Programmiertechniken

Sommersemester 2016

Lode Pollet

Information

- Prof. Dr. Lode Pollet
  - Office : A405, Theresienstr 37
  - Tel: 089/2180-4593
  - email: lode.Pollet@lmu.de
  - website: http://www.theorie.physik.uni-muenchen.de/lsschollwoeck/members/professors/pollet/

- Exercises
  - Jonas Greitemann (German, English)
  - Tobias Pfeffer (German, English)
  - Dario Hügel (German, English)
  - Dr. Jacopo Nespolo (English)

About…me

- Phd physics, Gent (2005)
- postdocs : ETH Zurich, UMass Amherst, Harvard
- since Oct 2011: professor at LMU Munich; tenured since 03/2015
- group website: http://www.theorie.physik.uni-muenchen.de/lsschollwoeck/pollet_group/index.html
- quantum Monte Carlo simulations, computational physics
- cold atoms, strongly correlated many-body physics, supersolid Helium-4, superconductors, …
- developer of ALPS project

About…you

- Bachelor students in physics?
- Master students in physics?
- other?

- semesters 2 - 8 ?
- M4 : Numerische Mathematik für Studierende der Physik?
- linear algebra? analysis? electromagnetism? quantum mechanics?
- who does not have a laptop?

About the course

- Time of the course: Mon 8-10 (H030) weekly
- Time of the exercises? see website (and do not forget to register)
  http://www.physik.uni-muenchen.de/lehre/vorlesungen/sose_16/programmiertechniken/index.html
- Computer accounts? see manual
- language of the course?
- website: contains all info
- exercises do not need to be handed in; no solutions provided online
- optional but recommended for all BSc students
  preparation for:
  - ✦ (advanced) Computational Physics course
  - ✦ lab work
  - ✦ Bachelor thesis
  - ✦ Master thesis
  - ✦ PhD thesis
  - ✦ well-paid job

Prerequisites : none

- Numerical Analysis (solving linear systems, eigenvalue problems, integration, differentiation, …)
- familiarity with computers useful
- programming experience (any language) not needed but an advantage
- knowledge of basic data structures (array, list, tree) will be reviewed

manual

- issues: different operating systems, levels of experience, etc

- policy: we make sure that everything works for Ubuntu linux. If you use windows, OSX, … follow the manual if you need help with installing the required software. If you do not follow the manual and you encounter installation issues we cannot give priority to your problems.

- CIP pool : exercises can be solved in the CIP pool (H037 und H022)

  http://www.it.physik.uni-muenchen.de/dienste/cip_pool/index.html

- first 2 weeks of the semester: exercise classes are a helpdesk

Contents

1. C++ basic language
2. macros, compilers, libraries
3. templates: generic programming
4. Basics of Classes
5. Classes again : operator overloading
6. The standard library (STL)
7. Basics of hardware
8. Inheritance : object oriented programming
9. Exceptions
10. Optimization
11. Lazy evaluation, expression templates, reference counting, proxy classes
12. Recent developments in C++11
13. Introduction to parallel programming
14. Introduction to Python
15. Python: numpy, scipy, matplotlib
16. Introduction to git and Cmake

Schedule

Mon, April 11 : lecture (C++) — H030

Thu, April 14: lecture (Python) — H030

Mon, April 18 : lecture (C++) — H030

Thu, April 21: lecture (Python) — H030

exercise class = helpdesk

for installing the manual

Mon, April 25 : lecture (C++) — H030

exercises start as usual in smaller groups; Thursday group NOT in H030

and so on (regular schedule) till the end of term

any announcements or changes appear on the website!

Goals

- efficient simulating of systems (C++)
- easy processing of large datasets (Python)
- basic hardware understanding (memory, caches, CPU)
- basic understanding of the modern language
- being technically able to solve typical problems in computational physics
- acquiring good programming style for scientific problems
- acquire skills to think in abstract concepts suitable for program design
- generic programming
- object oriented programming
- standard template library
- optimization

why C++

- high-level programming (lower error rate, better software reuse, easier debugging)
- control over objects and memory
- efficiency: (almost) as efficient as Fortran, faster than java, Pascal, …
- enhances chances on the job market
- modular programming
- generic programming
- object oriented programming
- after 30 years still one of the standards
- language flexibility and control

why Python

- focus on the science and let the computer scientists do the optimization
- easy to learn and use : fast development
- widespread support : numpy, scipy, matplotlib
- very readable and high-level coding
- object oriented
- no memory management
- interpreter : no compiling (this also means considerably slower than eg C++)
- no explicit variable names : extremely polymorphic
- free and open (unlike Matlab)
- can be excellent “glue” for C++ components

First program

```
/* A first C++ program */
#include <iostream>

using namespace std;          // also valid: using std::cout

int main() {
  cout << "Welcome students!\n";
  std::cout << "Hello world!" << std::endl;      // this notation works without "using"
                                                 // never use the "using" keyword globally in libraries
  return 0;
}
```

- comment block: */ … */ or rest of the line: //
- #include directive: angle brackets for <standard library>
- namespace: allows you to omit the class name
- main() : main program, no function arguments
- scope operator ::
- one unit delimited by curly braces { … }
- std::cout : standard output from the standard library
- << output stream
- statements end with a ;
- return 0 : program terminated normally

A little quiz

Who knows:

Python ?

- Numpy?
- Scipy?
- Matplotlib?

Java ?
C ?
C++ ?

- Classes?
- Inheritance?
- Templates?
- Generic Programming?
- Standard library?
- Optimization in C++?
- glvalues?
- Parallel programming?
- Polymorphism
- Proxy classes?

git?
Cmake?

A little quiz

what is your knowledge level of C++?

A. I have never programmed before
B. I have programmed before, but not in C nor C++
C. I know basic C
D. I know basic C++
E. I know C++ well
F. I am a C++ guru

A little quiz

what is the output of:

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

int main() {
  int a=0;
  std::cout << a++ << "\n";
  std::cout << ++a << "\n";
  std::cout << a << "\n";
  return 0;
}
```

A. 0 - 1 - 2
B. 0 - 2 - 2
C. 1 - 1 - 2
D. 1 - 2 - 2
E. 1 - 2 - 3

(quiz_pp.cpp)

A little quiz

what versions do not compile?

```
#include <iostream>

using namespace std;

void swap1 (int a, int b) { int t=a; a=b; b=t; }
void swap2 (int& a, int& b) { int t=a; a=b; b=t;}
void swap3 (int const & a, int const & b) { int t=a; a=b; b=t;}
void swap4 (int *a, int *b) { int *t=a; a=b; b=t;}
void swap5 (int* a, int* b) {int t=*a; *a=*b; *b=t;}


int main() {
  int a=1; int b=2;
  swap1(a,b); cout << a << " " << b << "\n";
  a=1; b=2;
  swap2(a,b); cout << a << " " << b << "\n";
  a=1; b=2;
  swap3(a,b); cout << a << " " << b << "\n";
  a=1; b=2;
  swap4(&a,&b); cout << a << " " << b << "\n";
  a=1; b=2;
  swap5(&a,&b); cout << a << " " << b << "\n";
  return 0;
}
```

A. version 1 does not compile
B. version 2 does not compile
C. version 3 does not compile
D. version 4 does not compile
E. version 5 does not compile
F. all versions compile

what versions actually accomplish the swap of the values?

A. version 1
B. version 2
C. versions 1 and 4
D. versions 2 and 4
E. versions 2 and 5
F. versions 2,4, and 5

(quiz_swap.cpp)

A topic discussed worldwide…

Integrating Computation into the Undergraduate Physics Curriculum

Sunday, March 13
1:00 p.m. - 5:30 p.m.
Baltimore, Maryland

Who Should Attend?
Physics educators

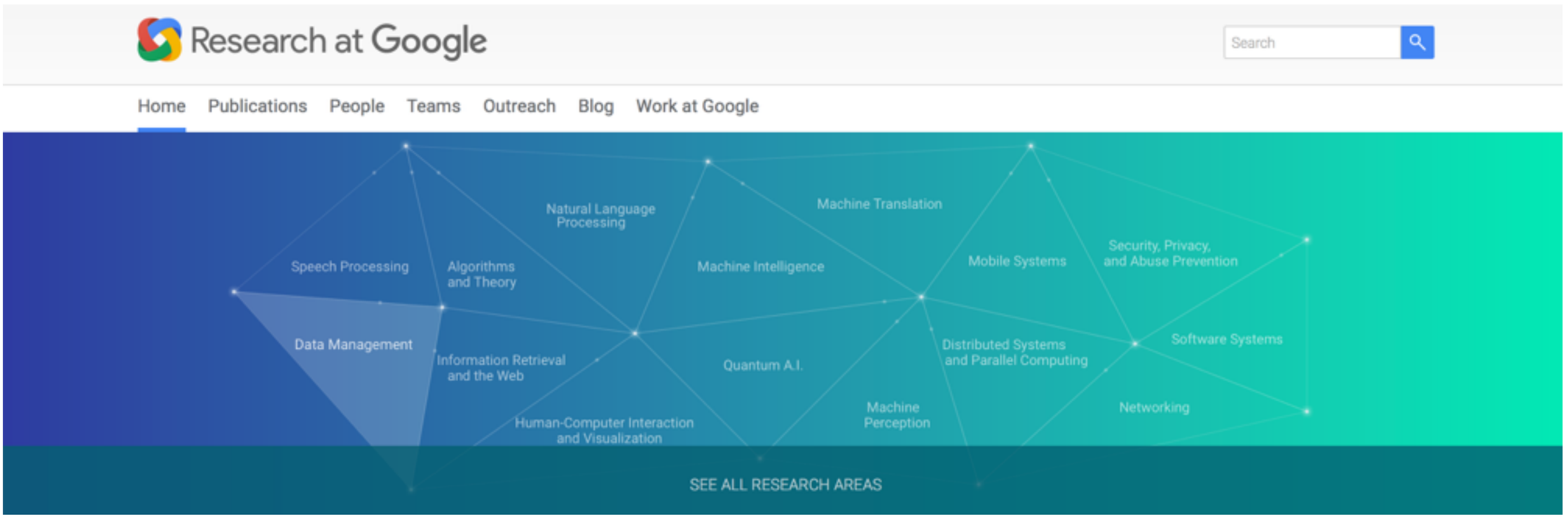
Overview
In this workshop we will discuss the importance of integrating computation into the physics curriculum and guide participants in discussing and planning how they would integrate computation into their courses. The PICUP partnership has developed materials for a variety of physics courses in a variety of platforms including C/C++, Fortran, VPython, Octave/MATLAB, and Mathematica. Participants will receive information on the computational materials that have been developed, ways to tailor the materials to their own classes, and available faculty opportunities and support through the PICUP partnership.

Please bring a laptop computer with the platform of your choice.

quantum revolution

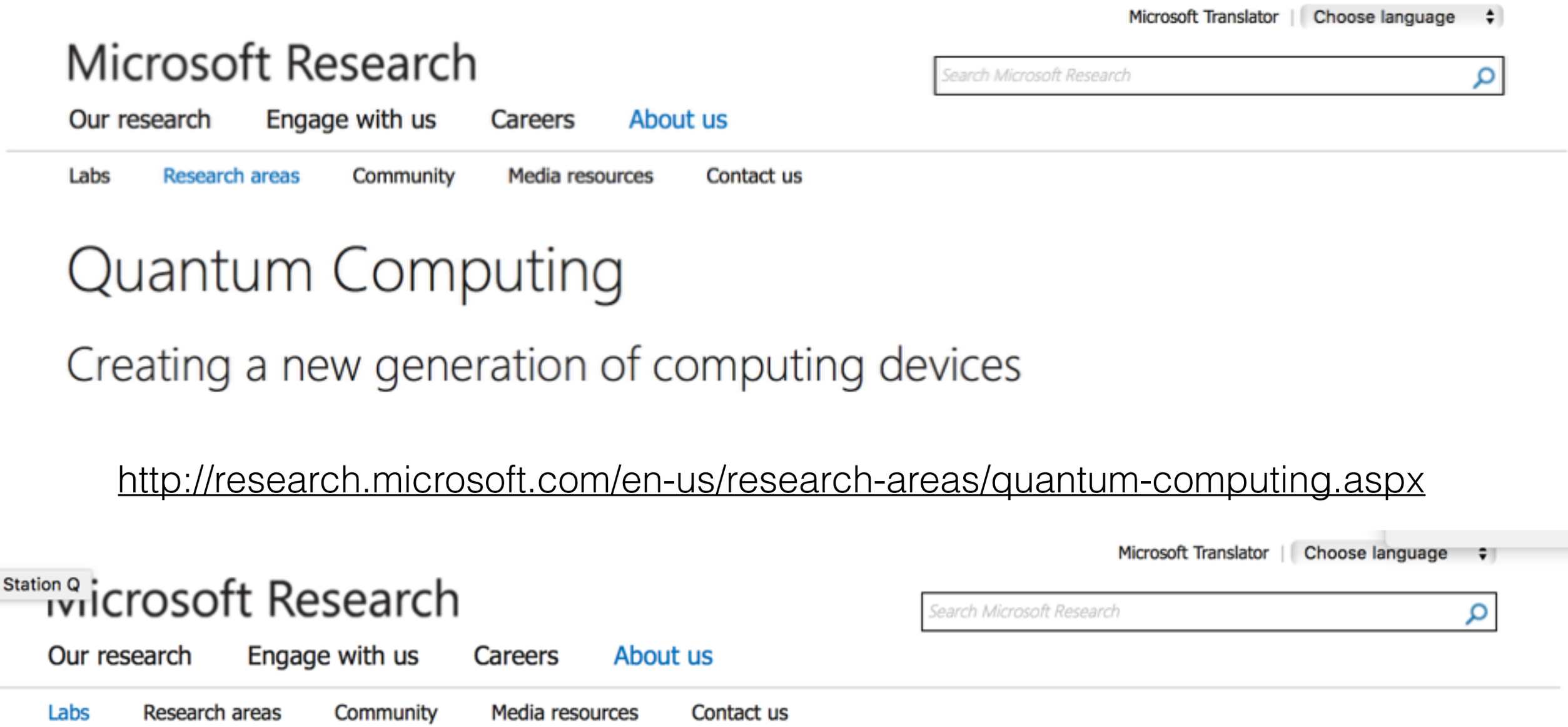
a world-wide massive investment in quantum technologies:

http://ti.arc.nasa.gov/quantum/index.html



https://research.google.com

quantum revolution



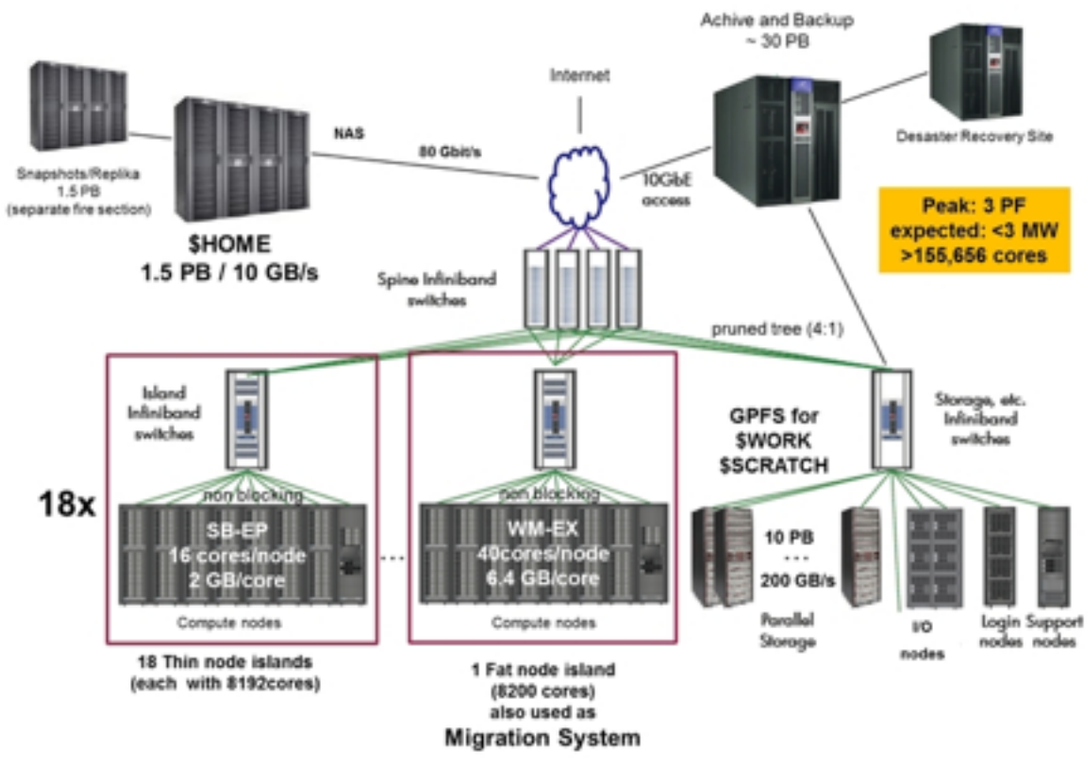
http://research.microsoft.com/en-us/research-areas/quantum-computing.aspx

http://research.microsoft.com/en-us/labs/stationq/

high performance computing

LRZ SuperMUC

https://www.lrz.de/services/compute/supermuc/systemdescription/



Book References

There are thousands of books on the market for C++. Some I can recommend are:

Andrew Koenig and Barbara E. Moo, *Accelerated C++,* Addison Wesley 2000 : Good and short introduction

Stanley B. Lippman, *Essential C++,* Addison Wesley 2000 : Good and short introduction

Bjarne Stroustrup, *The C++ Programming Language*, 4th edition, Pearson Education inc. The reference book

Stanley B. Lippman, J. Lajoie, B.E. Moo, *C++ primer*, 5th edition : rewritten for C++11

Scott Meyers*, Effective C++: 55 Specific Ways to Improve Your Programs and Designs (Professional Computing) (no C++11) :* some more advanced topics in the course are explained in detail here; good literature but not as an introduction

Scott Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* : when completing the course this is very useful

standard websites:

http://www.cplusplus.com

http://en.cppreference.com/w/

when not further specified, examples are taken from these sources

C++ data types

- the standard data types are:

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than `char`. At least 16 bits.
	char32_t	Not smaller than `char16_t`. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as `char`. At least 8 bits.
	signed **short** *int*	Not smaller than `char`. At least 16 bits.
	signed **int**	Not smaller than `short`. At least 16 bits.
	signed **long** *int*	Not smaller than `int`. At least 32 bits.
	signed **long long** *int*	Not smaller than `long`. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than `float`
	long double	Precision not less than `double`
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

http://www.cplusplus.com/doc/tutorial/variables/

C++11

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

int main()
{
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;
   return 0;
}
```

C++ data types

Types of fixed size can be found in <cstdint>
standard since C++11 <cstdint> : int8_t, int16_t, int32_t, int64_t
standard since C++11 <cstdint> : uint8_t, uint16_t, uint32_t, uint64_t
the more bits, the more distinct values can be represented but the higher the memory requirements; eg, for the unsigned data types

Size	Unique representable values	Notes
8-bit	256	= 2^8
16-bit	65 536	= 2^{16}
32-bit	4 294 967 296	= 2^{32} (~4 billion)
64-bit	18 446 744 073 709 551 616	= 2^{64} (~18 billion billion)

The properties of the fundamental types can be found in the numeric_limits classes defined in the <limits> header.

```
template <class T> class numeric_limits {          // first example of a template and a class
public:
  static const bool is_specialized = false;        // is true if information is provided
  static T min() throw();                          // minimum (largest negative value)
  static T max() throw();                          // maximum
  static const int  digits = 0;                    // number of bits (base - 2)
  static const int  digits10 = 0;                  // number of decimal digits
  static const bool is_signed = false;
  static const bool is_integer = false;
  static const bool is_exact = false;
  static const int radix = 0;
  static T epsilon() throw();                      // floating point precision, ie lowest number for which 1+ε != ε
  static T round_error() throw();
  // and many more
```

C++ data types

example from:

http://www.cplusplus.com/reference/limits/numeric_limits/

C++ shell

cpp.sh
online C++ compiler
about cpp.sh

```
1  // numeric_limits example
2  #include <iostream>       // std::cout
3  #include <limits>         // std::numeric_limits
4
5  int main () {
6    std::cout << std::boolalpha;
7    std::cout << "Minimum value for int: " << std::numeric_limits<int>::min() << '\n';
8    std::cout << "Maximum value for int: " << std::numeric_limits<int>::max() << '\n';
9    std::cout << "int is signed: " << std::numeric_limits<int>::is_signed << '\n';
10   std::cout << "Non-sign bits in int: " << std::numeric_limits<int>::digits << '\n';
11   std::cout << "int has infinity: " << std::numeric_limits<int>::has_infinity << '\n';
12   return 0;
13 }
14
15
16
```

Get URL

Run

options | compilation | execution

```
Minimum value for int: -2147483648
Maximum value for int: 2147483647
int is signed: true
Non-sign bits in int: 31
int has infinity: false


Exit code: 0 (normal program termination)
```

C++ Shell 2014-2015

C++ data types

- int a;
- int a = 5; is equivalent to int a(5);
- binary format : 00000000 00000101
- size of 1 byte : from 0 to 255 for unsigned char or -127 to 127 for signed char
- unsigned: x stored as n bits from 0 to 2^n - 1
- signed x:
  - stored as 2-complement from -2^{n-1} to $2^{n-1}-1$;
  - highest bit is the sign bit S
  - for positive x : S =0, rest is x
  - for x < 0 : S = 1, for the rest write (-x) in binary form, change all bits, add +1
  - advantage : signed variables can be added like unsigned variables
- conversions are possible, eg static_cast<int>(ch)

Address	Big-Endian representation of 1025	Little-Endian representation of 1025
00	00000000	00000001
01	00000000	00000100
02	00000100	00000000
03	00000001	00000000

- breaking up multi-bytes: big endian and little endian: how to order?
- endianness is machine dependent (eg my laptop is little endian). Write the simplest possible program to find out the endianness on your machine.

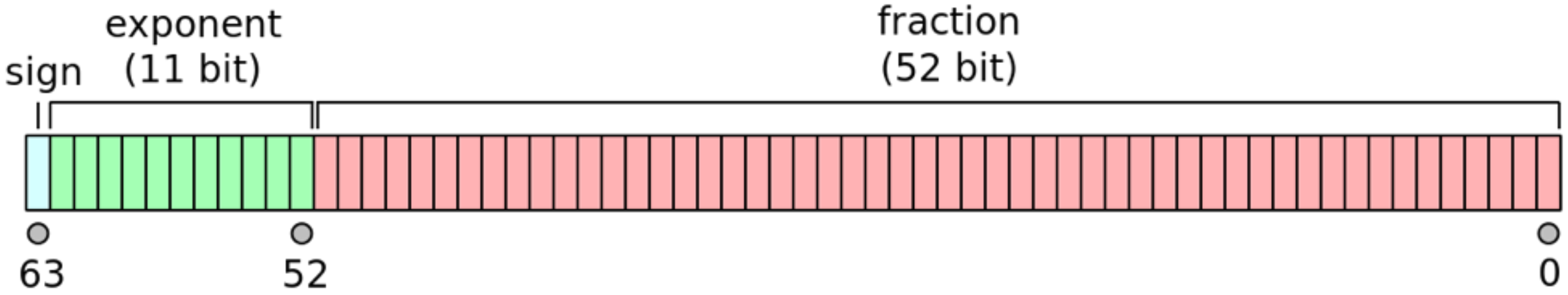
signed 2-complement

Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
⋮	⋮	⋮
01111110	126	126
01111111	127	127
10000000	-128	128
10000001	-127	129
10000010	-126	130
⋮	⋮	⋮
11111110	-2	254
11111111	-1	255

Storing floats and doubles

see http://en.wikipedia.org/wiki/IEEE_754-2008

binary double (64 bit):



(extended data formats : 80 bits)

$$(-1)^{\text{sign}}(1.b_{51}b_{50}...b_0)_2 \times 2^{e-1023}$$

$$(-1)^{\text{sign}}\left(1+\sum_{i=1}^{52} b_{52-i}2^{-i}\right) \times 2^{e-1023}$$

range: 10^{-308} to 10^{308} with full 15-17 digits

machine precision : 2^{-53}

C++ operators

is explained in detail on:

http://www.cplusplus.com/doc/tutorial/operators/

arithmetic:

operator	description
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

11 % 3 = 2
11 / 3 = 3
11 / 3. = 3.666667

compound assignment:

expression	equivalent to...
`y += x;`	`y = y + x;`
`x -= 5;`	`x = x - 5;`
`x /= y;`	`x = x / y;`
`price *= units + 1;`	`price = price * (units +1);`

also %=, >>=, <<=, &=, ^=, |= etc allowed

C++ operators

relational and comparison

operator	description
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!	logical not
&&	logical AND
||	logical OR

increment:

post-increment
post-decrement
pre-increment
pre-decrement

```
i++;    // same as i = i+1  or i += 1
i--;
++i;
--i;
```

C++ operators

bitwise operators:

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift bits left
>>	SHR	Shift bits right

note: there also exists a bitset class

comma operator, ternary conditional operator : see later

there exist many more special operators:

static_cast converts one type to another related type
dynamic_cast converts within inheritance hierarchies
const_cast adds or removes cv qualifiers
reinterpret_cast converts type to unrelated type
C-style cast converts one type to another by a mix of static_cast, const_cast, and reinterpret_cast
new allocates memory
delete deallocates memory
sizeof queries the size of a type
sizeof... queries the size of a parameter pack (since C++11)
typeid queries the type information of a type
noexcept checks if an expression can throw an exception (since C++11)
alignof queries alignment requirements of a type (since C++11)

operator precedence : is explained in detail on
http://en.cppreference.com/w/cpp/language/operator_precedence
when in doubt : use parentheses

declaration vs definition

a declaration declares a variable or function to be of a certain type. It does not have to be unique

```
int k;                          // declares variable to be of type int
double foo(const int );         // declares function fun to return a double and take a const int as argument
```

a definition instantiates this definition. It must be unique. If you forget it, the linker might complain

```
int y=4;
//3 = x;                                // error! a constant literal cannot be considered an lvalue


double foo(const int i) { return i+1;}      // no function definitions inside another function
```

The if statement

syntax is as follows:

```
if (speed > 30) {
  std::cout << "You are driving too fast\n";
}

if (speed > 30)
  std::cout << "You are driving too fast\n";
else
  std::cout << "OK";


if (speed > 50) {
  std::cout << "You are driving much too fast\n";
}
else if (speed > 30) {
  std::cout << "You are driving a little bit too fast\n";
}
else {
    std::cout << "OK";
}
```

For concise statements the ternary operator is a handy alternative. It counts as a single statement

```
// illustration of the ternary operator
int grade = 80;
std::cout << (grade < 50 ? "You failed the course\n" : "You passed the course\n");
```

The switch statement

syntax is as follows:

(statement_if.cpp)

```
// illustration of the switch statement
enum trafficlight_colors {red, yellow, green};
trafficlight_colors light = green;
switch (light) {
  case red:
    std::cout << "STOP!\n";
    break;          // DO NOT FORGET THE BREAK !!!! (what happens if you do?)
  case yellow:
    std::cout << "Watch out!\n";
    break;
  case green:
    std::cout << "Go!\n";
    break;
  default:
    std::cout << "New traffic rules?\n";
    abort();
}
```

also note the usage of the scoped enum. This code requires C++11.
The C++98 version looks like this:

```
enum {red, yellow, green};
int light = green;
switch (light) {
  case red:
    std::cout << "STOP!\n";
    break;          // DO NOT FORGET THE BREAK !!!! (what happens if you do?)
  case yellow:
    std::cout << "Watch out!\n";
    break;
  case green:
    std::cout << "Go!\n";
    break;
  default:
    std::cout << "New traffic rules?\n";
}
```

Loops

http://www.cplusplus.com/doc/tutorial/control/

There are 3 common ways of making a loop: for, while, and do…while:

```
for ( initialization ; condition ; increment ) {
   statements
}

while ( condition ) {
   statements
}

do {
   statements
} while ( condition );
```

the statements break interrupts the loop
the statement continue skips the rest of the current iteration

```
cout << "The for loop:\n";
for (int n=10; n>0; n--) cout << n << " ";
cout << "\n";

cout << "The while loop:\n";
int n = 10;
while (n > 0) {
  cout << n << " ";
  n--;
}
cout << endl;

cout << "The do loop:\n";
n = 10;
do {
  cout << n << " ";
  n--;
} while (n > 0);
cout << "\n";
```

(statement_for.cpp)

```
cout << "break:";
for (int n=10; n>0; n--) {
  cout << n << " ";
  if (n==3) {
    cout << "countdown aborted!";
    break;
  }
}

cout << "\n";

cout << "continue:";
for (int n=10; n>0; n--) {
  if (n==3) {
    continue;
  }
  cout << n << " ";
}
cout << endl;
```

A little quiz

```
// loop nr 1
for (int i=1;i<=n;++i)
  cout << i << "\n";

// loop nr 2
int i=0;
while (i<n)
  std::cout << ++i << "\n";
```

```
// loop nr 3
i=1;
do
  cout << i++ << "\n";
while (i<=n);

// loop nr 4
i=1;
while (true) {
 if(i>n) break;
 cout << i++ << "\n";
}
```

Which of the loops does not produce the correct output on all inputs?

A. loop nr 1
B. loop nr 2
C. loop nr 3
D. loop nr 4

Strings (accelerated_ch1.cpp)

```
// The program of Chapter 1 in the book "Accelerated C++" by Andrew Koenig and Barbara E. Moo

#include <iostream>
#include <string>

using namespace std;

int main() {
  const std::string hello = "hello";                          // valid string
  const std::string exclam = "!";                           // valid string literal
  const std::string message = hello + " , world" + exclam;      // valid concatenation of strings and string litera
  const std::string s = "a string";
  std::cout << s << std::endl;
  { const string s = "another string";
    std::cout << s << std::endl;
  }
  std::cout << s << std::endl;                              // the lines below are about the scope and curly braces
  { const std::string t = "a String";
    std::cout << t << std::endl;
    { const std::string t = "another String";
      std::cout << t << std::endl;}
    std::cout << t << std::endl;
  }
  { const std::string t = "a String";
    std::cout << t << std::endl;
    { const std::string t = "another String";
      std::cout << t << std::endl;};                          // note the additional ';'
    std::cout << t << std::endl;
  }
  std::cout << "What is your name ?";
  std::string name; std::cin >> name;                         // predict the output when answering Samuel Beckett
  std::cout << "Hello, " << name << std::endl << "And what is yours?";
  std::cin >> name;
  std::cout << "Hello, " << name << "; nice to meet you too!" << std::endl;
  return 0;
}
```

Strings

(accelerated_ch2.cpp)

```cpp
// The program of Chapter 2 in the book "Accelerated C++" by Andrew Koenig and Barbara E. Moo

#include <iostream>
#include <string>

// say what standard-library names we use
using std::cin;
using std::endl;
using std::cout;
using std::string;

int main() {
  // ask for the person's name
  cout << "Please enter your first name: ";

  // read the name
  string name;
  cin >> name;

  // build the messaging that we intend to write
  const string greeting = "Hello, " + name + "!";

  // the number of blanks surrounding the greeting
  const int pad = 1;

  // the number of rows and columns to write
  const int rows = pad * 2 + 3;
  const string::size_type cols = greeting.size() + pad * 2 + 2;     // size_type is unsigned!!

  // write a blank to separate the output from the input
  cout << endl;

  // write nr rows of output
  // invariant: we have r rows so far
  for (int r=0; r != rows; ++r) {
    string::size_type c = 0;
    // invariant: we have written c characters so far in the current row
    while (c != cols) {
      // is it time to write the greeting?
      if (r == pad + 1 && c == pad + 1) {
        cout << greeting;
        c += greeting.size();
      }
      else {
        // are we on the border?
        if (r ==0 || r == rows - 1 || c == 0 || c == cols -1)
          cout << "*";
        else
          cout << " ";
        ++c;
      }
    }
    cout << endl;
  }
  return 0;
}
```

Pointers

(ex_pointer.cpp)

```cpp
#include <iostream>
using namespace std;
int main() {
  int x1 = 3;
  int y1 = 5;

  /* pointer notation */
  int *p;  // also correct is int* p. p is now a pointer to int, but still uninitialized
  p = &x1;  // & takes the address of a variable

  *p = 1;  // * dereferences a pointer. Note that x is now set to 1
  std::cout << "x1 = " << x1 << std::endl;

  // Pointers can be dangerous to use:
  // p = 1; *p = 258; // this compiles but will most likely crash
  // style note: raw pointers should be avoided when possible ( and this almost always )
  // the risk of catastrophic errors otherwise is too great

  /* array */
  double v[10];  // allocates memory for 10 numbers;
  for (int i=0; i < 10; ++i) v[i] = i*2. - 5;
  for (int i=0; i < 10; ++i) std::cout << "i = " << i << " v[i] = " << v[i] << std::endl;

  unsigned int n; std::cout << "Type a positive integer number :\n"; std::cin >> n;
  // float x[n];  // will not compile because n is not known at compile time

  // solution : dynamic allocation
  double *x;
  x = new double[n]; // allocate memory
  x[0] = 5.;
  delete [] x; // deleting the memory for the array; x[i] is now undefined
  // note that 'delete' is for variables, 'delete []' for arrays

  /* pointer artihmetic */
  std::cout << "Element 1 of v : " << v[1] << " " << *(v + 1) << std::endl; // v[n] is the same as *(p+n)
  std::cout << "Element 0 of v : " << *v << " " << *(&v[0]) << std::endl;   // v is in effect the address as its
      first element
  double* pv = v;
  for (int i=0; i < 10; i++)
    std::cout << "Element " << i << " of v : " << *pv++ << std::endl;                  // increment and decrement
        also work

  return 0;
}
```

Pointers

static memory allocation

```
int x1 = 3;
int y1 = 5;
```

address	value	variable
0	3	x1
4	5	y1
8		
12		
16		
20		

dynamic memory allocation

```
int *v = new int[5];
for (int i=0 ; i < 5; ++i) v[i] = i;

int* pv = v;
for (int i=0 ; i < 5; ++i) *pv++;
```

address	value	name
0	12	v
4	12	pv
8		
12	0	
16	1	
20	2	

note: the size of a pointer is 4 bytes on 32-bit machines

A little quiz

what is the output of the following program?

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;
  p2 = &secondvalue;
  *p1 = 10;
  *p2 = *p1;
  p1 = p2;
  *p1 = 20;

  cout << "firstvalue is " << firstvalue << '\n';
  cout << "secondvalue is " << secondvalue << '\n';
  return 0;
}
```

A. 20 and 10
B. 10 and 10
C. 10 and 20
D. 20 and 20

References

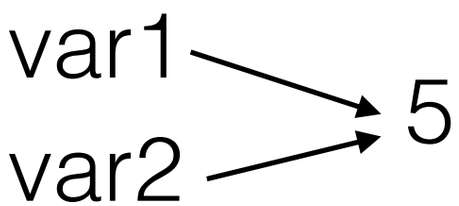
- are aliases for other variables
- characterized by the &
- they must immediately be initialized: a reference must refer to something!

```
// references
double var1 = 5.;
double &var2 = var1;          // var2 refers to the same memory allocation
var2 = 4.;
std::cout << "var 1 : " << var1 << "\tvar2 : " << var2 << std::endl;    // output is 4 4
double var3 = 2;
var2 = var3;
std::cout << "var 1 : " << var1 << "\tvar2 : " << var2 << "\tvar3 : " << var3 << std::endl;    // output is 2 2 2
var2 = 3.;
std::cout << "var 1 : " << var1 << "\tvar2 : " << var2 << "\tvar3 : " << var3 << std::endl;    // output is 3 3 2
```

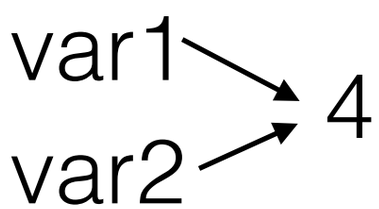
they are very useful as function arguments
and, in certain cases, as class function return values

References

step 1

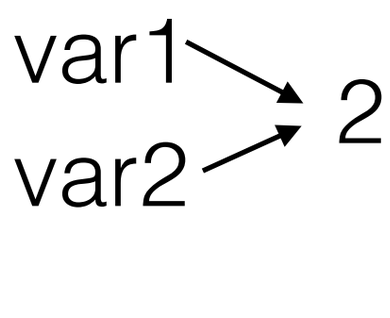


step 2



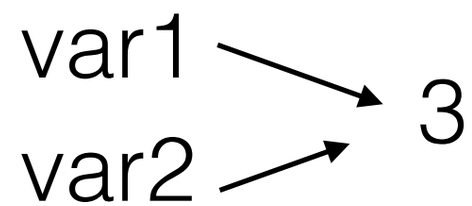
var3 —> 2

step 2



var3 —> 2

step 4



var3 —> 2

Policy of the course

- we will in this introductory course try to avoid raw pointers as much as possible

- almost all C++ code can be written with references alone

- this reduces the risk of memory leaks and is easier to debug

- read the next slide if you have more experience

References vs pointers

references and pointers are quite different:

- References refer always to something! They must be assigned upon initialization
- References cannot be reseated
- There is no “null-reference” unlike a null-pointer (null_ptr in C++11, 0 or NULL before)
- pointers use -> to access class members, references use .
- pointers have arithmetic, eg to iterate over an array, unlike references
- you can have a pointer to pointer (eg char**) etc (do not confuse with the rvalue references in C++11 written as T&&)
- you cannot have an array of references
- the overloaded operator[] should *almost always* return a reference

type casts

C++ knows 4 different types of casts:

→ static_cast<int>(5.0) : casts a double to an integer

const_cast<char *>(c) : in order to remove *const*ness, typically for function arguments

dynamic_cast<Derived*>(pointer_to_base) : only for pointers in the context of inheritance when going down the hierarchy tree (see later)

reinterpret_cast<b*>(a) : converts any pointer to any other pointer. May typically be found with function pointers.

we advise against the use of C-style casts, eg (int)(5.0) which are hard to read.

in general, type casts should only be used when no other option is possible

A little quiz

```
void increment(int i) {                    // variant 1
  i = i+1;
}

void increment_ref(int& i) {               // variant 2
  i = i+1;
}

void increment_ptr(int* i) {               // variant 3
  *i += 1;
}


int& increment_alt(int i) {                // variant 4
  int j = i+1;
  return j;
}


int main() {
  int j = 4;
  increment(j);       std::cout << j << "\n";
  increment_ref(j);   std::cout << j << "\n";
  increment_ptr(&j); std::cout << j << "\n";
  std::cout << increment_alt(j) << "\n";
  return 0;
}
```

which function call produces the correct output?

A. variant 1
B. variants 2 and 3
C. variants 2, 3 and 4
D. variant 4

A little quiz

the output of the program *may* look like:

```
Lode.Pollet@th-sv-clhead:~/Temp$ ./a.out
4
5
6
7
```

however, when compiling we got a warning:

```
Lode.Pollet@th-sv-clhead:~/Temp$ g++ quiz_fun.cpp
quiz_fun.cpp: In function ‘int& increment_alt(int)’:
quiz_fun.cpp:17:7: warning: reference to local variable ‘j’ returned [-Wreturn-local-addr]
   int j = i+1;
       ^
```

is this warning serious or can it be neglected?

unfortunately, variant 4 is flawed: it returns a reference to a local variable. This local variable goes out of scope when the function terminates, so the result is undefined. It may be that the address has not been overwritten by the program or the OS, but that would be mere luck (as in the output above)

A little quiz

a memory checker like valgrind easily spots the error:

```
Lode.Pollet@th-sv-clhead:~/Temp$ g++ -g quiz_fun.cpp
Lode.Pollet@th-sv-clhead:~/Temp$ valgrind ./a.out
…
==22407== Conditional jump or move depends on uninitialised value(s)
==22407==    at 0x4EBFCDE: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::ostreambuf_iterator<char,
std::char_traits<char> > >::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_base&, char, long) const (in /usr/lib/
x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4EC02BC: std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::do_put(std::ostreambuf_iterator<char,
std::char_traits<char> >, std::ios_base&, char, long) const (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4ECC06D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4008BE: main (quiz_fun.cpp:27)
==22407==
==22407== Use of uninitialised value of size 8
==22407==    at 0x4EBFBC3: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4EBFD05: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::ostreambuf_iterator<char,
std::char_traits<char> > >::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<char> >, std::ios_base&, char, long) const (in /usr/lib/
x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4EC02BC: std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::do_put(std::ostreambuf_iterator<char,
std::char_traits<char> >, std::ios_base&, char, long) const (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4ECC06D: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.19)
==22407==    by 0x4008BE: main (quiz_fun.cpp:27)
==22407==
```

error-free output looks like this:

```
==23387== Memcheck, a memory error detector
==23387== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==23387== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==23387== Command: ./a.out
==23387==
4
5
6
==23387==
==23387== HEAP SUMMARY:
==23387==     in use at exit: 0 bytes in 0 blocks
==23387==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==23387==
==23387== All heap blocks were freed -- no leaks are possible
==23387==
==23387== For counts of detected and suppressed errors, rerun with: -v
==23387== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Function calls

the example illustrates function calls and function arguments; pass by value, pass by reference, pass by const reference and pass by pointer

```cpp
#include <iostream>
#include <vector>


void print_message() {
  std::cout << "Welcome to the program!\n";
  // no return value when the function is void
}

double square(double x) { // computes the square
  return x*x;
}

void increment(int i) { // pass by value : a copy of the parameter i is made (the same name i is irrelevant)
  i = i+1;
  std::cout << "inside function increment i = " << i << std::endl;  // the local copy is 5
} // now the local variable goes out of scope

void increment_bis(int& i) { // pass by reference : no local copy is made
  i = i+1;
}
```

```cpp
void increment_vector(std::vector<double>& v) {
  // pass by reference, this avoids copying large sets of data, but data may be modified
  // etc
}

void increment_vector(std::vector<double> const &v) {
  // this avoids copying large sets of data and v may not be modified
  // etc
}

void increment_tres(int* i) {
    // pass by pointer, rarely needed in C++. Also avoids copying the data but has the risk of raw pointers
    (*i) += 1;
}

int main() {
  print_message();  // note that the function must have been declared before
  double x = 2.;
  std::cout << "x = " << x << " square : " << square(x) << std::endl;
  int i = 4;
  std::cout << "in main before increment: i = " << i << std::endl;
  increment(i);
  std::cout << "in main after increment: i = " << i << std::endl;   // still 4
  increment_bis(i);
  std::cout << "in main after increment_bis: i = " << i << std::endl;   // now it is 5
  increment_tres(&i); // we have to provide the address of the variable
  std::cout << "in main after increment_bis: i = " << i << std::endl;   // now it is 6
  // increment_bis(5); // will not compile because 5 is a literal constant
  return(0);
}
```

to be discussed later: templates, inline functions, function overloading, default arguments

to be discussed later: function pointers, functor objects, lambda functions

basic and advanced questions

- explain the difference between vector<int> v(50) and vector<int> v[50]
- how is -1 represented?
- how is the machine precision defined?
- what is the difference between static and dynamic memory? is there a physical difference?
- is your machine little endian or big endian? do you know of machines that use big endian?
- how does the standard library implement the swap function? has it changed in C++11?
- read the slide on references vs pointers
- check out the additional programs supplied with this lecture