∼

Next steps in C++

Scientific Computing Winter 2016/2017

Lecture 3

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

With material from from `http://www.cplusplus.com/`

made wit pandoc

$\sim$

Recap from last time

Von Neumann architecture

- Data and instructions in same memory
- decode-fetch-compute-store cycle
- Hierarchical memory system
    - Random access memory bandwith is slow
    - Latency (start-up time) of memory transfer is large
    - Fast but expensive cache memory used for intermediate storage of often used data
    - Data transfer between cache and RAM in larger chunks – `cache lines`

C++ : scalar data types

```
|--------------------+---------+-------+------+----------------------+----------------------|
| name               | fmt     | bytes | bits |                  min |                  max |
|--------------------+---------+-------+------+----------------------+----------------------|
| char               | %c (%d) |     1 |    8 |                 -128 |                  127 |
| unsigned char      | %c (%d) |     1 |    8 |                    0 |                  255 |
| short int          | %d      |     2 |   16 |               -32768 |                32767 |
| unsigned short int | %u      |     2 |   16 |                    0 |                65535 |
| int                | %d      |     4 |   32 |          -2147483648 |           2147483647 |
| unsigned int       | %u      |     4 |   32 |                    0 |           4294967295 |
| long int           | %ld     |     8 |   64 | -9223372036854775808 |  9223372036854775807 |
| unsigned long int  | %lu     |     8 |   64 |                    0 | 18446744073709551615 |
| float              | %e      |     4 |   32 |         1.175494e-38 |         3.402823e38 |
| double             | %e      |     8 |   64 |        2.225074e-308 |        1.797693e308 |
| long double        | %Le     |    16 |  128 |       3.362103e-4932 |       1.189731e4932 |
| bool               | %d      |     1 |    8 |                    0 |                    1 |
|--------------------+---------+-------+------+----------------------+----------------------|
```

- Type sizes are the "usual ones" on 64bit systems. The standard only guarantees that
  `sizeof(short ...) <= sizeof(...) <=sizeof(long ...)`
  - E.g. on embedded systems these may be different
- Declaration and output (example)

```cpp
#include <cstdio>
...
int i=3;
double x=15.0;
std::printf("i=%d, x=%e\n",i,x);
```

Scopes, Declaration, Initialization

- **All variables are typed and must be declared**
  - Declared variables “live” in scopes defined by braces `{ }`
  - Good practice: initialize variables along with declaration
  - “auto” is a great innovation in C++11 which is useful with complicated types which arise in template programming
    - type of *lvalue* (left hand side value) is detected from type of *rvalue* (value at the right hand side)

```
{
  auto i=15;  // int
  auto i=15u; // unsigned int
  auto i=15l; // long  int

  auto z=1.00e5; // double
  auto x=1.0e5f; // float


}
```

Functions

- Functions have to be *declared* as any other variable
- '`(...)` holds parameter list
  - each parameter has to be defined with its type
- type part of declaration describes type of return value
  - `void` for returning nothing

```
double multiply(double x, double y);
```

- Functions are *defined* by attaching a scope to the declaration
  - *Values of parameters are copied into the scope*

```
double multiply(double x, double y)
{
   return x*y;
}
```

- Functions are *called* by statements invoking the function with a particular set of parameters

```
{
    double s=3.0, t=9.0;
    double result=multiply(s,t);
    printf("s=%e, t=%e, s*t= %e\n",s,t,result); // s and t keep their values
}
```

Addresses and pointers

- Objects are stored in memory, and in order to find them they have an *address*
- We can determine the address of an object by the `&` operator
  - The result of this operation can be assigned to a variable called *pointer*
  - “pointer to type x” is another type denoted by `*x`
- Given an address (pointer) object we can refer to the content using the `*` operator

```
int i=15;  // i is an object
int *j= &i; // j is a pointer
int k=*j; // now, k=15
```

- The `nullptr` object can be assigned to a pointer in order to indicate that it points to “nothing”

```
int *p=nullptr;
```

- Instead of values, addresses can be passed to functions

```
void do_multiplication(double x, double y, double *result)
{
    *result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,&result) // result now contains 45
```

Language elements so far

- Scalar data types
- Addresses, pointers
- Functions
- Flow control
- Printing

Preprocessor, modules, compiler, arrays

The Preprocessor

- Before being sent to the compiler, the source code is sent through the *preprocessor*
- It is a legacy from C which is slowly being squeezed out of C++
- Preprocessor commands start with `#`
- Include contents of file `file.h` found on a default search path known to the compiler:

```
#include <file.h>
```

- Include contents of file `file.h` found on user defined search path

```
#include "file.h"
```

- Define a piece of text (mostly used for constants in pre-C++ times), Avoid! Use `const` instead.

```
#define N 15
```

- Define preprocessor macro for inlining code.
  Avoid! Use inline functions instead

```
#define  MAX(X,Y) (((x)>(y))?(x):(y))
```

Conditional compilation and pragmas

- Conditional compilation of pieces of source code, mostly used to dispatch between system dependent variant of code. Rarely necessary nowadays. . .

```
#ifdef MACOSX
   statements to be compiled only for MACOSX
#else
   statements for all other systems
#endif
```

- There are more complex logic involving constant expressions
- A pragma gives directions to the compiler concerning code generarion

```
#pragma omp parallel
```

Headers and namespaces

- If we want to use functions from the standard library we need to include a *header file* which contains their declarations
  - The #include statement comes from the C-Preprocessor and leads to the inclusion of the file referenced therein into the actual source
  - Include files with names in < > brackets are searched for in system dependent directories known to the compiler

```
#include <iostream>
```

- Namespaces allow to prevent clashes between names of functions from different projects
  - All functions from the standard library belong to the namespace std

```
namespace foo
{
    void cool_function(void);
}

namespace bar
{
    void cool_function(void);
}

...

{
    using namespace bar;
    foo::cool function()
    cool_function() // equivalent to bar::cool_function()
}
```

Emulating modules

- Until now C++ has no well defined module system.
- A module system usually is emulated using the preprocessor and namespaces. Here we show the ideal way to do this
- File `mymodule.h` containing interface declaratiions

```
#ifndef MYMODULE_H
#define MYMODULE_H
namespace mymodule
{
     void my_function(int i, double x);
}
#endif
```

- File `mymodule.cpp` containing function definitions

```
#include "mymodule.h"
namespace mymodule
{
      void my_function(int i, double x)
      {
          ...body of function definition...
      }
}
#endif
```

- File using `mymodule`:

```
#include "mymodule.h
...
mymodule::my_function(3,15.0);
```

main

Now we are able to write a complete program in C++

- `main()` is the function called by the system when running the program. Everything else needs to be called from there.
- Assume the follwing content of the file `run42.cxx`:

```cpp
#include <cstdio>

int main(int argc, char** argv)
{
   int i=4,j=2;
   int answer=10*4+2;
   printf("Hello world, the answer is %d!\n",answer);
   return 0;
}
```

Then the sequence of command line commands

```
$ g++ -o run42 run42.cxx
$ ./run42
```

gives the right answer to (almost) anything.

Command line instructions to control compiler

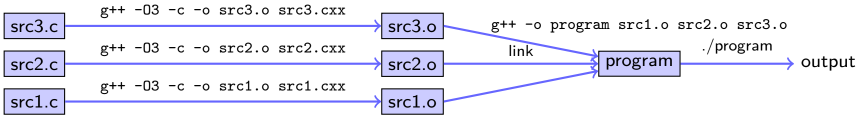
- By default, the compiler command performs the linking process as well
- Compiler command (Linux)

g++	GNU C++ compiler
g++-5	GNU C++ 5.x
clang++	CLANG compiler from LLVM project
icpc	Intel compiler

- Options (common to all of those named above, but not standardized)

-o name	Name of output file
-g	Generate debugging instructions
-O0, -O1, -O2, -O3	Optimization levels
-c	Avoid linking
-I<path>	Add <path> to include search path
-D<symbol>	Define preprocessor symbol
-std=c++11	Use C++11 standard

Compiling. . .


src3.c
g++ -O3 -c -o src3.o src3.cxx
src3.o
src2.c
g++ -O3 -c -o src2.o src2.cxx
src2.o
src1.c
g++ -O3 -c -o src1.o src1.cxx
src1.o
g++ -o program src1.o src2.o src3.o
link
program
./program
output

```
$ g++ -O3 -c -o src3.o src3.cxx
$ g++ -O3 -c -o src2.o src2.cxx
$ g++ -O3 -c -o src1.o src1.cxx
$ g++ -o program src1.o src2.o src3.o
$ ./program
```

Shortcut: invoke compiler and linker at once

```
$ g++ -O3 -o program src1.cxx src2.cxx src3.cxx
$ ./program
```

Arrays

- Focusing on numerical methods for PDEs results in work with finite dimensional vectors which are represented as *arrays* - sequences of consecutively stored objects
- Stemming from C, in C++ array objects represent just the fixed amount of consecutive memory. No size info or whatsoever
- No bounds check
- First array index is always 0

```
double x[9]; // uninitialized array of 9 elements
double y[3]={1,2,3}; // initialized array of 3 elements
double z[]={1,2,3};  // Same
double z[]{1,2,3};   //Same
```

- Accessing arrays
    - [] is the array access operator in C++
    - Each element of an array has an index

```
double a=x[3];  // undefined value because x was not initialized
double b=y[12]; // undefined value because out of bounds
y[12]=19;       // may crash program ("segmentation fault"),
double c=z[0];  // Acces to first element in array, now c=1;
```

Arrays, pointers and pointer arithmetic

- Arrays are strongly linked to pointers
- Array object can be treated as pointer

```
double x[]={1,2,3,4};
double b=*x; // now x=1;
double *y=x+2; // y is a pointer to third value in arrax
double c=*y;   // now c=3
ptrdiff_t d=y-x; // We can also do differences between pointers
```

- Pointer arithmetic is valid only in memory regions belonging to the same array

Arrays and functions

- Arrays are passed by passing the pointer referring to its first element
- As they contain no length information, we need to pass that as well

```
void func_on_array1( double[] x, int len);
void func_on_array2( double* x, int len); // same
void func_on_array3( const double[]  x, int len); // same, but does not allow to change x
...
double x[]={3,4,5};
int len=sizeof(x)/sizeof(x[0]);
func_on_array1(x,len);
```

- Be careful with array return

```
double * some_func(void)
{
  double a[]={-1,-2,-3};
  return a; // illegal as with the end of scope, the life time of a is over
            // smart compilers at least warn
}
```

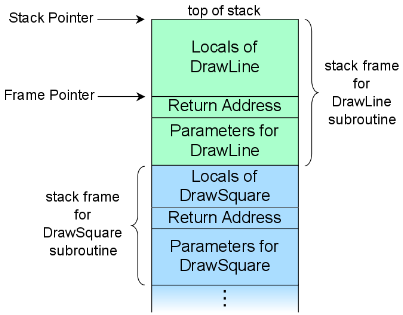
- This one is also illegal, though often compilers accept it

```
void another_func(int n)
{
    int b[n];
}
```

- Even in main() this will be illegal. How can we work on problems where size information is obtained only during runtime, e.g. user input ?

Memory: stack

- pre-allocated memory where `main()` and all functions called from there can put their data.
  - Any time a function is called, the current position in the instruction stream is stored in the stack as the return address, and the called function is allowed to work with the memory space after that



By R. S. Shaw, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1956587

- Stack space should be considered scarce - Stack size for a program is set by the system - On UNIX, use `ulimit -s` to check/set stack size
- All previous examples had their data on the stack, even large arrays;

Memory: heap

- Chunks from all remaining free system memory can be reserved – “allocated” – on demand in order to provide memory space for objects
- `new` reserves the memory and returns an address which can be assigned to a pointer variable
- `delete` (`delete[]`) for arrays releases this memory
- Compared to declarations on the stack, these operations are expensive
- Use cases:
  - Problem sizes unknown at compile time
  - Large amounts of data
  - ... so, yes, we will need this...

```
double *x= new double(5); // allocate space for a double and initialize this with 5
double *y=new double[5];  // allocate space of five doubles, uninitialized
x[3]=1;                   // Segmentation fault
y[3]=1;                   // Perfect...
delete x;                 // Choose the right delete!
delete[] y;                // Choose the right delete!
```

Multidimensional Arrays

- It is easy to declare multidimensional array on the stack when the size the array is known at compile time

```
double x[5][6];

for (int i=0;i<5;i++)
   for (int j=0;j<6;j++)
      x[i][j]=0;
```

- Determining array dimensions from function parameters may work with some compilers, but are not backed by the standard!

Intermediate Summary

- This was mostly all (besides structs) of the C subset of C++
  - Most “old” C libraries and code written in previous versions of C++ are mostly compatible to modern C++
- you will find many “classical” programs around which use the “(int size, double*data)" way of doing things, especially in numerics
  - UMFPACK, Pardiso direct solvers
  - PetsC library for distributed linear algebra
  - triangle, tetgen mesh generators
  - ...
- On this level it is possible to call Fortran programs from C++, and you might want to do this too:
  - BLAS, LAPACK dense matrix linear algebra
  - ARPACK eigenvalue computations
  - ...
- The C++ in C++ will follow anyway

Getting "real" with C++

Classes and members

- Classes are data types which collect different kinds of data, and methods to work on them.

```
class class_name
{
  private:
    private_member1;
    private_member2;
    ...
  public:
    public_member1;
    public_member2;
    ...
};
```

- If not declared otherwise, all members are private
- `struct` data types are defined in the same way as classes, but by default all members are public
- Accessing members of a class object:

```
class_name x;
x.public_member1=...
```

- Accessing members of a pointer to class object:

```
class_name *x;
(*x).public_member1=...
x->public_member1=...
```

Example class

- Define a class `vector` which holds data and length information

```
class vector
{
  private:
      double *data;
  public:
       int size;
      double get_value( int i) {return data[i];};
      void set_value( int i, double value); {data[i]=value;};
};

...

{
  vector v;
  v.data=new double(5);  // would work if data would be public
  v.size=5;
  v.set_value(3,5);

  b=v.get_value(3); // now, b=5
  v.size=6;  // we changed the size, but not the length of the data array...
             // and who is responsible for delete[] at the end of scope ?
}
```

- Methods of a class know all its members
- It would be good to have a method which constructs the vector and another one which destroys it.

Constructors and Destructors

```
class vector
{
  private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      double get_value( int i) {return data[i];};
      void set_value( int i, double value); {data[i]=value;};

Vector( int new_size) { data = new double[size];}
      ~Vector() { delete [] data;}
};

...

{
  vector v(5);
  for (int i=0;i<5;i++) v.set_value(i,0.0);

  v.set_value(3,5);
  b=v.get_value(3); // now, b=5
  v.size=6;  // Size is now private and can not be set;
  // Destructor is automatically called at end of scope.
  vector w(5);

  for (int i=0;i<5;i++) w.set_value(i,v.get_value(i));
}
```

- Constructors are declared as `classname(...)`
- Destructors are declared as `~classname()`

Interlude: References

- C style access to objects is direct or via pointers
- C++ adds another option - references
    - References essentially are alias names for already existing variables
    - Must always be initialized
    - Can be used in function parameters and in return values
    - No pointer arithmetics with them
- Declaration of refrence

```
double a=10.0;
double &b=a;

b=15; // a=15 now as well
```

- Reference as function parameter: no copying of data!

```
void do_multiplication(double x, double y, double &result)
{
    result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,result) // result now contains 45
```

Vector class again

- We can define () and [] operators!

```
class vector
{
  private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      double & operator()(int i) { return data[i]);
      double & operator[](int i) { return data[i]);
      vector( int new_size) { data = new double[size];}
      ~vector() { delete [] data;}
};

...

{
  vector v(5);
  for (int i=0;i<5;i++) v[i]=0.0;

  v[3]=5;
  b=v[3]; // now, b=5
  vector w(5);

  for (int i=0;i<5;i++) w(i)=v(i);

// Destructors are automatically called at end of scope.
}
```

Matrix class

- We can define `(i,j)` but not `[i,j]`

```
class matrix
{
  private:
      double *data=nullptr;
      int size=0;
      int nrows=0;
      int ncols=0;
  public:
      int get_size( return size);
      int get_nrows( return nrows);
      int get_ncols( return ncols);
      double & operator()(int i,int j) { return data[i*nrow+j]);
      matrix( int new_rows,new_cols) { nrows=new_rows;ncols=new_cols; size=nrows*ncols; data = new double
      ~matrix() { delete [] data;}
};

...
{
  matrix m(3,3);
  for (int i=0;i<3;i++)
  for (int j=0;j<3;j++)
      m(i,j)=0.0;
}
```

Generic programming: templates

- We want do be able to have vectors of any basic data type.
- We do not want to write new code for each type

```cpp
template <typename T>
class vector
{
  private:
      T *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      T & operator[](int i) { return data[i]);
      vector( int new_size) { data = new T[size];}
      ~vector() { delete [] data;}
};
...
{
  vector<double> v(5);
  vector<int> iv(3);
}
```

C++ template libray

- the standard template library (STL) became part of the C++11 standard
- whenever you can, use the classes available from there
- for one-dimensional data, std::vector is appropriate
- for two-dimensional data, things become more complicated
    - There is no reasonable matrix class
        - std::vector< std::vector> is possible but has to allocate each matrix row and is inefficient
    - it is not possible to create an std::vector from already existing data