Brief C++ recap

HPCSE - Autumn semester 2014

Andreas Hehn

Disclaimer

This is not a complete C++ tutorial!

More like: a jump-start for Matlab/Java/Python programmers.

Detailed C++ introduction see lecture:
Programming techniques for scientific simulations (Prof. Troyer)
(Slides are based on this lecture)
or references on last slide.

What we will touch

- Basics:
  - Types
  - Pointers, References
  - Dynamic memory
- Templates
- Classes
- Standard Library

A short program

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  // this is a comment
  cout << "Enter a number:\n";
  double x;
  cin >> x;
  cout << "The square root of
  " << x << " is "
  << sqrt(x) << "\n";
  return 0;
  /* another comment */
}
```

- #include includes external files
- Namespace `std` provides `std::cout`, etc.
- all program start with a function `main`

- a variable named 'x' of type '`double`' is declared
- a double value is read and assigned to x
- The square root is printed
- Exit with everything ok (`0`).

Fundamental data types

- Booleans (logical types): `bool` (may be true or false)
- Integer types
  - Letters/single byte: `char`
  - Signed: `short` (16bit), `int` (32bit), `long` (32 or 64bit), `int32_t`, `int64_t`, ...
  - Unsigned: `unsigned short`, `unsigned` ..., `uint32_t`, ...
- Floating point types
  - Single precision: `float` (32bit)
  - Double precision: `double` (64bit)
  - ...

Constants: add `const`

Initializing variables

```
double x;
cout << x;
//WARNING:
//output may be anything
```

- fundamental types are **not** initialized by default!

```
double x=0.0;
double y(1.0);
double z{5.0};
cout << x << ”,“ << y
     << “,” << z;
// good: output is 0,1,5
```

- Always initialize variables!

Advanced types

- **Enumerators** are integer which take values only from a certain set

```
enum trafficlight {red, orange, green};
enum occupation {empty=0, up=1, down=2, updown=3};
trafficlight light=green;
```

- **Arrays** of size n

```
int i[10]; double vec[100]; float matrix[10][10];
```

  - indices run from 0 … n-1! (FORTRAN: 1…n)

```
vec[0] = 0.4; ...; vec[99] = 0.1;
```

  - last index changes fastest (opposite to FORTRAN)
  - Better: `std::array` or `std::vector`

- Complex types can be given a new name

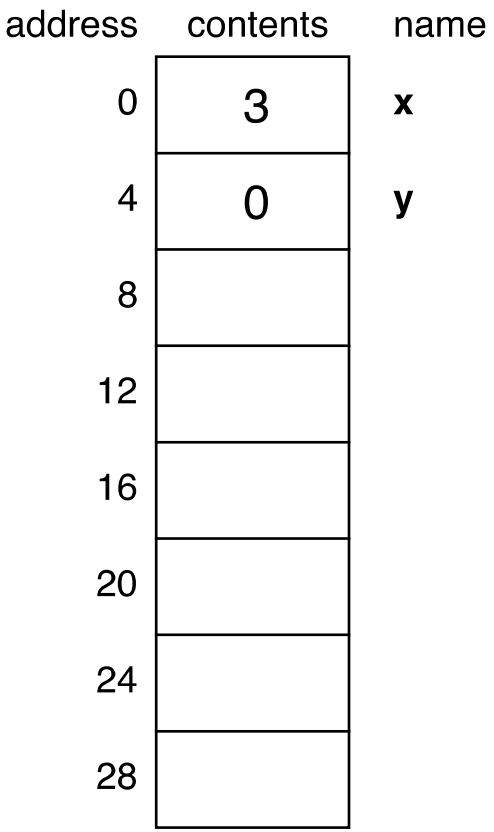
```
typedef double[10] vector10;
vector10 v={0,1,4,9,16,25,36,49,64,81};
```

Static memory allocation

- Declared variables are assigned to memory locations

```
int x=3;
int y=0;
```

- The variable name is a symbolic reference to the contents of some real memory location
  - It only exists for the compiler
  - No real existence in the computer



Pointers

- Pointers store the address of a memory location
  - are denoted by a * in front of the name

```
int* p; // pointer to an integer
```

  - Are initialized using the & operator

```
int i=3;
p = &i; // & takes the address of a variable
```

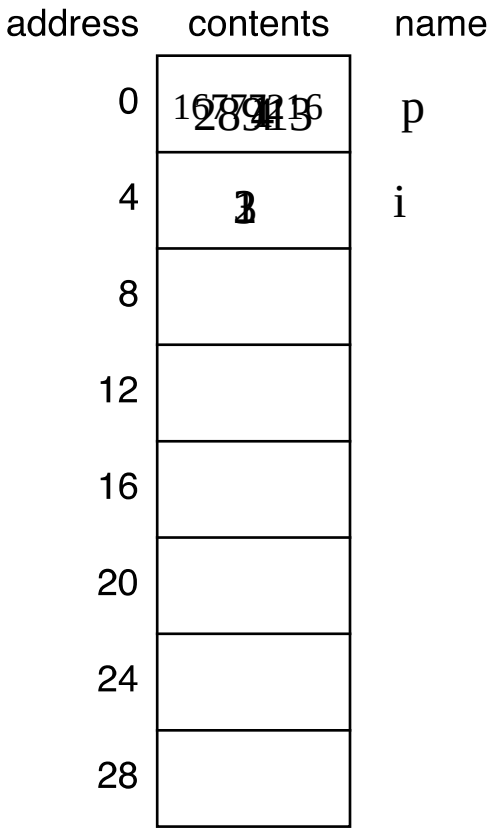
  - Are dereferenced with the * operator

```
*p = 1; // sets i=1
```

  - Can be dangerous to use

```
p = 1; // sets p=1: danger!
*p = 258; // now messes up everything, can
crash
```

- Take care: `int* p;` does not allocate memory!



Static Allocation

Automatic allocation

```
float x[10]; // allocates memory for 10 numbers
```

- will be deleted automatically when block `{}` is left

Allocation of flexible size

```
unsigned int n;
cin >> n;
float x[n];
// will not work
```

- The compiler has to know the number!

Dynamic allocation

- Solution: dynamic allocation

```
// allocate some memory for an array
float *x = new float[n];

x[0]=…;      // do some work

delete[] x; // delete the memory for the array.
// x[i], *x now undefined!
```

Pointer arithmetic

- for any pointer `T *p;` the following holds:
  - `p[n]` is the same as `*(p+n);`
- Adding and integer *n* to a pointer increments it by the *n* times the size of the type – and not by *n* bytes
- Be sure to only use valid pointers
  - initialize them
  - do not use them after the object has been deleted!
  - catastrophic errors otherwise

A look at memory: array example

- Array example

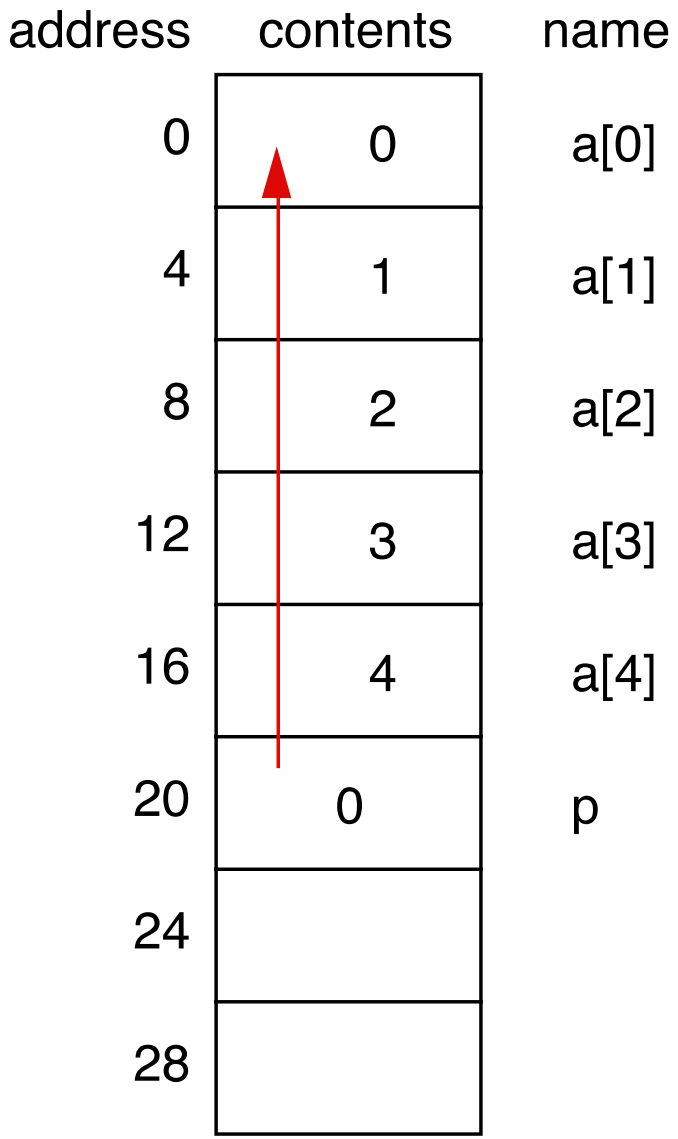
```
int array[5];

for (int i=0;i < 5; ++i)
  array[i]=i;
```

- Arrays are similar to pointers

```
int* p = array; // same as &array[0]
for (int i=0;i < 5; ++i)
  cout << *p++;


delete[] p; // will crash
array=0; // will not compile
p=0; // is OK
```



References

- are aliases for other variables:
  - are denoted by a `&` in front of the name

```
float very_long_variabe_name_for_number=0;

float& x = very_long_variabe_name_for_number;
// x refers to the same memory location

x=5; // sets very_long_variabe_name_for_number to 5;
```

- cannot be reset:

```
float y=2;
x=y; // sets very_long_variabe_name_for_number to 2;
     // does not set x to refer to y!
```

Function call

There are several kinds of function parameters:

- pass by value

```
double sqrt(double x)
```

- pass by reference

```
double sqrt(double& x)
```

- pass by const reference

```
double sqrt(double const& x)
```

- pass by pointer

```
double sqrt(double* x)
```

Pass by value

- The variable in the function is a copy of the variable in the calling program:

```
void f(int x) {
  x++; // increments x but not the variable of the calling
  program
  cout << x;
}

int main() {
  int a=1;
  f(a);
  cout << a; // is still 1
}
```

- Copying of variables time consuming for large objects like matrices

Pass by reference

- The function parameter is an alias for the original variable:

```
void increment(int& n) {
  n++;
}

int main() {
  int x=1; increment(x); // x now 2
  increment(5); // will not compile since 5
                // is literal constant!
}
```

- avoids copying of large objects:
  - `vector eigenvalues(Matrix& A);`
- but allows unwanted modifications!
  - the matrix A might be changed by the call to eigenvalues!

Pass by const reference

- Problem:
  - `vector eigenvalues(Matrix& A); // allows modification of A`
  - `vector eigenvalues(Matrix A); // involves copying of A`
- how do we avoid copying and prohibit modification?

```
vector eigenvalues (Matrix const &A);
```

  - now a reference is passed -> no copying
  - the parameter is const -> cannot be modified

Pass by pointer

- Similar to pass by reference
- Used mostly in C

```
vector eigenvalues(Matrix* m);
```

- rarely needed in C++

Templates and Function Overloading

Generic algorithms versus concrete implementations

- Algorithms are usually very generic:

  for min() all that is required is an order relation “<“

$$\min(x,y) = \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

- Most programming languages require concrete types for the function definition
  - C:

```
int min_int(int a, int b) { return a<b ? a : b;}
float min_float (float a, float b) { return a<b ? a : b;}
double min_double (double a, double b) { return a<b ? a : b;}
…
```

  - Fortran:

```
MIN(), AMIN(), DMIN(), …
```

Function overloading in C++

- solves one problem immediately: we can use the same name

```
int min(int a, int b) { return a<b ? a : b;}
float min (float a, float b) { return a<b ? a : b;}
double min (double a, double b) { return a<b ? a : b;}
```

- Compiler chooses which one to use

```
min(1,3); // calls min(int, int)
min(1.,3.); // calls min(double, double)
```

Generic algorithms using templates in C++

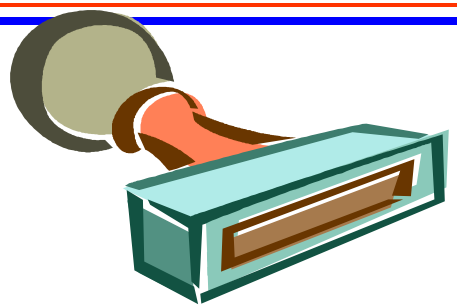
- C++ templates allow a generic implementation:

```
template <class T>
T min (T x, T y)
{
  return (x < y ? x : y);
}
```

$$\min(x,y) \text{ is } \begin{cases} x & \text{if } x < y \\ y & \text{otherwise} \end{cases}$$

Usage Causes Instantiation

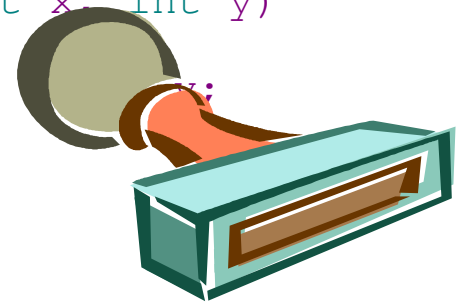
```
template <class T>
T min(T x, T y)
{
   return x < y ? x : y;
}
```



```
int x = min(3, 5);
int y = min(x, 100);
```

```
// T is int

int min<int>(int x, int y)
{
   return x < y ? x : y;
}
```



```
float z = min(3.14159f, 2.7182f);
```

```
// T is float

float min<float>(float x, float y)
{
   return x < y ? x : y;
}
```

Custom Types

Structs and Classes

Structs

- Plain old data structure (POD):

```
struct point
{
  double x;
  double y;
};
```

```
int main()
{
  point p;
  p.x = 1.0;
  p.y = 2.0;
  f(p)
}
```

Classes

```
class rectangle
{
public:
    rectangle(double bl_x,
        double bl_y, double tr_x, double tr_y)
    : x1(tl_x), y1(tl_y), x2(br_x), y2(br_y)
    {
        cout<< "new rectangle\n";
    }
    double area() const
    {
        return (x2-x1)*(y2-y1);
    }
    void move(double dx, double dy)
    {
        x1+=dx; x2+= dx; y1+=dy; y2+=dy;
    }
private:
    double x1,y1,x2,y2;
};
int main()
{
   rectangle x(0.0,0.0,1.0,2.0);
   double area = x.area();
   x.move(0.5,0.7);
   ...
}
```

- Custom types
- Constructor
- Member functions
  - `const` member functions can't modify data members
- `private`/`public`:

  Access from outside the class ?
- Data members

Special member functions

```
class rectangle
{
public:
   rectangle(double bl_x,
      double bl_y, double tr_x, double tr_y)
   :x1(tl_x),y1(tl_y),x2(br_x),y2(br_y)
   {
      cout << "new rectangle\n";
   }
   rectangle()
   :x1(0),y1(0),x2(0),y2(0)
   {
      cout << "default rectangle\n";
   }
   ~rectangle()
   {
      cout << "destroying rectangle\n";
   }
   rectangle(rectangle const& r)
   :x1(r.x1),y1(r.y1),x2(r.x2),y2(r.y2)
   {
      cout << "copying rectangle\n";
   }
   ...
int main()
{
   rectangle x(0.0,0.0,1.0,2.0);
   rectangle y;     // default construct
   rectangle z(x); // copy
}
```

Default Constructor

Destructor (how to destroy a rectangle)

Copy constructor (how to copy a rectangle)

Special member functions 2

```
class rectangle
{
public:
      ...
      rectangle(rectangle const& r)
      :x1(r.x1),y1(r.y1),x2(r.x2),y2(r.y2)
      {
            cout << "copying rectangle\n";
      }
      rectangle& operator = (rectangle const& r)
      {
            x1 = r.x1;
            y1 = r.y1;
            x2 = r.x2;
            y2 = r.y2;
            return *this
      }
```

Assignment operator

The object itself

returning it allows us to do

```
int main()
{
   rectangle y;
   rectangle x(0.0,0.0,1.0,2.0);
   rectangle z(5.3,4.5,8.9,9.9);
   rectangle a;
   a=z;
   x=y=z;
}
```

Standard Library

Standard library

- Containers:
  - `std::vector`
  - `std::map`
  - `std::string`
  - ...
- Algorithms:
  - `copy(...)`
  - `sort(...)`
  - `accumulate(...)`
  - ...
- Threads
- Input/Output
- ...

std::vector

- Flexible, resizeable vector/array

```
#include <vector>

std::vector<float> v(size, init_value);

v[2] = 2;
v[99999] = 3;  // Don't write beyond the end! No checks!
               // => CRASH!
v.resize(10);
v.push_back(0.4); // Append an element

cout << v.size(); // will print 11
v.clear();        // erase all elements
```

More functions on: cppreference.com

Iterators

```
vector<double> v(100);
```

- Fast way to iterate through an vector/array:

```
for(double* p=&v[0]; p < &v[0]+100; ++p)
   *p += 0.5;
```

- More generic concept: Iterators

```
for(vector<double>::iterator it=v.begin(); it != v.end(); ++it)
    *it += 0.5;
```

- Why cool? Works with different data structures too:

```
for(std::map<int,double>::iterator it = m.begin(); it != m.end(); ++it)
```

Iterators

```
vector<double> v(100);
```

- Loop is great... But looks ugly:

```
for(vector<double>::iterator it=v.begin(); it != v.end(); ++it)
    *it += 0.5;
```

- C++11:

```
for(double& x : v)
    x += 0.5;
```

Algorithms

```
#include <algorithm>
vector<double> v(100);
vector<double> v2(100);
```

- Usually built on iterators

```
copy(v.begin(), v.end(), v2.begin() );
sort(v.begin(), v.end());

double mean = accumulate(v.begin(),v.end(),0.0) / v.size();
```

- Sort by special criterion

```
sort(v.begin(), v.end(), std::greater<double>());
```

- Long list of useful algorithms: cppreference.com

Where to read on

- Tutorial:
  - cplusplus.com
- Reference:
  - cplusplus.com
  - cppreference.com
  - sgi.com/tech/stl
- Books:
  - Beginners:
    - Stanley B. Lippman, Essential C++, Addison Wesley 2000
    - Andrew Koenig and Barbara E. Moo, Accelerated C++, Addison Wesley 2000
  - More books: http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- Lecture:
  - Prof. Troyer: Programming techniques for scientific computing (this semester!)