Sequential Architecture, Programming languages (C++)

Scientific Computing Winter 2016/2017

Part I

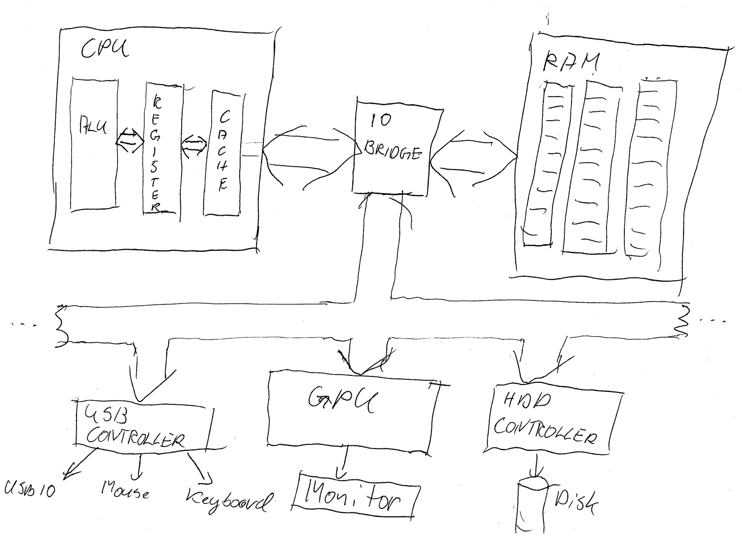
Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

With material from “Introduction to High-Performance Scientific Computing” by Victor Eijkhout (http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html), http://www.cplusplus.com/, “Expression templates revisited” by K. Iglberger

Sequential hardware

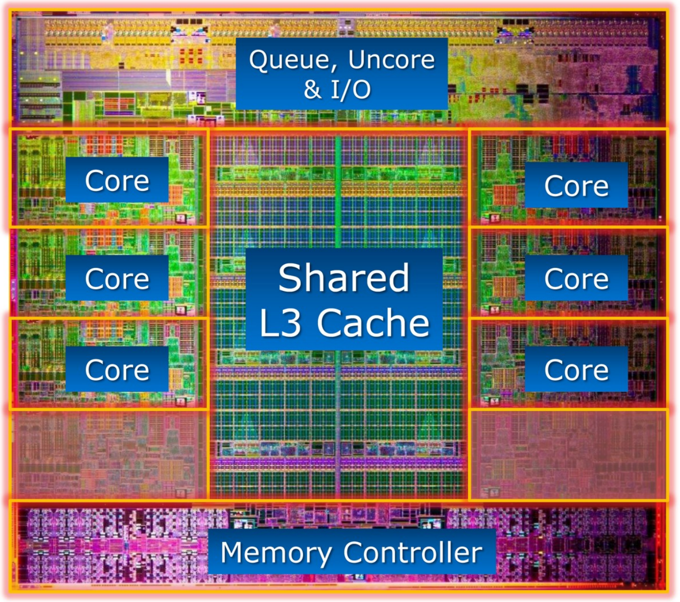
von Neumann Architecture



- Data and instructions from same memory
  - Instruction decode: determine operation and operands
  - Get operands from memory
  - Perform operation
  - Write results back
  - Continue with next instruction

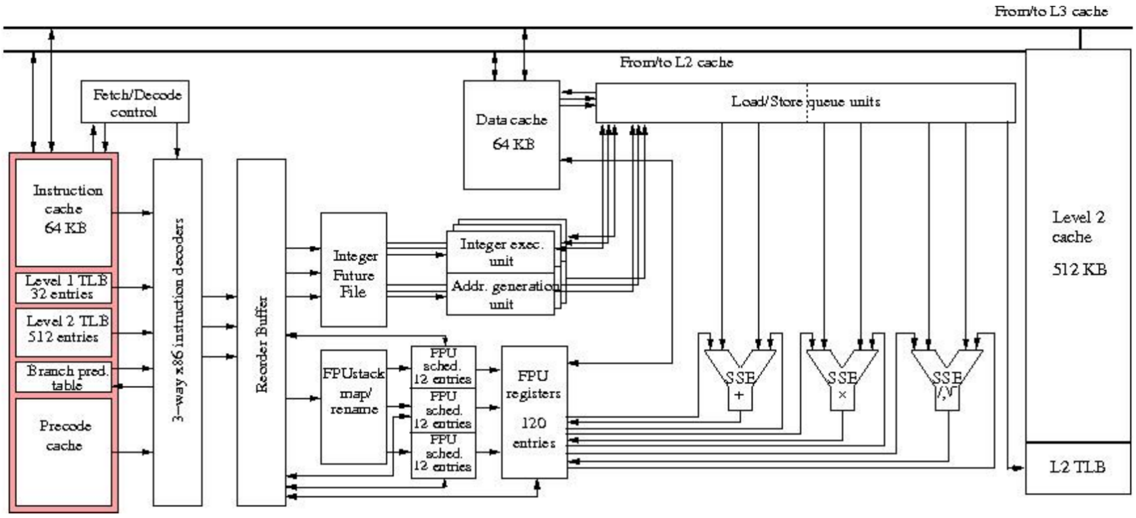
Contemporary Architecture

- Multiple operations simultaneously “in flight”
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively



Modern CPU. From: https://www.hartware.de/review_1411_2.html

What is in a “core” ?



From: Eijkhout

Modern CPU functionality

- Traditionally: one instruction per clock cycle
- Modern CPUs: Multiple floating point units, for instance 1 Mul + 1 Add, or 1 FMA
  - Peak performance is several operations /clock cycle
  - Only possible to obtain with highly optimized code
- Pipelining
  - A single floating point instruction takes several clock cycles to complete:
  - Subdivide an instruction:
    - Instruction decode
    - Operand exponent align
    - Actual operation
    - Normalize
  - Pipeline: separate piece of hardware for each subdivision
  - Like assembly line

Memory Hierachy

- Main memory access is slow compared to the processor
  - 100–1000 cycles latency before data arrives
  - Data stream maybe 1/4 floating point number/cycle;
  - processor wants 2 or 3
- Faster memory is expensive
- *Cache* is a small piece of fast memory for intermediate storage of data
- Operands are moved to CPU *registers* immediately before operation
- Data is always accessed through the hierarchy
  - From egisters where possible

```
    - Then the caches (L1, L2, L3)

               - Then main memory
```

Machine code

- Detailed instructions for the actions of the CPU
- Not human readable
- Sample types of instructions:
  - Transfer data between memory location and register
  - Perform arithmetic/logic operations with data in register
  - Check if data in register fulfills some condition
  - Conditionally change the memory address from where instructions are fetched ≡ "jump" to address
  - Save all register context and take instructions from different memory location until return ≡ "call"

```
534c 29e5 31db 48c1 fd03 4883 ec08 e85d
feff ff48 85ed 741e 0f1f 8400 0000 0000
4c89 ea4c 89f6 4489 ff41 ff14 dc48 83c3
0148 39eb 75ea 4883 c408 5b5d 415c 415d
415e 415f c390 662e 0f1f 8400 0000 0000
f3c3 0000 4883 ec08 4883 c408 c300 0000
0100 0200 4865 6c6c 6f20 776f 726c 6400
011b 033b 3400 0000 0500 0000 20fe ffff
8000 0000 60fe ffff 5000 0000 4dff ffff
```

Assembler code

- Human readable representation of CPU instructions
- Some write it by hand ...
    - Code close to abilities and structure of the machine
    - Handle constrained resources (embedded systems, early computers)
- Translated to machine code by *assembler*

```
    .file   "code.c"
    .section    .rodata
.LC0:
    .string "Hello world"
    .text
    ...
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
```

Registers

Processor instructions operate on registers directly - have assembly language names names like: `eax, ebx, ecx`, etc. - sample instruction: `addl  %eax, %edx`

- Separate instructions and registers for floating-point operations

Data caches

- Between the CPU Registers and main memory
- L1 Cache: Data cache closest to registers
- L2 Cache: Secondary data cache, stores both data and instructions
- Data from L2 has to go through L1 to registers
- L2 is 10 to 100 times larger than L1
- Some systems have an L3 cache, ~10x larger than L2

Cache line

- The smallest unit of data transferred between main memory and the caches (or between levels of cache)
- N sequentially-stored, multi-byte words (usually N=8 or 16).
- If you request one word on a cache line, you get the whole line
  - make sure to use the other items, you've paid for them in bandwidth
  - Sequential access good, "strided" access ok, random access bad
- Cache hit: location referenced is found in the cache
- Cache miss: location referenced is not found in cache
  - triggers access to the next higher cache or memory
- Cache thrashing
  - Two data elements can be mapped to the same cache line: loading the second "evicts" the first
  - Now what if this code is in a loop? "thrashing": really bad for performance
- Performance is limited by data transfer rate
  - High performance if data items are used multiple times

∼

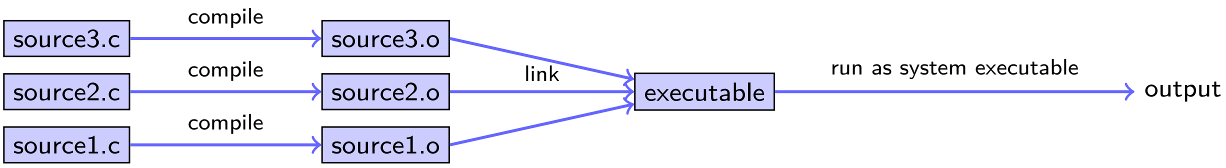
"Language Philosophy"

Compiled high level languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Translated to machine code (resp. assembler) by *compiler*

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  printf("Hello world");
}
```

- “Far away” from CPU ⇒ the compiler is responsible for creation of optimized machine code
- Fortran, COBOL, C, Pascal, Ada, Modula2, C++, Go, Rust, Swift
- Strongly typed
- Tedious workflow: compile - link - run



source3.c → compile → source3.o

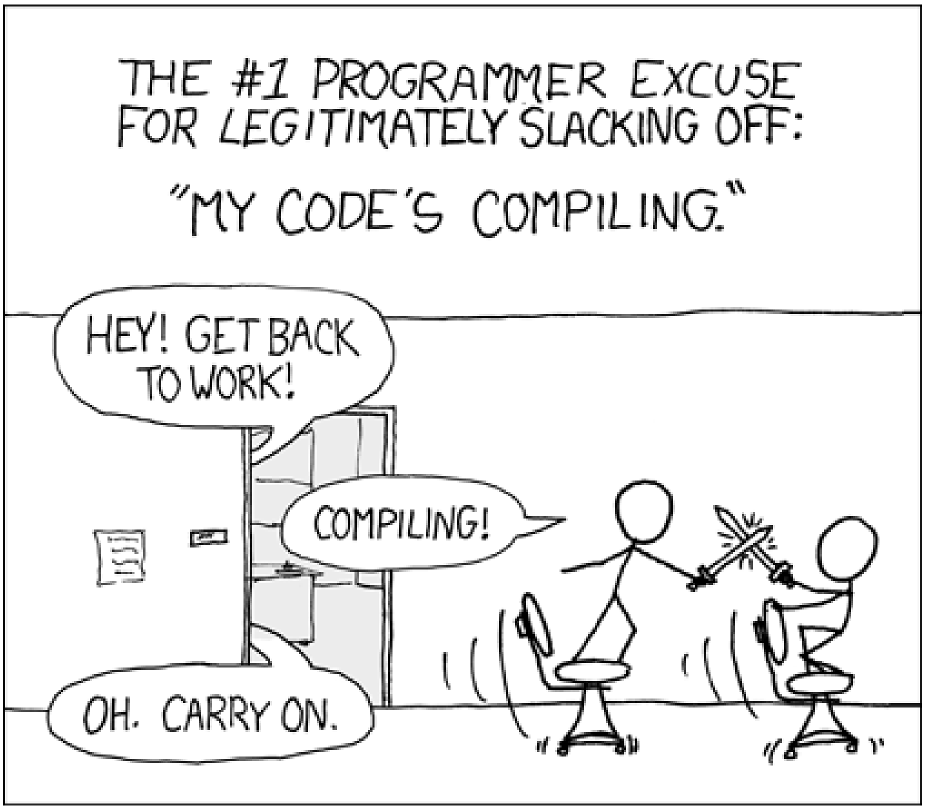
source2.c → compile → source2.o

source1.c → compile → source1.o

source3.o, source2.o, source1.o → link → executable

executable → run as system executable → output

Compiling. . .



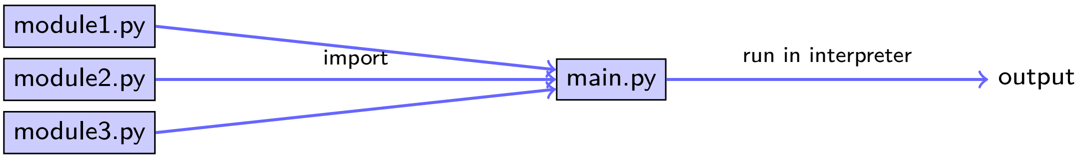
. . . from xkcd

High level scripting languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Need intepreter in order to be executed

```
print("Hello world")
```

- Very far away from CPU ⇒ usually significantly slower compared to compiled languages
- Matlab, Python, Lua, perl, R, Java, javascript
- Less strict type checking, often simple syntax, powerful introspection capabilities
- Immediate workflow: "just run"
  - in fact: first compiled to *bytecode* which can be interpreted more efficiently


module1.py

module2.py

module3.py

import

main.py

run in interpreter

output

JITting to the future ?

- As described, all modern interpreted language first compile to bytecode wich then is run in the interpreter
- Couldn't they be compiled to machine code instead? – Yes, they can: Use a just in time (JIT) compiler!
  - V8 → javascript
  - LLVM Compiler infrastructure → Python/NUMBA, **Julia** (currently at 0.5)
  - LuaJIT
  - Java
  - Smalltalk
- Drawback over compiled languages: compilation delay at every start, can be mediated by caching
- Potential advantage over compiled languages: *tracing* JIT, i.e. optimization at runtime
- Still early times, but watch closely. . .

Compiled languages in Scientific Computing

- Fortran: FORmula TRANslator (1957)
  - Fortran4: really dead
  - Fortran77: large number of legacy libs: BLAS, LAPACK, ARPACK . . .
  - Fortran90, Fortran2003, Fortran 2008
    - Catch up with features of C/C++ (structures,allocation,classes,inheritance, C/C++ library calls)
    - Lost momentum among new programmers
    - Hard to integrate with C/C++
    - In many aspects very well adapted to numerical computing
    - Well designed multidimensional arrays
- C: General purpose language
  - K&R C (1978) weak type checking
  - ANSI C (1989) strong type checking
  - Had structures and allocation early on
  - Numerical methods support via libraries
  - Fortran library calls possible
- C++: *The* powerful object oriented language
  - Superset of C (in a first approximation)
  - Classes, inheritance, overloading, templates (generic programming)
  - C++11: Quantum leap: smart pointers, threads, lambdas, initializer lists in standard
  - With great power comes the possibility of great failure. . .

Summary

- Compiled languages important for high performance
- Fortran lost its momentum, but still important due to huge amount of legacy libs
- C++ highly expressive, ubiquitous, significant improvements in C++11

∼

First steps in C++

- most things in this part (except iostreams and array initializers) are valid C as well

Printing stuff

- IOStream library
  - “Official” C++ output library
  - Type safe
  - Easy to extend
  - Clumsy syntax for format control

```
#include <iostream>
...

std::cout << "Hello world" << std::endl;
```

- C Output library
  - C Output library
  - Supported by C++-11 standard
  - No type safety
  - Hard to extend
  - Short, relatively easy syntax for format control
  - Same format specifications as in Python

```
#include <cstdio>
...

std::printf("Hello world\n");
```

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)

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

Typed constant expressions

- C++ has the ability to declare variables as constants:

```
const int i=15;
i++; // attempt to modify value of const object leads to
     // compiler error
```

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)

```
{
   int i=3;
   double x=15.0;
   auto y=33.0;
}
```

Arithmetic operators

- Assignment operator

```
a=b;
c=(a=b);
```

- Arithmetic operators +, -, *, /, %
- Beware of precedence which (mostly) is like in math!
- If in doubt, use brackets, or look it up!

```
 +  addition
 -  subtraction
 *  multiplication
 /  division
 %  modulo
```

- Compund assignment: +=, -=, *=, /=, %=

```
x=x+a;
x+=a; // equivalent to =x+a
```

- Increment and decrement: ++,--

```
y=x+1;
y=x++; // equivalent to y=x; x=x+1;
y=++x; // equivalent to x=x+1; y=x;
```

Further operators

- Relational and comparison operators ==, !=, >, <, >=, <=
- Logical operators !, &&, ||
  - short circuit evaluation:
    - if a in a&&b is false, the expression is false and b is never evaluated
    - if a in a||b is true, the expression is true and b is never evaluated
- Conditional ternary operator ?

```
c=(a<b)?a:b; // equivalent to the following
if (a<b) c=a; else c=b;
```

- Comma operator ,

```
c=(a,b); // evaluates to c=b
```

- Bitwise operators &, |, ^, ~, <<, >>
- sizeof: memory space (in bytes) used by the object resp. type

```
n=sizeof(char); //  evaluae
```

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
```

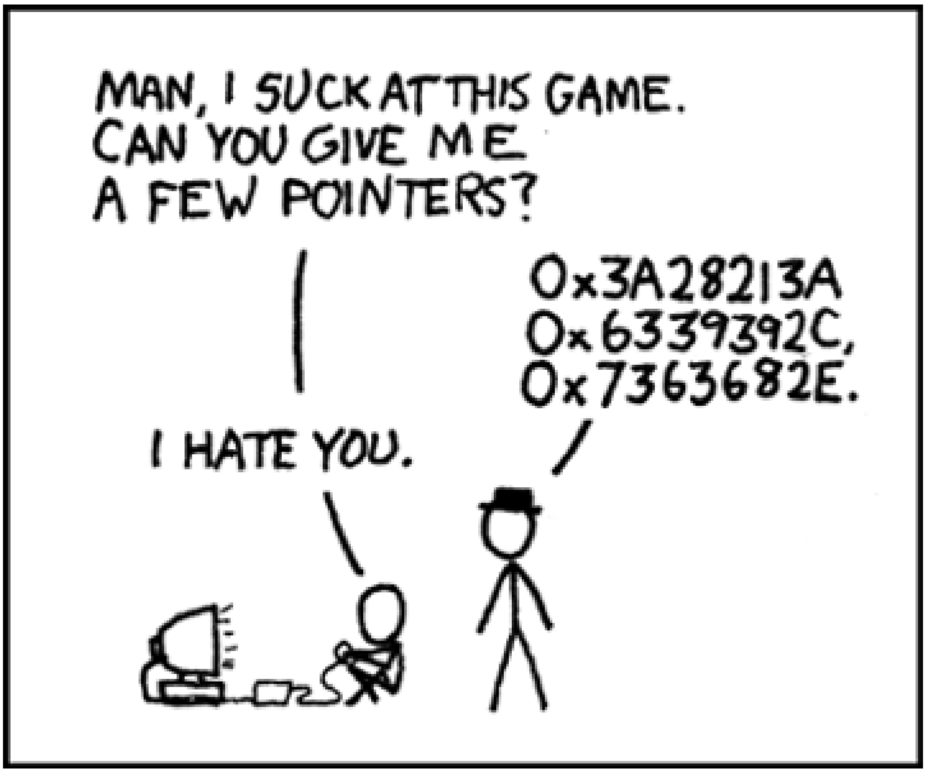
- The `nullptr` object can be assigned to a pointer in order to indicate that it points to "nothing"

```
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
```

Pointers. . .


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

. . . from xkcd

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
}
```

Functions: inlining

- Function calls sometimes are expensive compared to the task performed by the function
  - The compiler may include the content of functions into the instruction stream instead of generating a call

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

Flow control: Statements and simple loops

- Statements are individual expressions like declarations or instructions or sequences of statements enclosed in curly braces: `{}`:
  `{ statement1; statement2; statement3; }`
- Conditional execution: if
  `if (condition) statement;`
  `if (condition) statement; else statement;`

```
if (x>15)
    printf("error");
else
{
   x++;
}
```

- While loop:
  `while (condition) statement;`

```
i=0;
while (i<9)
{
  printf("i=%d\n",i);
  i++;
}
```

- Do-While loop: do statement while (condition);

Flow control: for loops

- This is the most important kind of loops for: numerical methods.
  for (initialization; condition; increase) statement;
  1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
  2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
  3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }
  4. increase is executed, and the loop gets back to step 2.
  5. The loop ends: execution continues by the next statement after it.
- All elements (initialization, condition, increase, statement) can be empty

```
for (int i=0;i<9;i++)   printf("i=%d\n",i); // same effect as previous slide
for(;;);  // completely valid, runs forever
```

Flow control: break, continue

- break statement: “premature” end of loop

```
for (int i=1;i<10;i++)
{
   if (i*i>15) break;
}
```

- continue statement: jump to end of loop body

```
for (int i=1;i<10;i++)
{
   if (i==5) continue;
   else do_someting_with_i;
}
```

Flow control: switch

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

equivalent to

```
if      (expression==constant1) {group-of-statements-1;}
else if (expression==constant2) {group-of-statements-2;}
...
else                            {default-group-of-statements;}
```

Language elements so far

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

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))
```

Why macros are evil ?

(Argumentation from stackoverflow)

- You can not debug macros.
  - a debugger allows to execute the the program statement by statement in order to find errors. Within macros, this is not possible
- Macro expansion can lead to strange side effects.

```
#define  MAX(x,y) (((x)>(y))?(x):(y))
auto a=5, b=4;
auto c=MAX(++a,b);       // gives c=7
auto d=std::max(++a,b);  // gives d=6
```

- Macros have no “namespace”, so it is easy to “replace” functions without notification. If one uses a function, the compiler would issue a warning.
- Macros may affect things you don’t realize. The semantics of macros is completely arbitrary and not detectable by the compiler

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`:

```
#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

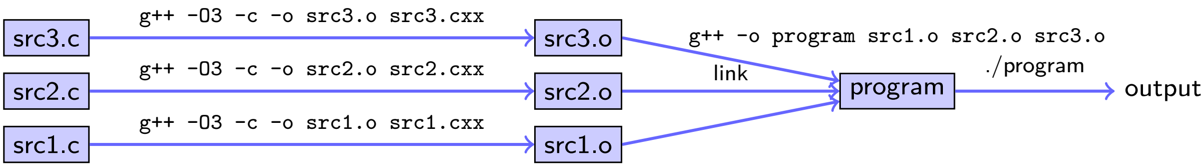
- By default, the compiler command performs the linking process as well
- Compiler command (Linux)

```
    | 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
src2.c
src1.c
g++ -O3 -c -o src3.o src3.cxx
g++ -O3 -c -o src2.o src2.cxx
g++ -O3 -c -o src1.o src1.cxx
src3.o
src2.o
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
}
```

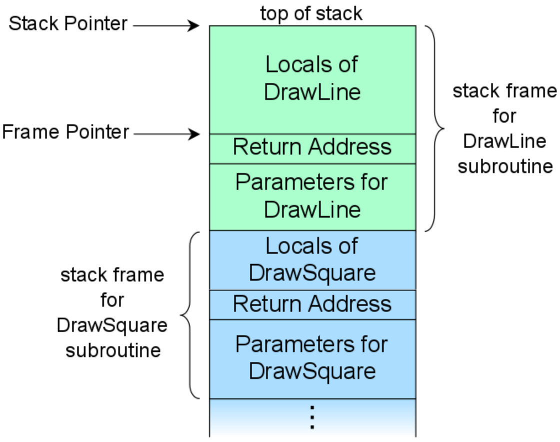
- This one is also illegal, though often compilers accept it

```
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

```
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

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class.
- The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

```
class vector2d
{
private:
    double *data;
    vector2d<int> shape;
    int size
public:
    double & operator(int i, int j);
    vector2d(int nrow, ncol);
    ~vector2d();template <t
}

class matrix: public vector2d
{
   public:
    apply(const vector1d& u, vector1d &v);
    solve(vector1d&u, const vector1d&rhs);
}
```

- All operations which can be performed with instances of `vector2d` can be performed with instances of `matrix` as well
- In addition, `matrix` has methods for linear system solution and matrix-vector multiplication

Smart pointers

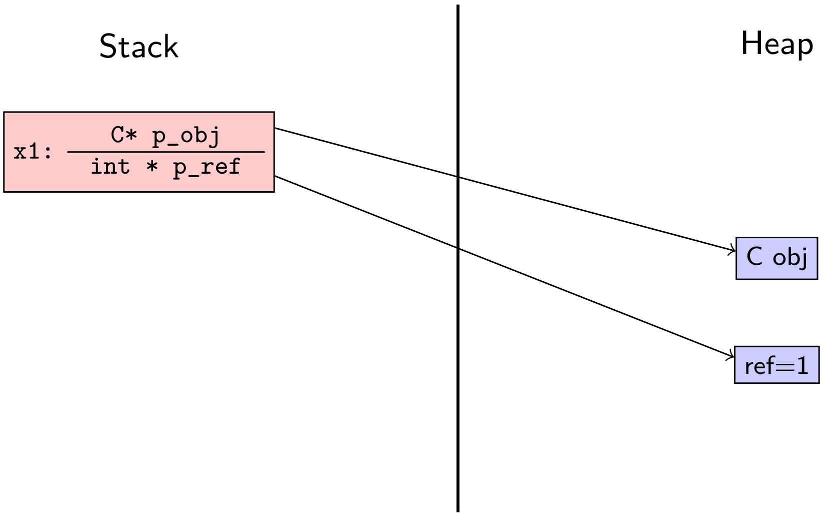
... with a little help from Timo Streckenbach from WIAS who introduced smart pointers into our simulation code.

- Automatic book-keeping of pointers to objects in memory.
- Instead of the meory addres of an object aka. pointer, a structure is passed around *by value* which holds the memory address and a pointer to a *reference count* object. It delegates the member access operator `->` and the address resolution operator `*` to the pointer it contains.
- Each assignment of a smart pointer increases this reference count.
- Each destructor invocation from a copy of the smart pointer structure decreses the reference count.
- If the reference count reaches zero, the memory is freed.
- `std::shared_ptr` is part of the C++11 standard

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

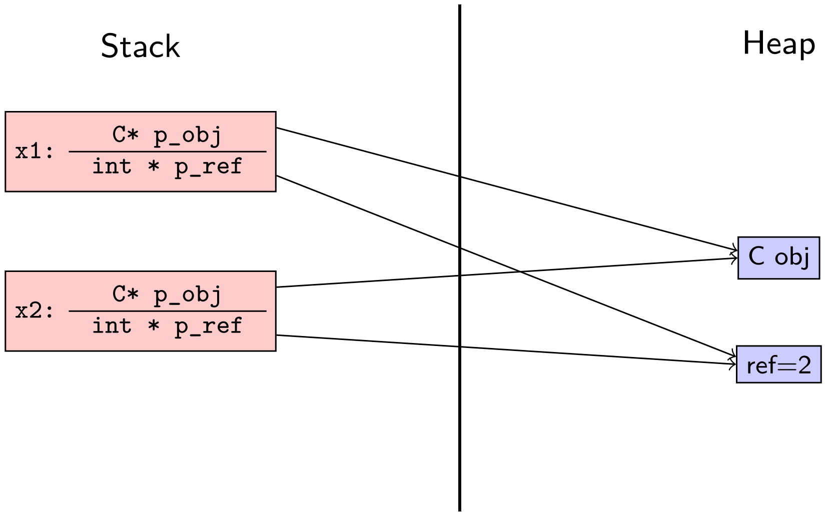


```
std::shared_ptr<C> x1= std::make_shared<C>();
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

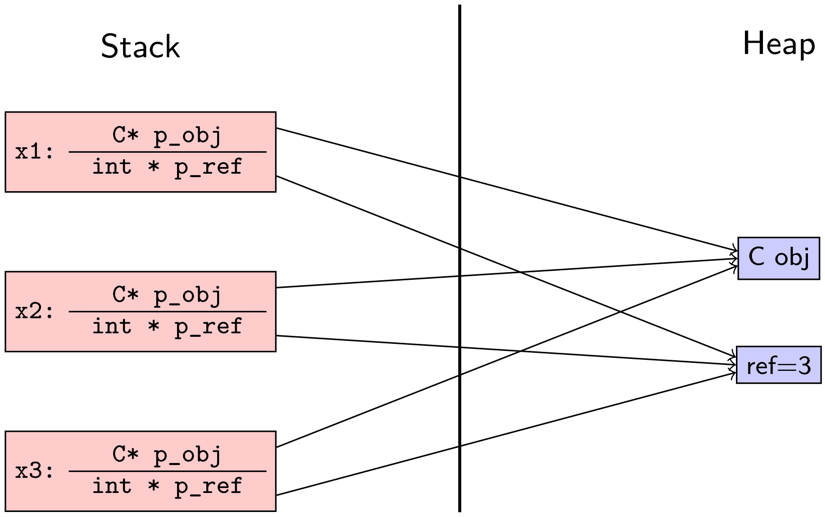


```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```



```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
std::shared_ptr<C> x3= x1;
```

Smart pointers vs. ∗-pointers

- When writing code using smart pointers, write

```
#include <memory>
class R;
std::shared_ptr<R> ReturnObjectOfClassR(void);
void PassObjectOfClassR(std::shared_ptr<R> o);
...
std::shared_ptr<R> o;
o->member=5;
...
{
   auto o=std::make_shared<R>();
   PassObjectOfClassR(o)
   // Smart pointer object is deleted at end of scope and frees memory
}
```

instead of

```
class R;
R* ReturnObjectOfClassR(void);
void PassObjectOfClassR(R* o);
...
R*o;
o->member=5;
...
{
   R* o=new R;
   PassObjectOfClassR(o);
   delete o;
}
```

Smart pointer advantages vs. ∗-pointers

- “Forget” about memory deallocation
- Automatic book-keeping in situations when members of several different objects point to the same allocated memory
- Proper reference counting when working together with other libraries, e.g. numpy

C++ topics not covered so far

- To be covered on occurence
  - character strings
  - overloading
  - optional arguments, variable parameter lists
  - Functor classes, lambdas
  - threads
  - malloc/free/realloc (C-style memory management)
  - cmath library
  - Interfacing C/Fortran
  - Interfacing Python/numpy
- To be omitted (probably)
  - Exceptions
  - Move semantics
  - Expression templates
    - Expression templates allow to write code like `c=A*b` for a matrix `A` and vectors `b,c`.
    - Realised e.g. in Eigen, Armadillo
    - Too complicated for teaching (IMHO)
  - GUI libraries
  - Graphics (we aim at python here)

C++ code using vectors, C-Style, with data on stack

File `/net/wir/examples/part1/c-style-stack.cxx`

```
#include <cstdio>

void initialize(double *x, int n)
{
    for (int i=0;i<n;i++) x[i]= 1.0/(double)(1+n-i);
}

double sum_elements(double *x, int n)
{
    double sum=0;
    for (int i=0;i<n;i++) sum+=x[i];
    return sum;
}

int main()
{
    const int n=1.0e7;
    double x[n];
    initialize(x,n);
    double s=sum_elements(x,n);
    printf("sum=%e\n",s);
}
```

- Large arrays may not fit on stack
- C-Style arrays do not know their length

C++ code using vectors, C-Style, with data on heap

File /net/wir/examples/part1/c-style-heap.cxx

```
#include <cstdio>
#include <cstdlib>
#include <new>

// initialize vector x with some data
void initialize(double *x, int n)
{
    for (int i=0;i<n;i++) x[i]= 1.0/(double)(1+n-i);
}

// calculate the sum of the elements of x
double sum_elements(double *x, int n)
{
    double sum=0;
    for (int i=0;i<n;i++) sum+=x[i];
    return sum;
}

int main()
{
    const int n=1.0e7;
    try  {  x=new double[n]; // allocate memory for vector on heap }
    catch (std::bad_alloc)  { printf("error allocating x\n");  exit(EXIT_FAILURE); }
    initialize(x,n);
    double s=sum_elements(x,n);
    printf("sum=%e\n",s);
    delete[] x;
}
```

- C-Style arrays do not know their length
- Proper memory management is error prone

C++ code using vectors, (mostly) modern C++-style

File /net/wir/examples/part1/cxx-style-ref.cxx

```
#include <cstdio>
#include <vector>

void initialize(std::vector<double>& x)
{
    for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);
}

double sum_elements(std::vector<double>& x)
{
    double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;
}

int main()
{
    const int n=1.0e7;
    std::vector<double> x(n); // Construct vector with n elements
                              // Object "lives" on stack, data on heap
    initialize(x);
    double s=sum_elements(x);
    printf("sum=%e\n",s);
    // Object destructor automatically called at end of lifetime
    // So data array is freed automatically
}
```

- Heap memory management controlled by object lifetime
- Recommended style *if we can completely stay within C++*

C++ code using vectors, (mostly) modern C++-style with smart pointers

File `/net/wir/examples/part1/cxx-style-sharedptr.cxx`

```
#include <cstdio>
#include <vector>
#include <memory>

void initialize(std::vector<double> &x)
{
    for (int i=0;i<x.size();i++) x[i]= 1.0/(double)(1+n-i);
}

double sum_elements(std::vector<double> & x)
{
    double sum=0;
    for (int i=0;i<x.size();i++)sum+=x[i];
    return sum;
}

int main()
{
    const int n=1.0e7;
    // call constructor and wrap pointer into smart pointer
    auto x=std::make_shared<std::vector<double>>(n);
    initialize(*x);
    double s=sum_elements(*x);
    printf("sum=%e\n",s);
    // smartpointer calls desctrutor if reference count
    // reaches zero
}
```

- Heap memory management controlled by smart pointer lifetime
- If method or function does not store the object, pass by reference ⇒ API stays the same as for previous case.

∼

Starting on unix

Some shell commands in the terminal window

```
| ls -l                       | list files in directory                               |
|                             | subdirectories are marked with ’d’                    |
|                             | in the first column of permission list                |
| cd  dir                     | change directory to dir                               |
| cd  ..                      | change directory one level up in directory hierachy   |
| cp  file1 file2             | copy file1 to file2                                   |
| cp  file1 dir               | copy file1 to directory                               |
| mv  file1 file2             | rename file1 to file2                                 |
| mv  file1 dir               | move file1 to directory                               |
| rm  file                    | delete file                                           |
| [cmd] *.o                   | perform command on all files with name ending with .o |
```

Editors & IDEs

- Source code is written with text editors
  (as compared to word processors like `MS Word` or `libreoffice`)
- Editors installed are
  - `gedit` - text editor of gnome desktop (recommended)
  - `emacs` - comprensive, powerful, a bit unusual GUI (my preferred choice)
  - `nedit` - quick and simple
  - `vi`, `vim` - the UNIX purist’s crowbar
    (which I avoid as much as possible)
- Integrated development environments (IDE)
  - Integrated editor/debugger/compiler
  - eclipse (need to get myself used to it before teaching)

Command line instructions to control compiler

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

```
| g++     | GNU C++ compiler                |
| clang++ | CLANG compiler from LLVM project |
| g++-5   | GNU C++ 5.x                     |
| 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               |
```

Expression templates

- This is a C++ technique which allows to implement expressions while avoiding introduction and copies of temporary objects

```
Vector a,b,c;
c=a+b;
```

- Has been realized in numcxx, allowing for more readable code and re-use of template based iterative solver library

Code with temporary objects

```
inline const Vector
operator+( const Vector& a, const Vector& b )
{
   Vector tmp( a.size() );
   for( std::size_t i=0; i<a.size(); ++i )
      tmp[i] = a[i] + b[i];
   return tmp;
}
```

Code with expression templates I

(K. Iglberger, “Expression templates revisited”)

Expression template:

```
template< typename A, typename B >
class Sum {
public:
    Sum( const A& a, const B& b ) : a_( a ), b_( b )    {}
    std::size_t size() const { return a_.size(); }
    double operator[]( std::size_t i ) const
    { return a_[i] + b_[i]; }
private:
    const A& a_;    // Reference to the left-hand side operand
    const B& b_;    // Reference to the right-hand side operand
};
```

Overloaded + operator:

```
template< typename A, typename B >
const Sum<A,B> operator+( const A& a, const B& b )
{
   return Sum<A,B>( a, b );
}
```

Code with expression templates II

Method to copy vector data from expression:

```
class Vector
{
   public:
   // ...
   template< typename A >
   Vector& operator=( const A& expr )
   {
    for( std::size_t i=0; i<expr.size(); ++i )
        v_[i] = expr[i];
    return *this;
   }
// ...
};
```

After template instantiation, the compiler will use

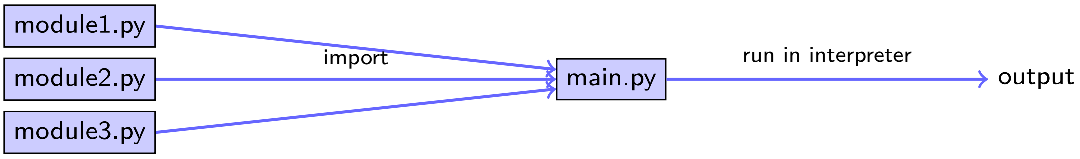
```
   for( std::size_t i=0; i<a.size(); ++i )
      c[i] = a[i] + b[i];
```

High level scripting languages

- Algorithm description using mix of mathematical formulas and statements inspired by human language
- Need intepreter in order to be executed

```
print("Hello world")
```

- Very far away from CPU ⇒ usually significantly slower compared to compiled languages
- Matlab, Python, Lua, perl, R, Java, javascript
- Less strict type checking, often simple syntax, powerful introspection capabilities
- Immediate workflow: “just run”
  - in fact: first compiled to *bytecode* which can be interpreted more efficiently



module1.py

module2.py

module3.py

import

main.py

run in interpreter

output

Python

- Developed since 1989, led by Guido van Rossum
- Can be seen as “open source” matlab
- Main advantage: huge ecosystem of packages for scientific computing
- Some use cases:
  - matlab replacement
  - glue language for different tools
  - system independent implementation of tools (e.g. mercurial)
  - driver language for software written in C/C++
    - quickly change parameters without recompiling etc.
    - make use of plotting capabilities
- Documentation: https://docs.python.org
  - current versions around: 2.7, 3.x
  - most python3 code works with 2.7
- Tutorial: `https://docs.python.org/3/tutorial/`

Numpy / Scipy /matplotlib

- numpy: add-on of an efficient array class for numerical computations, written in C
- Python lists would be too slow
- Interfacing to lapack etc. need dense arrays
- scipy: Scientific computation package with LAPACK etc.
- matplotlib: data plotting + visualization

Python and C++

- Architecture of python
  - Interpreter
  - Application programming interface (API) for interaction of C/C++ with python interpreter
    - register wrapper function with name
    - wrapper function knows how to fetch parameters/return values from/to python interpreter
    - wrapper function calls C++ function
  - Wrapper code with code to be wrapped linked to a “shared object” (UNIX), “dylib” (Mac), “DLL” (Windows)
  - Import of wrapper code makes it available in python
- Automatic tools for accessing API

SWIG

- Several possibilities
  - Cython (python dialect with possible inclusion of C)
  - pybind11 (C++11 classes for wrapping python)
  - **SWIG** (“classical tool” for wrapping interpreter)
- Simplified Wrapper and Interface Generator:
  - Tool to automatically create wrapper code from C++ style description
  - Create wrapper code in C++ which is linked together with library to be wrapped