Arrays and Dynamic Memory

Arrays and Dynamic Memory

Array Definition

So far, we had only datatypes with one entry per variable. Arrays with a fixed number of entries are defined as:

$$\langle \textit{datatype} \rangle\ \langle \textit{variablename} \rangle \mathbf{[} \langle \textit{number of entries} \rangle \mathbf{]};$$

where the number of entries is a constant, e.g.

```
int        n[ 5 ];
double     f[ 10 ];
const int  len = 32;
char       str[ len ];
```

Arrays can also be preinitialised. In that case, the array size can be omitted:

```
int  n1[5] = { 0, 1, 2, 3, 4, 5 };
int  n2[]  = { 3, 2, 1, 0 };       // automatically size of 4
```

Array Access

A single entry in an arrays is accessed by the index operator **[·]**:

```
double  f[5];
int     i;

f[0] = -1.0;
f[1] =  3.0;
f[4] =  f[1] * 42.0;

i    = 3;
f[i] = f[0] + 8.0;
```

In C++, indices are counted from zero. The valid index range is therefore:

$$[0, \dots, \text{array size} - 1]$$

```
for ( int i = 0; i < 5; i++ )
    f[i] = 2*i;
```

Array Access

There are normally no array boundary checks in C++, i.e. you can specify arbitrary, even negative indices, resulting in an undefined program behaviour.

Typical error:

```
double  f[5];

for ( int i = 0; i < 5; i++ )    // Ok
    f[i] = 2*i;

for ( int i = 0; i <= 5; i++ )   // Bug
    f[i] = 2*i;
```

Coding Principle No. 14

Always make sure, that you access arrays within the valid index range.

Array Operations

Unfortunately, there are no operators for arrays, e.g. no assignment, elementwise addition or multiplication like in other languages. All of these have to be programmed by yourself:

```
void  copy ( const double x[3], double y[3] )
{
    for ( int i = 0; i < 3; i++ )
        y[i] = x[i];
}

void  add ( const double x[3], double y[3] )
{
    for ( int i = 0; i < 3; i++ )
        y[i] += x[i];
}
```

Remark

Arrays can be used as function arguments like all basic datatypes. But not as function return types!

Multidimensional Arrays

So far, all arrays have been onedimensional. Multidimensional arrays are defined analogously by appending the corresponding size per dimension:

```
int     M[3][3];
double  T3[10][10][10];
long    T4[100][20][50];
```

The access to array elements in multidimensional arrays follows the same pattern:

```
M[0][0] =  1.0; M[0][1] = 0.0; M[0][2] = -2.0;
M[1][0] =  0.0; M[1][1] = 4.0; M[1][2] =  1.0;
M[2][0] = -2.0; M[2][1] = 1.0; M[2][2] = -1.5;

for ( int i = 0; i < 100; i++ )
    for ( int j = 0; j < 20; j++ )
        for ( int k = 0; k < 50; k++ )
            T3[i][j][k] = double(i+j+k);
```

Multidimensional Arrays

Example: Matrix-Vector multiplication

```
void
mulvec ( const double  M[3][3],
         const double  x[3],
         double        y[3] )
{
    for ( int i = 0; i < 3; i++ )
    {
        y[i] = 0.0;

        for ( int j = 0; j < 3; j++ )
            y[i] += M[i][j] * x[j];
    }
}
```

Arrays and Pointers

C++ does not support variable sized arrays as an intrinsic datatype. Hence, arrays with an unknown size at compile time are not possible with previous array types in C++.

But, in C++, there is no distinction between a pointer and an array. A pointer not only directs to some memory address, it is also the base point, e.g. index 0, of an array.

```
int   n[5] = { 2, 3, 5, 7, 11 };
int * p    = n;

cout << p[0] << endl;  // yields n[0]
cout << p[1] << endl;  // yields n[1]
cout << p[4] << endl;  // yields n[4]
```

The index operator **[i]** of a pointer `p` gives access to the `i`'th element of the array starting at address `p`.

Dynamic Memory

Since pointers and arrays are equivalent, one needs to initialise a pointer with the address of a memory block large enough to hold the wanted array. This is accomplished by dynamic memory management:

> *Memory of arbitrary size can be allocated and deallocated at runtime.*

In C++ this is done with the operators **new** and **new[·]** to allocate memory and **delete** and **delete[·]** to deallocate memory.
For a single element:

> ⟨*datatype*⟩ ***** *p* = **new** ⟨*datatype*⟩;
> **delete** *p*;

For more than one element:

> ⟨*datatype*⟩ ***** *p* = **new** ⟨*datatype*⟩**[**⟨*size*⟩**]**;
> **delete[]** *p*;

Arrays and Dynamic Memory

Dynamic Memory

Examples:

```
char *   s = new char[ 100 ];
int      n = 1024;
double * v = new double[ n ];
float *  f = new float;

for ( int i = 0; i < n; i++ )
    v[i] = double( square( i ) );

*f = 1.41421356237; // dereference f

...

delete[] v;  // new[] => delete[]
delete[] s;
delete   f;  // new   => delete
```

Remark

The size parameter to **new** *does not need to be a constant.*

Problems with Pointers

The corresponding array to a pointer has no information about the array size. Remember, that C++ performs no boundary checks. That opens the door to many errors (see Coding Principle No. 14).

```
double * v = new double[ 1000 ];

...

v[2000] = 1.0;
```

With the last instruction, you overwrite a memory position corresponding to completely other data. The program will only terminate, if the memory does not belong to the program (segmentation fault).

Problems with Pointers

The programmer does not know if the memory was allocated or deallocated, except if the pointer contains **NULL** (see Coding Principle No. 5).

```
double * v = new double[ 1000 ];

...

delete[]  v;

...

v[100] = 2.0;       // Bug: memory for v is deallocated
```

Again, the last instruction will be executed and will only result in an immediate error, if the memory is no longer part of the program.

Coding Principle No. 15

After calling **delete***, reset the pointer value to* **NULL***.*

Problems with Pointers

Memory addressed by forgotten pointers is lost for the program. C++ does not automatically delete memory with no references to it (garbage collection).

```
void f ()
{
    double * v = new double[ 1000 ];
    ...  // no delete[] v
}
// v is no longer accessible, memory is lost
```

This bug is not directly a problem, since no other data is overwritten. But if a lot of memory is not deleted after use, the program will have no available memory left.

Coding Principle No. 16

Always make sure, that allocated memory is deallocated after using.

Problems with Pointers

Remark

The aftermath of a pointer related bug, e.g. array boundary violation or accessing deleted memory, may show up much later than the actual position of the error.

Summary: pointers are dangerous and require careful programming.
But we have no choice ☹.

Problems with Pointers

Remark

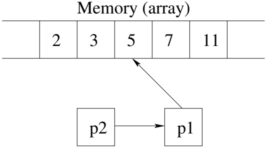
The aftermath of a pointer related bug, e.g. array boundary violation or accessing deleted memory, may show up much later than the actual position of the error.

Summary: pointers are dangerous and require careful programming. But we have no choice ☹. Well, almost ☺ (see later).

Multidimensional Arrays with Pointers

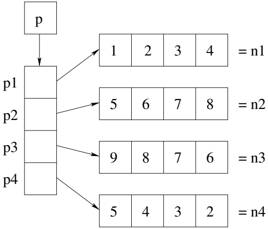
The analog of multidimensional arrays are pointers of pointers, i.e. pointers which direct to a memory address containing a pointer to another memory address:

```
int     n[5] = { 2, 3, 5, 7, 11 };
int *   p1   = & n[2];
int **  p2   = & p1;
```



This can be generalised to multiple dimensions:

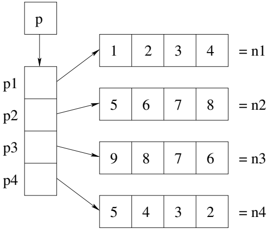
```
int   n1[4], n2[4], n3[4], n4[4];
int * p1   = n1;
int * p2   = n2;
int * p3   = n3;
int * p4   = n4;

int * p[4] = { p1, p2, p3, p4 };

cout << p[1][3] << endl; // yields 8
```



Multidimensional Arrays with Pointers

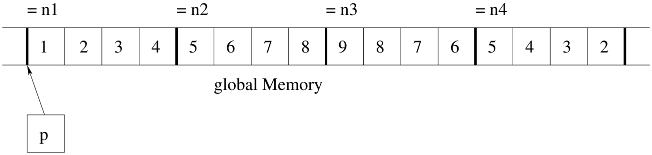
The same example with dynamic memory:

```
int *  p1 = new int[4];
int *  p2 = new int[4];
int *  p3 = new int[4];
int *  p4 = new int[4];

int ** p  = new int*[4];

p[0] = p1;
p[1] = p2;
p[2] = p3;
p[3] = p4;

p[0][0] = 1;
p[0][1] = 2;
...
p[2][2] = 7;
p[2][3] = 6;
p[3][0] = 5;
...
```



Multidimensional Arrays with Mappings

Working with pointers to pointers is only one way to implement multidimensional arrays. You can also map the multiple dimensions to just one:

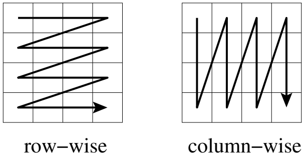



```
int  * p = new int[4*4];

p[ 2 * 4 + 1 ] = 8;  // p[2][1]
p[ 0 * 4 + 2 ] = 3;  // p[0][2]
```

Multidimensional Arrays with Mappings

In theory, one could use any mapping. In practise, two different mappings are standard:

- row-wise: standard in C, C++
- column-wise: standard in Fortran, Matlab



For a two-dimensional array, e.g. a matrix, with dimensions $n \times m$, the mappings are for index (i, j):

- row-wise: $i \cdot m + j$,
- column-wise: $j \cdot n + i$.

It is up to you, which mapping you prefer.

Example: $n \times m$ Matrix (row-wise)

```
void
set_entry ( const double * M,
            const int i, const int j,
            const int m, const double f )
{
    M[ i*m + j ] = f;
}

int
main ()
{
    int      n = 10;
    int      m = 20;
    double * M = new double[ n * m ];

    set_entry( M, 3, 1, m, 3.1415 );
    set_entry( M, 2, 7, m, 2.7182 );
}
```

Comparison: Pointers vs. Mapping

Two approaches have been introduced for multidimensional arrays: pointers of pointers and user defined mapping. Which is to be preferred?

A user defined mapping is faster since only simple arithmetic is performed for a memory access. The pointer based approach needs to follow each pointer individually, resulting in many memory accesses.

Pointers are more flexible, e.g. for triangular matrices, whereas a special mapping has to be defined for each shape.

My recommendation: use mappings, especially if you want fast computations.

Application: BLAS

Properties and requirements:

- vectors are onedimensional arrays, matrices implemented via mapping (row-wise),
- should provide functions for all standard operations, e.g. creation, access, linear algebra

Initialisation:

```
inline double *
vector_init ( const unsigned  i )
{
    return new double[i];
}

inline double *
matrix_init ( const unsigned  n, const unsigned  m )
{
    return new double[ n * m ];
}
```

Application: BLAS

Vector Arithmetic:

```
void fill  ( const unsigned  n, const double f, double * y );
void scale ( const unsigned  n, const double f, double * y );

void add   ( const unsigned  n, const double f, const double * x,
             double * y )
{ for ( unsigned i = 0; i < n; i++ ) y[i] += f * x[i]; }

double
dot ( const unsigned  n, const double * x, const double * y )
{
    double  d = 0.0;

    for ( unsigned i = 0; i < n; i++ ) d += x[i] * y[i];
    return d;
}

inline double
norm2 ( const unsigned  n, const double * x )
{ return sqrt( dot( n, x, x ) ); }
```

Application: BLAS

Matrix Arithmetic:

```
void
fill ( const unsigned  n, const unsigned  m,
       const double f, double * M )
{ fill( n*m, f, M ); }       // use vector based fill

void
scale ( const unsigned  n, const unsigned  m,
        const double f, double * M );

void
add ( const unsigned  n, const unsigned  m,
      const double f, const double * A, double * M );

inline double
normF ( const unsigned  n, const unsigned  m,
        double * M )
{ return norm2( n*m, M ); }  // use vector based norm2
```

Application: BLAS

Matrix-Vector Multiplication $y := y + \alpha A \cdot x$:

```
void
mul_vec ( const unsigned  n, const unsigned  m,
          const double alpha, const double * M, const double * x,
          double * y )
{
    for ( unsigned i = 0; i < n; i++ )
    {
        double  f = 0.0;

        for ( unsigned j = 0; j < m; j++ )
            f += get_entry( n, m, i, j, M ) * x[j];
        // alternative: f = dot( m, & M[ i * m ], x );

        y[i] += alpha * f;
    }
}
```

Remark

Compute dot product in local variable to minimize memory accesses.

Application: BLAS

Matrix-Matrix Multiplication $C := C + \alpha A \cdot B$:

```
void
mul_mat ( const unsigned n, const unsigned m, const unsigned k,
          const double alpha, const double * A, const double * B,
          double * C )
{
    for ( unsigned i = 0; i < n; i++ )
        for ( unsigned j = 0; j < m; j++ )
        {
            double  f = 0.0;

            for ( unsigned l = 0; l < k; l++ )
                f += get_entry( n, k, i, l, A ) *
                     get_entry( k, m, l, j, B );

            add_entry( n, m, i, j, f, M );
        }
}
```

Application: BLAS

```
double * M = matrix_init( 10, 10 );
double * x = vector_init( 10 );
double * y = vector_init( 10 );

fill( 10, 1.0, x );
fill( 10, 0.0, y );

...  // fill matrix M

cout << normF( 10, 10, M ) << endl;

mul_vec( 10, 10, -1.0, M, x, y );
```

Strings

One important datatype was not mentioned up to now: strings. Strings are implemented in C++ as *arrays of characters*, e.g.

char str[] or *char * str*

As arrays have no size information, there is no information about the length of a string stored. To signal the end of a string, by convention the character '0' is used (as an integer, not the digit), entered as '\0':

```
char  str[] = { 'S', 't', 'r', 'i', 'n', 'g', '\0' };
```

Constant strings can also be defined and used directly with ``...'':

```
char  str[] = "String";  // array initialisation
```

Here, '\0' is automatically appended.

Strings

If a string is too long for one input line, it can be wrapped by a backslash '\':

```
const char *  str = "This is a very long \
string";
```

C++ does not provide operators for string handling, e.g. concatenation or searching for substrings. All of that has to be implemented via functions:

```
char * concat ( const char * str1, const char * str2 )
{
    const unsigned  len1 = strlen( str1 );
    const unsigned  len2 = strlen( str2 );
    char *          res  = new char[ len1 + len2 + 1 ];
    int             pos  = 0, pos2 = 0;

    while ( str1[pos]  != '\0' ) { res[pos]   = str1[pos]; pos++; }
    while ( str2[pos2] != '\0' ) { res[pos++] = str2[pos2++]; }
    res[pos] = '\0';

    return res;
}
```

Strings

Usage:

```
const char * str1 = "Hallo ";
char *       str2 = concat( str1, "World" );

cout << str2 << endl;

delete[] str2;  // don't forget to deallocate!
```

It can not be emphasised too much:

Coding Principle No. 17

> *Always ensure, that strings are terminated by* $'\backslash 0'$.

Otherwise, operations on strings will fail due to array boundary violations.

Strings

'\0' is one example of a special character in C++ strings. Others are

Character	Result
'\n'	a new line
'\t'	a tab
'\r'	a carriage return
'\b'	a backspace
'\''	single quote '''
'\"'	double quote '"'
'\\'	backslash '\'

Examples:

```
cout << "First \t Second"  << endl;
cout << "line1 \n line2"   << endl;
cout << "special \"word\"" << endl;
cout << "set1 \\ set2"     << endl;
```