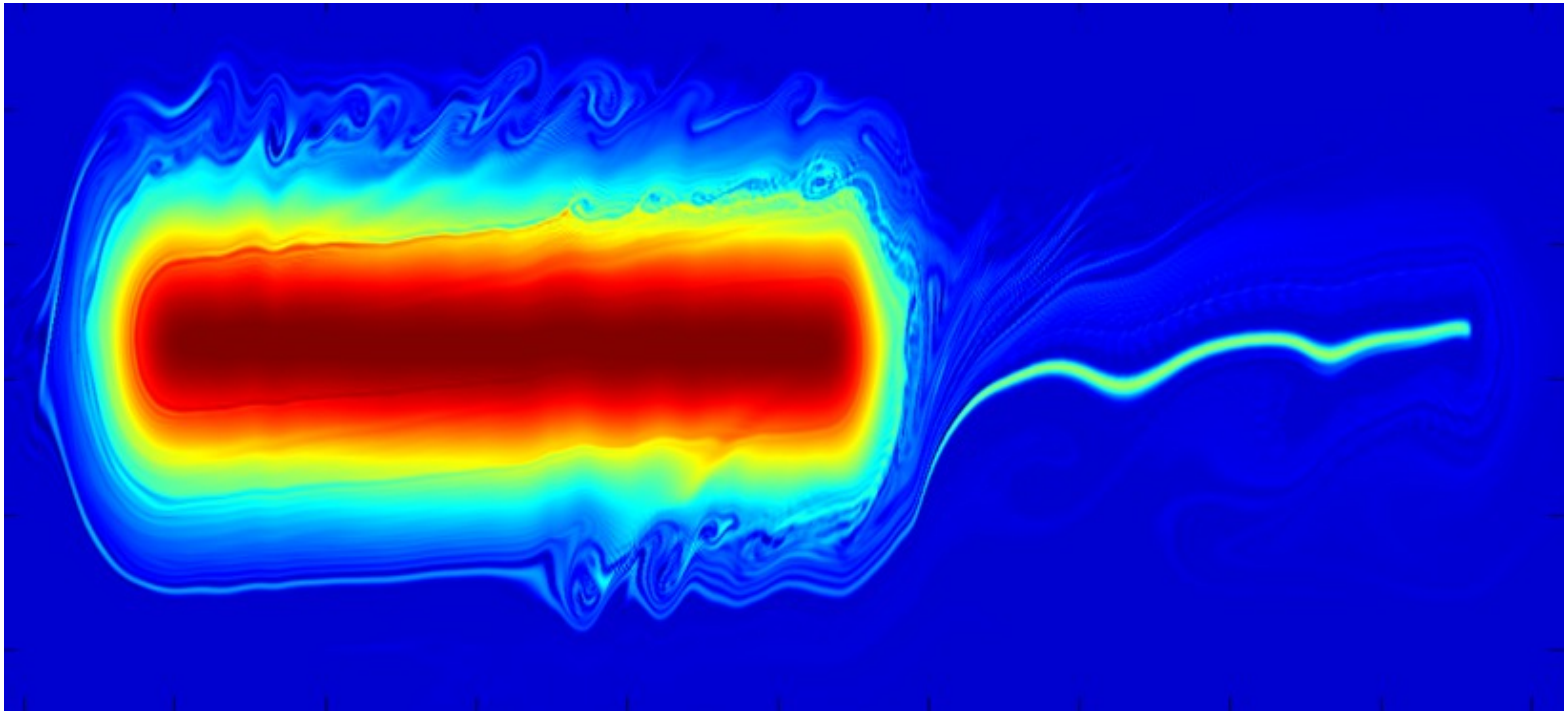


Numerical Simulations II - SS 2014
Chapter 2 - Arrays and Functions

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Complex Numbers

- The template class complex allows us to represent complex numbers and to do some math

```
#include <iostream>
#include <complex>

using namespace std;

int main(){

    complex<double> c,d,expc;

    // cf = 1.0 + 0.0i
    complex<float> cf = complex<float>(1.0, 0.0);

    // c = 0 + i
    c = complex<double>(0.0, 1.0);
    // d = 1.2 + i 0.5
    d = complex<double>(1.2, 0.5);

    expc = exp(c);

    cout << "c = " << c << ",\t d = " << d << endl;
    cout << "exp(x) = " << expc << endl;
    cout << "c*d =" << c*d << endl;
    cout << "|c*d| = " << norm(c*d) << endl;
    cout << "Re(c*d) = " << real(c*d) << ", \t Im(c*d) = " << imag(c*d) << endl;

    return 0;
}
```

Output:

```
c = (0,1),        d = (1.2,0.5)
exp(x) = (0.540302,0.841471)
c*d =(-0.5,1.2)
|c*d| = 1.69
Re(c*d) = -0.5,           Im(c*d) = 1.2
```

C++ - Arrays and Functions

Pointers and addresses

- The address-operator & can be used to find out the memory-address where the value of a variable is stored

```
#include <iostream>

using namespace std;

int main(){
    int i;
    double a,b,c;
    int j;

    cout << &i << endl;
    cout << &a << endl;
    cout << &b << endl;
    cout << &c << endl;
    cout << &j << endl;

    return 0;
}
```

Output (in hex):

```
0x7fff5fbff92c
0x7fff5fbff920
0x7fff5fbff918
0x7fff5fbff910
0x7fff5fbff928
```

Each double: 64 Bits = 8 Byte

Each int : 32 Bit = 4 Byte

Total: 32 Byte

c b

0x7fff5fbff9..																
	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
	20	21	22	23	24	25	25	27	28	29	2A	2B	2C	2D	2E	2F

a j i

Pointers and addresses

- What happens to a variable when we pass its value to a function?

```
#include <iostream>

using namespace std;

void f(double x){
    cout << &x << endl;
}

int main(){
    double a=5;

    cout << &a << endl;

    f(a);

    return 0;
}
```

Output:

```
0x7fff5fbff928
0x7fff5fbff920
```

a	28	29	2A	2B	2C	2D	2E	2F
x	20	21	22	23	24	25	26	27

- When we call a function, *new memory* is allocated to which a copy of the parameter values is stored. Within the function, we are thus working with a copy not the original data. *Modification of the local value does not affect the original variable.*

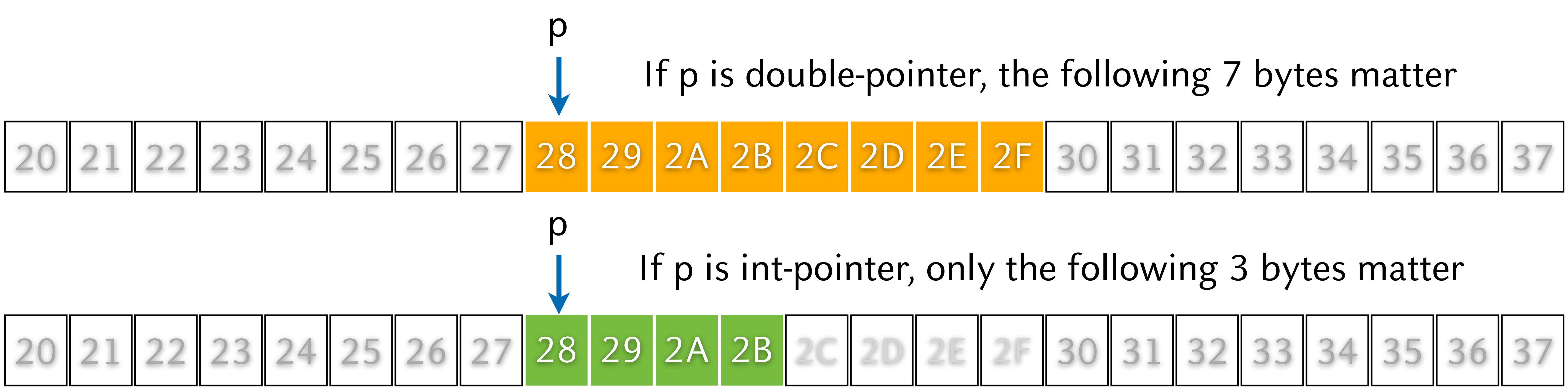
HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Pointers and addresses

- Pointers are datatypes which store the addresses of other variables.
- A pointer must always carry the information where the data is and what type of data there is
- Pointers are declared as:

```
double a=6;
double* p = &a;   // p is a pointer
int* pp = &a;  // this will not work
```

- Why do pointers carry a type?
  - It is to make sure the data at the address the pointer points to is interpreted correctly



HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Pointers and addresses

- Given we know the address of a variable we can manipulate the value which is stored there:

```
int a=5;
int* p = &a;
*(p) = 2;
```

- If we would just write

```
p=2;
```

  we would modify the address stored in p, not the value which is stored at the address p

- We need to *de-reference* the pointer using the * operator to write a new value at the address where p points to

- Another example:

```
double k=2, x=1.5, t;
double* p = &x;
t = k + *p;
```

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Pointers as function parameters

- When we call a function, we can pass the address of a parameter instead of the value of the parameter

```
#include <iostream>
using namespace std;

void f(double* x){
    cout << "x = " << x << endl;
    *x = 5;
}


int main(){
    double a=2;

    cout << "&a = " << &a << endl;
    f(&a);
    cout << "a = " << a << endl;

    return 0;
}
```

Output:

```
&a = 0x7fff5fbff928
x = 0x7fff5fbff928
a = 5
```

- Using this technique we can actually modify the value of the original variable from the main function.

Pointers and arrays

- We already used pointers when we discussed fixed size arrays. The statement
  double p[10];
  actually creates a double pointer to the beginning of a part of memory where we can sequentially store 10 double values.

```
int main(){
    double p[5];
    double* dp;

    p[0] = 1; p[2] = 2; p[4]=3;

    dp=p;

    cout << dp[0] << "\t" << dp[1] << "\t" << dp[4] << endl;

    return 0;
}
```

Output:

```
1   0   3
```

- The operator [] is related to the de-reference operator *, but it can do more.
  Based on the pointer type it knows how far behind the first memory address begins the second, third,... element.

Pointers and arrays

- Now we know how to pass an array as a function parameter: We pass the pointer to the first element of the array

```
#include <iostream>
using namespace std;

void f(double* x, int N){
    for(int i=0; i<N; i++)
        x[i] = i*i;
}


int main(){
    double p[5];
    int N = 5;

    f(p, N);

}
```

- Inside the function, we do not need to explicitly de-reference x, since we are using the [] operator which does the de-referencing for us
- We have to explicitly pass the array dimension, since the function can not infer this from the memory address alone

Calling a function...

- Up to now, we know two ways to pass parameters to a function:
  - pass a copy of the original variable
  - pass a pointer to the original variable
- Passing a copy allows to manipulate the copy without influencing the original data, but creating a copy needs time and space
- Passing the pointer to the original allows to manipulate the original (which we might want or not) and is usually fast

- Passing pointers is something we want to do frequently, but we always have to use the address operator & and the de-reference operator *, which makes code less readable.
- The solution are references

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

References

- References appear in function headers and are indicated by & after the datatype
  `void f(double* p, double& sum, double a, int N);`
- References are like pointers, but without the need for determining addresses and de-referencing. If we modify the value of a variable passed as reference in the function, we will change the value of the variable from the main function

```
void f(double* x, double& sum, double a, int N){
     sum=0;
     for(int i=0; i<N; i++){
          x[i] = pow(a,i);
          sum += x[i];
     }
}

int main(){
     double p[5];
     int N = 5;
     double a=2.3, sum;

     f(p,sum,a,N);

     for(int i=0; i<N; i++) cout << p[i] << endl;
     cout << "---" << endl << "sum = " << sum << endl;
}
```

No need to write
*sum += ...
here, because sum is a
reference.

Output:

```
1
2.3
5.29
12.167
27.9841
---
sum = 48.7411
```

more on references later...

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

const

- We often have variables which store a value that should never change. These values can be protected from change using the const statement

```
double area(const double r, const double PI){
    return r*r*PI;
}

int main(){
    const double PI = 3.14159265;
    const double PI2 = 2*PI;
    double r;

    cout << "r = "; cin >> r;
    const double u = PI2*r;

    cout << "circumference = " << u << ",\t area = " << area(r,PI) << endl;

}
```

- Use const wherever possible to minimize the risk of changing a variable unintentionally
- Const also allows the compiler to optimize code generation

const

- Using const together with pointers we have to distinguish two cases:
  - The pointer points to a variable which is constant,
  - The pointer itself is constant and can not be alterd, it always points to the same place, but the value at this place may change
- A `const double*` is a pointer which always points to a `const double`
- A `double* const` is a pointer which points to always the same double, but the value of the double may be altered
- It is possible to combine both to `const double* const`, which is a constant pointer that always points to the same constant double...

Const

```
void f(const double* x,  double* const y){
    // *x = 1; // error: assignment of read-only location
    *y = 3;

    const double q = 2;
    x = &q;

    double g=2;
    // y = &g; // error: assignment of read-only parameter 'y'
}

int main(){
    const double pi = 3.141;
      const double e = 2.714;
    double d=2;

    // double* p = &pi; // invalid conversion from 'const double*' to 'double*'

    const double* pp = &pi;
    // *pp = 1; // error: assignment of read-only location

    f(pp, &d);

    const double* const ppp = &pi;
    // ppp = &e; // assignment of read-only variable 'ppp'
}
```

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Dynamic arrays

- Often we need arrays for which the size is only known at runtime, then we need to dynamically reserve memory to store the array.
- To obtain a chunk of memory of the correct size, we need the new command
- `new double[n]` will return a double pointer to a chunk of memory large enough to hold n doubles

```
#include <iostream>

using namespace std;

int main(){
    const int N = 10;
    double p[N]; //static allocation,
                 //size known in advance

    int n;
    cout << "n = "; cin >> n;

    double* pn=new double[n]; //dynamic allocation

    for(int i=0; i<n; i++) pn[i] = 0;

    delete[] pn; //free up memory
}
```

Dynamic arrays

- For every new statement we need the according delete statement
- If we reserve memory just for a single variable

  ```
  double* d = new double;
  ```

  we only need to free this single memory slot via

  ```
  delete d;
  ```

- When we reserve memory for an array of data

  ```
  double *d = new double[n];
  ```

  we need to free the whole memory block via

  ```
  delete d[];
  ```

- When you forget to free memory again, this may lead to a crash of your program
- Always check programs for memory leaks (e.g. using Valgrind) and take memory leaks seriously!

C++ - Arrays and Functions

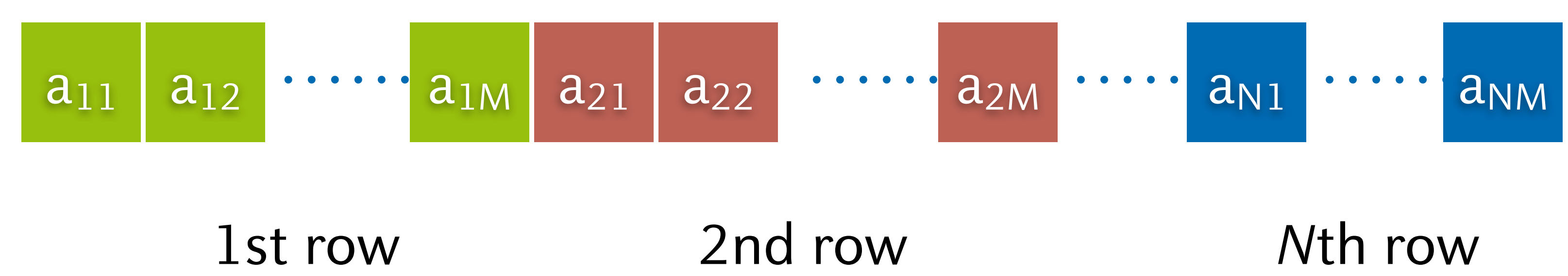
HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Multi-dimensional arrays

- Static allocation is easy:
  double p[100][20];

  p[99][19] = 10;
- When we pass a multi-dimensional, statically allocated array to a function we have to write the size of all dimensions but the first into the function header:
  `void f(double p[][20]);`
- How is the data stored in memory?
  - C++ uses row-major format (Fortran column-major!)

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & & & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NM} \end{pmatrix} \longrightarrow$$



Multi-dimensional arrays

- For us the more interesting case is a dynamically allocated multi-dimensional array. There is no such thing in C++.
- We could mimic the syntax a[i][j] for dynamic allocation of memory for a *N x M* matrix by dynamically allocating memory for *N* double pointers. Each double pointer would then be assigned via new to a 1D double array of length *M*.
  - This provides intuitive indexing, but we can not guarantee all double arrays to lie in a contiguous memory block ➠ this is bad for performance
- We need to map a 2D array onto a 1D array on our own
- Assume we want to store a Matrix of *Ny* x *Nx* double values.
  - Allocate an array of length $N = Ny * Nx$
  - Entry *(i,j)* of the Matrix is located at index $k = (i-1)*Nx + j - 1$ of the array (assuming all entries in one row of the matrix are stored sequentially - row-major order)
- Writing code that makes extensive use of Matrices becomes very cumbersome this way… We will need a much easier way to access these entries.