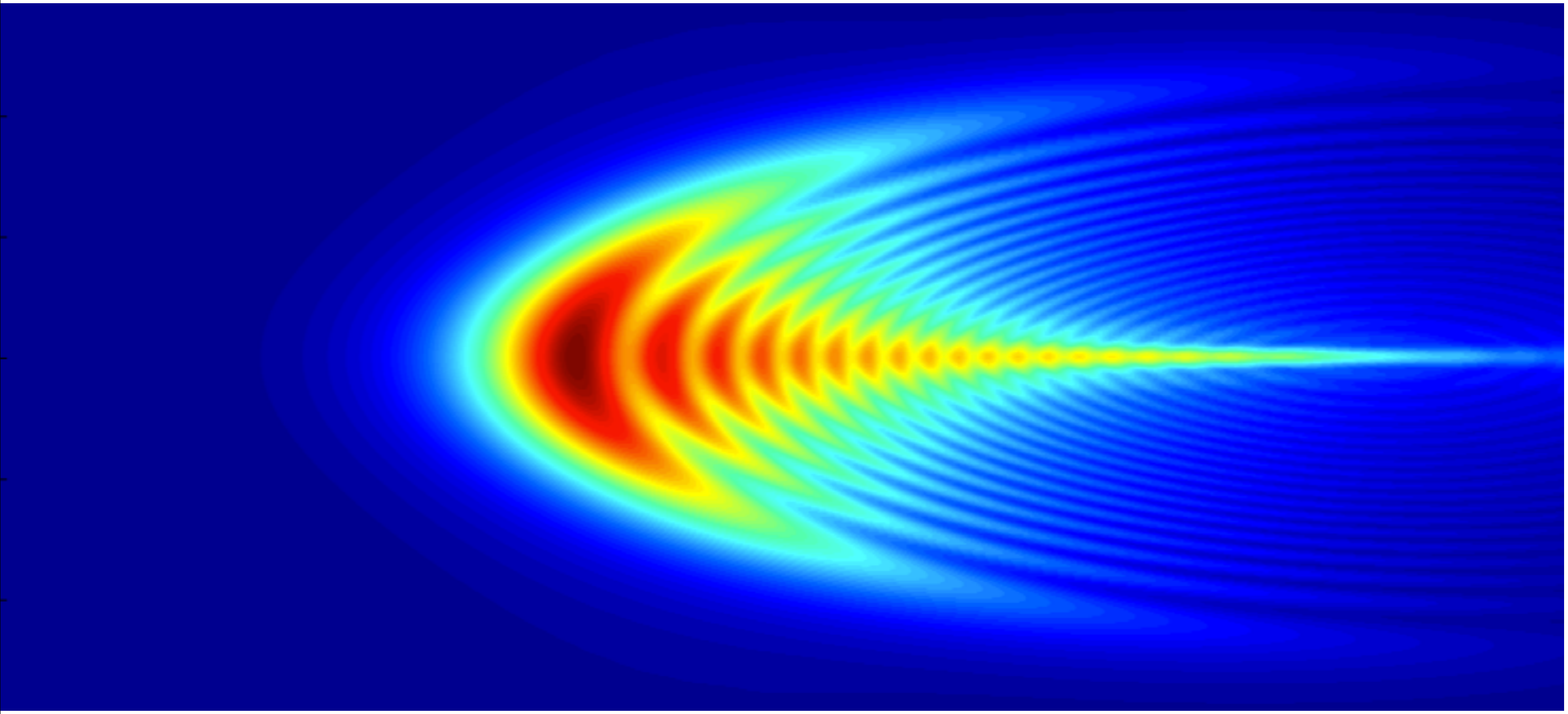


Computational Physics - WS 2015/16

Lecture on 18th Nov 2015

Homework #3 - sample solution

Statistics, random numbers, references

```
#include <iostream>
#include <cstdlib>

using namespace std;

void randomize(double* const, const int);
void stat(double&, double&, const double* const, const int);

int main(void){

  double mean = 0,   var = 0;
  int    N    = 100;
  double p[N];

  randomize(p, N);
  stat(mean, var, p, N);

  cout << "Mean value : " << mean << endl;
  cout << "Variance   : " << var  << endl;

  return 0;
}

void randomize(double* const p, const int N){
  for(int i = 0; i < N; i++)
    p[i] = double(rand()) / RAND_MAX;
}

void stat(double& mean, double& var, const double* const p, const
int N){
  for(int i = 0; i < N; i++)
    mean += p[i];
  mean /= N;

  for(int i = 0; i < N; i++)
    var += (p[i]-mean)*(p[i]-mean);
  var  /= N;
}
```

```
#include <ctime>
```

sometimes not necessary to include, depends on compiler

```
srand(time(NULL));
```

Output:

```
Mean value : 0.549589
Variance   : 0.0850506
```

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

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

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 altered, 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...

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

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'
}
```

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

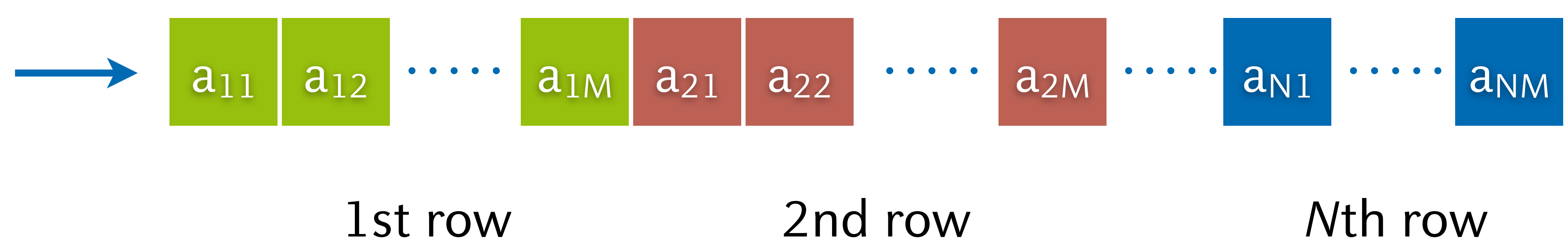
- For every new statement we need the according delete statement
- If we reserve memory just for a single variable
  `double* d = new double;`
  we only need to free this single memory slot via
  `delete d;`
- When we reserve memory for an array of data
  `double *d = new double[n];`
  we need to free 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!

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!)

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



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.

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Multi-dimensional arrays

In this example we have column-major (free choice!) and indices start at (1,1) and end at (N,M)

```
#include <iostream>

using namespace std;

int idx(const int, const int, const int, const int);

int main(void){
    int N, M, n, m;
    double* p;

    cout << "Enter matrix dim N (#columns)" << endl;
    cin  >> N;
    cout << "Enter matrix dim M (#rows)"    << endl;
    cin  >> M;

    p = new double[N*M];                                    // allocate memory

    cout << "which idx should be set to 5?" << endl;
    cin  >> n;
    cin  >> m;

    p[idx(n,m,N,M)] = 5.0;

    cout << "p(" << n << "," << m << ") = " << p[idx(n,m,N,M)] << endl;

    delete[] p;                                             // free memory
    return 0;
}

int idx(const int n, const int m, const int N, const int M){
    if ((n-1 < N) && (m-1 < M) && (n > 0) && (m > 0))    // control validity
        return (n-1)+(m-1)*N;
    else
        cout << "Error! n = " << n << ", m = " << m << " is not valid" << endl;
    exit(1);                                        // end program if not valid
}
```

Strings

- Sometimes we want to give a string to a function in order to specify e.g. the filename

```
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

void print_f(string);   // print-to-file function
void print_c(string);   // print-to-console function

int main(void){

  string word = "placeholder";

  print_f(word);
  print_c(word);

  return 0;
}

void print_f(string word){
  ofstream out(word.c_str());   // attention: out needs a char pointer
  out  << "word of the day is " << word << endl;
  out.close();
}

void print_c(string word){
  cout << "word of the day is " << word << endl;
  cout << "word of the day is " << word.c_str() << endl; // cout can handle both
}
```

Output:

```
word of the day is placeholder
word of the day is placeholder
```

+ the file „placeholder“ which includes

```
word of the day is placeholder
```

C++ - Writing and reading data

Stringstreams

- Sometimes we want to append characters to a preexisting string, e.g. for labeling multiple output files

```
#include <string>
#include <fstream>
#include <sstream>

using namespace std;

void print_f(string);

int main(void){

    int N = 5;
    string word = "placeholder";
    stringstream s;                         // create stringstream
    for (int i = 0; i < N; i++){
        s.str("");                          // empty it
        s << word << "_" << i << ".txt";    // fill it
        print_f(s.str());                   // convert to string
    }

    return 0;
}

void print_f(string word){
  ofstream out(word.c_str());   // make a char pointer out of the string
  out  << "useful content" << endl;
  out.close();
}
```

Output:

the files „placeholder_0.txt"
„placeholder_1.txt"
„placeholder_2.txt"
„placeholder_3.txt"
„placeholder_4.txt"

including „useful content"

C++ - Writing and reading data

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Ifstreams

output:
```
p[0] = 0.245
p[1] = 0.632
p[2] = 0.123
p[3] = 0.741
p[4] = 0.953
```

- reading data via input file stream object (*ifstream*)

Array contained in file „data“:

0.1	0.245
0.2	0.632
0.3	0.123
0.4	0.741
0.5	0.953

We want to store the second column into an array

```
#include <fstream>
#include <string>
#include <iostream>

using namespace std;

void reading(double* const, const int, const string);

int main(void){
    const int N = 5;
    const string filename = "data";
    double* p = new double[N];

    reading(p, N, filename);

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

    delete[] p;

    return 0;
}

void reading(double* const p, const int N, const string fname){
    ifstream in(fname.c_str());     // create input file stream
    double temp;                    // temp variable
    for (int i = 0; i < N; i++){
        in >> temp;                 // row major reading
        in >> p[i];                 // every 2nd entry is what we store
    }
    in.close();                     // close input file stream
}
```

HEINRICH HEINE UNIVERSITÄT DÜSSELDORF

Typedef

- Typedef allows us to introduce our own datatypes

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

typedef int* intp;
typedef complex<double> cmplx;

int main(void)
{
    int* nx,ny; //Only nx is a pointer,
                //ny is a regular int


    int *NX, *NY; //Both are pointers to int

    intp MX, MY; // Both are pointers to int

    cmplx c; // c is a double precision complex number

    cmplx* cp= new cmplx[*MX]; //cp is a pointer
                               //to an array of length
                               //*MX
}
```

Structs

- Structs are data structures that are made up from several components. Each component may have another datatype.
- Example: When we study the motion of a particle due to some force, we will need the values v_x, v_y, v_z, x, y, z. Maybe we will need additional values like mass m, charge q, ...
  - Structs allow to summarize these under the same name
- Structs are defined as

```
struct{
        component1;
        component2;
};
```

  - Don't forget the finalizing semi-colon!
- Any datatype we know can become component of a struct

Structs

- Once a struct is defined it can be used like any other datatype

```
#include <cmath>
#include <iostream>

using namespace std;

//------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};

//------------------------------
//------------------------------
int main(){
    particle p1;

    p1.x = 0.0; p1.y = 0.5; p1.z=0;
    p1.vx = 1.0; p1.vz = 0.2;
    p1.m = 1836; p1.q = -1;


    return 0;
}
```

Accessing members via pointers to structs

- If we have a pointer to a struct, access to the struct members is provided only by the -> operator
- No explicit de-referencing is necessary

```
#include <iostream>
using namespace std;
//--------------------------------
struct particle{
    int n;
    double x,y,z;
    double vx,vy,vz;
    double q,m;
};
//--------------------------------
int main(){
    particle p1;
    particle* pp = &p1;

    p1.x = 0.0; p1.y = 0.5; p1.z=0;
  p1.vx = 1.0; p1.vz = 0.2;
  p1.m = 1836; p1.q = -1;

  // pp.x = 1; // Will not work since pp is a pointer
  pp->x = 1;
  pp->y = 1;

  cout << "x = " << p1.x << ",\t y =" << p1.y << endl;
  return 0;
}
```