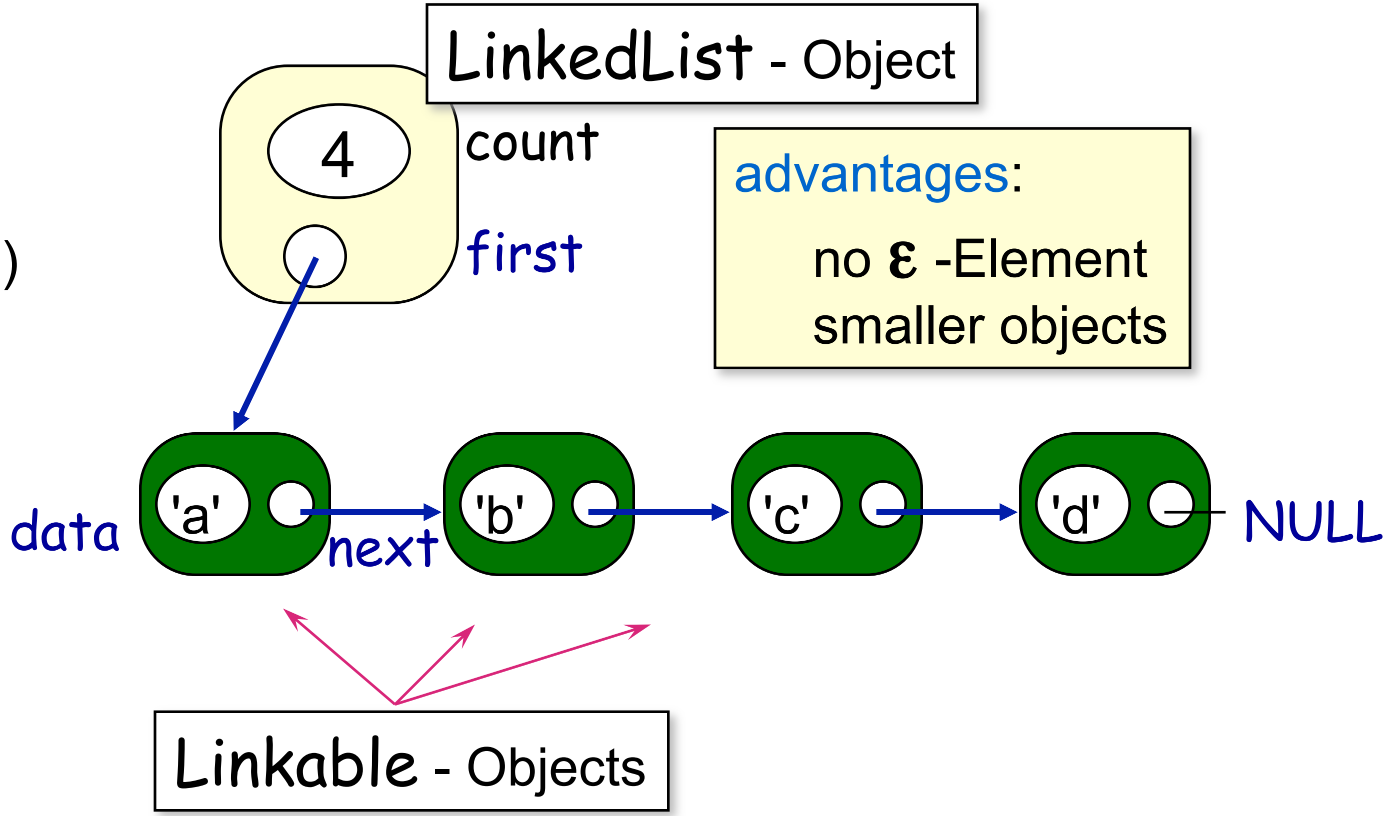
```
int main( int argc, char* argv[ ] ) {
  for ( int i = 1; i < argc; i++ ) {
    cout << "x = " << argv[ i ] << endl;
    for ( int j = 0; j < 4; j++ )
      cout << setw( 5 ) << fName[ j ] << "( x ) = "
           << setw( 12 ) << f[ j ]( atof( argv[ i ] ) ) << endl;
    cout << line << endl;
  } // for all arguments
  return 0;
} // main( )
```

applies each of the 4 functions to all arguments

```
x = 1
  sin( x ) =     0.841471
  cos( x ) =     0.540302
  sqr( x ) =            1
  log( x ) =            0
=========================
x = 10
  sin( x ) =    -0.544021
  cos( x ) =    -0.839072
  sqr( x ) =          100
  log( x ) =      2.30259
=========================
```

Pointer to objects: ADT LinkedList

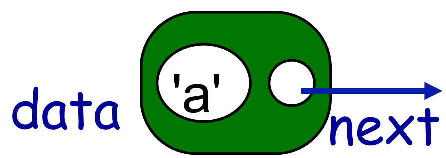
- dynamic structures (in opposition to array)
- holds variable amount of element data
- can be traversed sequentially
- operations
  - insert
  - append
  - print (traverse)



Linkable

- encapsulates data and knows its neighbor

```
class Linkable {
public:  // read
        inline string data( ) { return _data; }
        inline Linkable* next( ) { return _next; }
        // write
        inline void data( string d ) { _data = d; }
        inline void next( Linkable* n ) { _next = n; }
private:
        string _data;
        Linkable* _next;
}; // class Linkable
```



methods definied inside the class are implicitly *inline*

can be efficient for short methods

compiler replaces call with code

typical:

1. constructors
2. get/set methods
3. helper like min( ) sort( ), swap( ), …

LinkedList

- manages Linkable objects

```
class LinkedList {
public: // constructors
        LinkedList( int c=0, Linkable* f=NULL )
            : _count( c ), _first( f ) { }
        bool valid( int index ) { return 0<=index && index<_count;}
        int size( ) { return _count; }
private:
        int _count;
        Linkable* _first;
}; // class LinkedList
```

3 constructors

împlicitly inline

LinkedList

- manages Linkable objects

```
class LinkedList {
public:  // status read
         string at( int index=0 );
         int find( const string& data );
          // manipulation
         void insert( const string& data, int index=0 );
         void append( const string& data, int index=0 );
         void replace( const string& data, int index=0 );
         void erase( int index=0 );
         void display( );
private:  int _count;
          Linkable* _first;
}; // class LinkedList
```

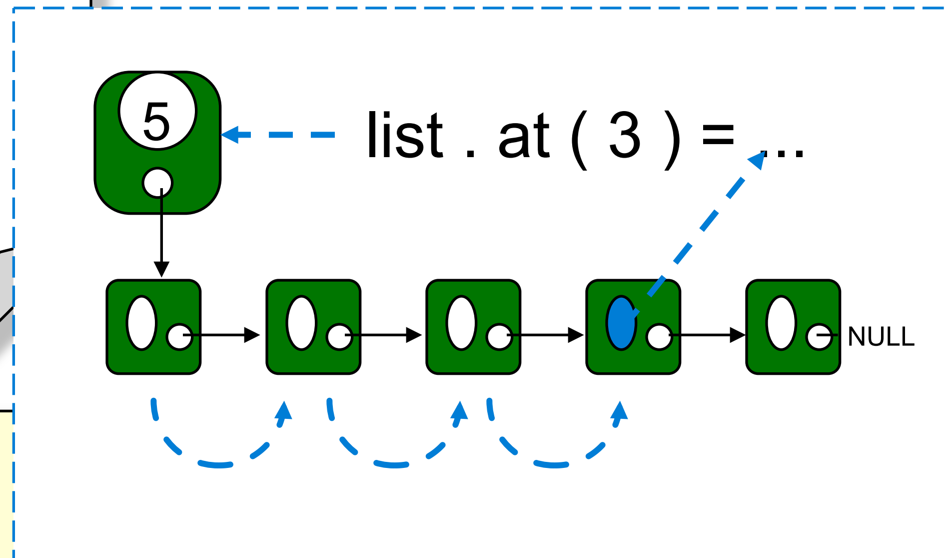
method prototypes

sequential access

```
string LinkedList::at( int index=0 ) {
    if ( valid( index ) ) {
      ① Linkable* cursor = _first; ②
        for ( int i=1; i<=index; i++ )
          ③ cursor = cursor->next( );
        return cursor->data( );
    } // if valid index
    return "";
} // at( )
```

not typical access for lists !!!

value of the i-th element



Pointer-Pattern 1

1. temporary Pointer
2. init with first
3. position to (next or) specific element in loop

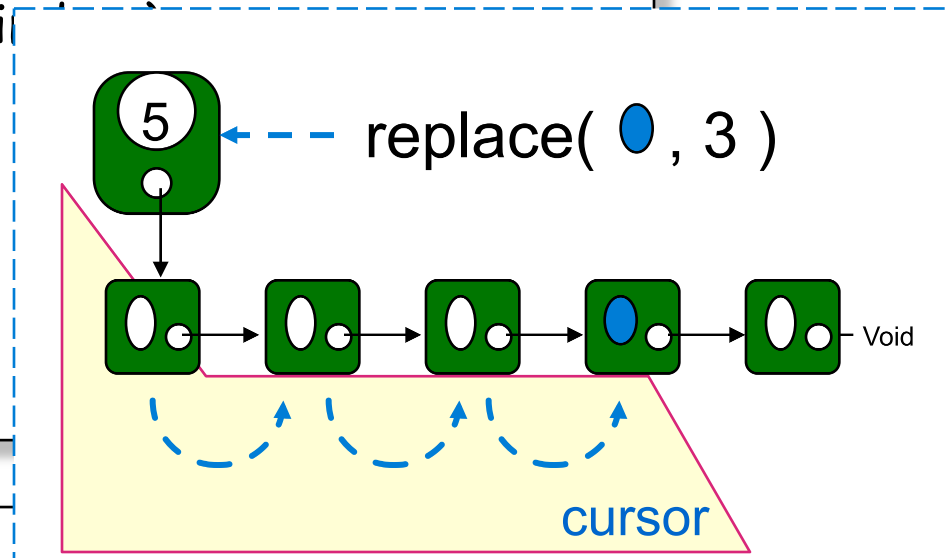
notation: obj->member

(*cursor).data( )

replace

```
void LinkedList::replace( const string& data, int index=0 ) {


    Linkable* cursor = elementAt( i

    if ( cursor ) {
        cursor->data( data );
    }

} // LinkedList::replace( )
```

```
class LinkedList {
...
private:
    Linkable* elementAt( int index );
```



internal method elementAt(...)

difference to method at( ) ?

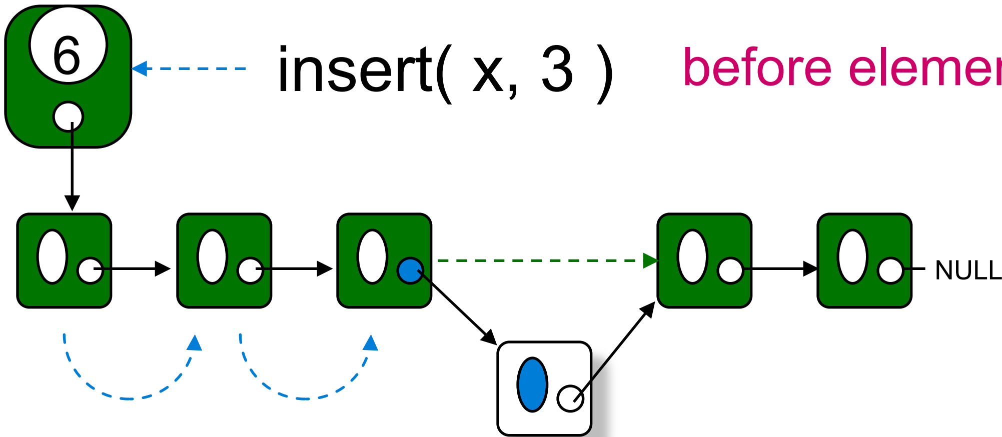
do we need to delete cursor ?

inserting elements

- what is the problem with this function?

?

insert(x, 3) before element at index 3



Pointer-Pattern 2

- ✓ position to element **before** action
- ✓ new with its next
- ✓ change next to new

```
void LinkedList::insert( const string& data,
    Linkable* cursor = elementBefore( index );
    Linkable* nEl = new Linkable( data, cursor->next( ) );
    cursor->next( nEl );
} // insert
```

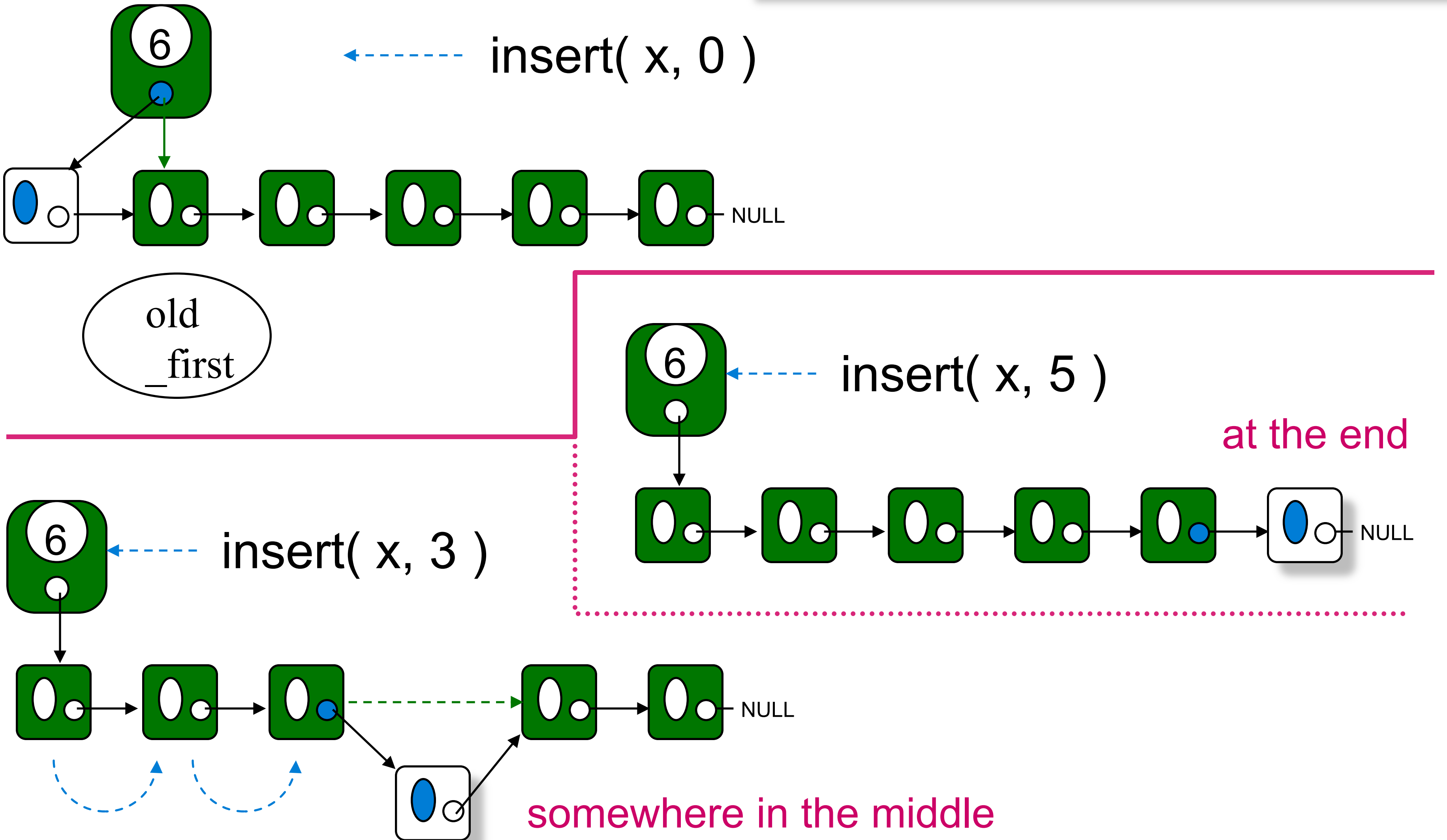
what about insert(x, 0) ?

first element has no predecessor of type Linkable!

inserting a new node

distinguish 3 (actually 2) cases

as first one



inserting

special case: if list is empty, then count == index == 0

```
void LinkedList::insert( const string& data, int index=0 ) {
    if ( valid( index )  || _count==0 && index==0 ) {
        Linkable* nEl = new Linkable( data );
        if ( index == 0 ) { // new first element
            nEl->next( _first );
            _first = nEl;
        } else { // insert before an existing element
            Linkable* cursor = elementBefore( index );
            nEl->next( cursor->next( ) );
            cursor->next( nEl );
        }
        _count++;
    } // if valid index
} // insert
```

new Linkable to be inserted

case 1: the new first element

case 2/3: insert before exist.

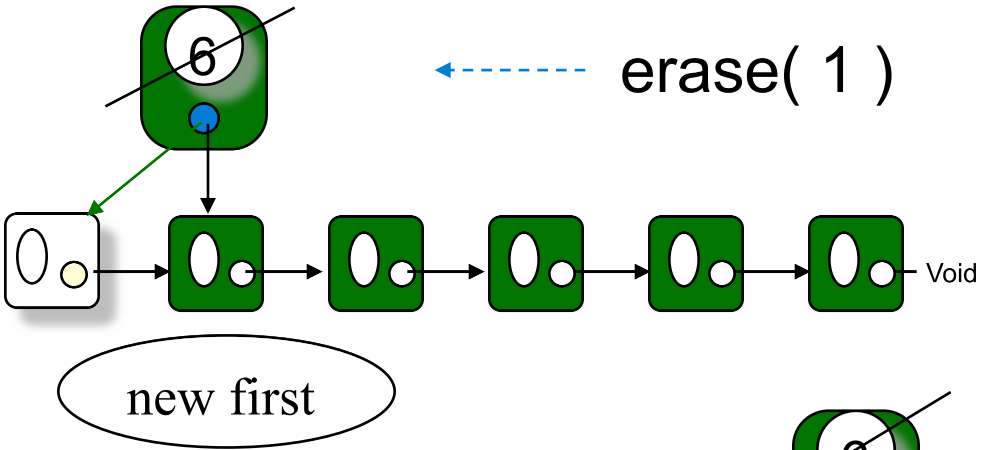
increment element counter

Pointer-Pattern 3

✓ distinguish head & tail elements

Erase an Element

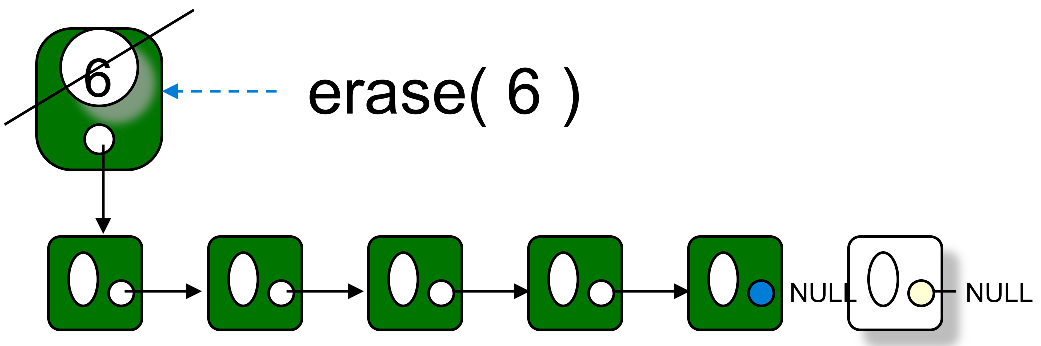
left for exercise

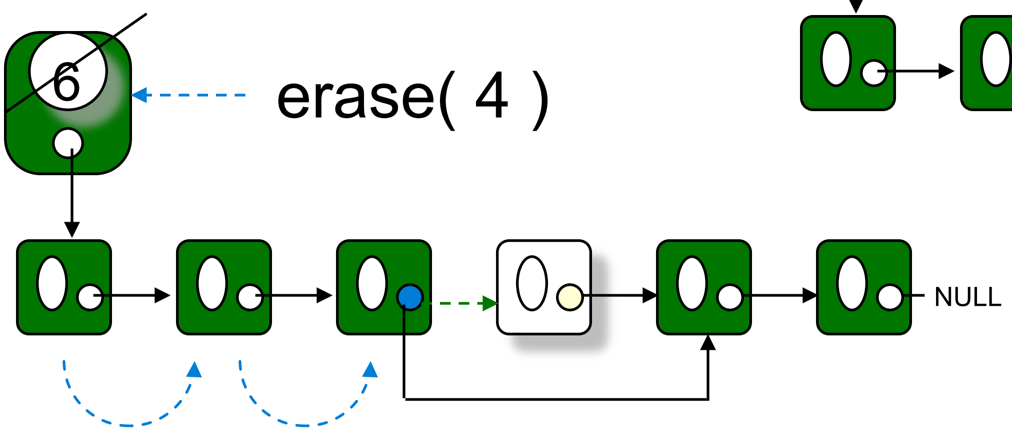


Pointer Pattern:

don't forget to free storage!!!

always associated *new* … *delete*





deleting

```
void LinkedList::delete( int index=0 ) {
    Linkable *pre, *d;
    if ( valid( index ) ) {
        if ( index == 0 ) { // first element
            d = _first;
            _first = d->next(); // unchain
        } else {
            pre = elementBefore( index );
            d = pre->next( );
            pre->next( d->next( ) );  // unchcain
        }
        delete d; // free storage!!!!

        _count--;

    } // if valid index
} // insert
```

case 1: delete first element

else: delete a following element

never forget !!!

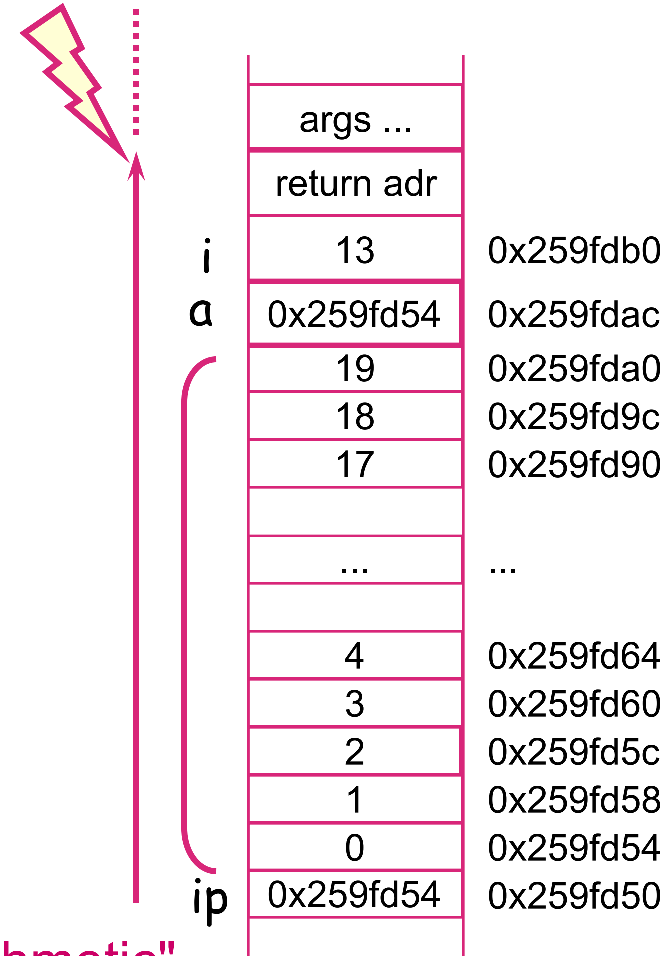
decrement element counter

Pointer and array

- an array is a interconnected field of (homogeneous) values
- same as `const` pointer to the first element
- C++ does not check for valid indices

```
int i = 13;
int a[20];  // local store, allocated
int *ip = new int; // global


a = ip; // *** NO!


ip = a; // ok
a[ i ] = 42;
*( a + i ) = 42; // all the same!
i[ a ] = 42;
```

```
for ( ip = a; ip  <40+a  ip++ )
      *ip = j++;
```



"pointer arithmetic"

be cautious, errorprone!

class string, char*, char[ ]

- Strings are no built in type, but have literals representations

```
"this is a string constant"
```

- Strings are equivalent to arrays of char (ending with '\0')

'r'	'w'	't'	'h'	'\0'

```
char a[ 5 ] = "rwth";
```

a   sizeof("rwth") == 5;   a[0] == 'r'   a[4] == '\0'

- arrays of char are equivalent to pointer

```
char a[ 5 ] = "rwth";
char* b = a;
cout << b;        → rwth
b[ 1 ] = 'u';
cout << a;        → ruth
```

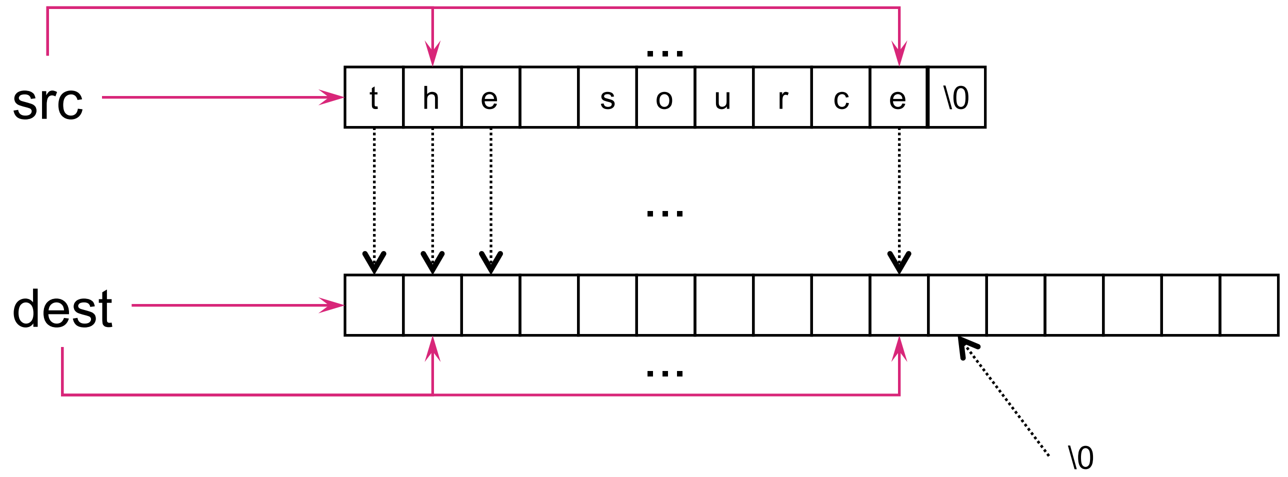
C++ also has class string (comparable to Java String)

pointer and array

C style: strcpy

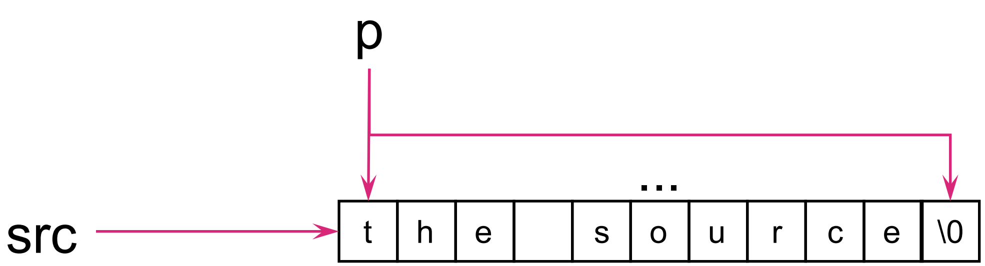
f(char [ ], ...) one dimension can be left undefined
Of course object must exist for the call

```
void strcpy(char* dest, char* src ) {

    while ( *dest++ = *src++ ) ;

} // strcpy -Version3: pointer
```



calculating with addresses

- C style: strlen




```
int strlen( char src[ ] ) {
    char *p = src;
    while ( *p++ );
    return ( p - str - 1 );
} // strlen
```

```
char* str___( char s1[ ], const char s2[ ] ) {
    char* p = s1 + strlen( s1 );
    strcp( p, s2 );
    return s1;
}
```

?

class string

- #include <string>
- Create:

```
string    s = "copy of this String literal.",
          t( s ),
          u( "literal" ),
          v( '=', 50 );
```

without '\0' at the end!!

copy constructor

constructor(char[ ])

constructor(char c, int anz)

- manipulate
  - concat – Ops: + and `s += " appended"` (like java)
  - compare: `s1 == s2` (Java: `s1.equals( s2 )`)
  - find substring: `int pos = s.find( "this" ); if ( pos != string::npos ) ...`
  - replace: `s.replace( pos, 4, "another" );`

string manipulation

```
#include <iostream> .. <string> .. <ctime>
using namespace std;
int main( ) {
    time_t sek;
    time( &sek );                 // get date & time (long)
    string t = ctime( &sek ); // convert to strin
    string dayOfWeek( t, 0, 3 ),
           month( t, 4, 3 ),
           day( t, 8, 2 ),
           year( t, t.size( ) - 5, 4 );
    cout << dayOfWeek+" " << day+". " << month+". "
         << year+"\n";
} // main( )
```

time_t, time( ), ctime( )

class string ...

1041015724

Fri Dec 27 19:27:45 2002

new substring from start, 3 chars long

?

Fri 27. Dec. 2002

Arrays

- differences to Java
  - C++ arrays are allocated statically (T name[constExprexssion];)
  - 2 dimensional thus are always rectangular, ...
  - size must be computed sizeof(a) / sizeof (base type) (no a.length)
- `char a[ ] = "implicit computation of size."`
- `int matrix [ ] [ 3 ] = { {1, 2}, { 3, 4 }, { 5, 6 } }`
- T a[..] ~ T* a
  - a[i] == *(a + i) == *(i + a) == i [a]
  - arrays are constant pointers to a field of n base type objects
- array assignment because of = overloading

  `int x[ 3 ], y[ 3 ]; ... init ...; x = y; // unlike java!!! Value Semantics (copy)`

argument vector

Matrix

```
int main( int argc, char* argv[ ] ) {
    char* p = argv[ argc-1 ] + strlen( argv[ argc-1 ] );
    while ( p >= argv[ 1 ] ) {
        cout<<*p--;
    }
    cout<<endl;
    return 0;
} // main
```

?

dynamic allocation of arrays

- compute size of array during runtime, or grow automatically, ...
- declare as pointer & allocate with new[ ] operator

```
int size=0, no=0, step=STEPSIZE;
float x, *pArr = NULL;

while ( cin >> x ) {
    if( no >= size ) { // needs resize
        float *p = new float[ size+step ];
        for( int i = 0; i < size; ++i )
            p[i] = pArr[i];
        delete [ ] pArr;
        pArr = p;    size += step;
    } // if resize
    pArr[ no++ ] = x;
} // while input
```

read in a bunch of numbers ...

allocate new vector for step more elements

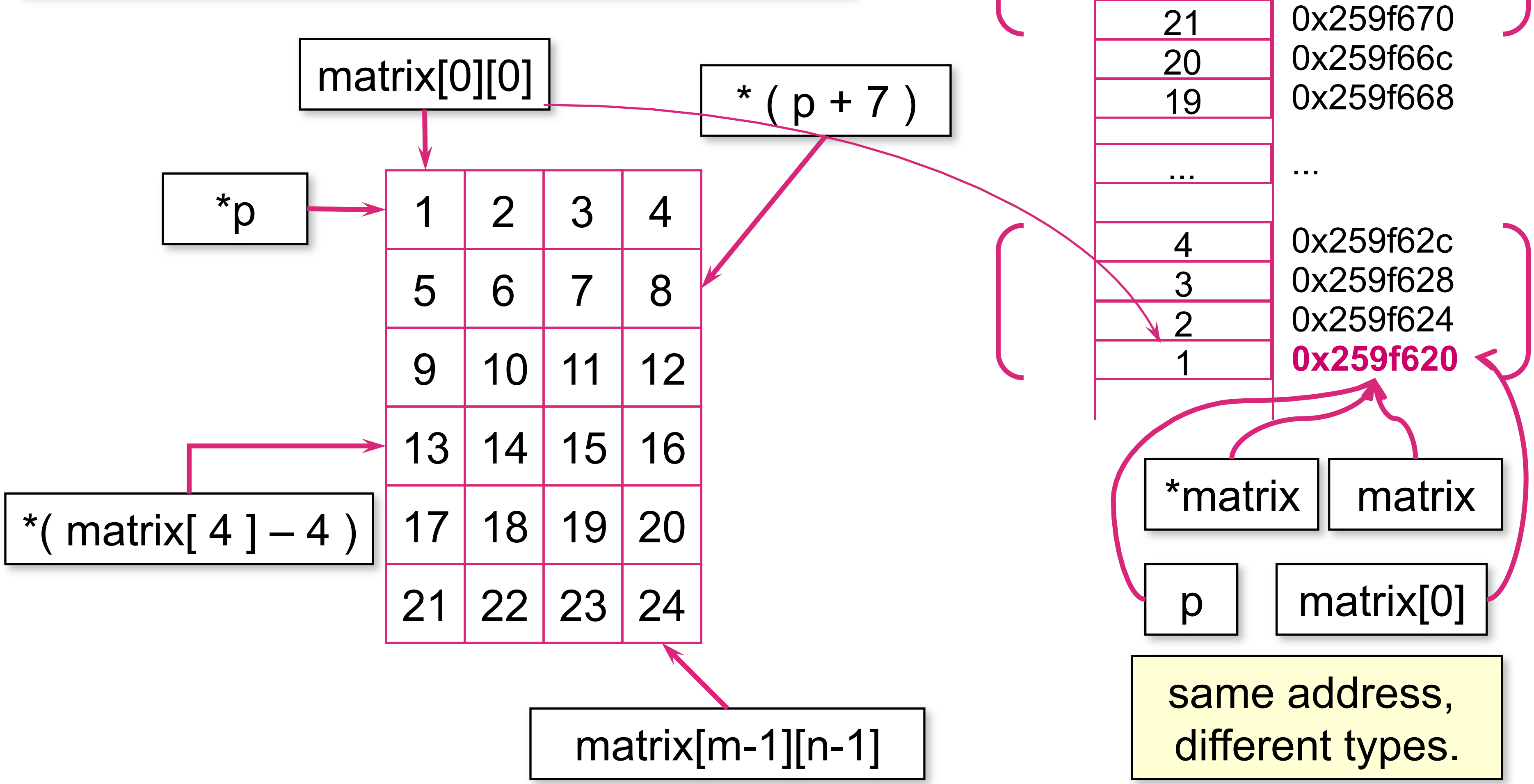
copy all values from old verctor

free store of old vector

set array & management info

2d vectors (matrix)

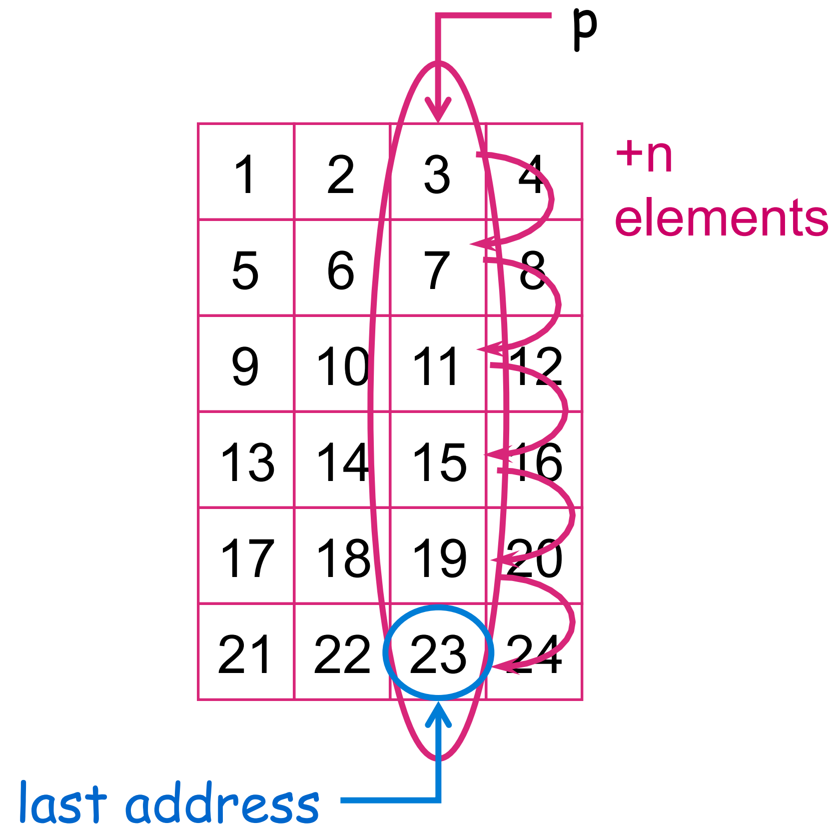
```
int matrix[ 6 ][ 4 ]; // ... fill with 1 .. 24
int *p=matrix[0];
```



Pointer arithmetics

- Task: print out values of the second to the last column:

```
for ( int * p =   & matrix[ 0 ][ n-2 ];
          p <= & matrix[ m-1 ][ n-2 ];
          p += n
      ) // increases n elements
    cout << *p << " ";
cout<<endl;
```



Allocation of Matrices

- dynamically and statically allocated matrices must be distinguished (are incompatible)

static allocation

initialization

```
int stat[3][5] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };
// same: { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15} }
for ( int i = 0; i < 3; i++ ) {
        for ( int j = 0; j < 5; j++ )
                cout << setw( 2 ) << stat[ i ][ j ] << " ";
        cout << endl;
} // for all Zeilen
```

```
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
```

Static Allocation of Matrices

- pointer iteration also possible

```
int stat[3][5] = { 1, ... };
...
for ( int* p = stat[0];
            p <= &( stat[2][4] );
            p++ )
      cout<<*p<<" ";
cout<<endl;
```

Print Procedure for Static Matrices

oder int p[ ]

übergeben als Vektor

```
void prS( int* p, int z, int s ) {
    for ( int i = 0; i < z; i++ ) {
        for ( int j = 0; j < s; j++ )
            cout<<setw( 2 ) << *( p + i*s + j ) << " ";
        cout << endl;
    } // for all lines
} // prS( )
```

can not be expressed as p[i][j], but means the same!

Aufruf:

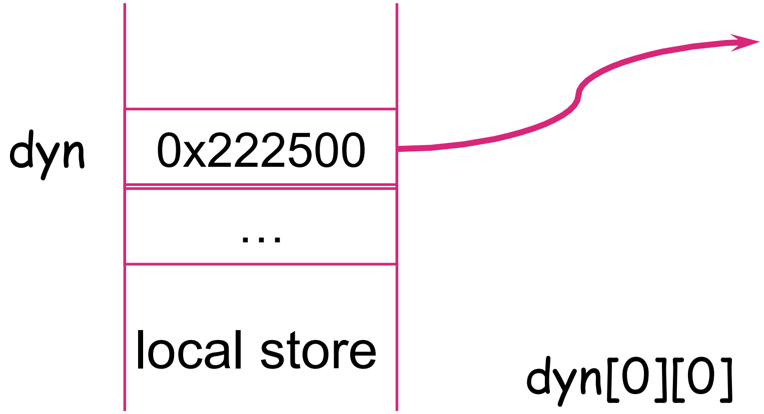
```
prS( stat[ 0 ], 3, 5 );
```

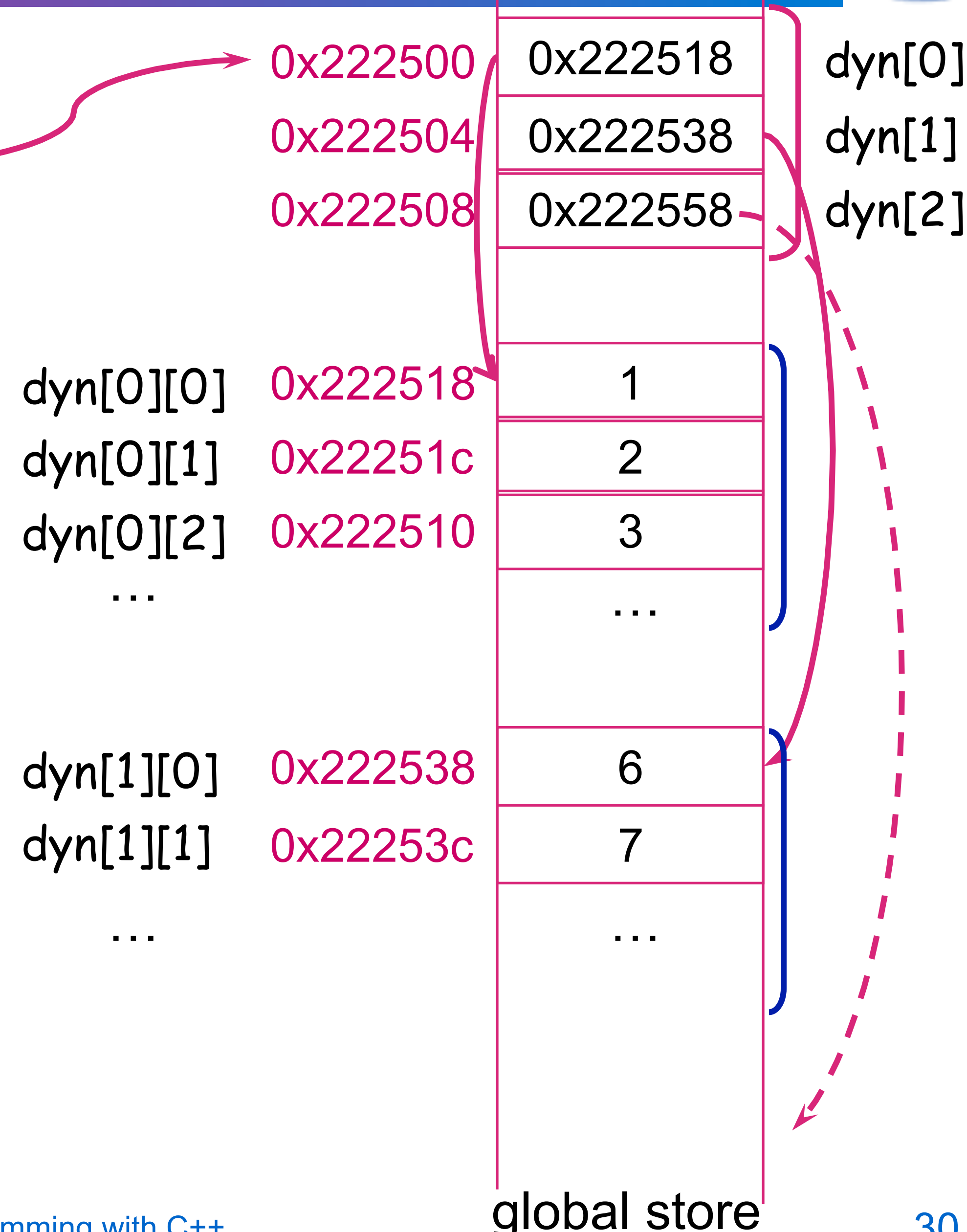
oder int** m

```
void pr( int* m[], int z, int s ) { ...
        cout << m[ i ][ j ] << .. }
...
pr( stat, 3, 5 );
```

passing `int (*)[5]' as argument 1 of `pr(int **, int, int)

Dynamic Allocation of Matrices






```
int** dyn = new int* [3];
for ( int i = 0; i < 3; i++ ) {
    dyn[i] = new int[5];
    for ( int j = 0; j < 5; j++ )
        dyn[ i ][ j ] = 5*i+j+1;
} // for all lines
```

Print Procedure for Dynamic Matrices

```
void pr( int** p, int z, int s ) {
    for ( int i = 0; i < z; i++ ) {
        for ( int j = 0; j < s; j++ )
            cout<<setw( 2 ) << p[ i ][ j ] << " ";
        cout << endl;
    } // for all lines
} // pr( )
```

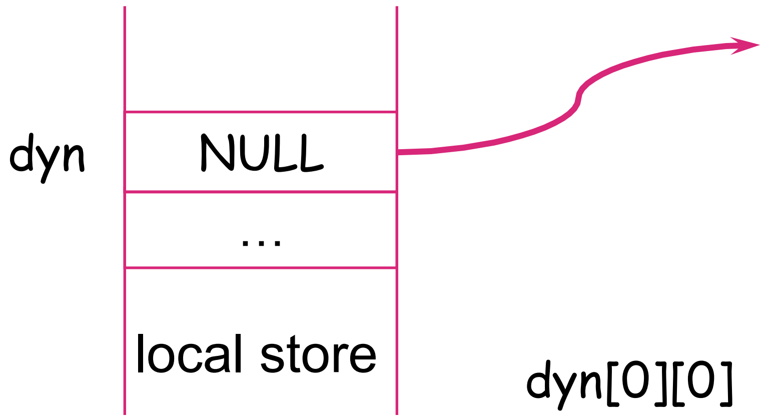
equivalent to
*(p + i*s + j)
*(p[i] + j)

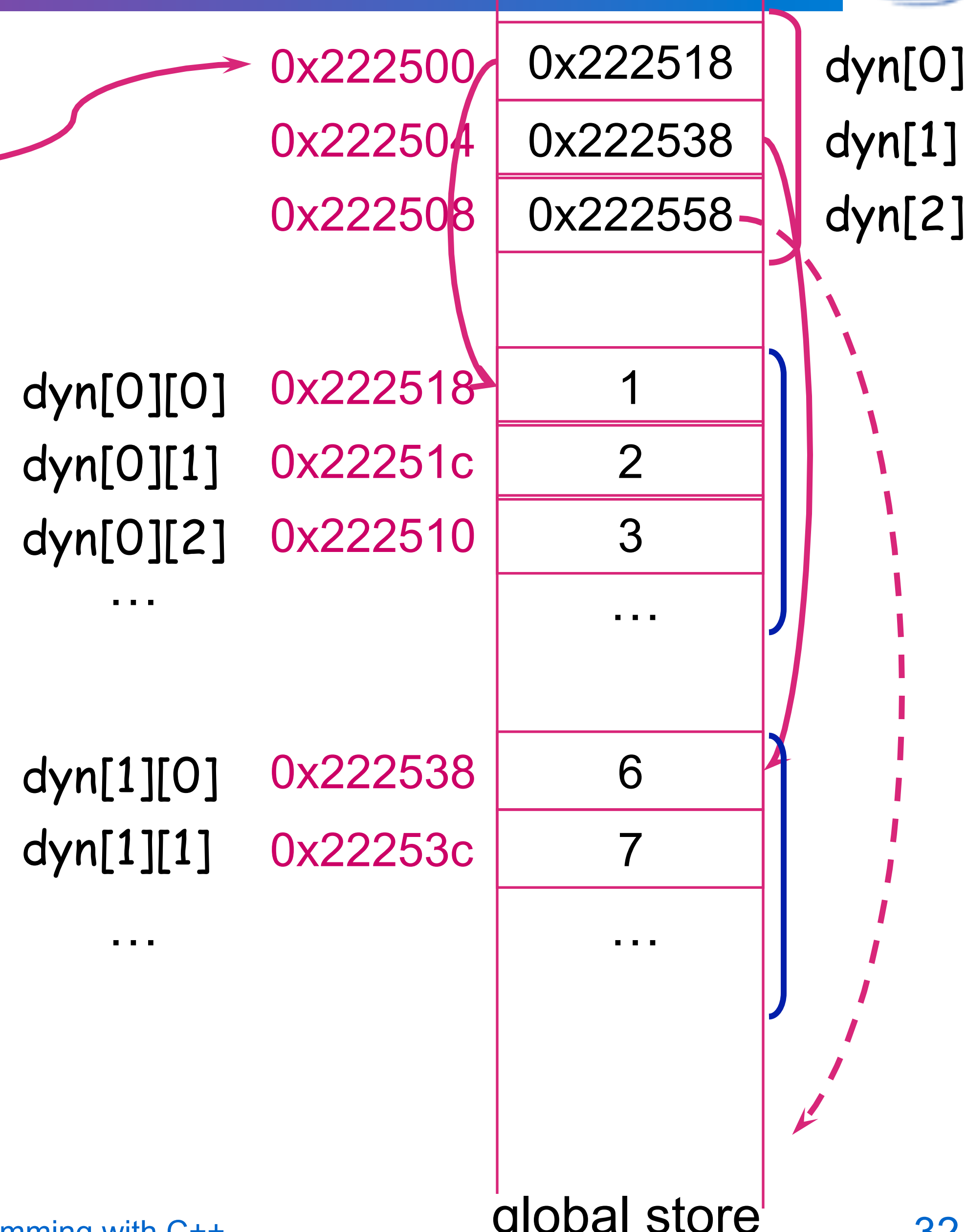
Aufruf:

```
pr( dyn, 3, 5 );
```

Free dynamically allocated storage

```
int** dyn = new int* [3];
      … new int[ 5 ] …
…
for ( int i = 0; i < 3; i++ )
      delete [ ] dyn[i];
delete[ ] dyn;
dyn = NULL;
…
```





Dynamic Matrices (Java)

- equivalent to Java arrays: non-rectangular matrices

```java
int a[ ][ ] = new int[ n ][ ];
for ( int i = 0; i < n; i++ ) {
    a[ i ] = new int[ i+1 ];
    for ( int j = 0; j <= i; j++ )
        if ( j == 0 || j == i )
            a[ i ][ j ] = 1; // frame-1
        else
            a[ i ][ j ] = a[ i-1 ][ j-1 ] + a[ i-1 ][ j ];
} // n binominial coefficients
```

```
$ java Pascal 7
                  1
               1     1
            1     2     1
         1     3     3     1
      1     4     6     4     1
   1     5    10    10     5     1
1     6    15    20    15     6     1
```

C++ implementation left for exercise

Pointer, References, Arrays

- Pointer: variables for addresses, calculating, `new` => global store => `delete`
- Reference: constant address of existing object
- Array: address of a continuous field of objects
  - Pointer for dynamic allocation: `new[ ]` and `delete[ ]`

- Storage:
  - local
  - global
    - global variables
    - static
    - explicitly allocated (with `new` operator) and freed (`delete` operator)