Programming Techniques for Scientific Simulations

Marc Maetz

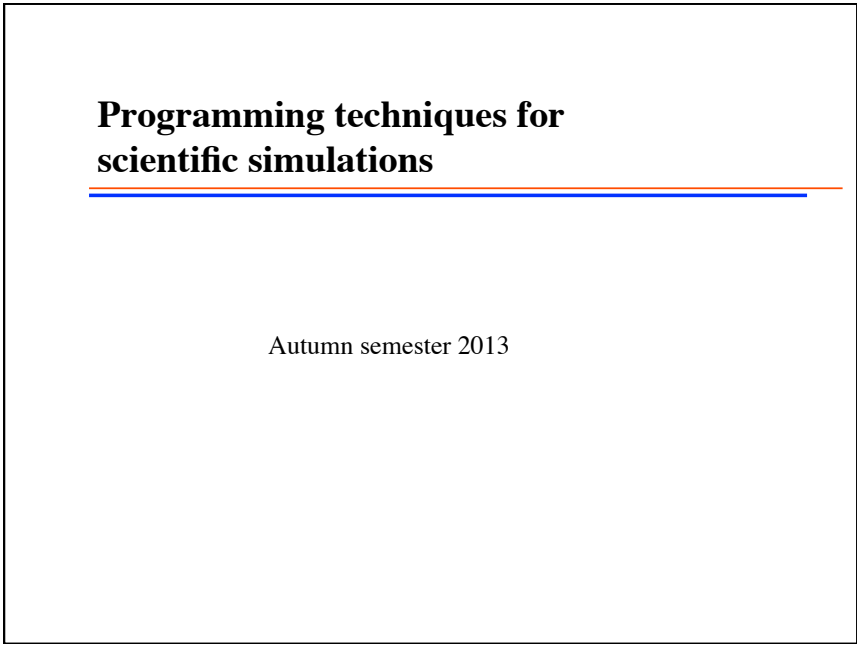
Prof. Dr. Matthias Troyer

January 2, 2014

Contents

Chapter 1

Preface



Preparing for the course

- D-PHYS account: https://admin.phys.ethz.ch/newaccount
- Software to install on your computer
  - All operating systems:
    - C++ compiler
    - git
    - CMake
  - Additionally for Linux:
    - make
  - Additionally for MacOS X:
    - Xcode with command line tools
- The assistants will help you in the exercise classes

Lecture homepage

- http://tinyurl.com/ethz-pt13
  - Enrollment key: *recursion*
- Sign up for an exercise class
- Updated regularly with lecture contents:
  - News about the class
  - Lecture notes
  - Exercise sheets
- Discussion forum: ask your classmates!

About the course

- RW (CSE) students
  - Mandatory lecture in the 3rd semester in the bachelor curriculum
- Physics students
  - Recommended course as preparation for:

    Computational Physics Courses:

    Introduction to Computational Physics (AS)
    Computational Statistical Physics (SS)
    Computational Quantum Physics (SS)

    Semester thesis in Computational Physics
    Masters thesis in Computational Physics
    PhD thesis in Computational Physics

Contents of the lecture

- Important skills for scientific software development
  - Version control
  - Build systems
  - Debugging
  - Profiling and optimization
- Advanced C++ programming
  - Object oriented programming
  - Generic programming and templates
  - Runtime and compile time polymorphism
- Libraries
  - High performance libraries: BLAS, LAPACK
  - C++ libraries: Standard library, Boost
  - Library design

Why C++?

- Generic high level programming
  - Shorter development times
  - Smaller error rate
  - Easier debugging
  - Better software reuse
- Efficiency
  - As fast or faster then FORTRAN
  - Faster than C, Pascal, …
- Job skills
  - We all need to find a job some day...

Generic programming

- Print a sorted list of all words used by Shakespeare

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <iterator>

using namespace std;

int main()
 {
   vector<string> data;
   copy(istream_iterator<string>(cin), istream_iterator<string>(), back_inserter(data));
   sort(data.begin(), data.end());
   unique_copy(data.begin(), data.end(), ostream_iterator<string>(cout,"\n"));
}
```

Why C++?

	C++	C	Java	FORTRAN	FORTRAN 95
Efficiency	√√	√	×	√√	√
Modular Programming	√	√	√	×	√
Object Oriented Programming	√	×	√	×	√
Generic Programming	√	×	×	×	×

A few quiz questions to get an overview of your knowledge

- Laptops:
  - http://eduapp.ethz.ch
- Smartphones (iOS, Android):
  - ETH Edu Application

A few quiz questions to get an overview of your knowledge

1. How are your C++ programming skills?

   A. I have never programmed at all
   B. I have never programmed in C nor C++
   C. I know some basic C
   D. I know some basic C++
   E. I know C++ well
   F. I am a C++ guru

A few quiz questions to get an overview of your knowledge

2. How is the integer value +1 represented in binary in a 16 bit integer

   A. 0000000000000000
   B. 0000000000000001
   C. 1000000000000000
   D. 1111111111111111
   E. 1000000000000001
   F. 1111111111111110

A few quiz questions to get an overview of your knowledge

3. How is the integer value -1 represented in binary in a 16 bit integer

A. 0000000000000000
B. 0000000000000001
C. 1000000000000000
D. 1111111111111111
E. 1000000000000001
F. 1111111111111110

A few quiz questions to get an overview of your knowledge

4. What is the size of the string “Hello”, i.e. the result of

```
sizeof("Hello")
```

A. 1
B. 5
C. 6
D. 7
E. 8

A few quiz questions to get an overview of your knowledge

5. What will the following code print:

```
int a=0;
std::cout << a++;
std::cout << ++a;
std::cout << a;
```

A. 012
B. 022
C. 112
D. 122
E. 123

A few quiz questions to get an overview of your knowledge

6. What is the machine precision ε?

A. The smallest floating point number that can be represented
B. The smallest positive floating point number
C. The largest number such that $1.0 + \varepsilon = 1.0$
D. The smallest number such that $1.0 + \varepsilon \neq 1.0$
E. The largest number such that $0.0 + \varepsilon = 0.0$
F. The smallest number such that $0.0 + \varepsilon \neq 0.0$

A loop example: what is wrong?

```
std::cout << "Enter a number: ";
unsigned int n;
std::cin >> n;

for (int i=1;i<=n;++i)
    std::cout << i << "\n";

int i=0;
while (i<n)
    std::cout << ++i << "\n";

i=1;
do
        std::cout << i++ << "\n";
   while (i<=n);

i=1;
while (true) {
   if(i>n) break;
   std::cout << i++ << "\n";
}
```

7. Does any of the loops not always print all positive numbers up to n?

A. All loops are wrong
B. The first loop is wrong
C. The second loop is wrong
D. The third loop is wrong
E. The fourth loop is wrong
F. All loops are correct

Consider the following five swap functions

- Five examples for swapping number

```
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;}
```

8. What will happen if we compile it?

A. All will compile
B. swap1 will not compile
C. swap2 will not compile
D. swap3 will not compile
E. swap4 will not compile
F. swap5 will not compile

Consider the following five swap functions

- Five examples for swapping number

```
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;}
```

- Now consider these calls
  - `int a=1; int b=2; swap1(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap2(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap3(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap4(&a,&b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap5(&a,&b); cout << a << " " << b << "\n";`
- 9. Which swap functions actually swap the values?

Consider the following five swap functions

- Five examples for swapping number

```
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;}
```

9. Which swap functions actually swap the values?

A. 1 and 4 will work
B. 2 and 5 will work
C. all but 3 will work
D. 1 and 2 will work
E. 3 will work
F. 4 and 5 will work

A few quiz questions to get an overview of your knowledge

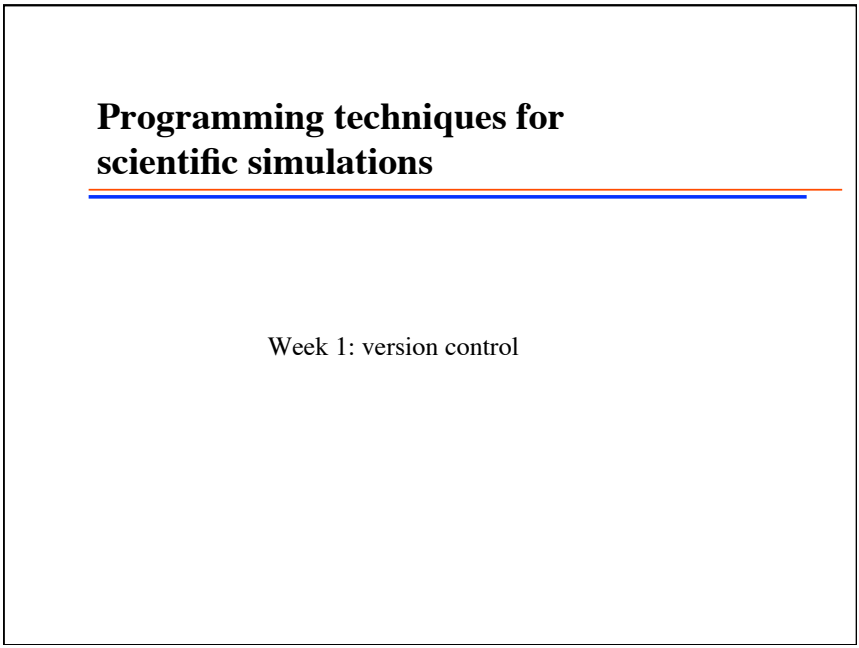
10. Do you know version control?

A. I have never heard about it
B. I have used CVS
C. I have used SVN
D. I have used GIT
E. I have used Copy&Paste
F. I know it well
G. I am a guru

A few quiz questions to get an overview of your knowledge

11. Do you know build systems?

A. I have never heard about it
B. I have used Automake
C. I have used Lego
D. I have used CMake
E. I have used Scons
F. I know it well
G. I am a guru


Programming techniques for
scientific simulations
Week 1: version control

Two people working on the same file

- What do we do if two people work on the same file?

Reference material

- We use some of the excellent material from the Software Carpentry project
- http://software-carpentry.org/v4/vc/index.html

Version control systems

- There are many different version control systems, some are
  - Distributed, one repository on a server
    - Subversion (SVN), used sometimes in this course
    - Concurrent Versioning System (CVS)
  - Decentralized, many local repositories that can be synced
    - Git, used primarily in this course
    - Mercurial (Hg)
- They can be used
  - Via command line tools
  - Via graphical user interfaces, e.g. SmartSVN and SmartGIT

Setting up git locally

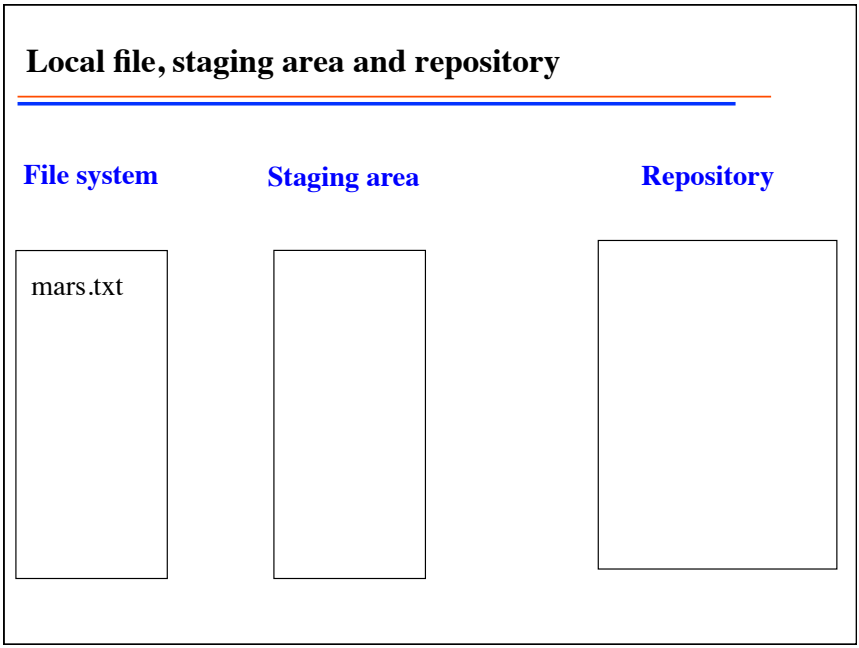
- Either use your GUI program or use the command line:
  - git config --global user.name ”Your name"
  - git config --global user.email userid@phys.ethz.ch
- More configuration options are available, check the documentation

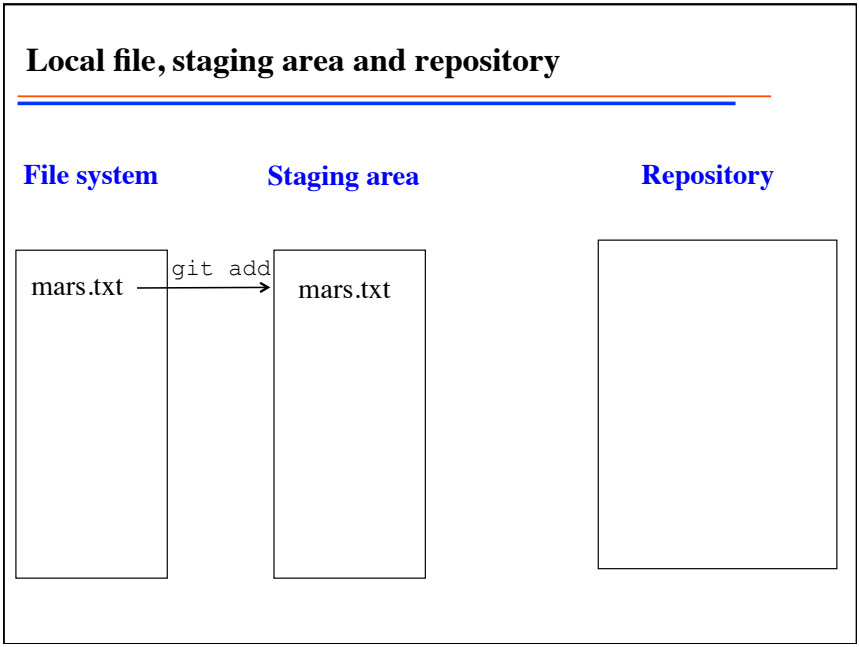
Starting to work with git

- Create a new directory and initialize a local git repository
  - mkdir project
  - cd project
  - git init
- Check the status and log
  - git status
  - git log
- Get help
  - man git
  - git --help

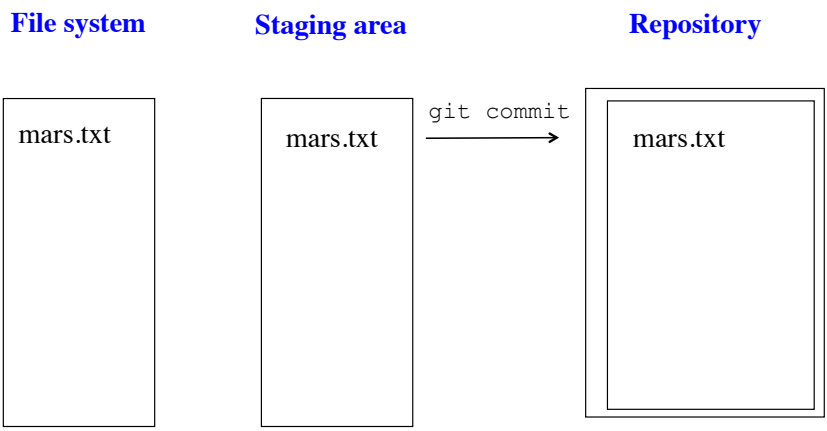
Add a file and commit it

- Write a file “mars.txt”
- Check the status!
- Now let’s add it to the “staging area”
  - git add mars.txt
- Check the status again!
- Now let us commit it to the repository, with a sensible comment
  - git commit -m "Starting to think about Mars”
- Check the status and log!


Local file, staging area and repository
File system
Staging area
Repository
mars.txt

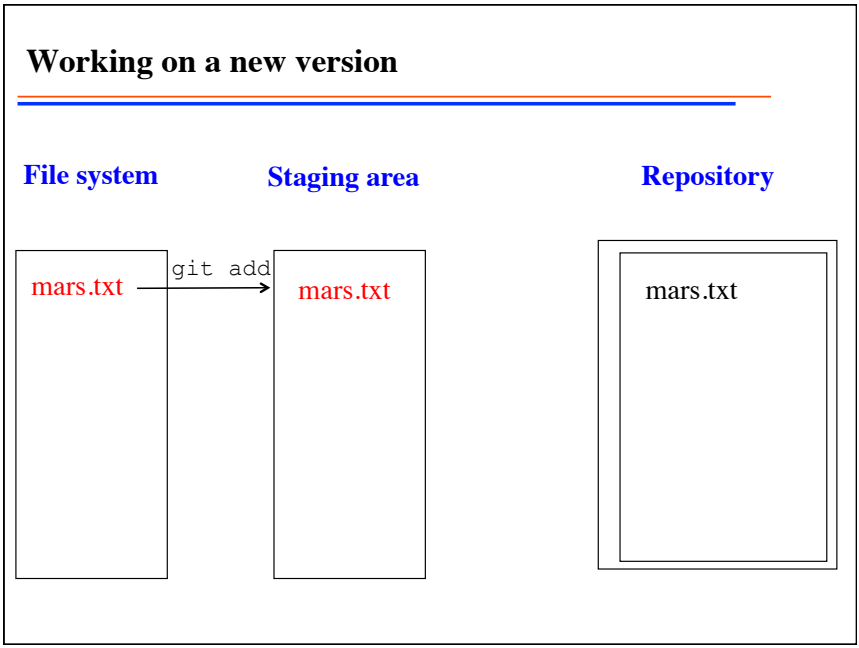

Local file, staging area and repository
File system
Staging area
Repository
mars.txt
git add
mars.txt

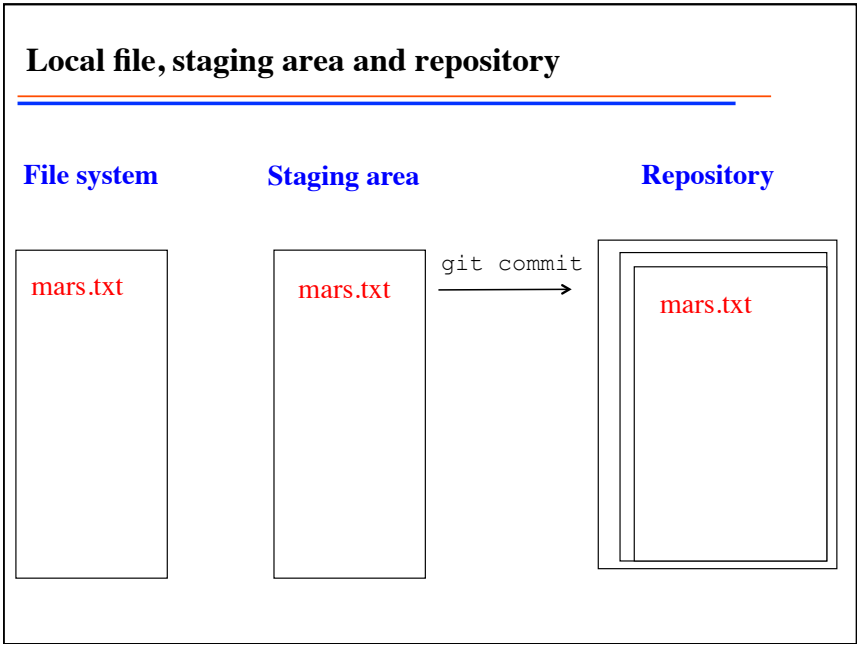
Local file, staging area and repository



Edit the file

- Change the file
- Check the status and the difference
  - git status
  - git diff
- Add it to the staging area
  - git add mars.txt
- Check the status and diff once more
- Commit the next version
  - git commit –m ”Thinking more about Mars”
- Check the status and log!


Working on a new version
File system
Staging area
Repository
mars.txt
git add
mars.txt
mars.txt


Local file, staging area and repository
File system
Staging area
Repository
mars.txt
mars.txt
git commit
mars.txt

Collaborating needs a common repository

- We need to publish our changes somewhere
- Your own server infrastructure (needs work to set up)
- github.com (free for open source access)
- gitlab.phys.ethz.ch
  - will be used for the exercises
  - lets you control who has read and write access
  - your data is in Switzerland and harder to get for the NSA

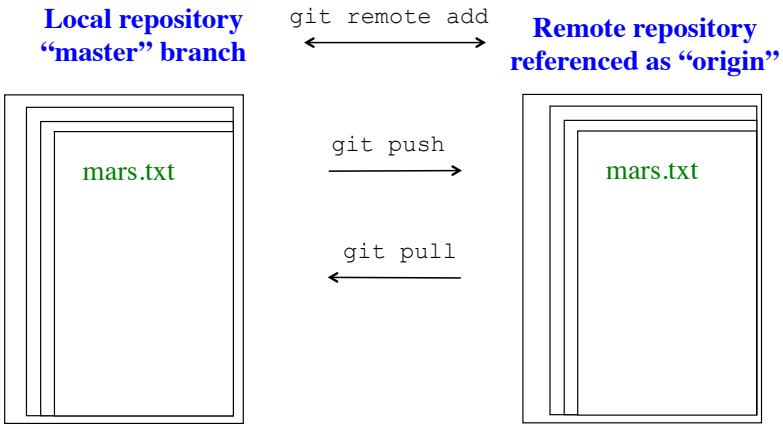
Setting up the gitlab repository

- Log on to gitlab.phys.ethz.ch and set up your account
- Create an SSH key pair (see exercise instructions) at http://tinyurl.com/ethz-pt13
- Upload the public key to gitlab
- Create a new project and follow the instructions

Pushing changes to the repository on the server

- Follow the instructions given by gitlab
- Connect the local and remote repository
  - git remote add origin git@gitlab.phys.ethz.ch:……
- And push the changes from the local to the remote repository
  - git push –u origin master

Connecting the repositories



Pulling changes and resolving conflicts

- Someone edits the file and pushes their changes
- We are prevented from pushing our changes before we merged theirs, thus let's pull the changes:
  - git pull
- If there are conflicts they are marked and not to be resolved before we can commit and push the files again
  - git add mars.txt
  - git commit -m "resolved conflicts"
  - git push

Removing and unstaging files

- Go back to the last comitted version
  - git checkout -- mars.txt
- Unstage a version
  - git rm --cached mars.txt
- Remove a file
  - git rm mars.txt

Undoing changes

- Roll back everything to the last commit
  - git reset --hard HEAD
- Roll back everything to one revision before the last commit
  - git reset --hard HEAD~1
- 5 revisions before
  - git reset --hard HEAD~5
- A specific revision (can be truncated to a few characters)
  - git reset --hard f22b25e3233b4645dabd0d81e651fe074bd8e73b
  - git reset --hard f22b25

Seeing differences to previous versions

- Use git diff but give a revision number (can be truncated to a few characters)
  - git diff f22b25e3233b4645dabd0d81e651fe074bd8e73b mars.txt
  - git diff f22b25 mars.txt
  - git diff HEAD~5 mars.txt

Branching

- Sometimes you want to make some changes but don't mess up the working version: create a new branch
  - git branch moons
- Now look at the branches
  - git branch
- Switch to the new branch "moons"
  - git checkout moons

Merging

- Create and add a new file "moons.txt", then switch back to master
  - git add moons.txt
  - git commit –m "Discussing moons"
  - git checkout master
- "moons.txt" is gone, since it's only on the moons branch
- We can merge the changes done on moons back to master
  - git merge moons
- Merging between branches can create conflicts that need to be resolved

More on working with branches

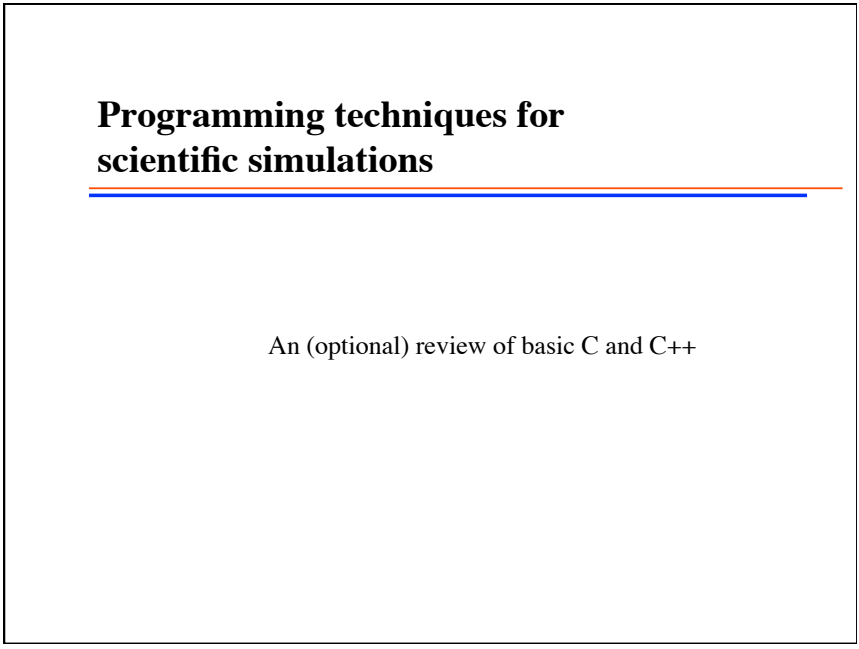
- Pushing a new branch to the remote repository referred to by “origin”
  - git push origin moons
- Deleting a local branch
  - git branch –d moons
- Deleting a remote branch from the remote repository referred to by “origin”
  - git push origin :moons

Advice

- Always use version control, for any project
- Commit often
- Use good commit messages
- Use branches to add new features to a working version
- Commit often
- Submit your solutions by using git, so that the assistants can easily see the code and comment in the files
- Give the assistants access to your repositories

Chapter 2

Introduction



A first C++ program

```
/* A first program */

#include <iostream>

using namespace std;

int main()
{
  cout << "Hello students!\n";
 // std::cout without the using declaration
 return 0;
}
```

- /* and */ are the delimiters for comments
- includes declarations of I/O streams
- declares that we want to use the standard library ("std")
- the main program is always called "main"
- "cout" is the standard output stream.
- "<<" is the operator to write to a stream
- statements end with a ;
- // starts one-line comments
- A return value of 0 means that everything went OK

More about the std namespace

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello\n";
}
```

```
#include <iostream>
using std::cout;
int main()
{
  cout << "Hello\n";
}
```

```
#include <iostream>
int main()
{
  std::cout << "Hello\n";
}
```

- All these versions are equivalent
- Feel free to use any style in your program
- Never use using statements globally in libraries!

A first calculation

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

using namespace std;

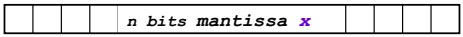
int main()
{
 cout << "The square root of 5 is"
 << sqrt(5.) << "\n";
 return 0;
}
```

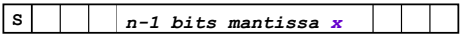
- `<cmath>` is the header for mathematical functions
- Output can be connected by `<<`
- Expressions can be used in output statements
- What are these constants?
  - `5.`
  - `0`
  - `"\n"`

Integral data types

- Signed data types
  - `short, int, long, long long`
  - since C++11: `int8_t, int16_t, int32_t, int64_t`
- Unsigned data types
  - `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`
  - since C++11: `uint8_t, uint16_t, uint32_t, uint64_t`
- Are stored as binary numbers of length
  - `short`: usually 16 bit
  - `int`: usually 32 bit
  - `long`: usually 32 bit on 32-bit CPUs and 64 bit on 64-bit CPUs
  - `long long`: usually 64 bits

Integer representations

- An *n*-bit integer is stored in *n*/8 bytes
  - Little-endian: least significant byte first
  - Big-endian: most significant byte first
  - Exercise: write a program to check the format of your CPU
- Unsigned 

  - ***x*** just stored as *n* bits, values from 0 … 2^n-1
- Signed 

  - Stored as 2's complement, values from -2^{n-1} … $2^{n-1}-1$
  - Highest bit is sign **S**
  - ***x*** ≥ 0 : **S**=0, rest is ***x***
  - ***x*** < 0 : **S**=1, rest is `~(-x -1)`
  - Advantage of this format: signed numbers can be added like unsigned

Integer constants

- Integer literals can be entered in a natural way
- Suffixes specify type (if needed)
  - int: `0`, `-3`, ….
  - unsigned int: `3u`, `7U` ,...
  - short: `0S`, `-5s`, ...
  - unsigned short: `1us`, `9su`, `6US`, ...
  - long: `0L`, `-5l`, ...
  - unsigned long: `1ul`, `9Lu`, `6Ul`, ...
  - long long: `0LL`, `-5ll`, ...
  - unsigned long long: `1ull`, `9LLu`, `6Ull`, ...

Characters

- Character types
  - Single byte: `char, unsigned char, signed char`
    - Uses ASCII standard
  - Multi-byte (e.g. for Japanese: 大): `wchar_t`
    - Unfortunately is not required to use Unicode standard
- Character literals
  - `'a', 'b', 'c', '1', '2', …`
  - `'\t'` … tabulator
  - `'\n'` … new line
  - `'\r'` … line feed
  - `'\0'` … byte value 0

Strings

- String type
  - C-style character arrays `char s[100]` should be avoided
  - C++ class `std::string` for single-byte character strings
  - C++ class `std::wstring` for multi-byte character strings
- String literals
  - `"Hello"`
  - Contain a trailing '\0', thus `sizeof("Hello")==6`

Boolean (logical) type

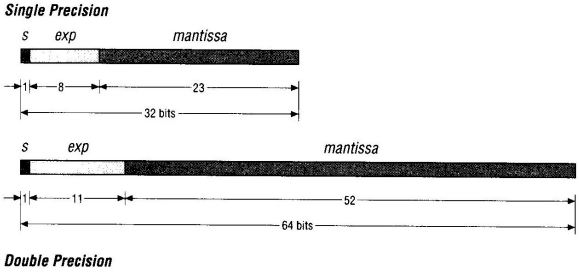
- Type
  - `bool`
- Literal
  - `true`
  - `false`

Floating point numbers

- Floating point types
  - single precision: `float`
    - usually 32 bit
  - double precision: `double`
    - Usually 64 bit
  - extended precision: `long double`
    - Often 64 bit (PowerPC), 80 bit (Pentium) or 128 bit (Cray)
- Literals
  - single precision: `4.562f`, `3.0F`
  - double precision: `3.1415927`, `0.`
  - extended precision: `6.54498467494849849489L`

IEEE floating point representation

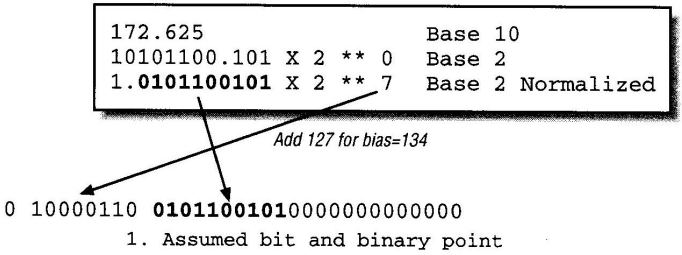
- The 32 (64) bits are divided into sign, exponent and mantissa



Type	Exponent	Mantissa	Smallest	Largest	Base 10 accuracy
`float`	8	23	1.2E-38	3.4E+38	6-9
`double`	11	52	2.2E-308	1.8E+308	15-17

Converting to/from IEEE representation

- Sign
  - Positive: 0, Negative: 1
- Mantissa
  - Left shifted until leftmost digit is 1, other digits are stored
- Exponent
  - Half of the range (127 for float, 1023 for double) is added



Floating point arithmetic

- Truncation can happen because of finite precision

$$\begin{array}{r} 1.00000 \\ 0.0000123 \\ \hline 1.00001 \end{array}$$

- Machine precision ε is smallest number such that $1+\varepsilon \neq 1$
  - Exercise: calculate ε for `float`, `double` and `long double` on your machine
- Be very careful about roundoff
  - For example: sum numbers starting from smallest to largest
  - See examples provided

Implementation-specific properties of numeric types

- defined in header `<limits>`
- `numeric_limits<T>::is_specialized` // is true if information available
- most important values for integral types
  - `numeric_limits<T>::min()` // minimum (largest negative)
  - `numeric_limits<T>::max()` // maximum
  - `numeric_limits<T>::digits` // number of bits (digits base 2)
  - `numeric_limits<T>::digits10` // number of decimal digits
  - and more: `is_signed`, `is_integer`, `is_exact`, ...
- most important values for floating point types
  - `numeric_limits<T>::min()` // minimum (smallest nonzero positive)
  - `numeric_limits<T>::max()` // maximum
  - `numeric_limits<T>::digits` // number of bits (digits base 2)
  - `numeric_limits<T>::digits10` // number of decimal digits
  - `numeric_limits<T>::epsilon()` // the floating point precision
  - and more: `min_exponent`, `max_exponent`, `min_exponent10`, `max_exponent10`, `is_integer`, `is_exact`
- first example of templates, use by replacing T above by the desired type: `std::numeric_limits<double>::epsilon()`

A more useful program

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

- a variable named 'x' of type 'double' is declared
- a double value is read and assigned to x
- The square root is printed

Variable declarations

- have the syntax: `type variablelist;`
  - `double x;`
  - `int i,j,k;` // multiple variables possible
  - `bool flag;`
- can appear anywhere in the program

```
int main() {
…
double x;
}
```

- can have initializers, can be constants
  - `int i=0;` // C-style initializer
  - `double r(2.5);` // C++-style constructor
  - `const double pi=3.1415927;`

Advanced types

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

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

- **Arrays** of size n

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

  - indices run from 0 … n-1! (FORTRAN: 1…n)
  - last index changes fastest (opposite to FORTRAN)
  - Should not be used in C++ anymore!!!

- Complex types can be given a new name

```
typedef double[10] vector10;
vector10 v={0,1,4,9,16,25,36,49,64,81};
vector10 mat[10]; // actually a matrix!
```

Expressions and operators

- Arithmetic
  - multiplication: `a * b`
  - division: `a / b`
  - remainder: `a % b`
  - addition: `a + b`
  - subtraction: `a - b`
  - negation: `-a`
- Increment and decrement
  - pre-increment: `++a`
  - post-increment: `a++`
  - pre-decrement: `--a`
  - post-decrement: `a--`
- Logical (result bool)
  - logical not: `!a`
  - less than: `a < b`
  - less than or equal: `a <= b`
  - greater than: `a > b`
  - greater than or equal: `a >= b`
  - equality: `a == b`
  - inequality: `a != b`
  - logical and: `a && b`
  - logical or: `a || b`
- Conditional: `a ? b : c`
- Assignment: `a = b`

Bitwise operations

- Bit operations
  - bitwise not: `~a`
    - inverts all bits
  - left shift: `a << n`
    - shifts all bits to higher positions, fills with zeros, discards highest
  - right shift: `a >> n`
    - shifts to lower positions
  - bitwise and: `a & b`
  - bitwise xor: `a ^ b`
  - bitwise or: `a | b`
- The **bitset** class implements more functions. We will use it later in one of the exercises.
- Interested students should refer to the recommended C++ books

- The shift operators have been redefined for I/O streams:
  - `cin >> x;`
  - `cout << "Hello\n";`
- The same can be done for all new types: "operator overloading"
- Example: **matrix operations**
  - `A+B`
  - `A-B`
  - `A*B`

Compound assignments

- `a *= b`
- `a /= b`
- `a %= b`
- `a += b`
- `a -= b`
- `a <<= b`
- `a >>= b`
- `a &= b`
- `a ^= b`
- `a |= b`

- `a += b` equivalent to `a=a+b`
- allow for simpler codes and better optimizations

Special operators

- scope operators: `::`
- member selectors
  - `.`
  - `->`
- subscript `[]`
- function call `()`
- construction `()`
- `typeid`
- casts
  - `const_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
  - `static_cast`
- `sizeof`
- `new`
- `delete`
- `delete[]`
- pointer to member select
  - `.*`
  - `->*`
- `throw`
- comma `,`
- all these will be discussed later

Operator precedences

- Are listed in detail in all reference books or look at http://www.cppreference.com/operator_precedence.html
- Arithmetic operators follow usual rules
  - `a+b*c` is the same as `a+(b*c)`
- Otherwise, *when in doubt use parentheses*

Statements

- simple statements
  - `;` // null statement
  - `int x;` // declaration statement
  - `typedef int index_type;` // type definition
  - `cout << "Hello world";` // all simple statements end with ;
- compound statements
  - more than one statement, enclosed in curly braces

```
{
 int x;
 cin >> x;
 cout << x*x;
}
```

The if statement

- Has the form

```
if (condition)
  statement
```

- or

```
if (condition)
  statement
else
  statement
```

- can be chained

```
if (condition)
  statement
else if(condition)
  statement
else
  statement
```

- Example:

```
if (light == red)
  cout << "STOP!";
else if (light == orange)
  cout << "Attention";
else {
  cout << "Go!";
}
```

The switch statement

- can be used instead of deeply nested if statements:

```
switch (light) {
    case red:
     cout << "STOP!";
     break;
    case orange:
     cout << "Attention";
     break;
    case green:
     cout << "Go!";
     go();
     break;
    default:
     cerr << "illegal color";
     abort();
}
```

- do not forget the `break`!
- always include a default!
  - the telephone system of the US east coast was once disrupted completely for several hours because of a missing default!
- also multiple labels possible:

```
switch(ch) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
     cout << "vowel";
    break;

  default:
     cout << "other character";
  }
```

The for loop statement

- has the form

```
for (init-statement ; condition ; expression)
    statement
```

- example:
  -
```
for (int i=0;i<10;++i)
   cout << i << "\n";
```

- can contain more than one statement in for(;;), but this is very bad style!
  -
```
double f;
int k;
for (k=1,f=1 ; k<50 ; ++k, f*=k)
   cout << k << "! = " << f<< "\n";
```

The while statement

- is a simpler form of a loop:

```
while (condition)
   statement
```

- example:

```
while (trafficlight()==red) {
      cout << "Still waiting\n";
      sleep(1);
   }
```

The do-while statement

- is similar to the while statement

```
do
   statement
while (condition);
```

- Example

```
do {
   cout << "Working\n";
   work();
} while (work_to_do());
```

The break and continue statements

- **break** ends the loop immediately and jumps to the next statement following the loop
- **continue** starts the next iteration immediately
- An example:

```
while (true) {
  if (light()==red)
    continue;
  start_engine();
  if(light()==orange)
    continue;
  drive_off();
  break;
}
```

A loop example: what is wrong?

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Enter a number: ";
  unsigned int n;
  cin >> n;

  for (int i=1;i<=n;++i)
    cout << i << "\n";

  int i=0;
  while (i<n)
    cout << ++i << "\n";

  i=1;
  do
    cout << i++ << "\n";
  while (i<=n);

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

The goto statement

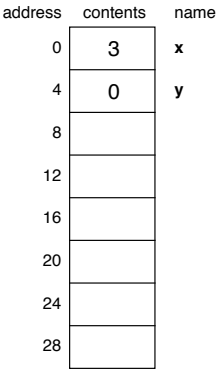
- will not be discussed as it should not be used
- included only for machine produced codes,
  e.g. FORTRAN -> C translators
- can always be replaced by one of the other control structures
- **we will not allow any `goto` in the exercises!**

Static memory allocation

- Declared variables are assigned to memory locations

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

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



Pointers

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

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

  - Are initialized using the & operator

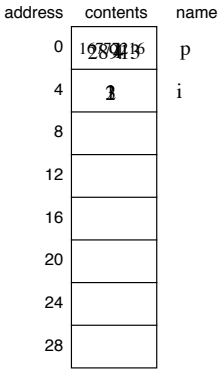
```
int i=3;
p =&i; // & takes the address of a variable
```

  - Are dereferenced with the * operator

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

  - Can be dangerous to use

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



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

Dynamic allocation

- Automatic allocation
  - `float x[10]; // allocates memory for 10 numbers`
- Allocation of flexible size
  - `unsigned int n; cin >> n; float x[n]; // will not work`
  - The compiler has to know the number!
- Solution: dynamic allocation
  - `float *x=new float[n]; // allocate some memory for an array`
  - `x[0]=…;… // do some work with the array x`
  - `delete[] x; // delete the memory for the array. x[i], *x now undefined!`
- Don't confuse
  - `delete`, used for simple variables
  - `delete[]`, used for arrays

Pointer arithmetic

- for any pointer `T *p;` the following holds:
  - `p[n]` is the same as `*(p+n);`
- Adding and integer *n* to a pointer increments it by the *n* times the size of the type – and not by *n* bytes
- Increment ++ and decrement -- increase/decrease by one element

- Be sure to only use valid pointers
  - initialize them
  - do not use them after the object has been deleted!
  - catastrophic errors otherwise

Arrays and pointers

- are very similar, but subtly different!

```
int array[5];

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

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


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

- see these examples!

```
int* pointer=new int[5];

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

int* p = pointer;
for (int i=0;i < 5; ++i)
  cout << *p++;
```

-
```
p=pointer;
delete[] p; // is OK
delete[] pointer; // crash
delete[] p; // will crash
p=0; // is OK
pointer=0; // is OK
```

A look at memory: array example

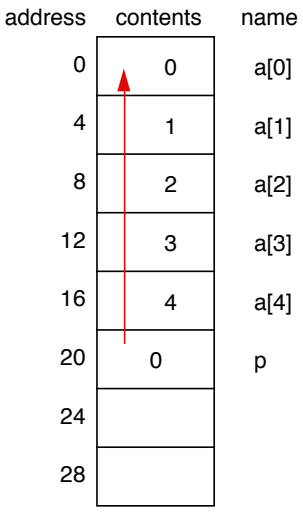
- Array example

```
int array[5];

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

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


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



A look at memory: pointer example

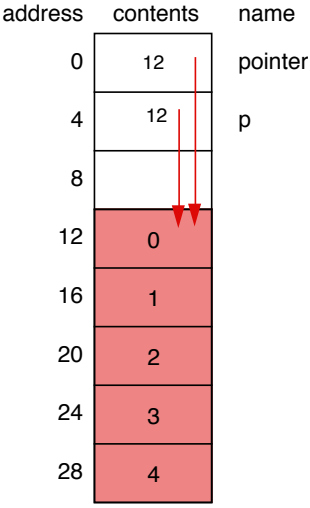
- Array example

```
int* pointer=new int[5];

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

int* p = pointer;
for (int i=0;i < 5; ++i)
  cout << *p++;

delete[] pointer; // is OK
delete[] pointer; // crash
delete[] p; // will crash
p=0; // is OK
pointer=0; // is OK
```



References

- are aliases for other variables:

```
float very_long_variabe_name_for_number=0;

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

x=5; // sets very_long_variabe_name_for_number to 5;

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

A more flexible program: function calls

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

float square(float  x) {
  return x*x;
}

int main() {
  cout << "Enter a number:\n";
  float x;
  cin >> x;
  cout << x << " " <<
  square(x) << "\n";
  return 0;
}
```

- a function "`square`" is defined
  - return value is `float`
  - parameter `x` is `float`
- and used in the program

Function call syntax

- syntax:

returntype functionname
(*parameters* **)**
{
functionbody
}

- *returntype* is "void" if there is no return value:

```
void error(char[] msg) {
  cerr << msg << "\n";
}
```

- There are several kinds of parameters:
  - pass by value
  - pass by reference
  - pass by const reference
  - pass by pointer
- Advanced topics to be discussed later:
  - inline functions
  - default arguments
  - function overloading
  - template functions

Pass by value

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

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

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

- Copying of variables time consuming for large objects like matrices

Pass by reference

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

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

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

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

Pass by const reference

- Problem:
  - `vector eigenvalues(Matrix& A);` // allows modification of A
  - `vector eigenvalues(Matrix A);` // involves copying of A
- how do we avoid copying and prohibit modification?
  - `vector eigenvalues (Matrix const &A);`
  - now a reference is passed -> no copying
  - the parameter is const -> cannot be modified

Pass by pointer

- Another method to pass an object without copying is to pass its address
- Used mostly in C

- `vector eigenvalues(Matrix *m);`

- disadvantages:
  - The parameter must always be dereferenced: `*m;`
  - In the function call the address has to be taken:

```
Matrix A;
cout << eigenvalues(&A);
```

- rarely needed in C++

A swap example

- Five examples for swapping number
  - `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;}`

- Which will compile?
- What is the effect of:
  - `int a=1; int b=2; swap1(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap2(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap3(a,b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap4(&a,&b); cout << a << " " << b << "\n";`
  - `int a=1; int b=2; swap5(&a,&b); cout << a << " " << b << "\n";`

Type casts: static_cast, reinterpret_cast

- Variables can be converted (cast) from one type to another
- **`static_cast`** converts one type to another, using the best defined conversion, e.g.
  - `float y=3.f;`
  - `int x = static_cast<int>(y);`
  - replaces the C construct `int x= (int) y;`
- **`reinterpret_cast`** converts one pointer type to another, but only useful for low-level programming, for example to look at representations of floating point numbers or check for endianness
  - `float y=3.f;`
  - `float *fp = &y;`
  - `int *ip = reinterpret_cast<int*>(fp);`
  - `std::cout << *ip;`

Type casts: const_cast

- const_cast can be used to remove const-ness from a variable
  - Example: need to pass a `double*` to a C-style function which does not change the value, but I only have a `const double*`

```
void legacy_c_function (double* d);

void foo(const double* d) {
   // remove the const
   double* nonconst_d = const_cast<double*>(d);
   // now call the function
   legacy_c_function(nonconst_d);
 }
```

  - Use it very sparingly. Usually the need for `const_cast` is a sign of bad software design
- Other casts to be discussed later:
  - `dynamic_cast`
  - `boost::lexical_cast`
  - `boost::numeric_cast`

Namespaces

- What if a `square` function is already defined elsewhere?
- C-style solution: give it a unique name; ugly and hard to type

```
float ETH_square(float);
```

- Elegant C++ solution: namespaces
  - Encapsulates all declarations in a module, called "namespace", identified by a prefix
  - Example:

```
namespace ETH
{
  float square(float);
}
```

- Namespaces can be nested
- Can be accessed from outside as:
  - `ETH::square(5);`
  - `using ETH::square; square(5);`
  - `using namespace ETH; square(5);`
- Standard namespace is `std`
- For backward compatibility the standard headers ending in .h import `std` into the global namespace. E.g. the file "iostream.h" is:

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

Default function arguments

- are sometimes useful

```
float root(float x, unsigned int n=2); // n-th root of x

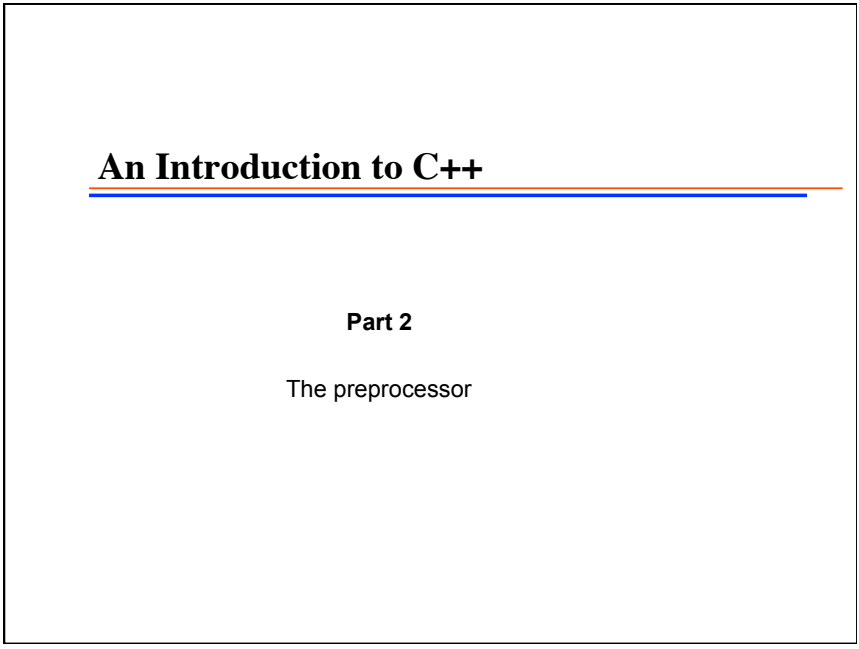
int main()
{
  root(5,3); // cubic root of 5
  root(3,2); // square root of 3
  root(3); // also square root of 3
}
```

- the default value must be a constant!

```
unsigned int d=2;
float root(float x, unsigned int n=d); // not allowed!
```

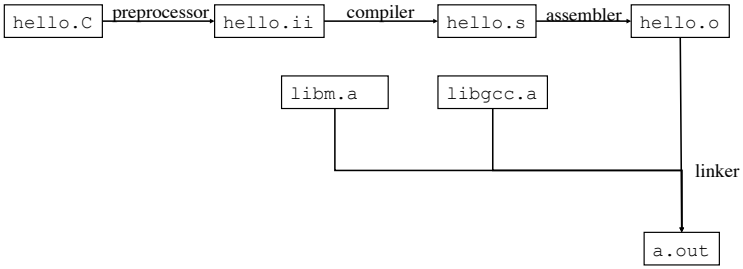
Chapter 3

Preprocessor



Steps when compiling a program

- What happens when we type the following?
  `g++ hello.C`
- Observe the steps by adding some extra flags:
  `g++ --verbose -save-temps hello.C`



The C++ preprocessor

- Is a text processor, manipulating the source code
- Commands start with #
  - `#define XXX`
  - `#define YYY 1`
  - `#define ADD(A,B) A+B`
  - `#undef ADD`
  - `#ifdef XXX`
    `#else`
    `#endif`
  - `#if defined(XXX) && (YYY==1)`
    `#elif defined (ZZZ)`
    `#endif`
  - `#include <iostream>`
  - `#include “square.h”`

#define

- Defines a preprocessor macro
  - `#define XXX “Hello”`
    `std::cout << XXX;`
  - Gets converted to
    `std::cout << “Hello”`
- Macro arguments are possible
  - `#define SUM(A,B) A+B`
    `std::cout << SUM(3,4);`
  - Gets converted to
    `std::cout << 3+4;`
- Definitions on the command line possible
  - `g++ -DXXX=3 -DYYY`
  - Is the same as writing in the first line:
    `#define XXX 3`
    `#define YYY`

#undef

- Undefines a macro
  - `#define XXX “Hello”`
    `std::cout << XXX;`
    `#undef XXX`
    `std::cout << “XXX”;`
  - Gets converted to
    `std::cout << “Hello”`
    `std::cout << “XXX”`
- Undefines on the command line are also possible
  - `g++ -UXXX`
  - Is the same as writing in the first line:
    `#undef XXX`

Looking at preprocessor output

- Running only the preprocessor:
  - `c++ -E`
- Running the full compile process but storing the preprocessed files
  - `c++ -save-temps`
- Look at the files pre1.C and pre2.C, then at the output of
  - `c++ -E pre1.C`
  - `c++ -E pre2.C`
  - `c++ -E -DSCALE=10 pre2.C`

#ifdef … #endif

- Conditional compilation can be done using #ifdef
  - ```
    #ifdef SYMBOL
      something
    #else
     somethingelse
    #endif
    ```
  - Becomes, if SYMBOL is defined:
    ```
    something
    ```
  - Otherwise it becomes
    ```
    somethingelse
    ```
- Look at the output of
  - `c++ -E pre3.C`
  - `c++ -DDEBUG -E pre3.C`
    ```

#if … #elif … #endif

- Allows more complex instructions, e.g.
  - ```
    #if !defined (__GNUC__)
      std::cout << " A non-GNU compiler";
    #elif __GNUC__<=2 && _GNUC_MINOR < 95
      std::cout << "gcc before 2.95";
    #elif __GNUC__==2
      std::cout << "gcc after 2.95";
    #elif __GNUC__>=3
      std::cout << "gcc version 3 or higher";
    #endif
    ```

#error

- Allows to issue error messages

```
#if !defined(__GNUC__)
#error This program requires the GNU compilers
#else
…
#endif
```

- Try the following
  - `c++ -c pre4.C`
```

#include "file.h" #include <iostream>

- Includes another source file at the point of invocation
- Try the following
  - `c++ -E pre5.C`
- `< >` brackets refer to system files, e.g. `#include <iostream>`
  - `c++ -E pre6.C`
- With -I you tell the compiler where to look for include files. Try:
  - `c++ -E pre7.C`
  - `c++ -E -Iinclude pre7.C`

Segmenting programs

- Programs can be
  - split into several files
  - Compiled separately
  - and finally linked together
- However functions defined in another file have to be declared before use!
- The function declaration is similar to the definition
  - but has no body!
  - parameters need not be given names
- Easiest solution are header files. Help maintain consistency.

- file "square.h"

```
double square(double);
```

- file "square.C"

```
#include "square.h"
double square(double x) {
  return x*x;
}
```

- file "main.C"

```
#include <iostream>
#include "square.h"

int main() {
  std::cout << square(5.);
}
```

Compiling and linking

- Compile the file square.C, with the -c option (no linking)
  - `c++ -c square.C`
- Compile the file main.C, with the -c option (no linking)
  - `c++ -c main.C`
- Link the object files
  - `c++ main.o square.o`

Include guards

- The following fails to compile :
  - ```
    #include “incl.h”
    #include “incl.h”
    ```
- Try it:
  - `c++ -c guard.C`
- Add include guards to incl.h and try again:
  - ```
    #ifndef SQUARE_H
    #define SQUARE_H

    int x;
    #endif
    ```

Assert in header <cassert>

- are a way to check preconditions, postconditions and invariants
- <cassert> looks something like:

```
#ifdef NDEBUG
#define assert(e)        ((void)0)
#else
#define assert(e) …
#endif
```

- If the expression is false the program will abort and print the expression with a notice that this assertion has failed
- Try it
  - `c++ assert.C`

Making a library on Linux/Unix/MacOS X

- Often used *.o files can be packed into a library, e.g.:
  -
```
ar ruc libtest.a square.o
ranlib libtest.a
g++ main.C -L. -ltest
```
- `ar` creates an archive, more than one object file can be specified
  - The name must be `lib`*`something`*`.a`
- `ranlib` adds a table of contents (not needed on some platforms)
- `-L` specifes the directory where the library is located
- `-l`*`something`* specifies looking in the library `lib`*`something`*`.a`

How libraries work

- What is done here:
  - `c++ main.C -L. -ltest`
- After compilation the object files are linked
- If there are undefined functions (e.g. `square`) the libraries are searched for the function, and the needed functions linked with the object files
- Note that the order of libraries is important
  - if `liba.a` calls a function in `libb.a`, you need to link in the right order: `-la -lb`

Documenting your library

- After you finish your library, document it with
  - Synopsis of all functions, types and variables declared
  - Semantics
    - what does the function do?
  - Preconditions
    - what must be true before calling the function
  - Postconditions
    - what you guarantee to be true after calling the function if the precondition was true
  - What it depends on
  - Exception guarantees (will be discussed later)
  - References or other additional material

Example documentation

- Header file “square.h” contains the function “square”:
  - Synopsis:
    ```
    double square(double x);
    ```
  - `square` calculates the square of x
  - Precondition: the square can be represented in a double
    ```
    std::abs(x) <= std::sqrt(std::numeric_limits<double>::max())
    ```
  - Postcondition: the square root of the return value agrees with the absolute value of x within floating point precision:
    ```
    std::sqrt(square(x)) - std::abs(x) <=
    std::abs(x) *std::numeric_limits<double>::epsilon()
    ```
  - Dependencies: none
  - Exception guarantee: no-throw

The cost of a function call

- A function call is expensive:
  - Values in registers might need to be saved in memory
  - Function arguments might need to be stored in memory
  - A jump to the function is done, stopping all pipelines
  - Function arguments might need to be read from memory
  - Only then can the function start to execute
- Let us look at the assembly code of a simple example
  - `g++ -c -save-temps -O0 functioncall.c`
  - `g++ -c -save-temps -O3 functioncall.c`
  - `g++ -c -save-temps -finline-functions functioncall.c`
- Look at functioncall.s - What can you observe?
  - Can you observe automatic “inlining”?

Inlining

- A function call takes several hundred CPU cycles
- For simple functions that are called often this is a big waste of time:
  - `float square(float);`

```
int main() {
  float sq[10000];
  for (int k=0;k<10000;++k)
    sq[k] = square(k);
}
```

- It is better to inline the function
  - `inline float square(float x) {return x*x;}`
- This leads to the same optimized code as:
  - `sq[k] = float(k)*float(k);`
- Note that for an inline function not only the declaration but the complete function body must be in the header file!

Chapter 4

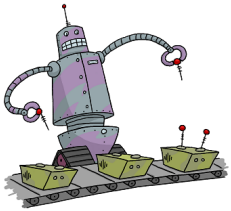
Automated Builds

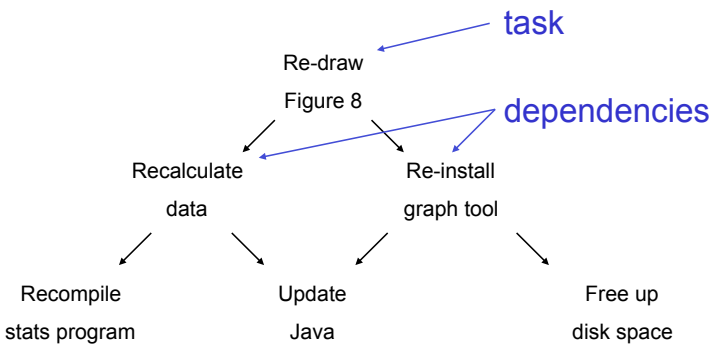
Automated Builds

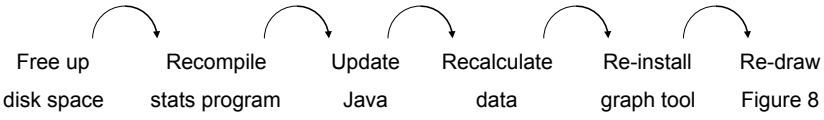
Introduction

software carpentry

A typical working day







This pattern arises frequently

New data collected?

Recalculate statistics

Source files changed?

Recompile program

New content written?

Update web site

Hard or impossible to keep track of:

– what depends on what

– what's up-to-date and what isn't

"Anything worth repeating is worth automating."

So use a *build manager* to automate the process

Describe dependencies in a *build file*

Along with commands used to update things

Build manager does the rest

software carpentr

Most widely used build manager is Make

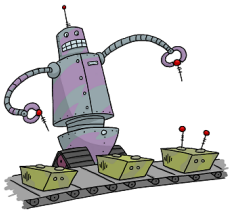
Invented by a student intern at Bell Labs in 1975

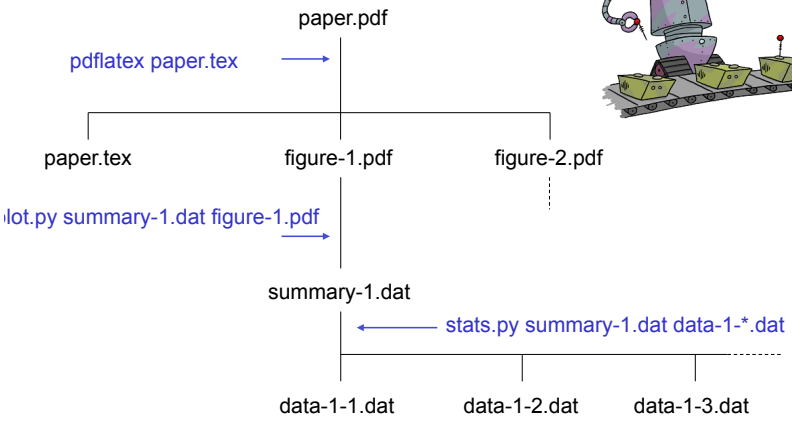
Has grown into a little programming language

A very cryptic little language, without a debugger...

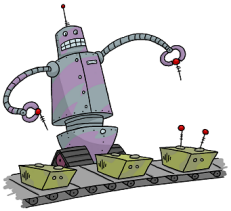
...that requires an understanding of the Unix shell

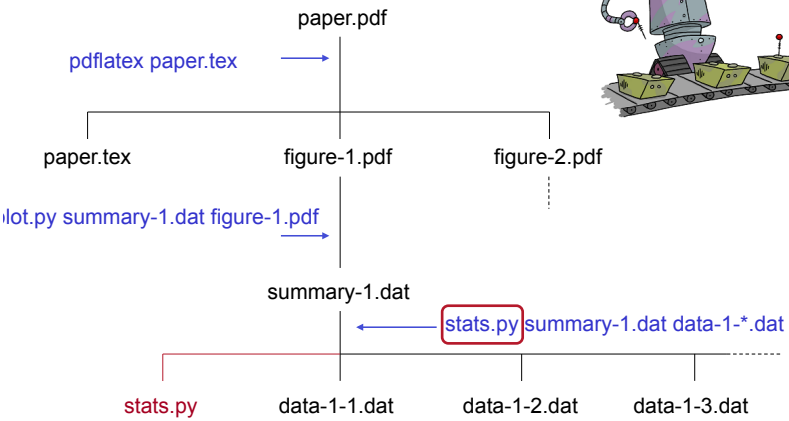
Manage tasks and dependencies





Manage tasks and dependencies





1. What is newer than what?

```
$ ls -t *.dat *.pdf
summary-1.dat    figure-1.pdf
$
```

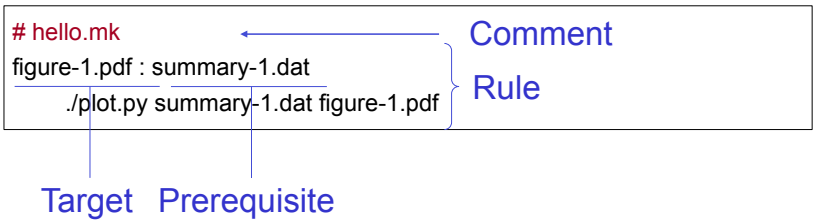
Data file is newer than SVG image

software carpentry

1. What is newer than what?

```
$ ls -t *.dat *.pdf
summary-1.dat    figure-1.pdf
$
```

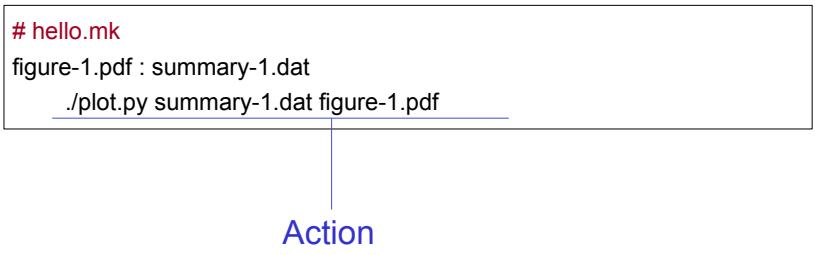
2. Put this in a *Makefile* called hello.mk



1. What is newer than what?

```
$ ls -t *.dat *.pdf
summary-1.dat    figure-1.pdf
$
```

2. Put this in a *Makefile* called hello.mk



1. What is newer than what?

```
$ ls -t *.dat *.pdf
summary-1.dat    figure-1.pdf
$
```

2. Put this in a *Makefile* called hello.mk

```
# hello.mk
figure-1.pdf : summary-1.dat
	./plot.py summary-1.dat figure-1.pdf
```

Must indent with a single tab character

3. Run Make from the shell

```
$ make -f hello.mk
```

-f filename

3. Run Make from the shell

```
$ make -f hello.mk
./plot.py summary-1.dat figure-1.pdf
$
```

Prerequisite is newer than target

So Make executes the action

3. Run Make from the shell

```
$ make -f hello.mk
./plot.py summary-1.dat figure-1.pdf
$
```

4. Run Make again

```
$ make -f hello.mk
$
```

Target is newer than prerequisite

So action not executed

Usually have multiple targets per file

```
# double.mk

figure-1.svg : summary-1.dat
	sgr -N -r summary-1.dat > figure-1.svg

figure-2.svg : summary-2.dat
	sgr -N -r summary-2.dat > figure-2.svg
```

Usually have multiple targets per file

```
# double.mk

figure-1.pdf : summary-1.dat
	./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : summary-2.dat
	./plot.py summary-2.dat figure-2.pdf
```

Force it to run

```
$ touch *.dat
$
```

← Update timestamps on files

Usually have multiple targets per file

```
# double.mk

figure-1.pdf : summary-1.dat
	./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : summary-2.dat
	./plot.py summary-2.dat figure-2.pdf
```

Force it to run

```
$ touch *.dat
$ make -f double.mk
./plot.py summary-1.dat figure-1.pdf
$
```

Why isn't figure-2.pdf rebuilt?

First rule in Makefile is *default rule*

Make only does this unless told otherwise

Force it to rebuild figure-2.pdf explicitly

```
$ make -f double.mk figure-2.pdf
./plot.py summary-2.dat figure-2.pdf
$
```

Better than typing commands

one by one, but only slightly

Introduce a *phony target*

Doesn't correspond to a file

So never up to date

But things can depend on it

```
# phony.mk

all: figure-1.pdf figure-2.pdf

figure-1.pdf : summary-1.dat
	./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : summary-2.dat
	./plot.py summary-2.dat figure-2.pdf
```

"make all" rebuilds everything

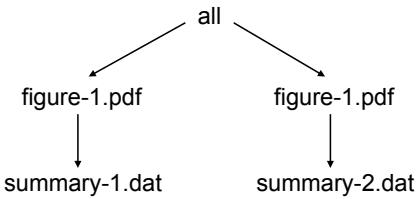
```
$ touch *.dat
$ make -f phony.mk
./plot.py summary-1.dat figure-1.pdf
./plot.py summary-2.dat figure-2.pdf
$
```

Order is not guaranteed

Could re-create figure-2.pdf first

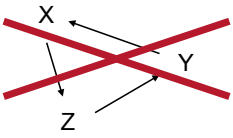
Or re-create both in parallel

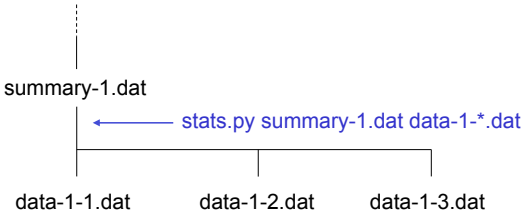
Things can be both targets and prerequisites



A *directed graph*

Must be *acyclic*



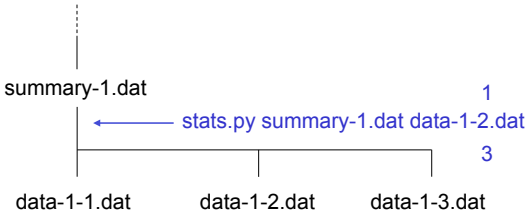



```
# multiple.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
	./stats.py summary-1.dat data-1-1.dat data-1-2.dat data-1-3.dat
```

How to generalize to any number of files?

And get rid of repeated filenames



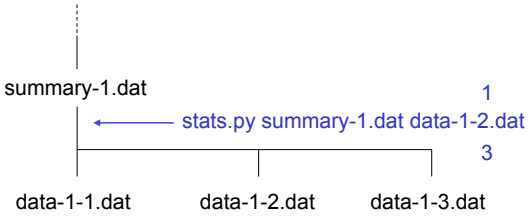

```
# target-variable.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
	./stats.py summary-1.dat data-1-1.dat data-1-2.dat data-1-3.dat
	./stats.py $@ data-1-1.dat data-1-2.dat data-1-3.dat
```

Automatic variable

"the target of this rule"

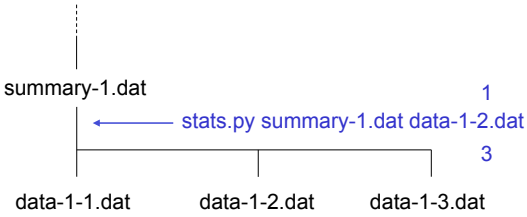
No, there isn't a more readable form




```
# target-variable.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
	./stats.py $@ data-1-1.dat data-1-2.dat data-1-3.dat
```

Still a lot of redundancy



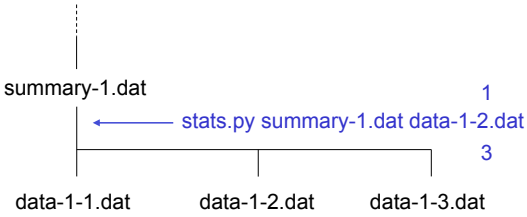

```
# variables.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
        ./stats.py $@ data-1-1.dat data-1-2.dat data-1-3.dat
        ./stats.py $@ $^
```

All prerequisites of this rule

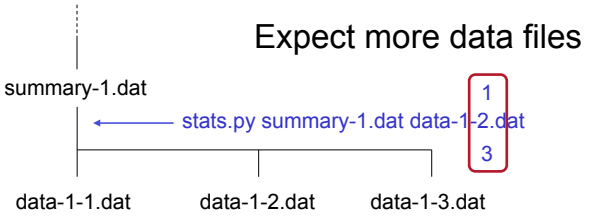
$< is "the first prerequisite"

$? is "all out-of-date prerequisites"



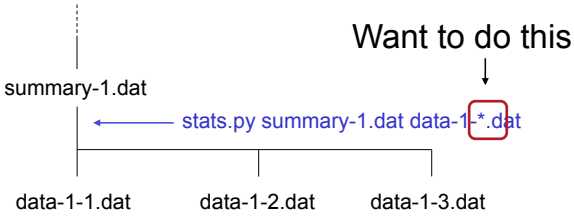

```
# variables.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
        ./stats.py $@ $^
```



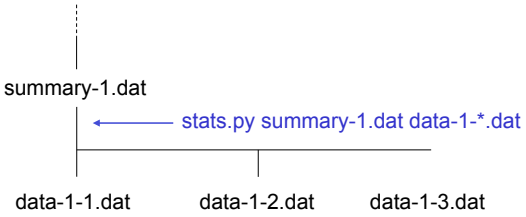

```
# variables.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
	./stats.py $@ $^
```




```
# variables.mk

summary-1.dat : data-1-1.dat data-1-2.dat data-1-3.dat
	./stats.py $@ $^
```




```
# wildcard.mk

summary-1.dat : data-1-*.dat
        ./stats.py $@ $^
```

Just like shell wildcard

Must use $^ in action, since filenames not fixed in advance

The makefile so far

```
paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        pdflatex $<

figure-1.pdf : summary-1.dat
        ./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : summary-2.dat
        ./plot.py summary-2.dat figure-2.pdf

summary-1.dat : data-1-*.dat
        ./stats.py $@ $^

summary-2.dat : data-2-*.dat
        ./stats.py $@ $^
```

Still some redundancy

Fix later

Doesn't handle dependency on stats.py and plot.py

Option 1: add to existing rules

```
paper.pdf : paper.tex figure-1.pdf figure-2.pdf
	pdflatex $<

figure-1.pdf : plot.py summary-1.dat
	./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : plot.py summary-2.dat
	./plot.py summary-2.dat figure-2.pdf

summary-1.dat : stats.py data-1-*.dat
	./ stats.py $@ $^

summary-2.dat : stats.py data-2-*.dat
	./stats.py $@ $^
```

Option 1: add to existing rules

```
⋮

summary-1.dat : stats.py data-1-*.dat
	stats.py $@ $^

summary-2.dat : stats.py data-2-*.dat
	stats.py $@ $^

⋮
```

$^ is now stats.py data-1-1.dat data-1-1.dat ...

So the invocation of stats.py is wrong

Having it ignore one argument is an ugly hack

Option 2: make data files depend on stats.py

```
figure-2.pdf : summary-2.dat
        ./plot.py summary-2.dat figure-2.pdf

summary-1.dat : data-1-*.dat
        stats.py $@ $^

summary-2.dat : data-2-*.dat
        stats.py $@ $^

data-1-1.dat : stats.py
        touch $@

data-1-2.dat : stats.py
        touch $@
```

Option 2: make data files depend on stats.py

```
⋮

data-1-1.dat : stats.py
        touch $@

data-1-2.dat : stats.py
        touch $@

⋮
```

A *false dependency*

Updating raw data files triggers update of summary

Back to listing all raw data files explicitly…

Option 3: add additional dependencies

```
paper.pdf : paper.tex figure-1.pdf figure-2.pdf
	pdflatex $<

figure-1.pdf : plot.py summary-1.dat
	./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : plot.py summary-2.dat
	./plot.py summary-2.dat figure-2.pdf

summary-1.dat : stats.py data-1-*.dat
	./ stats.py $@ $^

summary-2.dat : stats.py data-2-*.dat
	./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Option 3: add additional dependencies

```
⋮

summary-1.dat : data-1-*.dat
	stats.py $@ $^

summary-2.dat : data-2-*.dat
	stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py

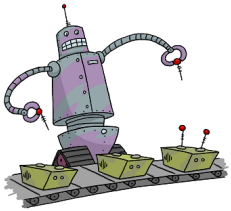
⋮
```

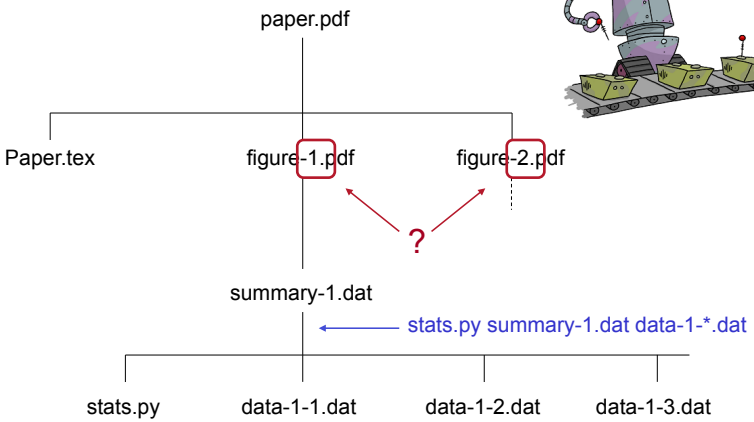
Full set of dependencies is union of lists

But $^ in the action is still just data-1-*.dat

software carpentry

Manage tasks and dependencies





Makefile so far

```
paper.pdf : paper.tex figure-1.pdf figure-2.pdf
      pdflatex $<

figure-1.pdf : summary-1.dat
      ./plot.py summary-1.dat figure-1.pdf

figure-2.pdf : summary-2.dat
      ./plot.py summary-2.dat figure-2.pdf

summary-1.dat : stats.py data-1-*.dat
     ./ stats.py $@ $^

summary-2.dat : stats.py data-2-*.dat
      ./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Eliminate this redundancy

Use a *pattern rule* to capture the common idea

```
# pattern-rule.mk

figure-%.pdf : summary-%.dat
	./plot.py $^ $@

summary-1.dat : data-1-*.dat
	./stats.py $@ $^

summary-2.dat : data-2-*.dat
	./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

- % is a wildcard
- Has the same value on both sides
- Undefined in the action
- Have to use $@, $^, etc.

Try running it

```
$ make -f pattern-rule.mk
./stats.py summary-1.dat data-1-1.dat data-1-2.dat data-1-3.dat
$
```

Why didn't other commands run?

software carpentry

Pattern rule doesn't create dependencies itself

```
# pattern-rule.mk

figure-%.pdf : summary-%.dat
	./plot.py $^ $@

summary-1.dat : data-1-*.dat
	./stats.py $@ $^

summary-2.dat : data-2-*.dat
	./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

If Make wants to create figure-1.svg, it can use this rule

Still have to tell Make what it wants to do

Put the rule for paper.pdf back in the file

```
# use-pattern.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
	pdflatex paper.tex

figure-%.pdf : summary-%.dat
	./plot.py $^ $@

summary-1.dat : data-1-*.dat
	./stats.py $@ $^

summary-2.dat : data-2-*.dat
	./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Make now knows that it needs to create the figures, so it finds and uses the rule

This doesn't work!

```
# doesnt-work.mk

paper.pdf : paper.tex figure-*.pdf
        pdflatex paper.tex

figure-%.pdf : summary-%.dat
        ./plot.py $^ $@

summary-1.dat : data-1-*.dat
        ./stats.py $@ $^

summary-2.dat : data-2-*.dat
        ./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Figures don't exist when Make starts to run, so this is empty

Get rid of more redundancy

```
# all-patterns.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        pdflatex paper.tex

figure-%.pdf : summary-%.dat
        ./plot.py $^ $@

summary-1.dat : data-1-*.dat
        ./stats.py $@ $^

summary-2.dat : data-2-*.dat
        ./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

software carpentry

Get rid of more redundancy

```
# all-patterns.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        pdflatex paper.tex

figure-%.pdf : summary-%.dat
        ./plot.py $^ $@

summary-%.dat : data-%-*.dat
        ./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Get rid of more redundancy

```
# all-patterns.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        pdflatex paper.tex

figure-%.pdf : summary-%.dat
        ./plot.py $^ $@

summary-%.dat : data-%-*.dat
        ./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Make wildcard

Get rid of more redundancy

```
# all-patterns.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        pdflatex paper.tex

figure-%.pdf : summary-%.dat
        ./plot.py $^ $@

summary-%.dat : data-%-*.dat
      ./stats.py $@ $^

summary-1.dat : stats.py
summary-2.dat : stats.py
```

Shell wildcard

But this doesn't work!

```
# all-patterns.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        pdflatex paper.tex

figure-%.pdf : summary-%.dat
        ./plot.py $^ $@

summary-%.dat : data-%-*.dat
      ./stats.py $@ $^

summary-%.dat : stats.py
```

summary-*.dat still only depends on data-*-*.dat

Make only uses the first pattern rule it finds

Back to using false dependencies...

```
# false-dependencies.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
	pdflatex paper.tex

figure-%.pdf : summary-%.dat
	./plot.py $^ $@

summary-%.dat : data-%-*.dat
	./stats.py $@ $^

data-*-*.dat : stats.py
	touch $@
```

It's a less-than-perfect tool...

Makefile so far

```
# false-dependencies.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
	pdflatex paper.tex

figure-%.pdf : summary-%.dat
	./plot.py $^ $@

summary-%.dat : data-%-*.dat
	./stats.py $@ $^

data-*-*.dat : stats.py
	touch $@
```

At home Python and LaTeX are somewhere else

```
# with-commands-at-home.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        /usr/local/bin/pdflatex paper.tex

figure-%.pdf : summary-%.dat
        /usr/local/bin/python plot.py $^ $@

summary-%.dat : data-%-*.dat
        /usr/local/bin/python stats.py $@ $^

data-*-*.dat : stats.py
        touch $@
```

Usually don't list "system" files explicitly

But what about the lab?

1. ~~Write~~ two Makefiles

   Write and maintain

2. Comment and uncomment lines

   Consistently every time

   Will create lots of noise in version control

3. Refactor

Use a *macro*

```
# with-macro.mk

LATEX = /opt/local/bin/pdflatex
PYTHON = /opt/local/bin/python

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
        ${LATEX} paper.tex

figure-%.pdf : summary-%.dat
        ${PYTHON} plot.py $^ $@

summary-%.dat : data-%-*.dat
        ${PYTHON} stats.py $@ $^

data-*-*.dat : stats.py
        touch $@
```

Only have one thing to change

✓ Consistency

✗ But still have noise

Must use ${MACRO} or $(MACRO), *not* $MACRO

Make reads $MACRO as($M)ACRO

Which is probably just "ACRO"

Which is probably not what you want

yet another legacy wart

Now put the first macros in a separate file

```
# config.mk

LATEX = /opt/local/bin/pdflatex
PYTHON = /opt/local/bin/python
```

And include it from the main file

```
# with-include.mk
include config.mk

paper.pdf : paper.tex figure-1.pdf figure-2.pdf
	${LATEX}  paper.tex

figure-%.pdf : summary-%.dat
	${PYTHON} plot.py $^ $@

summary-%.dat : data-%-*.dat
	${PYTHON}  stats.py $@ $^

data-*-*.dat : stats.py
	touch $@
```

software carpentr

Actually create *two* configuration files

```
# config-home.mk
LATEX = /opt/local/bin/pdflatex
PYTHON = /opt/local/bin/python
```

```
# config-lab.mk
LATEX = pdflatex
PYTHON = python
```

These two files stay in version control

Copy to create config.mk per machine

software carpentr

Actually create *two* configuration files

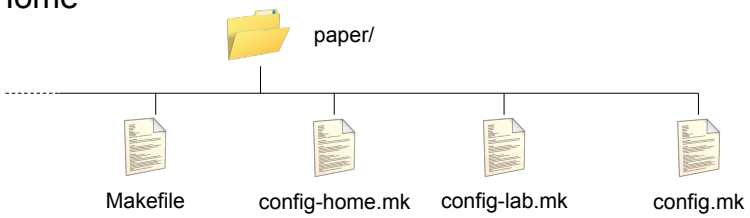
```
# config-home.mk
LATEX = /opt/local/bin/pdflatex
PYTHON = /opt/local/bin/python
```

```
# config-lab.mk
LATEX = pdflatex
PYTHON = python
```

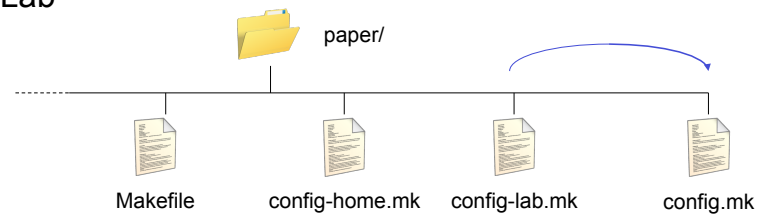
These two files stay in version control

Copy to create config.mk per machine

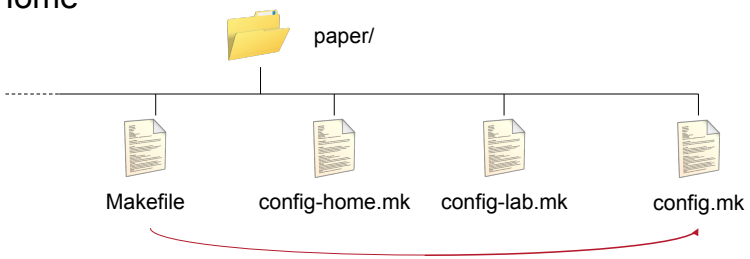
Home



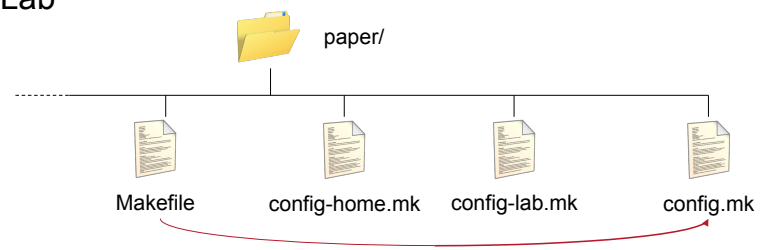
Lab



Home



Lab



Many other approaches

CMake and Autoconf/Automake: compile higher-level specification into a Makefile (or equivalent)

✓ Automatically discover/manage differences between machines

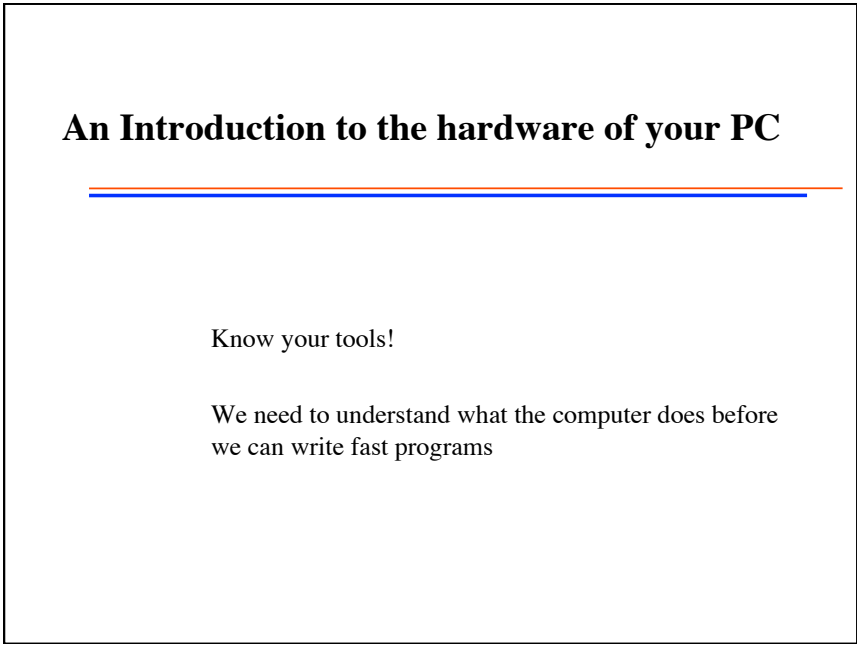
✗ But even harder to debug

A build file is a program

Requires the same degree of respect

Chapter 5

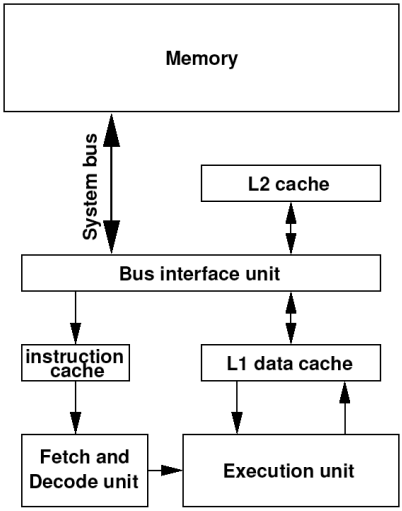
Introduction to hardware of the PC



Understanding hardware is important

- Steps in executing a program
  - We write our code in a high-level language
  - The compiler translates the program to machine language
  - The computer executes the machine language program
- We want to write a fast program
  - Need to understand hardware limitations
  - Need to understand what the compiler does
- This week
  - Introduction to main hardware components
  - Understanding the limitations

Schematic diagram of a computer



Components of the CPU

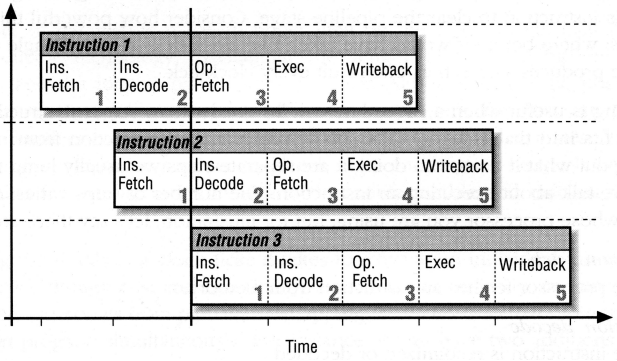
- The main components of the central processing unit (CPU) are:
  - Memory controller
    - Manages loading from and storing to memory
  - Registers
    - Can store integer or floating point numbers
    - Values can be set to specifed constants
    - Values can be loaded from or stored into memory
  - Arithmetic and logical units (ALU)
    - Performs arithmetic operations and comparisons
    - Operates on values in the registers (very fast)
    - On some CPUs they can operate on contents of memory (slow)
  - Fetch and decode unit
    - Fetches the next instruction from memory
    - Interprets the numerical value of the instruction and decides what to do
    - Dispatches operations to ALU and memory controller to perform the operation
- Be aware that modern CPUs are more complex (see later)

Machine code and assembly language

- The CPU performs instructions read from memory
  - Instructions are given in machine code
  - These are just numbers which are interpreted as instructions
  - Ugly and nearly impossible to interpret
- Assembly language
  - Is a one-to-one translation from machine code to a readable text form
  - Is non-portable: differs depending on CPU-type
- Typical instructions
  - Load values into registers
  - Load data from memory into register or store registers into memory
  - Perform arithmetic and logical instructions on registers
  - Jump (branch) to another instruction

Pipelining

- Is used to speed up execution
  - Second (independent) instruction can be started before first one finishes



Example of a pipeline

- Imagine a loop

```
for (int i=0; i <102400; ++i)
  a[i]=b[i]+c[i];
```

- Consecutive iterations are independent and can be executed in parallel after unrolling

```
for (int i=0; i <102400; i+=4){
  a[i]=b[i]+c[i];
  a[i+1]=b[i+1]+c[i+1];
  a[i+2]=b[i+2]+c[i+2];
  a[i+3]=b[i+3]+c[i+3];
}
```

Let us look at some examples

- Check out week 4 from the repository
- Example1: simpleadd.C
  - Add two floating point numbers
- Example 2: loopadd.C
  - Add two arrays of floating point numbers

Looking at the assembly code

- Run, one after the other
  - `g++ -save-temps -c -O3 simpleadd.C`
  - `g++ -save-temps -c -O3 loopadd.C`
  - `g++ -save-temps -c -O3 -funroll-loops loopadd.C`
- Let us take a look at the created intermediate *.s files
- simpleadd.s
  - Can you understand the addition?
- loopadd.s
  - Can you see the loop?
  - Can you see the unrolling and potential for pipelining?

Branch prediction

- At each branch (`if`-statement, …) the pipelines stall
  - Have to wait for end of execution before starting one of the branches
- Solution: branch prediction
  - Predict (clever compiler, clever hardware) which branch is more likely
    - E.g. in loop will usually repeat the loop
  - Start executing more likely branch
    - If correct prediction: pipeline runs on without any cost
    - If wrong prediction: abort pipeline and start right branch

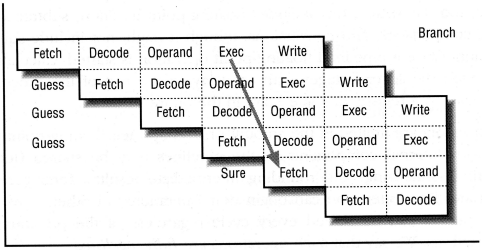
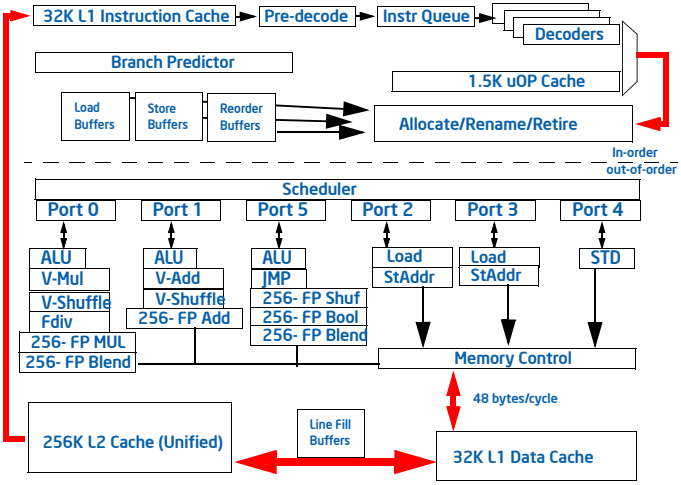


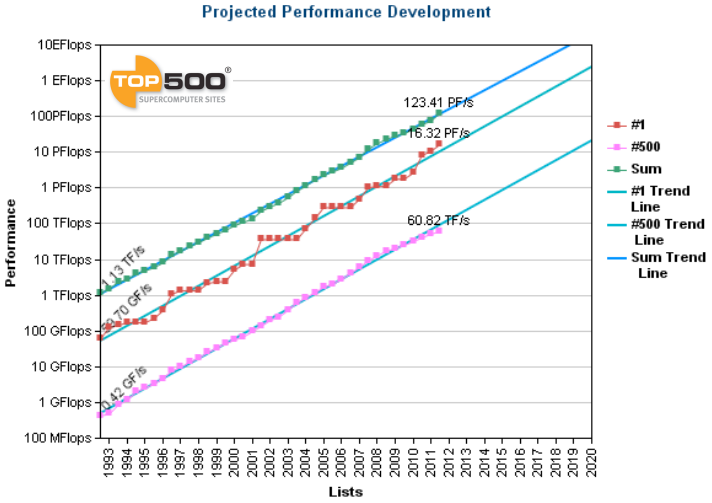
Diagram of an Intel Sandy Bridge CPU Core



Moore's law

- "The number of transistors on a chip doubles every 18 months"
  - More transistors means smaller transistors
  - Smaller transistors => shorter distances => faster signals
  - Smaller transitors => fewer charges => faster switching
  - Thus also the CPU speed increases exponentially
- Has worked for the past 30 years!
- How long will it continue?
  - Current prototype chips at 10 GHz
  - Insulating layers only 4 atoms thick!
  - Can we still reduce the size??
  - Moore's law will probably stop working in the next decade
  - Software optimization will become more important

Moore's law for supercomputers



Moore's gap

Moore's Law ran smoothly until 2002, when the gap between performance and gate count started to appear.

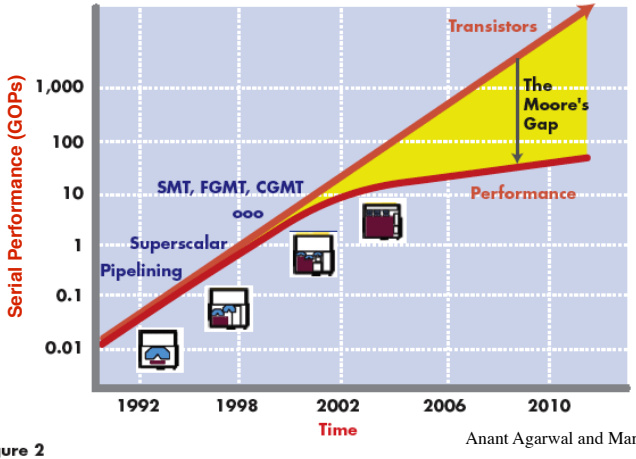


Figure 2

Anant Agarwal and Markus Levy

What has changed in last few years?

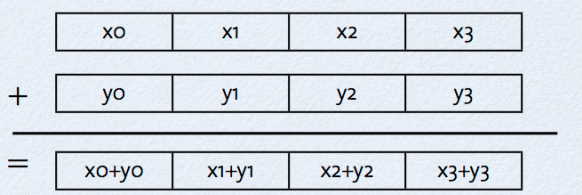
Model	Freq.	Release	Benchmark
Intel Pentium 4	1.4 GHz	November 2000	0.25x
Intel Pentium 4 HT 471	3.8 GHz	June 2005	1x
Intel Core i7 - 970	3.2 GHz	July 2010	15x

source: cpubenchmark.net

- The performance used to be increased only by increasing the clock frequency
  - very expensive in terms of power
    other components cannot reach the same operating frequency
- Nowadays performance is increased more efficiently:
  - Better pipelining and branch prediction
  - Better use of caches
  - Parallelism (multicore chipsets)

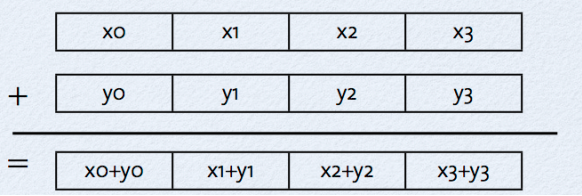
Parallelization 1: SIMD vector operations

- SIMD (Single Instruction Multiple Data) instructions perform the same operation on many values at once
- Since 1999 part of all Intel CPUs
  - SSE (Streaming SIMD Extensions)
  - AVX (Advanced Vector Extensions)
- Example: adding four floats with one “packed floating point” instruction
  - 4 additions can execute in one cycle (Sandy Bridge)



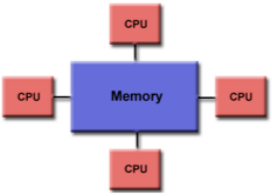
Parallelization 1: SIMD vector operations

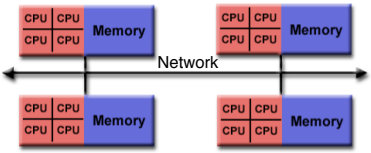
- The compiler can automatically generate SIMD code
  - `g++ -save-temps -c -O3 –ftree-vectorize loopadd.C`
- Optimal performance requires special care
  - memory layout and alignment
  - data dependencies
- This is a topic of the lecture “High Performance Computing for Science and Engineering (HPCSE)”



Parallelization 2: Shared & distributed memory

- Even laptop processors have several "cores" which can perform independent operations
  - all cores can access the same memory: **"shared memory architecture"**
- Today's supercomputers consist of many separate shared memory nodes
  - Data needs to be passed explicitly between nodes: **"distributed memory architecture"**

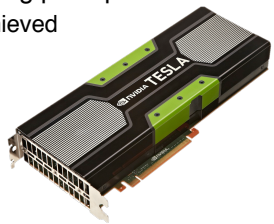




- This lecture concentrates on single-core applications
  - parallelization is taught in the HPCSE lecture

GPUs

- General-purpose GPUs offer immense floating point performance
  - About 3x performance gain can often be achieved
- Example: NVIDIA Tesla K20X
  - 2688 cores
  - Almost 4 TFlop/s



- SIMD (Single Instruction Multiple Data) programming style
- Harder to program
  - Many cores
  - Inhomogeneous and small memory
  - Transfer between CPU and GPU is expensive
- Programming environments: CUDA, OpenCL

Summary of CPUs

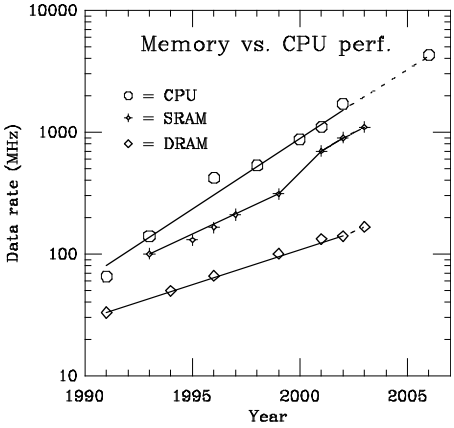
- CPU complexity is increasing
  - Branch prediction in hardware
  - Out-of-order execution
  - SIMD units
  - Multi-core architectures, non-uniform memory access
- High performance computing
  - Parallelization on several levels (SIMD units, multi-core chips, clusters of many multi-core nodes)
  - Hybrid systems (CPU+GPU) will very likely be the future
- We have very fast CPUs, but the rest of the system cannot keep up with the speed

How about the other components of a computer?

- Transistor density doubles every 18 months
- PC speed doubles every 2-2.5 years
  - Are now as fast as supercomputers were a decade ago
- Supercomputer speed doubles every year
  - PCs will not catch up with supercomputers
- But the rest of the system does not catch up
  - RAM speed increases slower
  - Disk speed increases even slower

Memory versus CPU speed

- DRAM has gotten cheap over the past decades but not much faster



Memory (RAM)

- SRAM (static random access memory)
  - Very fast access but very expensive
  - Data stored in state of transistors (flip-flop)
  - Data stays as long as there is power
- DRAM (dynamic random access memory)
  - Much cheaper than SRAM but slower
  - Data stored in tiny capacitor which discharge slowly
  - Capacitors need to be recharged regularly (hence dynamic)
- SDRAM (synchronous dynamic random access memory)
  - Variant of DRAM, with a clock synchronized with caches,
  - allows faster reading of successive data

Faster RAM technologies

- DDR RAM (double data rate)
  - Can send data twice per clock cycle
  - Send data on rising and falling edge of clock signal
- Interleaved memory systems
  - Use more than one bank of memory chips
  - Used in vector machines and most 64-bit systems
  - Can read simultaneously from each bank
    - increases bandwidth
    - Does not change latency (access time)

Improving memory speed by using caches

- Are added to speed up memory access (Opteron Barcelona)
  - Many GByte of slow DRAM
  - 2 MByte of fast and expensive L3-Cache
  - 512 kByte of even faster and more expensive L2-Cache per core
  - 2x64 kByte of the fastest and most expensive L1-Cache (instruction and data cache) per core
- Problems needing little memory will run faster!

Comparison of memory/cache speeds

	Cycles	Normalised
L1 cache	2	1
L2 cache	15	7.5
L3 cache	75	37.5
Other L1/L2	130	65
Memory	~300	~150
1-hop remote L3	190	95
2-hop remote L3	260	130

AMD "Barcelona" @ 2GHz

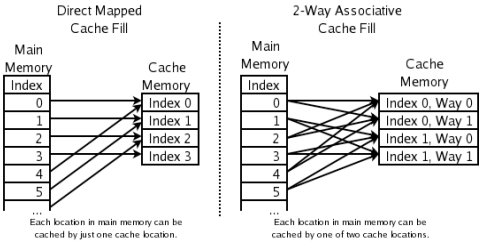
T. Roscoe, A. Baumann

How does a cache work?

- CPU requests a word (e.g. 4 bytes) from memory
  - A full "cache line" (Opteron: 64 bytes) is read from memory and stored in the cache
  - The first word is sent to the CPU
- CPU requests another word from memory
  - Cache checks whether it has already read that word as part of the previous cache line
  - If yes, it is sent quickly from cache to CPU
  - If not, a new cache line is read
- Once the cache is full, the oldest data is overwritten
- *Locality of memory references is important for speed*

Types of caches

- Direct mapped
  - Each memory location can be stored only in one cache location
  - “cache thrashing” occurs if we access in strides of the cache size, always replacing the previous date
- *n*-way associative
  - Each memory location can be stored in *n* cache locations
  - Better performance, more expensive
- Fully associative
  - Each memory location can be stored anywhere
  - Best but most expensive



Exercises about caches

- Exercise 1:
  - Write a program to measure the number and size of caches in your machine
- Exercise 2 (bonus):
  - Write a program to determine the type of associativity of your L1-cache. Is it
    - Direct mapped?
    - *n*-way associative?
    - Fully associative?

Virtual memory: memory is actually even slower

- What if more than one program runs on a machine?
- What if we need more memory than we have RAM?

- Solution 1: virtual memory
  - Programs run in a “logical” address space
  - Hardware maps “logical” to “physical” address

- Solution 2: swap space
  - Some physical memory may be on a hard disk
  - If accessed it is first read from disk into memory
  - This is even slower!

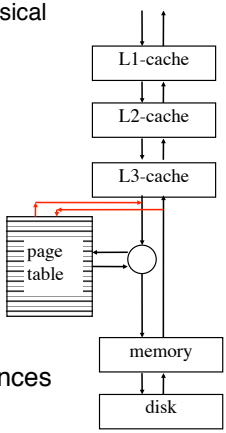
Virtual memory logic:

- Memory is organized in “pages” of e.g. 4 Kbyte
  - Addresses are translated from logical to physical address space
  - Lookup in page table
    - If in memory, access to memory
    - If on disk, read from disk first (slow!!!)

- Access to page table needs reading from memory

- Solution: translation lookaside buffer (TLB)
  - Is a cache for the page table

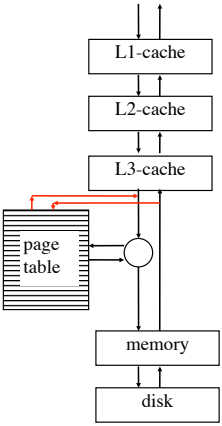
- It is again important to keep memory references local



Virtual memory: the worst case

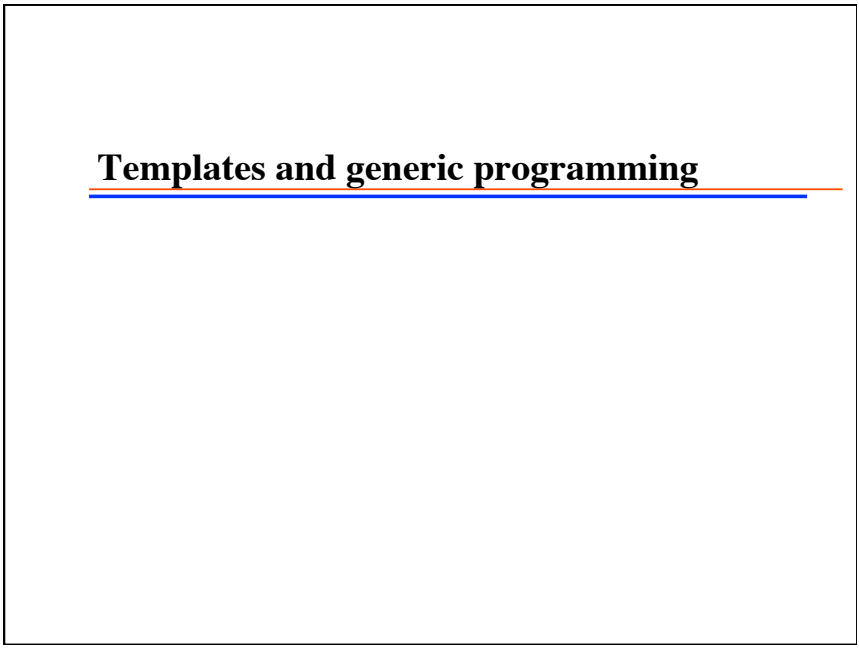
- Request an address
- Cache miss in L1
- Cache miss in L2
- Cache miss in L3
- Lookup physical address
  - Cache miss in TLB
  - Request page table entry
  - Load page table from memory (slow)
- Page fault in page table
  - Store a page to disk (extremely slow)
  - Create and initialize a new page (very slow)
  - Load page from disk (extremely slow)
- Load value from memory (slow)

- Try to reuse data as much as possible



Chapter 6

Templates and generic programming



Improving on last week's assignment

- Quiz: How did you calculate the machine precision?

1. Did you just have a main() function
2. Did you have three functions with different names?
```
1. epsilon_float()
2. epsilon_double()
3. epsilon_long_double()
```
3. Did you have three functions with the same name?
```
1. epsilon(float x)
2. epsilon(double x)
3. epsilon(long double x)
```
4. Or did you have just one function that could be used for any type?
```
1. epsilon()
```

Generic algorithms versus concrete implementations

- Algorithms are usually very generic:
  for min() all that is required is an order relation "<"

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

- Most programming languages require concrete types for the function definition
  - C:
```
int min_int(int a, int b) { return a<b ? a : b;}
float min_float (float a, float b) { return a<b ? a : b;}
double min_double (double a, double b) { return a<b ? a : b;}
…
```
  - Fortran:
```
MIN(), AMIN(), DMIN(), …
```

Function overloading in C++

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

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

- Compiler chooses which one to use

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

- However be careful:

```
min(1,3.1415927); // Problem! which one?
min(1.,3.1415927); // OK
min(1,int(3.1415927)); // OK but does not make sense
```

or define new function `double min(int,double);`

How can several functions have the same name?

1. Why should it be a problem?
2. I don't know
3. The compiler uses magic
4. It is a problem, but I know how it can be solved

C++ versus C linkage

- How can three different functions have the same name?
  - Look at what the compiler does
    ```
    c++ -c -save-temps -O3 min.cpp
    ```
  - Look at the assembly language file min.s and also at min.o
    ```
    nm min.o
    ```
- The functions actually have different names!
  - Types of arguments appended to function name
- C and Fortran functions just use the function name
  - Can declare a function to have C-style name by using `extern "C"`
    ```
    extern "C" { short min(short x, short y);}
    ```

Using macros (is dangerous)

- We still need many functions (albeit with the same name)
- In C we could use preprocessor macros:
  - `#define min(A,B) (A < B ? A : B)`
- However there are serious problems:
  - No type safety
  - Clumsy for longer functions
  - Unexpected side effects:

```
min(x++,y++);  // will increment the smaller number twice!!!
               // since this is: (x++ < y++ ? x++ : y++)
```

- Look at it:
  - `c++ -E minmacro.cpp`

Generic algorithms using templates in C++

- C++ templates allow a generic implementation:

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

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

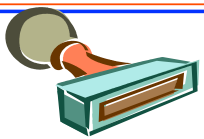
- Using templates we get functions that
  - work for many types `T`
  - are optimal and efficient since they can be inlined
  - are as generic and abstract as the formal definition
  - are one-to-one translations of the abstract algorithm

Usage Causes Instantiation

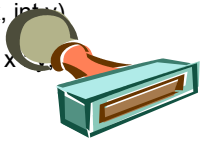
```
template <class T>
T min(T x, T y)
{
    return x < y ? x : y;
}

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

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



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



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

Polymorphism

- **Definition**: Using many different types through the same interface
- What are the advantages?

Generic programming process

- Identify useful and efficient algorithms
- Find their generic representation
  - Categorize functionality of some of these algorithms
  - What do they need to have in order to work **in principle**
- Derive a set of (minimal) requirements that allow these algorithms to run (efficiently)
  - Now categorize these algorithms and their requirements
  - Are there overlaps, similarities?
- Construct a framework based on classifications and requirements
- Now realize this as a software library

Generic Programming Process: Example

- (Simple) Family of Algorithms: min, max
- Generic Representation

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

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

- Minimal Requirements?
- Find Framework: Overlaps, Similarities?

Generic Programming Process: Example

- (Simple) Family of Algorithms: min, max
- Generic Representation

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

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

- Minimal Requirements yet?
- Find Framework: Overlaps, Similarities?

Generic Programming Process: Example

- Possible Implementation

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

- What are the Requirements on **T**?
  - operator < , result convertible to bool

Generic Programming Process: Example

- Possible Implementation

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

- What are the Requirements on **T**?
  - operator < , result convertible to bool
  - Copy construction: need to copy the result!

Generic Programming Process: Example

- Possible Implementation

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

- What are the Requirements on **T**?
  - operator < , result convertible to bool
  - that's all!

The problem of different types: manual solution

- What if we want to call min(1,3.141)?

```
template <class R,U,T>
R const& min(U const& x, T const& y)
{
  return (x < y ? static_cast<R>(x) : static_cast<R>(y));
}
```

- Now we need to specify the first argument since it cannot be deduced.

```
min<double>(1,3.141);
min<int>(3,4);
```

Concepts

- A concept is a set of requirements, consisting of valid expressions, associated types, invariants, and complexity guarantees.
- A type that satisfies the requirements is said to model the concept.
- A concept can extend the requirements of another concept, which is called refinement.
- A concept is completely specified by the following:
  - Associated Types: The names of auxiliary types associated with the concept.
  - Valid Expressions: C++ expressions that must compile successfully.
  - Expression Semantics: Semantics of the valid expressions.
  - Complexity Guarantees: Specifies resource consumption (e.g., execution time, memory).
  - Invariants: Pre and post-conditions that must always be true.

Assignable concept

- Notation
  - X — A type that is a model of Assignable
  - x, y — Object of type X

Expression	Return type	Semantics	Postcondition
x=y;	X&	Assignment	x is equivalent to y

swap(x,y)	void	Equivalent to
		{
		X tmp = x;
		x = y;
		y = tmp;
		}	

CopyConstructible concept

- Notation
  - X — A type that is a model of CopyConstructible
  - x, y — Object of type X

Expression	Return type	Semantics	Postcondition
X(y)	X&		Return value is equivalent to y
X x(y);		Same as X x; x=y;	x is equivalent to y
X x=y;		Same as X x; x=y;	

Documenting a template function

- In addition to
  - Preconditions
  - Postconditions
  - Semantics
  - Exception guarantees
- The documentation of a template function must include
  - Concept requirements on the types
- Note that the complete source code of the template function must be in a header file

Argument Dependent Lookup (Koenig Lookup)

- Applies only to *un*qualified calls

  abs(x) ~~std::abs(x)~~

- Looks in "associated classes and namespaces": namespaces of the arguments

- Originally for operators, e.g. operator<<(std::ostream&, T);

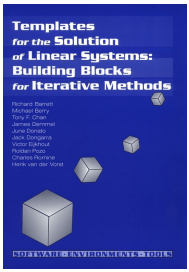
```
namespace lib  {
    template <class T> T abs(T x)
        { return x > 0 ? x : -x; }

    template <class T>
    T compute(T x) {
        …
        return abs(x);
    }
}
namespace user {
    class Num {};
    Num abs(Num);
    Num x = lib::compute(Num());
}
```

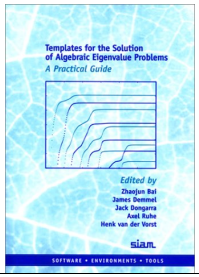
?

Examples: iterative algorithms for linear systems

- Iterative template library (ITL)
  - Rick Lee *et al*, Indiana
- generic implementation of iterative solvers for linear systems from the "Templates" book



- Iterative Eigenvalue Template Library (IETL)
  - Prakash Dayal *et al*, ETH
- generic implementation of iterative eigensolvers. partially implements the eigenvalue templates book



The power method

- Is the simplest eigenvalue solver
  - returns the largest eigenvalue and corresponding eigenvector

ALGORITHM 4.1: **Power Method for HEP**

(1) *start with vector* $y = z$, *the initial guess*
(2) **for** $k = 1, 2, \ldots$
(3) $v = y/\|y\|_2$
(4) $y = Av$
(5) $\theta = v^* y$
(6) **if** $\|y - \theta v\|_2 \leq \epsilon_M |\theta|$, **stop**
(7) **end for**
(8) *accept* $\lambda = \theta$ *and* $x = v$

- Only requirements:
  - A is linear operator on a Hilbert space
  - Initial vector y is vector in the same Hilbert space
- Can we write the code with as few requirements as possible?

Generic implementation of the power method

- A generic implementation is possible

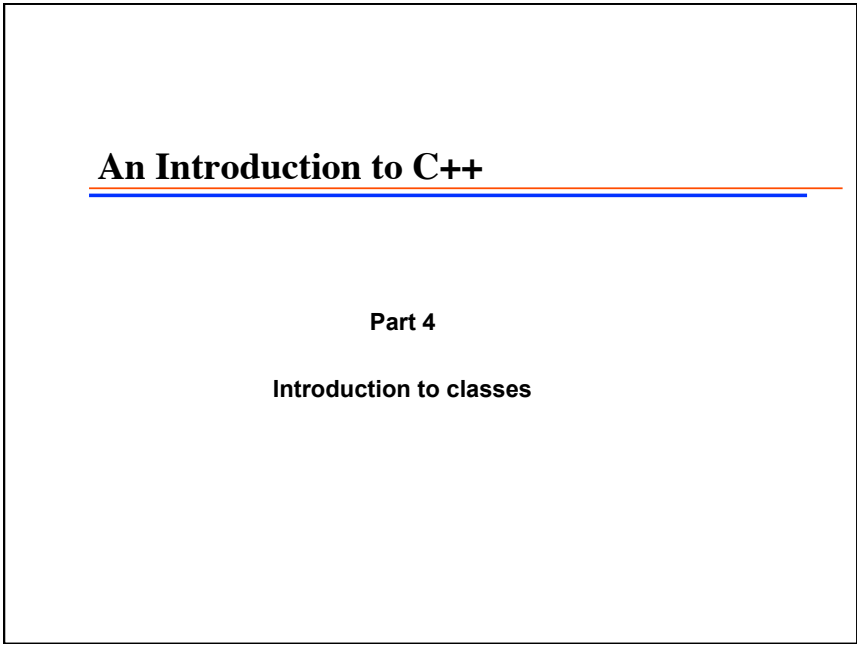
```
OP A;
V v,y;
T theta, tolerance, residual;
…
do {
  v = y / two_norm(y);          // line (3)
  y = A * v;                    // line (4)
  theta = dot(v,y);             // line (5)
  v *= theta;                   // line (6)
  v -= y;
  residual = two_norm(v); // ||q v - Av||
} while(residual>tolerance*abs(theta));
```

Concepts for the power method

- The triple of types (T,V,OP) models the Hilbert space concept if
  - `T` must be the type of an element of a field
  - `V` must be the type of a vector in a Hilbert space over that field
  - `OP` must be the type of a linear operator in that Hilbert space
- All the allowed mathematical operations in a Hilbert space have to exist:
  - Let `v`, `w` be of type `V`
  - Let `r`, `s` of type `T`
  - Let `a` be of type `OP`.
  - The following must compile and have the same semantics as in the mathematical concept of a Hilbert space:
    `r+s`, `r-s`, `r/s`, `r*s`, `-r` have return type `T`
    `v+w`, `v-w`, `v*r`, `r*v`, `v/r` have return type `V`
    `a*v` has return type `V`
    `two_norm(v)` and `dot(v,w)` have return type `T`
    …
  - Exercise: complete these requirement

Chapter 7

Introduction to classes



Classes

- Are a method to create new data types
  - E.g. a vector or matrix type
- Object oriented programming:
  - Instead of asking: "What are the subroutines?"
  - We ask:
    - What are the abstract entities?
    - What are the properties of these entities?
    - How can they be manipulated?
  - Then we implement these entities as classes
- Advantages:
  - High level of abstraction possible
  - Hiding of representation dependent details
  - Presentation of an abstract and simple interface
  - Encapsulation of all operations on a type within a class
    - allows easier debugging

A first simulation: biological aging

- a simple model for death: $dN = -\lambda N dt$
- what about aging?
  - the remaining lifetime does not depend on current age.
  - true for radioactive decay
  - not true for biology
- what about age distribution?
  - exponential distribution!
  - also not what is seen in nature!
- what is missing?
  - some kind of aging
  - we need to develop a model containing aging

The Penna model

- A very simple model of biological aging
  - T.J.P. Penna, J. Stat. Phys **78**, 1629 (1995)
- Three important assumptions
  - finite age of adulthood
  - mutations of genetic material
  - limited resources
- This allows to model many features of biological population dynamics:
  - pacific salmon dies after giving birth
  - redwood trees have offsprings for hundreds of generations
  - catastrophic decline of cod in Atlantic due to small increase in fishing
- All these issues cannot be modeled without aging effects!

Details of the Penna model

- Each animal contains genes determining the survival rate
  - Each gene relevant for one year of its life
  - animal dies when it has collected T bad genes
- Limitation of resources
  - An animal that would survive because of its genes dies with a probability of N/N_0
    - N…current population
    - N_0… maximum sustainable population
- Children
  - from an age of R years an animal gets a child asexually with a probability b (birthrate)
- Mutations
  - The children have the genes of the parents but with M random mutations

Program for the Penna model

- First step: find the entities
- What are the abstract ideas?
  - Genes
  - Animal
  - Population
- Exercise: write a list of the properties of each of these entities
  - Representation (internal state)
  - Properties
  - Operations
  - Construction/destruction

What are classes?

- Classes are collections of "members" representing one entity
- Members can be
  - functions
  - data
  - types
- These members can be split into
  - `public`, accessible interface to the outside.
    Should not be modified later!
  - `private`, hidden representation of the concept.
    Can be changed without breaking any program using the class
- Objects of this type can be modified only through these member functions -> localization of access, easier debugging

How to design classes

- ask yourself some questions
- what are the logical entities (nouns)?
  -> classes
- what are the internal state variables ?
  -> private data members
- how will it be created/initialized and destroyed?
  -> constructor and destructor
- what are its properties (adjectives)?
  -> public constant member functions
- how can it be manipulated (verbs)?
  -> public operators and member functions

A first class example: a traffic light

- Property
  - The state of the traffic light (green, orange or red)
- Operation
  - Set the state
- Construction
  - Create a light in a default state (e.g. red)
  - Create a light in a given state
- Destruction
  - Nothing special needs to be done
- Internal representation
  - Store the state in a variable
  - Alternative: connect via a network to a real traffic light

A first class example: a traffic light

- Converting the design into a class

```
class Trafficlight  {
};
```

A first class example: a traffic light

- Add a public type member

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member
};
```

A first class example: a traffic light

- Add a private data member (variable) of that type:

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member




  private: // this is hidden
    light state_; // data member
};
```

A first class example: a traffic light

- Add a const member function to access the state

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member



    light state() const; //function member

  private: // this is hidden
    light state_; // data member
};
```

A first class example: a traffic light

- Add a non-const member function to change the state

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member


    light state() const; //function member
    void set_state(light);
  private: // this is hidden
    light state_; // data member
};
```

A first class example: a traffic light

- Add a default constructor to initialize it in the default way
  a constructor has the same name as the class

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member
    Trafficlight(); // default constructor


    light state() const; //function member
    void set_state(light);
  private: // this is hidden
    light state_; // data member
};
```

A first class example: a traffic light

- Add a second constructor to construct it from a light

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member
    Trafficlight(); // default constructor
    Trafficlight(light=red); // constructor


    light state() const; //function member
    void set_state(light);
  private: // this is hidden
    light state_; // data member
};
```

A first class example: a traffic light

- And finish by adding a destructor (called to cleanup at destruction)

a destructor has the same name as the class, prefixed by ~

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member

    Trafficlight(light=red); // constructor
    ~Trafficlight(); // destructor

    light state() const; //function member
    void set_state(light);
  private: // this is hidden
    light state_; // data member
};
```

Data hiding and access

- The concept expressed through the class is representation - independent
- Programs using a class should thus also not depend on representation
- Access declarators
  - public: only representation-independent interface, accessible to all
  - private: representation-dependent functions and data members
  - friend declarators allow related classes access to representation
- Note: Since all data members are representations of concepts (numbers, etc.) they should be hidden (private)!
- By default all members are private
  In a `struct` by default all are public

Member access

```
class Trafficlight  {
public:
  enum light
  { green, orange, red};

  Trafficlight();
  Trafficlight(light);
  ~Trafficlight();

  light state() const;
  void set_state(light);
private:
  light _state;
};
```

- Usage:
```
Trafficlight
x(Trafficlight::green);
Trafficlight::light l;

l = x.state();
l = Trafficlight::green;
```
- Members accessed with

  variable_name.member_name
- Type members accessed with

  class_name::*member_name*

  as they are not bound to specific object but common to all.

Special members

- Constructors
  - initialize an object
  - same name as class
- Destructors
  - do any necessary cleanup work when object is destroyed
  - have the class name prefixed by ~
- Conversions
- Operators

Illustration: a point in two dimensions

- Internal state:
  - x- and y- coordinates
  - is one possible representation
- Construction
  - default: (0,0)
  - from x- and y- values
  - same as another point
- Properties:
  - distance to another point
  - x- and y- coordinates
  - polar coordinates
- Operations
  - Inverting a point
  - assignment

```
class Point {
private:
  double x_,y_;
public:
  Point(); // (0,0)
  Point(double, double);
  Point(const Point&);
  double dist(const Point& )
   const;
  double x() const;
  double y() const;
  double abs() const;
  double angle() const;
  void invert() ;
  Point& operator=(const Point&);
};
```

Constructors and Destructors

- Let us look at the point example:
  - `public:`
    ```
    Point(); // default constructor
    Point(double, double); // constructor from two numbers
    Point(const Point&); // copy constructor
    ```
- Most classes should provide a default constructor
- Copy constructor automatically generated as memberwise copy, unless otherwise specified
- Destructor normally empty and automatically generated
- Nontrivial destructor only if resources (memory) allocated by the object. This usually also requires nontrivial copy constructor and assignment operator. (example: array class)

Default members

- Some member functions are implicitly created by the compiler
  - Copy constructor
    `A::A(A const&);`
    defaults to member-wise copy if not specified
  - Assignment operator
    `A::operator=(A const&);`
    also defaults to member-wise copy
  - Destructor
    `A::~A()`
    defaults to empty function

Declaration, Definition and Implementation

- Declaration

```
class Point;
```

- Definition

```
class Point {
private:
  double x_,y_;
public:
  Point(); // (0,0)
  Point(double, double);
  ...
};
```

- Implementation

```
double Point::abs() const {
  return std::sqrt(x_*x_+y_*y_)
}
```

- Constructors

```
Point::Point(double x, double y)
 : x_(x), y_(y)
{} // preferred method
```

- or

```
Point::Point(double x, double y)
{ x_ = x; y_ = y;}
```

Initializing a reference or a const member

- The simple-minded way fails

```
class A {
  private:
    int& x;
    const int y;
  public:
    A(int& r, int s) {
      x=r; // does not work
        // what does x refer to ?
      y=s; // does not compile
        // y is const!
    }
};
```

- We need the initialization syntax

```
class A {
  private:
    int& x;
    const int y;
  public:
    A(int& r, int s)
      : x(r),
        y(s)
    {
    }
};
```

- Stylistic advice: initialize all members in this way

const and volatile

- const
  - Variables or data members declared as `const` cannot be modified
  - Member functions declared as `const` do not modify the object
  - Only `const` member functions can be called for `const` objects
- volatile:
  - Volatile variables
    `volatile int x;`
    can be modified from outside the program!
  - Examples: I/O ports
  - No optimization or caching allowed!
  - Only member functions declared `volatile` can be called for `volatile` objects
  - With 99.9% probability you will never need to use it

mutable

- Problem:
  - want to count number of calls to `age()` function of animal
- Original source:

```
class Animal() {
public:
  age_t age() const;
private:
  long     cnt_;
  age_t age_;
};

age_t Animal::age() const {
  cnt_++; // error: const!
  return age_;
}
```

- Solution:
  - mutable qualifier allows modification of member even in const object!
- Modified source:

```
class Animal() {
  …
private:
  mutable long cnt_;
  …
};

age_t Animal::age() const {
  cnt_++;  // now OK!
  return age_;
}
```

friends

- Consider geometrical objects: points, vectors, lines,...
  - ```
    class Point {
    …
    private:
      double x,y,z;
    };

    class Vector {
    …
    private:
      double x,y,z;
    };
    ```
- For an efficient implementation these classes should have access to each others internal representation
- Using friend declaration this is possible:
  - ```
    class Vector;
    class Point {
    …
    private:
      double x,y,z;
      friend class Vector;
    };
    class Vector {
    …
    private:
      double x,y,z;
      friend class Point;
    };
    ```
- also functions possible:
  - `friend Point::invert(…);`

this

- Sometimes the object needs to be accessed from a member function
- `this` is a pointer to the object itself:
- ```
  const Array& Array::operator=(const Array& o) {

    … copy the array …

   return *this;
  }
  ```
  ```

Inlining of member functions

- For speed issues member functions can be inlined
- Avoid excessive inlining as it leads to code-bloat
- Either in-class definition:

```
class complex {
    double re_, im_;

  public:
     double real() const
       {return re_;}

     double imag() const
       {return im_;}
     …
};
```

- or out-of-class:

```
class complex {
   double re_, im_;

  public:
       inline double real() const;
       inline double imag() const;
      …
};

double complex::real() const
{
  return re_;
}

double complex::imag() const
{
  return im_;
}
```

Static members

- are shared by all objects of a type
- Act like global variables in a name space
- exist even without an object, thus :: notation used:

```
Genome::gene_number
Genome::set_mutation_rate(2);
```

- Static member functions can only access static member variables! Reason: which object's members to use???
- must be declared and defined!
  - will not link otherwise

```
class Genome  {
public:
 Genome(); // constructor

  static const unsigned short
                    gene_number=64;
              // static data member
 Genome clone() const;

  static void set_mutation_rate
              (unsigned short);
private:
   unsigned long gene_;
   static unsigned short
                 mutation_rate_;
   };

// in source file:
unsigned short
   Genome::mutation_rate_=2; //
   definition
```

Class templates

- same idea as function templates, classes for any given type T
- Learn it by studying examples:
  - Array of objects of type T
  - Complex numbers based on real type T
  - Statistics class for observables of type T

- Take care with syntax, where `<T>` must be used!
  - ```
    template <class T> class Array {
      …
      Array(const Array<T>&); // constructor!
      …
    };
    ```
  - ```
    template <class T>
    Array<T>::Array(const Array<T>& cp)
    { … }
    ```

The complex template

- The standard complex class is defined as a template

- `template <class T> class complex;`

- It is specialized and optimized for
  - `complex<float>`
  - `complex<double>`
  - `complex<long double>`
- but in principle also works for `complex<int>`, …

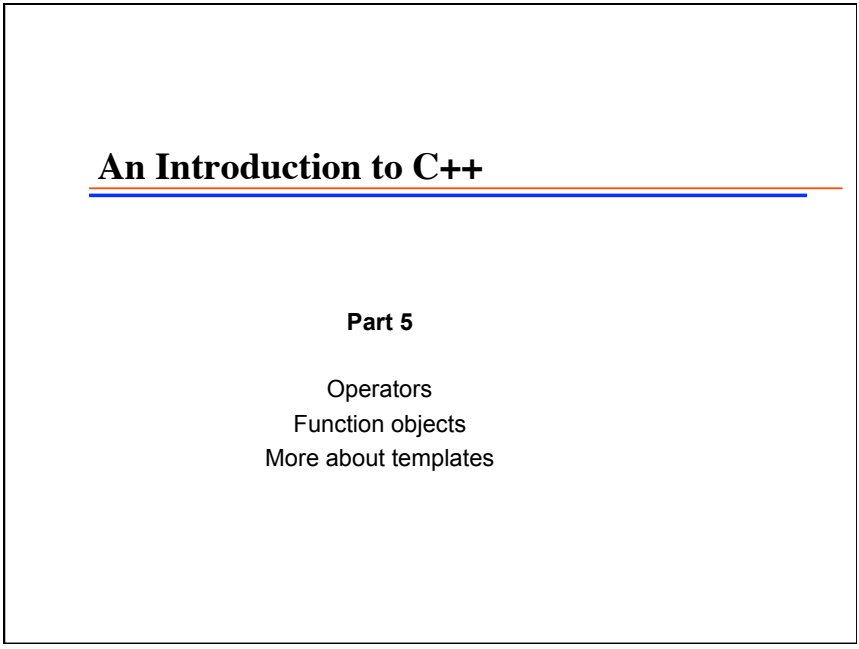
- it is a good exercise in template class design to look at the `<complex>` header

Do not avoid `typedef`!

- Do not store the age of an animal in an int
- Instead define a new type age_type
  - ```
    class Animal {
      public:
        typedef unsigned short age_t;
        age_t age() const;
      private:
        age_t age_;
    };
    ```
- Allows easy modifications. If we want to allow older ages, just change the typedef to:
  - `typedef unsigned long age_t;`
- The rest of the code can remain unchanged!
    ```

Chapter 8

Operators, Function objects, more about templates



Special members

- Constructors
  - initialize an object
  - same name as class
- Destructors
  - do any necessary cleanup work when object is destroyed
  - have the class name prefixed by ~
- **Conversion of object A to B**
  - two options:
  - constructor of B taking A as argument
  - conversion operator to type B in A:
    - operator B();
- **Operators**
- Default versions exist for some of these

Operators as functions

- Most operators can be redefined for new classes
- Same as functions, with function name:

```
operator symbol(...)
```

- Example:

```
Matrix A,B,C;
C=A+B;
```

- is converted to
  - either `C.operator=( A.operator+(B));`
  - or `C.operator= (operator+(A,B));`

Assignment operators

- The assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `^=`, `&=`, `|=`, `%=`
  - can only be implemented as member functions
  - should always return a const reference to allow expressions like

```
a=b=c;
a=b+=c;
```

- Example:
  -
```
class Point {
    double x_,y_;
  public:
    const Point& operator+=(const Point& rhs) {
      x_ += rhs.x_;
      y_ += rhs.y_;
      return *this;
    }
};
```

Symmetric operators

- Symmetric operators, e.g. `+`, `-`, … are best implemented as free functions
- Either the simple-minded way

  -
```
Point operator+(const Point& x, const Point& y) {
  Point result(x.x() + y.x(), x.y() + y.y());
  return result;
}
```

- Or the elegant way

  -
```
Point operator+(Point x, const Point& y) {
  return x+= y;
}
```

Extending classes with operators

- Extensions to existing classes can only be implemented as free functions
- Example: extending the iostream library

```
std::ostream& operator <<(std::ostream& out,
                          const Point& p) {
  out << "( " << p.x() << ", " << p.y() << " )";
  return out;
}
```

- We can now print a `Point`:

```
Point p;
std::cout << "The point is " << p << std::endl;
```

More comments about operators

- `A a; ++a;` uses
  `const A& A::operator++();`
  or `const A& operator++(A&);`
- `A a; a++;` uses
  `A A::operator++(int);`
  or `A operator++(A&,int);`
  The additional int argument is just to distinguish the two
- `A b; b=a;` uses the assignment
  `const A& A::operator=(const A&);`
- `A b=a;` and `A b(a);` both use the copy constructor
  `A::A(const A&);`

Conversion operators

- conversion of `A -> B` as in:
  - `A a; B b=B(a);`
- can be implemented in two ways
  - constructor `B::B(const A&);`
  - conversion operator `A::operator B();`
- Automatic conversions:
  - `char -> int`
  - `unsigned -> signed`
  - `short -> int -> long`
  - `float -> double -> long double`
  - Integer -> floating point
  - as in: `double x=4;`

Array subscript operator: `operator[]`

- In an array or vector class we want to be use the array subscript syntax:

```
Array a;
for (int i=0;i<a.size();++i)
  std::cout << a[i] << std::endl;
```

- We need to implement both const and non-const `operator[]`:

```
class Array {
  public:
    …
    double operator[](unsigned int i) const
    { return p[i];}
    double& operator[](unsigned int i)
    { return p[i];}
  private:
    double* p;
};
```

Pointer operators: `operator*` and `operator->`

- We will get to know classes acting like pointers
  - Iterators
  - Smart pointers (e.g. reference counted or checked pointers)
- In such classes we want to use the pointer syntax
  - `*p`
  - `p->f()`
- We need to implement const and non-const versions of these operators:

```
class P {
  …
  double* operator->() { return p_;}
  const double* operator->() const { return p_;}
  double& operator*() { return *p_;}
  const double& operator*() const { return *p_;}
private:
  double* p_;
};
```

The function call operator: `operator()`

- We sometimes want to use an object like a function, e..g

```
Potential V;
double distance;
std::cout << V(distance);
```

- This works only if Potential is a function pointer, or if we define the function call operator:

```
class Potential {
  double operator()(double d) { return 1./d;}
  …
};
```

- Don't get confused by the two pairs of ()()
  - The first is the name of the operator
  - The second is the argument list

Template specialization

- Consider our `Array<T>`
  - An array of size n takes `n*sizeof(T)` bytes
- Consider `Array<bool>`
  - An array of size n takes `n*sizeof(T) = n` bytes
  - An optimized implementation just needs one bit per entry and thus only n/8 bytes
- How can we define an optimized version `for bool`?
- Solution: template specialization

```
template <class T>
class Array {
  // generic implementation
  …
  };
```

```
template <>
class Array<bool> {
  //optimized version for bool
  …
  };
```

Traits types

- We want to allow the addition of two arrays:
  - `template <class T>`
    `Array<T> operator+ (const Array<T>&, const Array<T>&)`
- How do we add two different arrays?
  Array<int> + Array<double> makes sense!
  - `template <class T, class U>`
    `Array<?> operator +(const Array<T>&, const Array<U>&)`
- What is the result type?
  - We want to calculate with types!
- The solution is a technique called traits. Used quite often
  - numeric_limits traits class for numeric data types
  - can also be used here:
  - `template <class T, class U>`
    `Array< typename sum_type<T,U>::type >`
    `operator +(const Array<T>&, const Array<U>&)`

typename

- The keyword typename is needed here so that C++ knows the member is a type and not a variable or function.

```
template <class T, class U>
Array< typename sum_type<T,U>::type >
operator +(const Array<T>&, const Array<U>&)
```

- This is required to parse the program code correctly – it would not be able to check the syntax otherwise

Traits types (continued)

- We want to use traits like
  - `template <class T, class U>`
    `Array< typename sum_type<T,U>::type >`
    `operator +(const Array<T>&, const Array<U>&)`
- Definition of `sum_type`:
  - empty template type to trigger error messages if used
    - `template< class T, class U > class sum_type {};`
  - Partially specialized valid templates:
    - `template <class T> struct sum_type<T,T> {typedef T type;};`
  - Fully specialized valid templates:
    - `template <> struct sum_type<double,float> {typedef double type;};`
    - `template <> struct sum_type<float,double> {typedef double type;};`
    - `template <> struct sum_type<float,int> {typedef float type;};`
    - `template <> struct sum_type<int,float> {typedef float type;};`

Old style traits

- In C++98 traits were big "blobs":

```
template<>
struct numeric_limits<int> {
  static const bool is_specialized = true;
  static const bool is_integer = true;
  static const bool is_signed = true;
 ……

};
```

- Later it was realized that this was ugly:
  - A traits class is a "meta function", a function operating on types
  - A blob like numeric limits takes one argument, and returns many different values
  - This is not the usual design for functions!

New style traits

- Since C++03 all new traits are single-valued functions
  - Types are returned as the `type` member:

```
template<class T>
struct sum_type { typedef T type; };

template<>
struct sum_type<int> { typedef double type; };
```

  - Constant values are returned as the `value` member:

```
template<class T>
struct is_integral { static const bool value=false; };

template<>
struct is_integral<int> { static const bool value=true; };
```

Another application of traits

- Imagine an average() function:

```
template <class T>
T average(const Array<T>& v) {
  T sum;
  for (int n=0;n<v.size();++n)
    sum += v[n];
  return sum/v.size();
}
```

- Has problems with Array<int>, as the average is in general a floating point number:
  - v = (1,4,3)
  - Average would be int(8/3)=2
- Solution: traits
- The better version:

```
template <class T>
typename average_type<T>::type
average(const Array<T>& v) {
 typename average_type<T>::type sum;
  for (int n=0;n<v.size();++n)
    sum += v[n];
  return sum/v.size();
}
// the general traits type:
template <class T>
struct average_type {
  typedef T type;
};
// the special cases:
template<>
struct average_type <int> {
  typedef double type;};
…
// repeat for all integer types
```

An automatic solution for all integral types

```
template <class T> struct average_type {
  typedef typename
           helper1<T, std::numeric_limits<T>::is_specialized>::type type;
};
// the first helper:
template<class T, bool F>
struct helper1 { typedef T type };

//  the first helper if numeric_limits is defined: a partial specialization
template<class T>
struct helper1<T,true> {
  typedef typename
           helper2<T, std::numeric_limits<T>::is_integer>::type type;
};
// the second helper:
template<class T, bool F>
struct helper2 { typedef T type };

//  the second helper if the type is integer: a partial specialization
template<class T>
struct helper2<T,true> { typedef double type;}
```

Procedural programming

```
double integrate( double (*f) (double)),
                  double a, double b, unsigned int N)
{
  double result=0;
  double x=a;
  double dx=(b-a)/N;
  for (unsigned int i=0; i<N; ++i, x+=dx)
    result +=f(x);
  return result*dx;
}
```

```
double func(double x) {return x*sin(x);}
cout << integrate(func,0,1,100);
```

- same as in C, Fortran, etc.

Generic programming

```
template <class T, class F>
T integrate(F f, T a, T b, unsigned int N)
{
  T result=T(0);
  T x=a;
  T dx=(b-a)/N;
  for (unsigned int i=0; i<N; ++i, x+=dx)
    result +=f(x);
  return result*dx;
}
```

```
inline double func(double x) {return x*sin(x);}
std::cout << integrate(func,0.,1.,100);
```

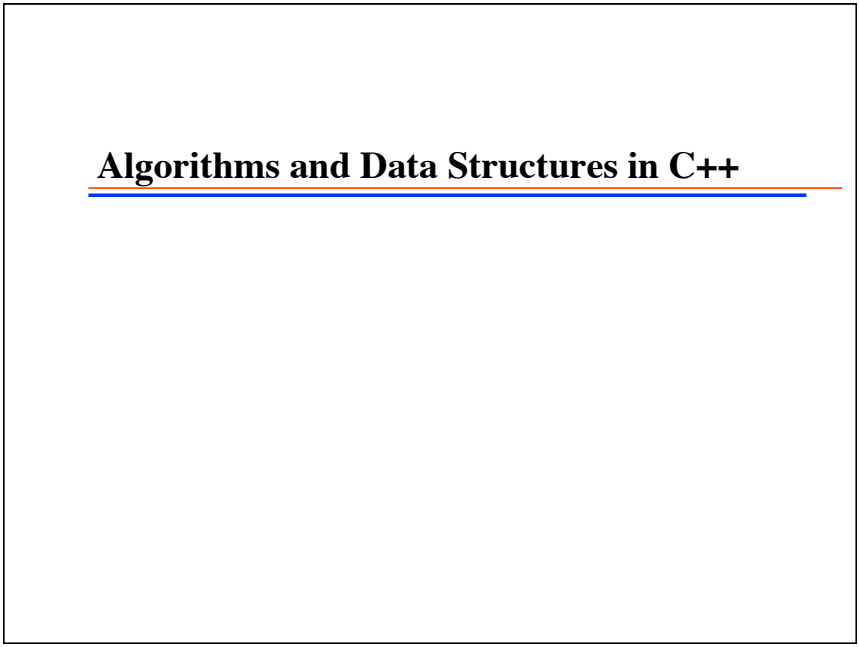
- allows inlining!
- works for any type T and F!

Function objects

- Assume a function with parameters: $f(x;\lambda) = \exp(-\lambda x)$
  - ```
    double func(double x, double lambda) {
      return exp(-lambda*x);
    }
    ```
  - cannot be used with integrate template!
- Solution: use a function object
  - ```
    class MyFunc {
      const double lambda;
    public:
      MyFunc(double l) : lambda(l) {}
      double operator() (double x) {return exp(-lambda*x);}
    };
    ```
  - ```
    MyFunc f(3.5)
    integrate(f,0.,1.,1000);
    ```
  - uses object of type MyFunc like a function!
- **Very useful and widely used technique**
    ```

Chapter 9

Algorithms and Data Structures in C++



Complexity analysis

- Answers the question "How does the time needed for an algorithm scale with the problem size *N*?"
  - Worst case analysis: maximum time needed over all possible inputs
  - Best case analysis: minimum time needed
  - Average case analysis: average time needed
  - Amortized analysis: average over a sequence of operations
- Usually only worst-case information is given since average case is much harder to estimate.

The O notation

- Is used for worst case analysis:

  An algorithm is $O(f(N))$ if there are constants c and N_0, such that for $N \geq N_0$ the time to perform the algorithm for an input size N is bounded by $t(N) < c\, f(N)$

- Consequences
  - $O(f(N))$ is identically the same as $O(a\, f(N))$
  - $O(a\, N^x + b\, N^y)$ is identically the same as $O(N^{\max(x,y)})$
  - $O(N^x)$ implies $O(N^y)$ for all $y \geq x$

Notations

- Ω is used for best case analysis:

  An algorithm is $\Omega(f(N))$ if there are constants c and N_0, such that for $N \geq N_0$ the time to perform the algorithm for an input size N is bounded by $t(N) > c\,f(N)$

- Θ is used if worst and best case scale the same

  An algorithm is $\Theta(f(N))$ if it is $\Omega(f(N))$ and $O(f(N))$

Time assuming 1 billion operations per second (1Gop)

Complexity	N=10	10^2	10^3	10^4	10^5	10^6
1	1 ns	1 ns	1 ns	1 ns	1 ns	1ns
ln N	3 ns	7 ns	10 ns	13 ns	17 ns	20 ns
N	10 ns	100 ns	1 μs	10 μs	100 μs	1 ms
N log N	33 ns	664 ns	10 μs	133 μs	1.7 ms	20 ms
N^2	100 ns	10 μs	1 ms	100 ms	10 s	17 min
N^3	1 μs	1 ms	1 s	17 min	11.5 d	31 a
2^N	1 μs	10^{14} a	10^{285} a	10^{2996} a	10^{30086} a	10^{301013} a

Time assuming 10 petaoperations per second (10 Pop/s)

Assume a parallel machine with 10 peta operations per second and perfect parallelization but one operation still needs at least 1ns

Complexity	N=10	10^2	10^3	**10^6**	**10^9**	**10^{12}**
1	1 ns	1 ns	1 ns	1 ns	1 ns	1 ns
ln N	1 ns	1 ns	1 ns	1 ns	1 ns	1 ns
N	1 ns	1 ns	1 ns	1ns	100 ns	100 μs
N log N	1 ns	1 ns	1 μs	1.33 ns	177 s	200 μs
N^2	1 ns	1 ns	1 ns	100 μs	100 s	3a
N^3	1 ns	1 ns	100 ns	100 s	3000 a	10^{12} a
2^N	1 ns	10^7 a	10^{278} a			

Which algorithm do you prefer?

- When do you pick algorithm A, when algorithm B? The complexities are listed below

Algorithm A	Algorithm B	Which do you pick?
O(ln N)	O(N)	
O(ln N)	N	
O(ln N)	1000 N	
ln N	O(N)	
1000 ln N	O(N)	
ln N	N	
ln N	1000 N	
1000 ln N	N	

Complexity: example 1

- What is the O, Ω and Θ complexity of the following code?

```
double x;
std::cin >> x;
std::cout << std::sqrt(x);
```

Complexity: example 2

- What is the O, Ω and Θ complexity of the following code?

```
unsigned int n;
std::cin >> n;
for (int i=0; i<n; ++i)
  std::cout << i*i << "\n";
```

Complexity: example 3

- What is the O, Ω and Θ complexity of the following code?

```
unsigned int n;
std::cin >> n;
for (int i=0; i<n; ++i) {
  unsigned int sum=0;
  for (int j=0; j<i; ++j)
    sum += j;
  std::cout << sum << "\n";
}
```

Complexity: example 4

- What is the O, Ω and Θ complexity of the following two segments?
  - Part 1:

```
unsigned int n;
std::cin >> n;
double* x=new double[n]; // allocate array of n numbers
for (int i=0; i<n; ++i)
  std::cin >> x[i];
```

  - Part 2:

```
double y;
std::cin >> y;
for (int i=0; i<n; ++i)
  if (x[i]==y) {
    std::cout << i << "\n";
    break;
  }
```

Complexity: adding to an array (simple way)

- What is the complexity of adding an element to the end of an array?
  - allocate a new array with N+1 entries
  - copy N old entries
  - delete old arrray
  - write (N+1)-st element
- The complexity is O(N)

Complexity: adding to an array (clever way)

- What is the complexity of adding an element to the end of an array?
  - allocate a new array with 2N entries, but mark only N+1 as used
  - copy N old entries
  - delete old arrray
  - write (N+1)-st element
- The complexity is O(N), but let's look at the next elements added:
  - mark one more element as used
  - write additional element
- The complexity here is O(1)
- The amortized (averaged) complexity for N elements added is

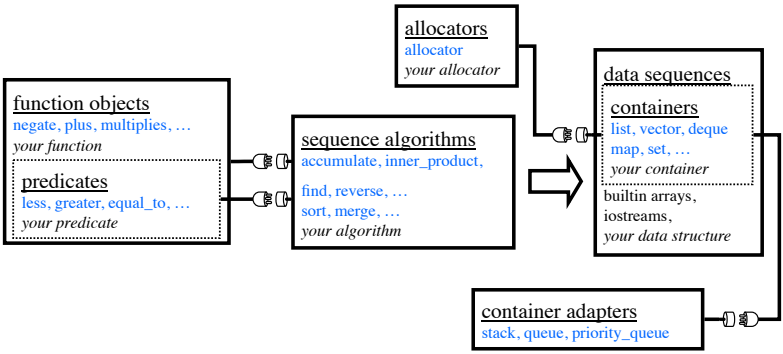
$$\frac{1}{N}\left(O(N)+(N-1)O(1)\right)=O(1)$$

STL: Standard Template Library

- Most notable example of generic programming
- Widely used in practice
- Theory: Stepanov, Musser; Implementation: Stepanov, Lee


- *Standard* Template Library
  - Proposed to the ANSI/ISO C++ Standards Committee in 1994.
  - After small revisions, part of the official C++ standard in 1997.

The standard C++ library



The `string` and `wstring` classes

- are very useful class to manipulate strings
  - `string` for standard ASCII strings (e.g. “English”)
  - `wstring` for wide character strings (e.g. “日本語”)
- Contains many useful functions for string manipulation
  - Adding strings
  - Counting and searching of characters
  - Finding substrings
  - Erasing substrings
  - …
- Since this is not very important for numerical simulations I will not go into details. Please read your C++ book

The `pair` template

```
template <class T1, class T2> class pair {
public:
  T1 first;
  T2 second;
  pair(const T1& f, const T2& s)
   : first(f), second(s)
  {}
};
```

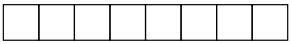
- will be useful in a number of places

Data structures in C++

- We will discuss a number of data structures and their implementation in C++:
- Arrays:
  - C array
  - `vector`
  - `valarray`
  - `deque`
- Linked lists:
  - `list`
- Trees
  - `map`
  - `set`
  - `multimap`
  - `multiset`
- Queues and stacks
  - `queue`
  - `priority_queue`
  - `stack`

The array or vector data structure

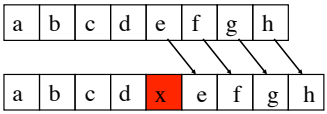
- An array/vector is a consecutive range in memory



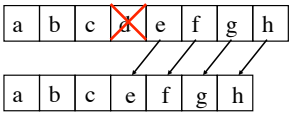
- Advantages
  - Fast $O(1)$ access to arbitrary elements: `a[i]` is `*(a+i)`
  - Profits from cache effects
  - Insertion or removal at the end is $O(1)$
  - Searching in a sorted array is $O(\ln N)$
- Disadvantage
  - Insertion and removal at arbitrary positions is $O(N)$

Slow O(*N*) insertion and removal in an array

- Inserting an element
  - Need to copy O(N) elements

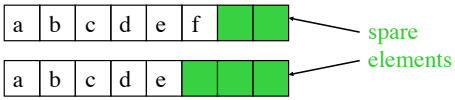


- Removing an element
  - Also need to copy O(N) elements

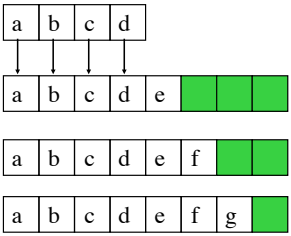


Fast O(1) removal and insertion at the end of an array

- Removing the last element
  - Just change the size
    - Capacity 8, size 6:
    - Capacity 8, size 5:



- Inserting elements at the end
  - Is amortized O(1)
    - first double the size and copy in O(N):
    - then just change the size:

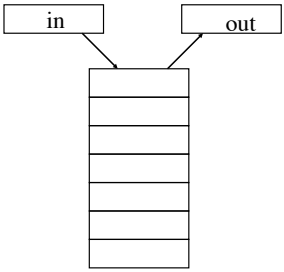


The deque data structure (double ended queue)

- Is a variant of an array, more complicated to implement
  - See a data structures book for details
- In addition to the array operations also the insertion and removal at beginning is $O(1)$
- Is needed to implement queues

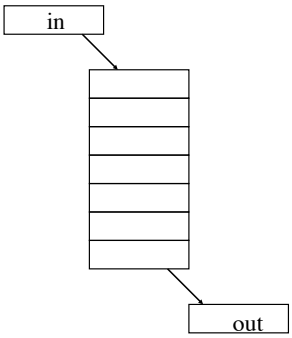
The stack data structure

- Is like a pile of books
  - LIFO (last in first out): the last one in is the first one out
- Allows in $O(1)$
  - Pushing an element to the top of the stack
  - Accessing the top-most element
  - Removing the top-most element



The queue data structure

- Is like a queue in the Mensa
  - FIFO (first in first out): the first one in is the first one out
- Allows in $O(1)$
  - Pushing an element to the end of the queue
  - Accessing the first and last element
  - Removing the first element

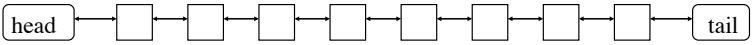


The priority queue data structure

- Is like a queue in the Mensa, but professors are allowed to go to the head of the queue (not passing other professors though)
  - The element with highest priority (as given by the < relation) is the first one out
  - If there are elements with equal priority, the first one in the queue is the first one out
- There are a number of possible implementations, look at a data structure book for details

The linked list data structure

- An linked list is a collection of objects linked by pointers into a one-dimensional sequence



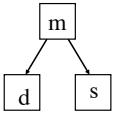
- Advantages
  - Fast $O(1)$ insertion and removal anywhere
    - Just reconnect the pointers
- Disadvantage
  - Does not profit from cache effects
  - Access to an arbitrary element is $O(N)$
  - Searching in a list is $O(N)$

The tree data structures

- An array needs
  - $O(N)$ operations for arbitrary insertions and removals
  - $O(1)$ operations for random access
  - $O(N)$ operations for searches
  - $O(\ln N)$ operations for searches in a sorted array
- A list needs
  - $O(1)$ operations for arbitrary insertions and removals
  - $O(N)$ operations for random access and searches
- What if both need to be fast? Use a tree data structure:
  - $O(\ln N)$ operations for arbitrary insertions and removals
  - $O(\ln N)$ operations for random access and searches

A node in a binary tree

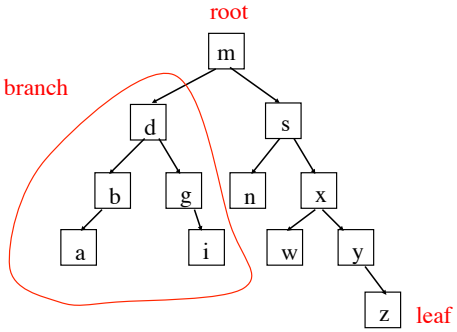
- Each node is always linked to two child nodes
  - The left child is always smaller
  - The right child node is always larger



A binary tree

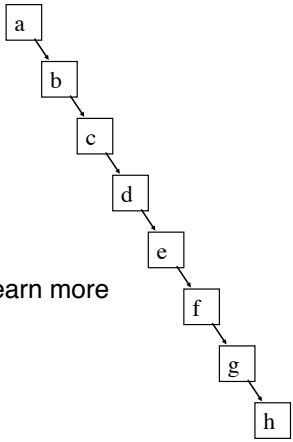
- Can store $N=2^n-1$ nodes in a tree of height n
- Any access needs at most $n = O(\ln N)$ steps

- Example: a tree of height 5 with 12 nodes



Unbalanced trees

- Trees can become unbalanced
  - Height is no longer O(ln N) but O(N)
  - All operations become O(N)
- Solutions
  - Rebalance the tree
  - Use self-balancing trees
- Look into a data structures book to learn more



Tree data structures in the C++ standard

- Fortunately the C++ standard contains a number of self-balancing tree data structures suitable for most purposes:
  - `set`
  - `multiset`
  - `map`
  - `multimap`
- But be aware that computer scientists know a large number of other types of trees and data structures
  - Read the books
  - Ask the experts

The container concept in the C++ standard

- Containers are sequences of data, in any of the data structures
  - `vector<T>` is an array of elements of type T
  - `list<T>` is a doubly linked list of elements of type T
  - `set<T>` is a tree of elements of type T
  - …
- The standard assumes the following requirements for the element T of a container:
  - default constructor `T()`
  - assignment `T& operator=(const T&)`
  - copy constructor `T(const T&)`
  - Note once again that assignment and copy have to produce identical copy: in the Penna model the copy constructor should not mutate!

Connecting Algorithms to Sequences

```
find( s, x ) :=
    pos ← start of s
    while pos not at end of s
        if element at pos in s == x
            return pos
        pos ← next position
    return pos
```

```
int find( char const(&s)[4], char x )
{
    int pos = 0;
    while (pos != sizeof(s))
    {
        if ( s[pos] == x )
            return pos;
        ++pos;
    }
    return pos;
}
```

```
struct node
{
    char value;
    node* next;
};
```

```
node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

Connecting Algorithms to Sequences

```
find( s, x ) :=
    pos ← start of s
    while pos not at end of s
        if element at pos in s == x
            return pos
        pos ← next position
    return pos
```

```
char* find(char const(&s)[4], char x)
{
    char* pos = s;
    while (pos != s + sizeof(s))
    {
        if ( *pos == x )
            return pos;
        ++pos;
    }
    return pos;
}
```

```
struct node
{
    char value;
    node* next;
};
```

```
node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

Connecting Algorithms to Sequences

```
find( s, x ) :=
    pos ← start of s
    while pos not at end of s
        if element at pos in s == x
            return pos
        pos ← next position
    return pos
```

```
char* find(char const(&s)[4], char x)
{
    char* pos = s;
    while (pos != s + sizeof(s))
    {
        if ( *pos == x )
            return pos;
        ++pos;
    }
    return pos;
}
```

```
struct node
{
    char value;
    node* next;
};
```

```
node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

F. T. S. E.

Fundamental Theorem of Software Engineering

"We can solve any problem by introducing an extra level of indirection"

--Butler Lampson

Andrew Koenig

Iterators to the Rescue

- Define a common interface for
  - traversal
  - access
  - positional comparison
- Containers provide iterators
- Algorithms operate on pairs of iterators

```
template <class Iter, class T>
Iter find( Iter start, Iter finish, T x )
{
   Iter pos = start;
   for (; pos != finish; ++pos)
   {
      if ( *pos == x )
         return pos;
   }
   return pos;
}
```

```
struct node_iterator
{
   // ...
   char& operator*() const
   { return n->value; }

   node_iterator& operator++()
   { n = n->next; return *this; }
private:
   node* n;
};
```

Describe Concepts for std::find

```
template <class Iter, class T>
Iter find(Iter start, Iter finish, T x)
{
    Iter pos = start;
    for (; pos != finish; ++pos)
    {
        if ( *pos == x )
            return pos;
    }
    return pos;
}
```

- Concept Name?
- Valid expressions?
- Preconditions?
- Postconditions?
- Complexity guarantees?
- Associated types?

Traversing an array and a linked list

- Two ways for traversing an array
  - Using an index:

```
T* a = new T[size];
for (int n=0;n<size;++n)
  cout << a[n];
```

  - Using pointers:

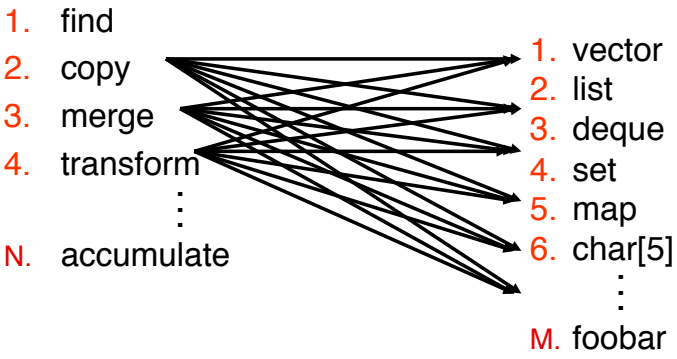
```
for (T* p = a;
     p !=a+size;
     ++p)
  cout << *p;
```

- Traversing a linked list

```
template <class T> struct node
{
  T value; // the element
  node<T>* next; // the next Node
};

template<class T> struct list
{
  node<T>* first;
};
list<T> l;
…
for (mode<T>* p=l.first;
       p!=0;
       p=p->next)
    cout << p->value;
```

NxM Algorithm Implementations?



Generic traversal

- Can we traverse a vector and a list in the same way?

- Instead of

```
for (T* p = a;
     p !=a+size;
     ++p)
  cout << *p;
```

- We want to write

```
for (iterator p = a.begin();
     p !=a.end();
     ++p)
  cout << *p;
```

- Instead of

```
for (node<T>* p=l.first;
     p!=0;
     p=p->next)
  cout << p->value;
```

- We want to write

```
for (iterator p = l.begin();
     p !=l.end();
     ++p)
  cout << *p;
```

Implementing iterators for the array

```
template<class T>
  class Array {
  public:
    typedef T* iterator;
    typedef unsigned size_type;
    Array();
    Array(size_type);

    iterator begin()
    { return p_;}
    iterator end()
    { return p_+sz_;}

  private:
    T* p_;
    size_type sz_;
  };
```

- Now allows the desired syntax:

```
for (Array<T>::iterator p =
  a.begin();
     p !=a.end();
     ++p)
  cout << *p;
```

- Instead of

```
for (T* p = a.p_;
     p !=a.p_+a.sz_;
     ++p)
  cout << *p;
```

Implementing iterators for the linked list

```
template <class T>
  struct node_iterator {
  Node<T>* p;
  node_iterator(Node<T>* q)
   : p(q) {}

  node_iterator<T>& operator++()
  { p=p->next;}

  T* operator ->()
  { return &(p->value);}

  T& operator*()
  { return p->value;}

  bool operator!=(const
        node_iterator<T>& x)
  { return p!=x.p;}

  // more operators missing ...
  };
```

```
template<class T>
  class list {
   Node<T>* first;
  public:
    typedef node_iterator<T> iterator;

    iterator begin()
    { return iterator(first);}

    iterator end()
    { return iterator(0);}
  };
```

- Now also allows the desired syntax:

```
for (List<T>::iterator p = l.begin();
     p !=l.end();
     ++p)
  cout << *p;
```

Iterators

- have the same functionality as pointers
- including pointer arithmetic!
  - `iterator a,b; cout << b-a;` // # of elements in [a,b[
- exist in several versions
  - forward iterators … move forward through sequence
  - backward iterators … move backwards through sequence
  - bidirectional iterators … can move any direction
  - input iterators … can be read: `x=*p;`
  - output iterators … can be written: `*p=x;`
- and all these in const versions (except output iterators)

Container requirements

- There are a number of requirements on a container that we will now discuss based on the handouts

Containers and sequences

- A container is a collection of elements in a data structure
- A sequence is a container with a linear ordering (not a tree)
  - vector
  - deque
  - list
- An associative container is based on a tree, finds element by a key
  - map
  - multimap
  - set
  - multiset
- The properties are defined on the handouts from the standard
  - A few special points mentioned on the slides

Sequence constructors

- A sequence is a linear container (vector, deque, list,…)
- Constructors
  - `container()` … empty container
  - `container(n)` … n elements with default value
  - `container(n,x)` … n elements with value x
  - `container(c)` … copy of container c
  - `container(first,last)` … first and last are iterators
    - container with elements from the range [first,last[
- Example:

```
std::list<double> l;
// fill the list
...
// copy list to a vector
std::vector<double> v(l.begin(),l.end());
```

Direct element access in deque and vector

- Optional element access (not implemented for all containers)
  - `T& container[k]` … k-th element, no range check
  - `T& container.at(k)` … k-th element, with range check
  - `T& container.front()` … first element
  - `T& container.back()` … last element

Inserting and removing at the beginning and end

- For all sequences: inserting/removing at end
  - `container.push_back(T x)` // add another element at end
  - `container.pop_back()` // remove last element
- For list and deque (stack, queue)
  - `container.push_front(T x)` // insert element at start
  - `container.pop_front()` // remove first element

Inserting and erasing anywhere in a sequence

- List operations (slow for vectors, deque etc.!)
  - `insert (p,x)` // insert x before p
  - `insert(p,n,x)` // insert n copies of x before p
  - `insert(p,first,last)` // insert [first,last[before p
  - `erase(p)` // erase element at p
  - `erase(first,last)` // erase range[first,last[
  - `clear()` // erase all

Vector specific operations

- Changing the size
  - `void resize(size_type)`
  - `void reserve(size_type)`
  - `size_type capacity()`
- Note:
  - `reserve` and `capacity` regard memory `allocated` for vector!
  - `resize` and `size` regard memory currently used for vector data
- Assignments
  - `container = c` … copy of container c
  - `container.assign(n)` …assign n elements the default value
  - `container.assign(n,x)` … assign n elements the value x
  - `container.assign(first,last)` … assign values from the range [first,last[
- Watch out: assignment does not allocate, do a resize before!

The `valarray` template

- acts like a vector but with additional (mis)features:
  - No iterators
  - No reserve
  - Resize is fast but erases contents
- Many numeric operations are defined:

```
std::valarray<double> x(100), y(100), z(100);
x=y+exp(z);
```

  - Be careful: it is not the fastest library!
  - We will learn about faster libraries later

Sequence adapters: `queue` and `stack`

- are based on deques, but can also use vectors and lists
  - `stack` is first in-last out
  - `queue` is first in-first out
  - `priority_queue` prioritizes with < operator
- stack functions
  - `void push(const T& x)` … insert at top
  - `void pop()` … removes top
  - `T& top()`
  - `const T& top() const`
- queue functions
  - `void push(const T& x)` … inserts at end
  - `void pop()` … removes front
  - `T& front()`, `T& back()`, `const T& front()`, `const T& back()`

`list` -specific functions

- The following functions exist only for std::list:
  - `splice`
    - joins lists without copying, moves elements from one to end of the other
  - `sort`
    - optimized sort, just relinks the list without copying elements
  - `merge`
    - preserves order when "splicing" sorted lists
  - `remove(T x)`
  - `remove_if(criterion)`
    - criterion is a function object or function, returning a bool and taking a `const T&` as argument, see Penna model
    - example:

```
bool is_negative(const T& x) { return x<0;}
```

    - can be used like

```
list.remove_if(is_negative);
```

The `map` class

- implements associative arrays
  -

```
map<std::string,long> phone_book;
phone_book["Troyer"] = 32589;
phone_book["Heeb"] = 32591;
if(phone_book[name])
  cout << "The phone number of " << name <<  " is "
       << phone_book[name];
else
  cout << name << "\'s phone number is unknown!';
```

- is implemented as a tree of `pair`s
- Take care:
  - `map<T1,T2>::value_type` is `pair<T1,T2>`
  - `map<T1,T2>::key_type` is `T1`
  - `map<T1,T2>::mapped_type` is `T2`
  - `insert`, `remove`, ... are sometimes at first sight confusing for a map!

Other tree-like containers

- `multimap`
  - can contain more than one entry (e.g. phone number) per key
- `set`
  - unordered container, each entry occurs only once
- `multiset`
  - unordered container, multiple entries possible
- extensions are no problem
  - if a data structure is missing, just write your own
  - good exercise for understanding of containers

Search operations in trees

- In a map<K,V>, K is the key type and V the mapped type
  - Attention: iterators point to pairs
- In a set<T>, T is the key type and also the value_type
- Fast O(log *N*) searches are possible in trees:
  - `a.find(k)` returns an iterator pointing to an element with key k or end() if it is not found.
  - `a.count(k)` returns the number of elements with key k.
  - `a.lower_bound(k)` returns an iterator pointing to the first element with key >= k.
  - `a.upper_bound(k)` returns an iterator pointing to the first element with key > k.
  - `a.equal_range(k)` is equivalent to but faster than `std::make_pair(a.lower_bound(k) , a.upper_bound(k))`

Search example in a tree

- Look for all my phone numbers:
  - // some typedefs

```
typedef multimap<std::string, int> phonebook_t;
typedef phonebook_t::const_iterator IT;
typedef phonebook_t::value_type value_type;

// the phonebook
phonebook_t phonebook;

// fill the phonebook
phonebook.insert(value_type("Troyer",32589));
…

// search all my phone numbers
pair< IT,IT> range =  phonebook.equal_range("Troyer");

// print all my phone numbers
for (IT it=range.first; it != range.second;++it)
  cout << it->second << "\n";
```

Almost Containers

- C-style array
- `string`
- `valarray`
- `bitset`

- They all provide almost all the functionality of a container
- They can be used like a container in many instances, but not all
  - `int x[5] = {3,7,2,9,4};`
    `vector<int> v(x,x+5);`
  - uses `vector(first,last)`, pointers are also iterators!

The generic algorithms

- Implement a big number of useful algorithms
- Can be used on any container
  - rely only on existence of iterators
  - "container-free algorithms"
  - now all the fuss about containers pays off!
- Very useful
- Are an excellent example in generic programming
- We will use them now for the Penna model
  That's why we did not ask you to code the Population class for the Penna model yet!

Example: `find`

- A generic function to find an element in a container:

```
list<string> fruits;
list<string>::const_iterator found =
  find(fruits.begin(),fruits.end(),"apple");
if (found==fruits.end()) // end means invalid iterator
 cout << "No apple in the list";
else
  cout << "Found it: " << *found << "\n";
```

- find declared and implemented as

```
template <class In, class T>
  In find(In first, In last, T v) {
    while (first != last  && *first != v)
      ++first;
    return first;
  }
```

Example: `find_if`

- takes predicate (function object or function)
  - `bool favorite_fruits(const std::string& name)`
    `{ return (name==“apple” || name == “orange”);}`
- can be used with `find_if` function:
  - ```
    list<string>::const_iterator found =
      find_if(fruits.begin(),fruits.end(),favorite_fruits);
    if (found==fruits.end())
      cout << “No favorite fruits in the list”;
    else
      cout << “Found it: “ << *found << “\n”;
    ```
- find_if declared and implemented as as
  - ```
    template <class In, class Pred>
      In find_if(In first, In last, Pred p) {
        while (first != last && !p(*first) )
          ++first;
        return first;
    }
    ```

Member functions as predicates

- We want to find the first pregnant animal:
  - ```
    list<Animal> pop;
    find_if(pop.begin(),pop.end(),is_pregnant)
    ```
- This does not work as expected, it expects
  - `bool is_pregnant(const Animal&);`
- We want to use
  - `bool Animal::pregnant() const`
- Solution: mem_fun_ref function adapter
  - ```
    find_if(pop.begin(),pop.end(),
                    mem_fun_ref(&Animal::pregnant));
    ```
- Many other useful adapters available
  - Once again: please read the books before coding your own!

push_back and back_inserter

- Attention:
  - `vector<int> v,w;`
    `for (int k=0;k<100;++k){`
    `v[k]=k; //error: v is size 0!`
    `w.push_back(k); // OK:grows the array and assigns`
    `}`
- Same problem with copy:
  - `vector<int> v(100), w(0);`
    `copy(v.begin(),v.end(),w.begin()); // problem: w of size 0!`
- Solution1: vectors only
  - w.resize(v.size()); copy(v.begin(),v.end(),w.begin());
- Solution 2: elegant
  - copy(v.begin(),v.end(),back_inserter(w)); // uses push_back
- also push_front and front_inserter for some containers

Penna Population

- easiest modeled as
  - `class Population {`
    `private: std::list<Animal> l;`
    `}`
- Removing dead:
  - `l.remove_if(mem_fun_ref(&Animal::is_dead));`
- Removing dead, and others with probability N/N0:
  - `l.remove_if(animal_dies(N/N0));`
  - where `animal_dies` is a function object taking N/N0 as parameter
- Inserting children:
  - cannot go into same container, as that might invalidate iterators:

```
std::vector<Animal> children;
for(const_iterator a=l.begin();a!=l.end();++a)
  if(a->pregnant())
    children.push_back(a->child());
std::copy(children.begin(),children.end(),
    back_inserter(l);
```

The binary search

- Searching using binary search in a sorted vector is $O(\ln N)$
- Binary search is recursive search in range [begin,end[
  - If range is empty, return
  - Otherwise test middle=begin+(end-begin)/2
    - If the element in the middle is the search value, we are done
    - If it is larger, search in [begin,middle[
    - If it is smaller, search in [middle,end[
- The search range is halved in every step and we thus need at most $O(\ln N)$ steps

Example: lower_bound

```
template<class IT, class T>
IT lower_bound(IT first, IT last, const T& val) {
  typedef typename iterator_traits<IT>::difference_type dist_t;
  dist_t len = distance(first, last); // generic function for last-first
  dist_t half;
  IT middle;
  while (len > 0) {
    half = len >> 1; // faster version of half=len/2
    middle = first;
    advance(middle, half);// generic function for middle+=half
    if (*middle < val) {
      first = middle;
      ++first;
      len = len - half - 1;
    }
    else
      len = half;
  }
  return first;
}
```

Algorithms overview

- Nonmodifying
  - `for_each`
  - `find, find_if, find_first_of`
  - `adjacent_find`
  - `count, count_if`
  - `mismatch`
  - `equal`
  - `search`
  - `find_end`
  - `search_n`
- Modifying
  - `transform`
  - `copy, copy_backward`
  - `swap, iter_swap, swap_ranges`
  - `replace, replace_if, replace_copy, replace_copy_if`
  - `fill, fill_n`
  - `generate, generate_n`
  - `remove, remove_if, remove_copy, remove_copy_if`
  - `unique, unique_copy`
  - `reverse, reverse_copy`
  - `rotate, rotate_copy`
  - `random_shuffle`

Algorithms overview (continued)

- Sorted Sequences
  - `sort,stable_sort`
  - `partial_sort, partial_sort_copy`
  - `nth_element`
  - `lower_bound, upper_bound`
  - `equal_range`
  - `binary_search`
  - `merge, inplace_merge`
  - `partition, stable_partition`
- Permutations
  - `next_permutation`
  - `prev_permutation`
- Set Algorithms
  - `includes`
  - `set_union`
  - `set_intersection`
  - `set_difference`
  - `set_symmetric_difference`
- Minimum and Maximum
  - `min`
  - `max`
  - `min_element`
  - `max_element`
  - `lexicographical_compare`

Exercise

- Code the population class for the Penna model based on a standard container
- Use function objects to determine death
- In the example we used a loop.
  - Can you code the population class without using any loop?
  - This would increase the reliability as the structure is simpler!
- Also add fishing in two variants:
  - fish some percentage of the whole population
  - fish some percentage of adults only
- Read Penna's papers and simulate the Atlantic cod!
  Physica A, **215**, 298 (1995)

stream iterators and Shakespeare

- Iterators can also be used for streams and files
  - `istream_iterator`
  - `ostream_iterator`
- Now you should be able to understand Shakespeare:

```
int main()
 {
    vector<string> data;
    copy(istream_iterator<string>(cin),istream_iterator<string>(),
         back_inserter(data));
    sort(data.begin(), data.end());
    unique_copy(data.begin(),data.end(),ostream_iterator<string>(cout,"\n"));
}
```

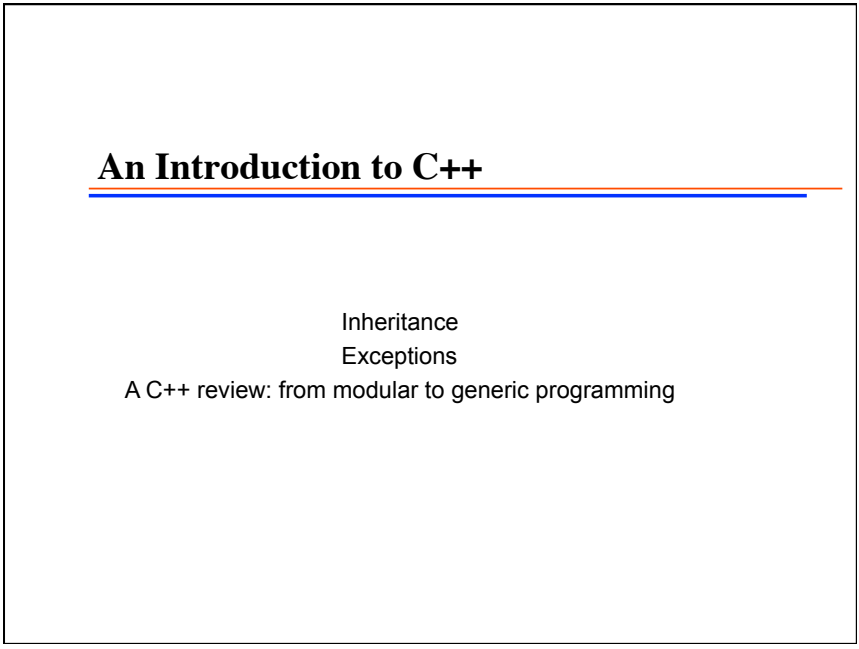
Summary

- Please read the sections on
  - containers
  - iterators
  - algorithms
- in Stroustrup or Lippman (3rd editions only!)

- Examples of excellent class and function designs
- Before writing your own functions and classes:
  Check the standard C++ library!
- When writing your own functions/classes:
  Try to emulate the design of the standard library
- Don't forget to include the required headers:
  - `<algorithm>`, `<functional>`, `<map>`, `<iterators>`, ... as needed

Chapter 10

Inheritance, Exceptions, A C++ review: from modular to generic programming



Inheritance

- is another very important feature
- it models the concept:
  objects of type B are the same as A, but in addition have…
- Examples
  - A shape is a 2D figure which has an area and can be drawn, although I know neither generally
  - A triangle is a shape, but its area is … and it looks like …
  - A square is a shape, but its area is … and it looks like …
  - A complex figure is a shape and consists of an array of shapes
  - A monoid is a semigroup, but in addition contains a unit element
  - A group is a monoid, but in addition has an inverse
  - A simulation can be run but I don't know how generally
  - A Penna simulation is run this way …

Abstract base classes

- are good for expressing common ideas
- We want to have a function that can run a simulation and print some information:
  - ```
    void perform(Simulation& s) {
      std::cout << "Running the simulation "
                << s.name() << "\n";
      s.run(); // run it
    }
    ```
- This class must have an `name()` and a `run()` member function
  - ```
    class Simulation{
    public:
      Simulation () {};
      virtual std::string name() const =0;
      virtual void run() =0;
    };
    ```
  - `virtual` means that this function depends on concrete simulation
  - `=0` means that this function **must** be provided for any concrete simulation

Concrete derived classes

- PennaSim and IsingSim are both Simulations:
  - ```
    class PennaSim: public Simulation {
    public:
      std::string name() const;
      void run();
    };
    ```
  - ```
    class IsingSim: public Simulation{
    public:
      std::string name() const;
      void run()
    };
    ```
- Examples
  - `Simulation x;`
    // Error since it is abstract! `name()` and `run()` not defined
  - `PennaSim p;` // OK!
  - `IsingSim i;` // OK!
  - `Simulation& sim=p;` // also OK, since it is a reference!

Using inheritance

- recall the function `void perform(Simulation&);`
- let us call it for two simulations

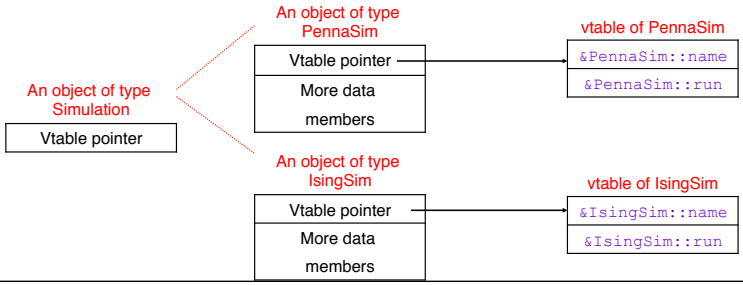
```
PennaSim p;
IsingSim i;
perform(p); // will use PennaSim::name() and PennaSim::run()
perform(i); // will use IsingSim::name() and IsingSim::run()
```

- All virtual function can be redefined by derived class
- In addition a derived class can define additional members

- There exists a third access specifier: `protected`
  - means `public` for derived classes
  - means `private` for others

The virtual function table

- How does the program know the concrete type of an object?
  - The compiler creates a virtual function table (vtable) for each class
    - The table contains pointers to the functions
  - A pointer to that table is stored in the object, before the other members
  - The program checks the virtual function table of the object for the address of the function to call
    - Needs two memory accesses and cannot be inlined



Using templates instead

- The same could be done with templates:

```
template <class SIMULATION>
void perform(SIMULATION& s) {
  std::cout << s.name() << "\n";
  s.run();
}
```

```
class PennaSim {
public:
  std::string name() const;
  void run();
};
```

```
PennaSim p;
perform(p); // instantiates the template for PennaSim
```

- But type of `SIMULATION` must be known at compile time!

Comparing virtual functions and templates

- Object Oriented Programming:
  - ```
    void perform(Simulation& s)
    {
      s.run();
    }
    ```
  - Object needs to be derived from Simulation
  - Concrete type decided at runtime
- Generic programming:
  - ```
    template <class SIM> void perform(SIM& s)
    {
     s.run();
    }
    ```
  - Object needs to have a run function
  - Concrete type decided at compile time

Runtime and compile time polymorphism

Runtime polymorphism

- uses virtual functions
- decision at run-time
- works for objects derived from the common base
- one function created for the base class -> saves space
- virtual function call needs lookup in type table -> slower
- extension possible using only definition of base class
- Most useful for application frameworks, user interfaces, “big” functions

Compile time polymorphism

- uses templates
- decision at compile-time
- works for objects having the right members
- a new function created for each class used -> more space
- no virtual function call, can be inlined -> faster
- extension needs definitions and implementations of all functions
- useful for small, low level constructs, small fast functions and generic algorithms

When to use which?

- Generic programming allows inlining
  - faster code
- Object oriented programming more flexible
  - how to age an Array of animals of different types?

```
void show(std::vector<Animal*> a) {
  for (int i=0; i<a.size(); ++i)
    a[i]->grow();
}
```

  - This works for array of mixed animals, e.g. fish, sheep, …

Example: random number generators

- We want to be able to switch random number generators at runtime: use virtual functions
- First attempt: rng1.h
  - Make `operator()` a virtual function
  - Problem: virtual function calls are slow
- Second attempt: rng2.h
  - Store a buffer of random numbers
  - `operator()` uses numbers from that buffer
  - Only when buffer is used up, a virtual function `fill_buffer()` is called to create many random numbers
  - This reduces the cost of inheritance since the virtual function is called only rarely

Example: Penna model with fishing

- The Penna population class was:

```
class Population {
    … constructors and more …
    void simulate(int years); // the full simulation
    void step(); // one year
}
```

```
void Population::simulate(int years)
{
    while (years--)
      step();
}
```

```
void Population::step()
{
    … lots of work to do one year …
}
```

Example: Penna with fishing (part 2)

- Now we want to do a new simulation with fishing
- We want to reuse code and not copy&paste

```
class FishingPopulation : public Population
{
    … constructors …
    void step(); // one year
}
```

```
void FishingPopulation::step()
{
    Population::step(); // do the normal aging
    … then implement fishing …
}
```

Example: Penna with fishing (part 3)

- This will not do what we want since `Population::simulate` does not call the step function of `FishingPopulation`
  - ```
    void Population::simulate(int years)
    {
      while (years--)
        step();
    }
    ```
- Solution: need to make step() a virtual function
  - ```
    class Population {
      … constructors and more …
      void simulate(int years); // the full simulation
      virtual void step(); // one year
    }
    ```

How to deal with runtime errors?

- What should our integration library do if the user passes an illegal argument?
  - Return 0?
  - Return infinity?
  - Abort?
  - Set an error flag?
- Neither of these is ideal
  - Return values of 0 or infinity cannot be distinguished from good results
  - Aborting the program is no good idea for mission critical programs
  - Error flags are rarely checked by the users
- Solution
  - C++ exception handling

C++ Exceptions

- The solution are exceptions
- The library recognizes an error or other exceptional situation.
  - It does not know how to deal with it
  - Thus it *throws* an exception
- The calling program might be able to deal with the exception
  - It can *catch* the exception and do whatever is necessary
- If an exception is not caught
  - The program terminates

How to throw an exception

- What is an exception?
  - An object of any type
- Thrown using the `throw` keyword:
  - ```
    if(n<=0)
        throw "n too small";
    ```
  - ```
    if(index >= size())
        throw std::range_error("index");
    ```
- Throwing the exception
  - causes the normal execution to terminate
  - The call stack is unwound, the functions are exited, all local objects destroyed
  - Until a catch clause is found

The standard exception base class

- Is in the header `<exception>`
  - ```
    class exception {
    public:
      exception() throw();
      exception(const exception&) throw();
      exception& operator=(const exception&) throw();
      virtual ~exception() throw();
      virtual const char* what() const throw();
    };
    ```
- The function qualifier `throw()` indicates that these functions do not throw any exceptions

The standard exceptions

- are in `<stdexcept>`, all derived from `std::exception`
- Logic errors (base class `std::logic_error`)
  - `domain_error`: value outside the domain of the variable
  - `invalid_argument`: argument is invalid
  - `length_error`: size too big
  - `out_of_range`: argument has invalid value
- Runtime errors (base class `std::runtime_error`)
  - `range_error`: an invalid value occurred as part of a calculation
  - `overflow_error`: a value got too large
  - `underflow_error`: a value got too small
- All take a string as argument in the constructor
    ```

Catching exceptions

- Statements that might throw an exception are put into a `try` block
- After it `catch()` clauses can catch some or all exceptions
- Example:

```
int main()
{
  try {
    std::cout << integrate(sin,0,10,1000);
  }
  catch (std::exception& e) {
    std::cerr << "Error: " << e.what() << "\n";
  }
  catch(...) {// catch all other exceptions
    std::cerr << "A fatal error occurred.\n";
  }
}
```

Exceptions example

```
int main() {

  bool done;
  do {
    done = true;
    try {
      double a,b;
      unsigned int n;
      std::cin >> a >> b >> n;
      std::cout << simpson(sin,a,b,n);
    }
    catch (std::range_error& e) {
      // also catches derived exceptions
      std::cerr << "Range error: " << e.what() << "\n";
      done=false;
    }
  // all other exceptions go uncaught
  } while (!done);
}
```

More exception details

- Exceptions and inheritance
  - A `catch(ExceptionType& t)` clause also catches exceptions derived from `ExceptionType`
- Rethrowing excpeptions
  - If a catch() clause decides it cannot deal with the exception it can re-throw it with `throw;`
- More details in text books
  - Uncaught exceptions
  - throw() qualifiers
  - Exceptions thrown while dealing with an exception
  - Exceptions in destructors
    - Can be very bad since the destructor is not called!

C++ review

- Stack class
  - procedural
  - modular
  - object oriented
  - generic

Procedural stack implementation: stack1.C

```
void push(double*& s, double v)
{
  *s++=v;
}

double pop(double *&s)
{
  return *--s;
}
```

```
int main() {

double stack[1000];
double* p=stack;

push(p,10.);

std::cout << pop(p) << "\n";
std::cout << pop(p) << "\n";
// error of popping below
// beginning goes undetected!
}
```

Modular stack implementation: stack2.C

```
namespace Stack {
struct stack {
  double* s;
  double* p;
  int n;};

void init(stack& s, int l) {
  s.s=new double[l];
  s.p=s;
  s.n=l;}

void destroy(stack& s) {
  delete[] s.s;
}
```

```
void push(stack& s, double v) {
  if (s.p==s.s+s.n-1) throw
    std::runtime_error("overflow");
  *s.p++=v;
}

double pop(stack& s) {
  if (s.p==s.s) throw
    std::runtime_error("underflow");
  return *--s.p;
}

int main() {
Stack::stack s;
Stack::init(s,100); // must be called
Stack::push(s,10.);
Stack::pop(s);
Stack::pop(s); // throws error
Stack::destroy(s); // must be called
}
```

Object oriented stack implementation: stack3.C

```
namespace Stack {
class stack {
  double* s;
  double* p;
  int n;
public:
  stack(int=1000); // like init
  ~stack(); // like destroy
  void push(double);
  double pop();
};
```

```
int main() {
  Stack::stack s(100);
  // initialization done automatically
  s.push(10.);
  std::cout << s.pop();
  // destruction done automatically
}
```

Generic stack implementation: stack4.C

```
namespace Stack {
template <class T>
class stack {
  T* s;
  T* p;
  int n;
public:
  stack(int=1000); // like init
  ~stack(); // like destroy
  void push(T);
  T pop();
};
```

```
int main() {
  Stack::stack<double> s(100);
  // works for any type!
  s.push(1.3);
  cout << s.pop();
}
```

Summary of Programming Styles

- Procedural implementation
  - possible in many languages
- Modular implementation
  - allows transparent change in underlying data structure without breaking the user's program. E.g. we can add range checks
- Object oriented implementation
  - provides data encapsulation
  - makes sure that initialization and cleanup functions are called whenever needed
- Generic implementation
  - works for any data type

Review of the numerical integration exercise

- The numerical integration exercise demonstrates all four programming styles:
  - 1st part: procedural programming
  - 2nd part: modular programming
    - We built a library
  - 3rd part generic programming
    - We used templates
  - 4th part: object oriented programming
    - We derive from a base class
- After you have coded all four versions, perform benchmarks
  - Which version is fastest?
  - Which version is the most flexible?

Procedural programming

```
double integrate( double (*f) (double),
                  double a, double b, unsigned int N)
{
  double result=0;
  double x=a;
  double dx=(b-a)/N;
  for (unsigned int i=0; i<N; ++i, x+=dx)
    result +=f(x);
  return result*dx;
}
```

```
double func(double x) {return x*sin(x);}
cout << integrate(func,0,1,100);
```

- same as in C, Fortran, etc.

Generic programming

```
template <class T, class F>
T integrate(F f, T a, T b, unsigned int N)
{
  T result=T(0);
  T x=a;
  T dx=(b-a)/N;
  for (unsigned int i=0; i<N; ++i, x+=dx)
    result +=f(x);
  return result*dx;
}
```

```
struct func {
  double operator()(double x) { return x*sin(x); }
};
cout << integrate(func(),0.,1.,100);
```

- allows inlining!
- works for any type T

Object oriented programming

```
struct SimpleFunction {  // base class provides function interface
    Function () {}
    virtual double operator()(double) const=0;
};
```

```
double integrate(SimpleFunction const& f,
                 double a, double b, unsigned int N)
{
  double result=0;
  double x=a;
  double dx=(b-a)/N;
  for (unsigned int i=0; i<N; ++i, x+=dx)
    result +=f(x);
  return result*dx;
}
```

- Lets us choose the function at runtime

Object oriented programming

```
struct MyFunc : public SimpleFunction {  // derived class
public:
  MyFunc() {}
  double operator()(double x) {return x*sin(x);}  //implements
function
};
```

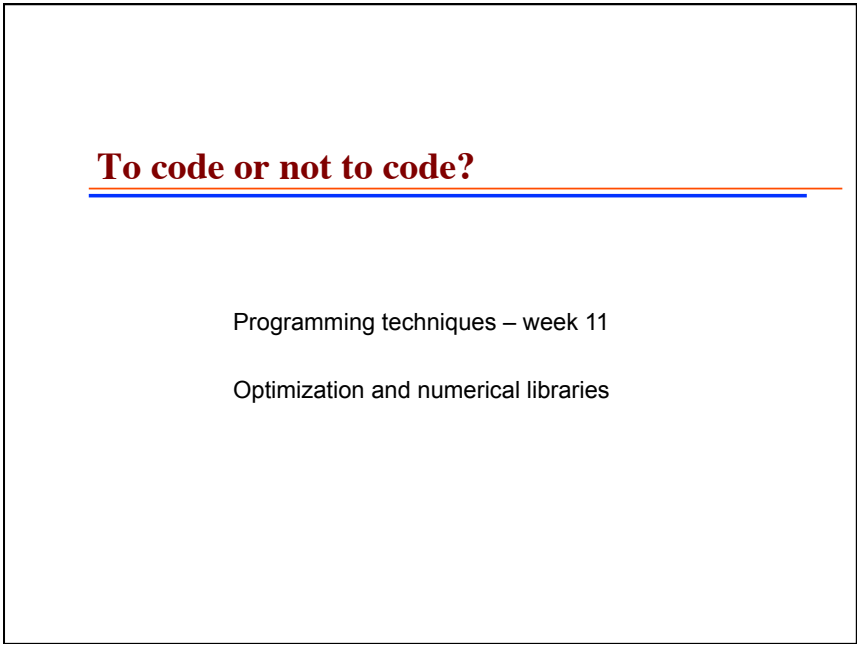
```
MyFunc f;
integrate(f, 0,1,1000);
```

C++11 provides much nicer ways!

- Stay tuned

Chapter 11

Optimization and numerical libraries



Optimization

- First rule: Do not optimize!
- What if the program is too slow?
  - Use compiler optimization flags
  - find optimal algorithm
  - use libraries
- What if the program is still too slow?
  - use profiling to determine which parts are slow
  - investigate slow part and check that data structures are optimal
    - are arrays, lists or trees better?
  - is the algorithm optimal?
    - check literature for better algorithms
    - use libraries
  - only then think about optimizing
- Consider parallelization or vectorization

Profiling

- is used to determine how much time is spent in which program parts
- Three easy steps:
  - compile the program with the `-p` option
  - run the program
  - use `prof` to look at the performance data
- Alternative using gprof:
  - compile with the `-pg` option
  - run the program
  - use `gprof` to look at the performance data
  - includes time spent in called functions
- See the man pages for details about these programs

Choice of data structures

- choose your data structures depending on the use
- was discussed before and in the exercises:
  - if you need random access use an array
  - if you need to insert in the middle use a list
  - if you need both use a tree
- use the standard C++ library containers wherever possible. They are (nearly) optimal.
- if you need a container not included:
  - design your own in the STL style
  - make it available to others

Example: the best data structure for the Penna model

- We picked a linked list because removal from the middle of a vector is slow
- However a vector might be faster:
  - We do not care about the order of the animals
  - We can implement a special remove_if:
    - Replaces the removed animal with the last one
    - This makes removal fast
- We can code a container derived from vector with a special remove_if
  - Will be faster than a std::list
  - Will require only a one-line change in the Penna code
- Look at penna_vector.h

Choice of algorithms

- Look at the scaling of the algorithms with problem size:
- Fourier transform
  - Simple: O(N^2)
  - Fast Fourier Transform: O(N log N)
- Matrix-Matrix multiplication
  - Simple: O(N^3)
  - Strassen: O($N^{2.8}$)
  - Coppersmith and Winograd: O($N^{2.376}$), but the constants are too large to make it useful for any matrix that we can actually multiply
- Eigenvalues:
  - all eigenvalues, dense matrix: O(N^3)
  - some eigenvalues, dense matrix: O(N^2)
  - some eigenvalues, sparse matrix: O(N)

The Strassen algorithm

- is one example why you should use libraries even for trivial-looking operations
- Normal matrix-matrix multiplication is order O(N^3)
- Strassen algorithm is O($N^{\log 7/\log 2}$)=O($N^{2.8}$)
  - write matrix as four submatrices

$$C = AB \qquad \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

  - use a clever scheme

$$c_{11} = Q_1 + Q_4 - Q_5 + Q_7$$
$$c_{21} = Q_2 + Q_4$$
$$c_{12} = Q_3 + Q_5$$
$$c_{22} = Q_1 + Q_3 - Q_2 + Q_6$$

$$Q_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$
$$Q_2 = (a_{21} + a_{22})b_{11}$$
$$Q_3 = a_{11}(b_{12} - b_{22})$$
$$Q_4 = a_{22}(-b_{11} + b_{21})$$
$$Q_5 = (a_{11} + a_{12})b_{22}$$
$$Q_6 = (-a_{11} + a_{21})(b_{11} + b_{12})$$
$$Q_7 = (a_{12} - a_{22})(b_{21} + b_{22})$$

Comparing matrix multiplication algorithms

- Standard algorithm is $O(N^3)$
  - N^3 multiplications
  - $N^2(N\text{-}1)$ additions
- Strassen algorithm takes
  - 7 multiplications of matrices of size $N/2$
  - 18 additions of matrices of size $N/2$
- What is the complexity $T_{strassen}(N)$?
  - $T_{strassen}(N) = 7\ T_{strassen}(N/2) + 18/4\ N^2$
- Assuming $T_{strassen}(N) > O(N^2)$
  - $O(T_{strassen}(N)) = 7\ O(T_{strassen}(N/2))$
  - $O(T_{strassen}(2N)) = 7\ O(T_{strassen}(N))$
  - $\Rightarrow T_{strassen}(N) = O(N^{\log 7/\log 2})$

How do we find the best algorithm?

- Look in books of Knuth
- Search the SIAM journals
- Do not trust the Numerical Recipes too much
- But the easiest solution is: use a library
  - bug free (less buggy than your codes)
  - optimized (probably better than you can do)
  - well documented (do you ever document your codes?)
  - supported on most architectures
- A huge collection is available on netlib at http://www.netlib.org/
- In the next weeks we will introduce a variety of useful libraries

How to optimize

- Generally you should use a library instead of optimizing yourself.
- But as computational scientists you will sometimes
  - have to write libraries
  - enter new research fields and algorithms where there is no library
- We will learn how to optimize
  - Optimization using assembly language
  - Classical optimization techniques for any language
  - Optimization in C++
- And look at libraries using these optimization techniques

Optimization in assembly language

- Sometimes the CPU possesses machine language instructions that cannot be used directly from a high level language
  - Bit counts
  - Vector instructions (discussed in next year's HPC lecture)
- Assembly language instructions can be mixed with C++
  - Advantage: can speed up code
  - Disadvantage: code becomes non-portable
  - useful only in very rare cases, but can potentially make a big difference
- Best approach
  - Encapsulate assembly language call in a library

Example: counting leading zeroes in an integer

- Problem: count the number of leading zeroes in a 32-bit integer
  - Can be used to get the position of the highest bit set
  - Can be used to calculate the logarithm base 2 of an integer
- Solution in C++: requires a loop

```
int count_leading_zeroes(int x) {
  for (int i = 0 ; i<32 ;++i)
    if (x&(1<<(31-i)))
      return i;
  return 32;
}
```

- Solution in PowerPC-assembler: (powerpc_asm.C)

```
inline int count_leading_zeros (int x) {
  int c;
  asm ("cntlzw %1,%0" : "=r" (c) : "r" (x) );
  return c;
}
```

Inline assembly statements

- We used an inline assembly statement, which mixes assembly language with C++:

```
asm ("cntlzw %1,%0" : "=r" (c) : "r" (x) );
```

- Explaining the syntax:
  - `asm(…)` : inserts an inline assembly language statement
  - `cntlzw r9,r15` : puts the number of leading zeroes in register 9 into register 15
  - `cntlzw %1, %0` : we do not know which register the compiler will use and thus use placeholders `%0` and `%1` (use `%2` … if more registers are needed)
  - `: "=r" (c)` : puts the variable `c` into the register marked by `%0` (and after the execution assigns the value of the register `%0` to `c`
  - `: "r" (x) :` : The second `:` marks the input variables that will not be modified. This statement tells the compiler to load variable `x` into register `%1`
- To learn more, search the webs to find processor-specific instructions
  - But be warned that it is tricky

Another example: long integers

- How is 64-bit addition implemented on a 32-bit machine?
- Just as you learned adding numbers in primary school:
  - Add the low words and remember the carry
  - Add the high words and the carry
- Example: `add64.C`
  - `g++ -c -save-temps -O add64.C`
  - Look at `add64.s`
- Compare to a 64-bit machine
  - Addition done in one step!

128 bit integers in `int128.C`

- If we need 128 bit integers we need to define a new class:
  - Build a 128 bit integer from two 64 bit ones:

```
struct int128 {
  unsigned long long low;
  long long high;
};
```

- How do we add them?
  - Adding low and high words separately will not be correct since the carry is not used

```
int128 operator+(int128 x, int128 y) {
  int128 result;
  result.low=x.low+y.low;
  result.high=x.high+y.high;
  // wrong result: this does not use carry of previous addition
  return result;
}
```

  - Inline assembly language can be used to change "add without carry" to "add with carry"

A step in between: compiler intrinsics

- Many compilers for support a long list of intrinsic functions that look like functions but get mapped directly to assembly statements.
- Example:

```
int bitcount(unsigned int x) {
    int cnt=0;
    for (int i = 0 ; i<32 ;++i)
      if (x&1)
        ++cnt;
      x>>=1;
    return cnt;
}
```

- Can instead be called as just the intrinsic function
  - `_mm_popcnt(unsigned int x);`
- Will be discussed in detail in the High Performance Computing lectures

Helping the compiler optimize

- Using an optimizing compiler is easier than writing fast code in assembly language
- We will now discuss techniques to optimize code.
- Some can be done by the compiler
  - You need to know about them to realize which optimizations you do not need to perform
  - Not optimizing manually what the compiler can do for you can help keep the code cleaner
- Some have to be done by you
  - But only after you have determined by profiling that a function is the bottleneck

Optimization options

- Let the compiler optimize for you by adding command line options:
  - `-DNDEBUG` switch off all asserts and other tests in production code
  - `-O` turn on optimizations
  - `-O1` turn on some optimizations
  - `-O2` turn on more optimizations
  - `-O3` turn on even more optimizations
  - …
  - `-O5` turn on all optimizations
  - `-Os` optimize for size instead of speed
- Code may get slower when optimizing more!
- `-O3` and higher increases the risk of compiler bugs creating wrong code
- Options for debugging:
  - `-O0 -g` switch off all optimizations, for debugging and testing
  - `-Og` enable all optimizations that do not interfere with debugging (gcc >= 4.8)

Homework

- -O1 … –O5 are just shortcuts for many individual optimization techniques. Explore your compiler's manual to find out about all the individual techniques available and answer the following questions for you compiler:
  - From which level on, or with which option, are functions inlined when marked by "inline"?
  - From which level on, or with which option, may functions be inlined even when not marked by "inline"?
  - From which level on, or with which option, are loops unrolled?
  - From which level on, or with which option, are arithmetical inaccuracies accepted when compiling floating point code?
  - From which level on, or with which option does the compiler use all advanced features of your CPU model instead of creating generic code?

Copy propagation (automatic)

- is usually done by any modern compiler and need not be done by you.
- It changes

```
x = y;
z = 1 + x;
```

- to

```
x = y;
z = 1 + y;
```

- and allows pipelining of the two statements

Constant folding (automatic)

- Is also done by modern compilers and need not be done by you.
- It changes

```
const int x = 100;
int z = 2*x;
```

- to

```
const int x = 100;
int z = 200;
```

- And performs the multiplication at compile-time

Dead code removal (automatic)

- Is most useful in connection with template parameters. The compiler can detect if a statement is never executed
- It changes

```
int n = 100;
if (n<1)
  std::cerr << "n less than one";
…
```

- to

```
int n = 100;
…
```

- thus removing the code that will never be executed

Strength reduction (automatic)

- The compiler often realizes how to simplify expressions, making them faster
- It changes

```
x = 2 * y;
```

- to

```
x = y + y;
```

- or (for integer y)

```
x= ( y << 1 );
```

- And performs the faster operation

Variable renaming (automatic)

- Is also often done by the compiler to expose potentials for pipelining
- It changes

```
int x = y * z;
int q = r + x * x;
    x = a + b;
```

- to

```
int x0 = y * z;
int q = r + x0 * x0;
int x = a + b;
```

- And can now pipeline the last two statements

Common subexpression elimination (automatic)

- Can be done by the compiler in simple cases:
- It changes

```
d = c * (a + b);
e = (a + b) / 2;
```

- to

```
temp = (a + b);
d = c * temp
e = temp / 2;
```

- And saves one addition

Common subexpression elimination (manual)

- If a function call is involved you have to perform common subexpression elimination manually!
- You have to manually change

```
d = c * f(x);
e = f(x) / 2;
```

- to

```
temp = f(x);
d = c * temp
e = temp / 2;
```

- Since the compiler does not know whether `f(x)` is always the same number
  - maybe `f` is your name for a random number generator ….

Loop invariant code motion (automatic)

- Scientific programs spend most of their time in loops. We have to minimize the work done in those loops. A compiler can help in simple loops:
- It changes

```
for (int i=0; i<n; ++i) {
  a[i] = b[i] + c * d;
  e = g[k];
}
```

- to

```
temp = c * d;
for (int i=0; i<n; ++i) {
  a[i] = b[i] + temp;
}
e = g[k];
```

Loop invariant code motion (manual)

- In complex loops or if function calls are involved, we have to manually optimize
- We have to manually change

```
for (int i=0; i<n; ++i) {
  a[i] = b[i] + f(x);
  e = g(y);
}
```

- to

```
temp = f(x);
for (int i=0; i<n; ++i) {
  a[i] = b[i] + temp;
}
e = g(y);
```

Induction Variable Simplification (automatic / manual)

- Induction variable simplification is changing

```
for (int i=0; i<n; ++i) {
  k =  4*i + m;
  …
}
```

- to

```
k = m;
for (int i=0; i<n; ++i) {
  …
  k += 4;
}
```

Importance of Induction Variable Simplification

- Take care of hidden complexities in array subscripts: the code

```
for (int i=0; i<n; ++i) {
  x[4*i] = …
 }
```

- Is actually

```
for (int i=0; i<n; ++i) {
   *(x+4*i) = …
}
```

- And is faster coded as

```
for (T* p=x; p<x+4*n; p+=4) {
   *p = …
}
```

Loop unrolling (automatic / manual)

- The loop for a scalar product

```
double s=0.;
for (int i=0; i<3; ++i)
   s += x[i] * y[i];
```

- Is much faster when unrolled as

```
double s = x[0] * y[0] + x[1] * y[1] + x[2] * y[2];
```

- For two reasons:
  - No loop control statements
  - Easy pipelining
- Simple loops can be unrolled by compilers with high enough optimization settings (-funroll-loops on gcc)

Partial loop unrolling (automatic / manual)

- The loop for an array product

```
for (int i=0; i<N; ++i)
  a[i] = b[i] * c[i];
```

- Is much faster when partially unrolled as (for `N` a multiple of 4)

```
for (int i=0; i<N; i+=4) {
  a[i]   = b[i]   * c[i];
  a[i+1] = b[i+1] * c[i+1];
  a[i+2] = b[i+2] * c[i+2];
  a[i+3] = b[i+3] * c[i+3];
 }
```

- Because pipelining can again be used

Aiming for unit stride (manual)

- The loop for a matrix sum

```
for (int i=0; i<N; ++i)
  for (int j=0; j<N; ++j)
    a[i][j] = b[i][j] + c[i][j];
```

- Is much faster than

```
for (int i=0; i<N; ++i)
  for (int j=0; j<N; ++j)
    a[j][i] = b[j][i] + c[j][i];
```

- Because the unit stride (sequential memory access) in the inner loop uses the cache much better

In-cache matrix-matrix multiplications

- The matrix multiplication

```
for (int i=0; i<N; ++i)
  for (int j=0; j<N; ++j)
    for (int k=0; k<N; ++k)
      a[i][j] += b[i][k] * c[k][j];
```

- Is better changed to get unit stride in the inner loop

```
for (int i=0; i<N; ++i)
  for (int k=0; k<N; ++k) {
    temp = b[i][k];
    for (int j=0; j<N; ++j)
      a[i][j] += temp * c[k][j];
  }
```

Out-of-cache matrix multiplications: blocking

- Performance degrades if the matrix does not fit into the cache
- Split the matrix into smaller blocks and perform in-cache multiplications of the blocks:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

- The size of the blocks a_{ij}, b_{ij} and c_{ij} depends on the types and sizes of the caches.
- This is tricky and we will learn about libraries doing it for you next week

Libraries for linear algebra

- Fortran libraries
  - BLAS
  - LAPACK
- C++ libraries
  - Blitz++
  - uBlas
  - ITL and IETL
  - POOMA
- The Fortran libraries are well optimized but difficult to call
- The C++ libraries are easier to use but not as complete yet
- Fortran can also be called from C++, as we will do in one of the exercises

Calling Fortran from C++

- declare the function `extern "C"`
- pass all parameters by pointers or reference
- The naming depends on the machine
  - Fortran `FUNC` -> C `func_` with GNU or Intel compilers
  - Fortran `FUNC` -> C `func` with IBM or Cray compilers
- Program has to be linked with Fortran runtime libraries
- Take care of:
  - Fortran real is float on most workstations but double on Cray
  - Fortran integer is usually an int
  - Array indices in Fortran usually start from 1
  - Storage order of matrices is reversed
  - Fortran `a(i,j)` is C `a[j-1][i-1]`

A calling example: DDOT

- The DDOT function in the BLAS library calculates the scalar (dot) product of two double precision vectors:
  - ```
    DOUBLE PRECISION FUNCTION DDOT(N,X,INCX,Y,INCY)
    DOUBLE PRECISION X(*),Y(*)
    INTEGER INCX,INCY,N
    ```
- To call DDOT from C++ we need to declare it as:
  - ```
    extern "C" double ddot_(int& n, double *x, int& incx,
                            double *y, int& incy);
    ```
- To link we need to add the following options:
  - On the D-PHYS Linux machines: -lblas -lg2c -lm
  - On MacOS X: -framework vecLib
  - How to find options for other machines will be explained in the exercises

BLAS

- is short for Basic Linear Algebra Subroutines
- is a Fortran library
- BLAS level 1
  - vector operations: addition, dot product, …
- BLAS level 2
  - matrix-vector operations
- BLAS level 3
  - matrix-matrix operations
- use the BLAS wherever possible
  - optimized assembler code versions available on most machines
  - generic Fortran version available on www.netlib.org
- Homework: if you have a Unix or Linux machine at home download and install BLAS and LAPACK

ATLAS

- We learned in the last weeks that optimizing matrix operations can be tricky:
  - For which sizes should we use Strassen's algorithm?
  - How large should we choose sub-blocks to be get optimal cache effects by blocking?
- The Fortran BLAS on netlib works on all machines and thus cannot be optimized to the CPU, cache size and cache type of your machine
- On supercomputers the vendors provide a hand-optimized BLAS
- ATLAS is the solution for the rest of us:
  - A self-tuning library
  - When being installed it benchmarks hundreds of blocking strategies until it finds the optimal one for your machine
  - It then compiles a BLAS with these optimal settings

ATLAS benchmark example 1

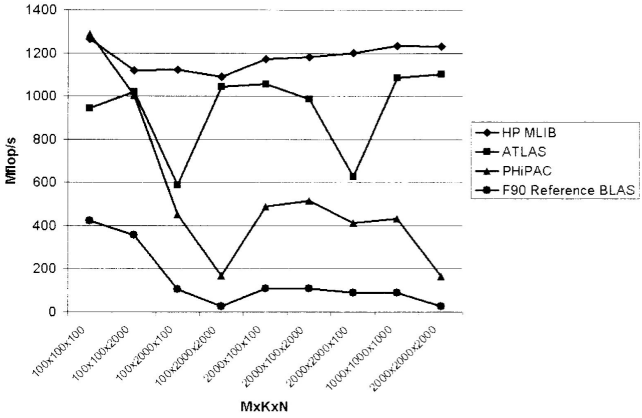


Figure 9-1 Comparison of matrix multiplication performance with various software packages.

ATLAS benchmark example 2

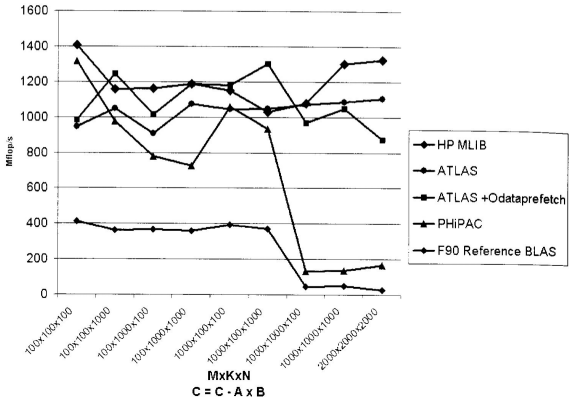


Figure 9-2 Comparison of transposed matrix multiplication performance using various software packages.

LAPACK Overview

- is a Linear Algebra PACKage
- ScaLAPACK is the parallel version
- has functions for
  - eigenvalues and -vectors
  - linear equation solvers
  - matrix inversions
  - determinants
  - ...
- special functions for
  - symmetric or Hermitian matrices
  - tridiagonal matrices
  - banded matrices

LAPACK & BLAS naming conventions

- functions are of the form
  - PTTXXX
- where P denotes the precision
  - **S** single precision real
  - **D** double precision real
  - **C** single precision complex
  - **Z** double precision complex
- TT denotes the matrix type:
  - **GE** general,
  - **SY** symmetric
  - **HE** Hermitian
  - **...**
- XXX denotes the operation to be performed
- Example: DGEEV is the double precision general eigensolver

Important LAPACK functions

- Eigensolvers: ***EV
  - we will use DSYEV or SSYEV for the exercises
- Linear equation solvers: ***SV
- Linear least squares: ***LS
- Factorizations:
  - LQ: ***LQF
  - QL: ***QLF
- Matrix inverse: ***TRI

FFTW

- The fastest open source Fourier transfrom library is the self-tuning FFTW ("Fastest Fourier Transform in the West")
- Available from http://www.fftw.org/

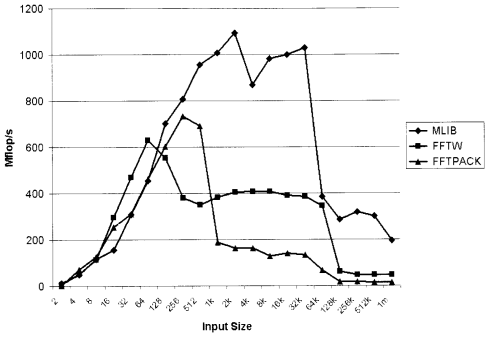


Figure 9-3 Comparison of one-dimensional, complex double precision FFT performance using various software packages.

Commercial libraries: NAG, IMSL, …

- add many more functions, like:
  - optimizations
  - non-linear root solvers
  - interpolation
  - statistical functions
  - …
- They are however not free but commercial libraries
  - cost a lot of money
  - not suitable for private use
  - ETH has a site license: you can us them in your research

To code or not to code? Part II

Week 12

Optimization in C++

C++ optimization

- The previous optimizations were for all languages
- Now we will discuss about C++-specific optimizations
  - Inlining (already known)
  - Template meta programs
  - Lazy Evaluation
  - Expression templates

What is a meta program?

- What is “meta”?
- Answer from “The Free On-line Dictionary of Computing”
  - A prefix meaning one level of description higher. If X is some concept then meta-X is data about, or processes operating on, X.
  - For example, a metasyntax is syntax for specifying syntax, metalanguage is a language used to discuss language, meta-data is data about data, and meta-reasoning is reasoning about reasoning.
- A meta program is a program writing programs

The C++ compiler : a Turing machine

- Erwin Unruh has shown that a C++ compiler with templates is a Turing machine
  - It can perform any arbitrary calculation that a classical computer can perform
  - The halting problem is undecidable: it is impossible to decide whether a C++ compiler will ever finish compiling a program
- In particular we can
  - Perform loops at compile time
  - Do branches at compile time
    - Using partial template specialization
- This is called "template meta programming"

Unruh's famous prime number program: `unruh.C`

- The following program prints all prime numbers as error messages

```
// Erwin Unruh, untitled program, ANSI X3J16-94-0075/ISO WG21-462, 1994.
template<int i> struct D { D(void*); operator int(); };

template<int p, int i> struct is_prime {
  enum { prim = (p%i) && is_prime<(i > 2 ? p : 0), i-1>::prim };
};

template<int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i,i-1>::prim };
  void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum { prim = 1 }; };
struct is_prime<0,1> { enum { prim = 1 }; };
struct Prime_print<2> {
   enum { prim = 1 };
   void f() { D<2> d = prim; }
};

void foo()
{ Prime_print<10> a; }
```

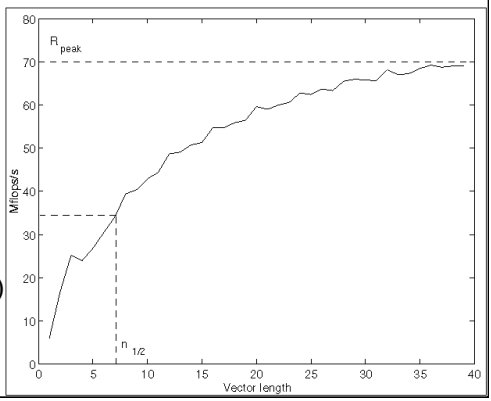
Performance bottleneck in dot products

- The dot product

```
double dot(const double* a, const double* b, int N)
{
  double result = 0.;
  for (int i=0; i < N; ++i)
    result += a[i] * b[i];
  return result;
}
```

does not reach peak performance on small vectors because the loop overhead is too large

(figure © by Todd Veldhuizen)



Unrolled loops reach peak performance

- The unrolled loop is optimal:

```
inline double dot3(const double* a, const double* b) {
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
}
```

- Question: how can we unroll for vectors of size N, where N is a template parameter such as in `tinyvector.h` ?

```
template<class T, int N>
class TinyVector {
public:
  T& operator[](int i) { return data[i]; }
  T operator[](int i) const { return data[i]; }

private:
  T data[N];
};
```

- Answer: template meta programming

An unrolled dot product by template meta programming

```
// The dot() function invokes meta_dot -- the metaprogram
template<class T, int N>
inline T dot(TinyVector<T,N>& a,
  TinyVector<T,N>& b)
{ return meta_dot<N-1>::f(a,b); }

// The metaprogram
template<int I>
struct meta_dot {
  template<class T, int N>
  static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
  { return a[I]*b[I] + meta_dot<I-1>::f(a,b); }
};

template<> // the end of the recursion
struct meta_dot<0> {
  template<class T, int N>
  static T f(TinyVector<T,N>& a, TinyVector<T,N>& b)
  { return a[0]*b[0]; }
};
```

How does this work?

- Here is what the compiler does with the code:

```
TinyVector<double,4> a, b;
double r = dot(a,b);

         = meta_dot<3>::f(a,b);

         = a[3]*b[3] + meta_dot<2>::f(a,b);

         = a[3]*b[3] + a[2]*b[2] + meta_dot<1>::f(a,b);

         = a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + meta_dot<0>::f(a,b);

         = a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + a[0]*b[0];
```

Operator overloading on a vector class: simplevector.h

```
template <class T>
class simplevector {
public:
  simplevector(int s=0);
  simplevector(const simplevector&
   v);
  ~simplevector()

  simplevector& operator=(const
   simplevector& v);
  simplevector operator +=(const
   simplevector& v);

  int size() const
  T operator[](std::size_t i) const;
  T& operator[](std::size_t i);

private:
  value_type* p_;
  size_type sz_;
};
```

- And add an `operator+`

```
template <class T>
simplevector<T> operator+(
   const simplevector<T>& x,
   const simplevector<T>& y )
{
  simplevector<T> result=x;
  result += y;
  return result;
}
```

Why operator overloading is slow: simplevector.h

- Problem: The expression

  ```
  A=B+C+D
  ```

- for `std::valarray` or for the `simplevector` class is evaluated as

  ```
  TEMP1 = B+C;
  TEMP2 = TEMP1+D;
  A     = TEMP2;
  ```

  needs two extra write and two extra read operations. Time the programs `timesimple.C` and `timesimple2.C` for a first benchmark.

- Expression templates, developed by Todd Veldhuizen solve this. The expression is evaluated just as:

  ```
  for (int j=0;j<A.size();++j)
    A[j] = B[j] + C[j] + D[j];
  ```

Lazy evaluation: lazyvector.h and timelazy2.C

- Consider

```
A = B + C;
```

- We define a vectorsum object:

```
template <class T>
class vectorsum {
public:
  vectorsum(const lazyvector<T>& x,
   const lazyvector<T>& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return left_[i] + right_[i]; }

private:
  const lazyvector<T>& left_;
  const lazyvector<T>& right_;
};
```

- operator+ just returns the vectorsum object describing the sum:

```
template <class T>
inline vectorsum<T> operator+(
   const lazyvector<T>& x,
   const lazyvector<T>& y)
{ return vectorsum<T>(x,y); }
```

- The evaluation is only done when assigning, there is no temporary

```
template <class T>
class lazyvector {
  …
  operator=(const vectorsum<T>& v) {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
  }
  …
};
```

Making it more flexible: lazyvectorflex.h

- We would need to write a class for each type of expression:
  - vectorsum
  - vectordifference
  - vectorproduct
  - …
- Instead let us define operation classes:

```
struct plus {
  template <class T> static inline T
    apply(T a, T b)
    { return a+b;}
};

struct minus {
 template <class T> static inline T
     apply(T a, T b)
     { return a-b;}
};
```

- template vectorsum on the operation:

```
template <class T, class Op>
class vectorsum {
public:
  vectorsum(const lazyvector<T>& x,
   const lazyvector<T>& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return Op::apply(
          left_[i],right_[i]);
  }

private:
  const lazyvector<T>& left_;
  const lazyvector<T>& right_;
};
```

Making it more flexible: lazyvectorflex.h

- Define operator+ and operator- by just this one class:

```
template <class T>
inline vectorsum<T,plus> operator+
   (
   const lazyvector<T>& x,
   const lazyvector<T>& y)
{ return vectorsum<T,plus>(x,y);

template <class T>
inline vectorsum<T,minus>
   operator-(
   const lazyvector<T>& x,
   const lazyvector<T>& y)
{ return vectorsum<T,minus>(x,y);
```

- And add the template also to the assignment:

```
template <class T>
class lazyvector {
  …
  template <class Op>
  operator=(const vectorsum<T,Op>& v)
   {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
   }
  …
};
```

Expression templates

- Lazy evaluation solves the problem for expressions with only two operands:
  - A = B + C
  - A = B - C
  - A = B * C
  - A = B / C
- How can we make this more general, to allow also expressions such as:
  - A = B + C + D
  - A = B * (C + D) + exp(E)
- The answer is: templates! Here called “expression templates”

Step 1: generalizing vectorsum into an expression class

- Let us replace

```
template <class T, class Op>
class vectorsum {
public:
  vectorsum(const lazyvector<T>& x,
   const lazyvector<T>& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return Op::apply(
          left_[i],right_[i]);
  }

private:
  const lazyvector<T>& left_;
  const lazyvector<T>& right_;
};
```

- By a more general class

```
template <class L, class Op, class R>
class X {
public:
  X(const L& x, const R& y)
  : left_(x), right_(y) {}

  T operator[](int i) const
  { return Op::apply(
          left_[i],right_[i]);
  }

private:
  const L& left_;
  const R& right_;
};
```

Step 2: generalizing the operators

- Replace

```
template <class T>
inline vectorsum<T,plus> operator+(const lazyvector<T>& x,
                                   const lazyvector<T>& y)
{
  return vectorsum<T,plus>(x,y);
}
```

- By the more general

```
template <class L, class T>
inline X<L,plus,etvector<T> > operator+(const L& x,
                                        const etvector<T>& y)
{
  return X<L,plus,etvector<T> >(x,y);
}
```

Step 3: generalizing the assignment

- Replace

```
template <class T>
class lazyvector {
  …
  template <class Op>
  operator=(const vectorsum<T,Op>& v) {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
   }
};
```

- by

```
template <class T>
class etvector {
  …
  template <class L, class Op, class R>
  operator=(const X<L,Op,R>& v) {
    for (int i=0;i<size();++i)
      p_[i]=v[i];
   }
};
```

How does this work

- First the compiler parses the expression:

```
D = A + B + C;
  = X<etvector,plus,etvector>(A,B) + C;
  = X<X<etvector,plus,etvector>,plus,etvector >
          (X<etvector,plus,etvector>(A,B),C);
```

- Then it matches the `operator=`:

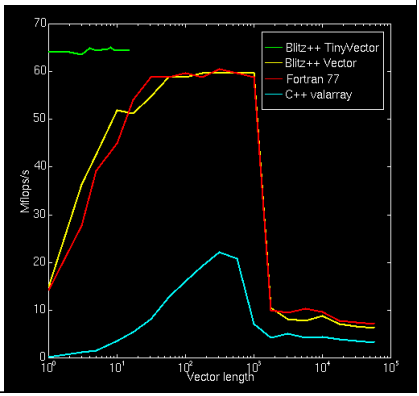
```
D.operator=(X<X<etvector,plus,etvector>,plus,etvector>
             (X< etvector,plus,etvector >(A,B),C) v) {
  for (int i=0; i < sz_; ++i)
    p_[i] = v[i];
}
```

- And for each index the expression is evaluated:

```
p_[i] = plus::apply(X<etvector,plus,etvector >(A,B)[i], C[i]);
      = plus::apply(A[i],B[i]) + C[i];
       = A[i] + B[i] + C[i];
```

Some old performance tests

- C++ with expression templates
  - Has more than 90% of the performance of Fortran 77
  - Is faster than Fortran 90/95 or C



C++ libraries

- Fortran libraries are well tested and optimized but hard to use
- C++ libraries are rare but are being developed more intensely now
  - Blitz++: expression templates for array operations and stencils
  - POOMA: parallel array operations and stencils
  - Boost uBlas: matrix library
  - ITL and IETL: iterative algorithms for linear systems
  - Boost Graph library: graph algorithms
  - POOMA: particle and field simulations
  - ALPS: simulations of classical and quantum lattice models
- see http://oonumerics.org/ and http://www.boost.org/ for an overview

Blitz++

- was developed by Todd Veldhuizen
- available from http://oonumerics.org/blitz/
- was the first application of expression templates

- contains optimized classes for
  - d-dimensional arrays:
    `template <class T, int D>  class Array<T,D>;`
    is a d-dimensional array
  - short vectors of fixed length
    `template <class T, int N> class TinyVector<T,N>;`
    is a short vector of length N
  - small matrices, ...
- supports mathematical expressions with arrays

Blitz++: TinyVector

- TinyVector uses template meta programs to achieve optimal performance

- Example: a ray reflection: $\vec{o} = \vec{i} - 2(\vec{i} \cdot \vec{n})\vec{n}$
  with $\vec{n} = (0,0,1)$
- can be coded in Blitz++ as:
  - ```
    TinyVector<double,3> i;
    i = …// set i
    TinyVector<double,3> n=0,0,1;
    TinyVector<double,3> o=i-2*sum(i*n)*n;
    ```
- This will give the *optimal* code:
  - ```
    o[0]=i[0];
    o[1]=i[1];
    o[2]=-i[2];
    ```

Blitz++: array expressions

- Consider the free energy and inner energy of a statistical system with energy levels energy[i]:

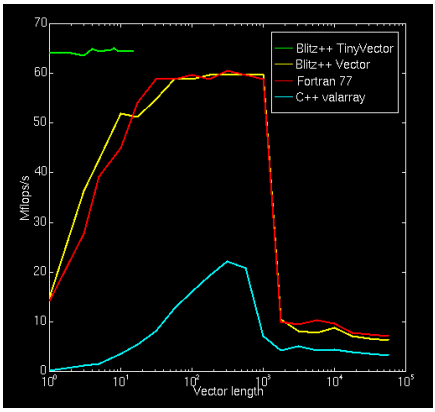
$$F = -T \log \sum_i \exp(-E_i / T)$$

$$U = \sum_i E_i \exp(-E_i / T) \Big/ \sum_i \exp(-E_i / T)$$

- Can be coded in Blitz++ as:
  - ```
    double F(const Array<double,1>& energy,double T) {
      return -T*log(sum(-exp(energy/T)));
    }
    ```
  - ```
    double U(const Array<double,1>& energy,double T)  {
      return sum(energy*exp(-energy/T)) /
             sum(exp(-energy/T));
    }
    ```
- That's easy, isn't it?

Blitz++: array benchmark

- Faster than simple-minded C++, about as fast as Fortran-77



Blitz++: Range objects

- A 1-D heat equation solver would be:
  - ```
    Array<double,1> oldT(N), newT(N);
    for (int iter=0; iter<niter; ++iter) {
      for (int i=1;i<N-1;++i)
        newT(i)=oldT(i)+c * (oldT(i-1)+oldT(i+1)-2*oldT(i));
      swap(oldT,newT);
    }
    ```
- Using a `Range` object the inner loop is simplified to:
  - ```
    Array<double,1> oldT(N), newT(N);
    Range I(1,N-2);
    for (int iter=0; iter<niter; ++iter) {
       newT(I)=oldT(I)+c * (oldT(I-1)+oldT(I+1)-2*oldT(I));
       swap(oldT,newT);
    }
    ```
- The Range expression `newT(I)= …` gets expanded into an optimzied loop

Stencils

- The expression `oldT(I-1)+oldT(I+1)-2*oldT(I)` is quite common as a second order derivative
- Blitz++ allows "stencils" to be coded to simplify the use of such expressions
- Example: heat equation in 2D:
  - ```
    Array<double,2> oldT(N,N), newT(N,N);
    Range I(1,N-2); // the interior region
     // create a view of the interior
    Array<double,2> oldI = oldT(I,I), newI = newT(I,I);
    for (int iter=0; iter<niter; ++iter) {
        // apply the stencil
        newI = oldI+c * Laplacian2D(oldI);
        swap(oldT,newT);
    }
    ```
- That's easy now!
    ```

Blitz++: benchmark of 3D wave propagation

- http://www.oonumerics.org/blitz/benchmarks/acou3d.html

	Time [in seconds]	Lines of code
Blitz++	200	9
Fortran 90	294	12
Fortran 77	168	18
Blitz++ (tuned)	123	8
Fortran 90 (tuned)	140	21
Fortran 77 (tuned)	120	27

POOMA: particles and fields

- was originally developed by the Advanced Computing Lab of the Los Alamos National Laboratories
- has classes for parallel simulations of fields and particles
- contains expression templates and stencils like Blitz++
- can be used for
  - partial differential equations (e.g. parallelized heat equation)
  - Particle-in-Cell codes for particle simulations
- available from http://www.nongnu.org/freepooma/

Parallel heat equation using POOMA II

```
Interval<2> I2(N,N); // 2D index domain NxN
Loc<2> blocks(2,2);  // domain decomposition into 2x2 blocks
UniformGridPartition<2> partition(blocks);
UniformGridLayout<2> layout(I2, UniformGridPartition<2>(blocks),
   DistributedTag());
//Create containers to store temperature
Array<2,double, MultiPatch<UniformTag,Remote<Brick> > >
  oldT(layout), newT(layout);

Interval<1> I(1, N-2);
Interval<1> J(1, N-2);

// initialize variables
…
// iterate
for (i=0;i<iter;++i) {
  newT(I,J) = oldT(I,J) + c*0.25*(oldT(I+1,J) + oldT(I-1,J) +
                                  oldT(I,J+1) + oldT(I,J-1));
  oldT = newT;
}
```

The Boost uBlas library

- available from http://www.boost.org/
- contains
  - 1-d vectors
  - 2-d matrices:
    - Fortran-style, C-style, arbitrary style layout
    - Sparse and dense matrix types
  - full BLAS functionality
  - Simple interface to BLAS and LAPACK under development
- Everything is in the namespace `boost::numeric::ublas`

Boost uBlas matrices

- Boost uBlas is very flexible and has classes for
  - Special matrices
    - Zero matrix
    - Identity matrix
  - Dense matrices
    - General
    - Symmetric
    - Hermitian
  - Sparse matrices
    - Banded
    - A number of sparse storage formats
- In addition you can choose, if you wish, the
  - Storage layout (Fortran or C-style)
  - Storage container

The Boost uBlas matrix class

- Is a class for dense matrices of arbitrary type T:
  - `template <class T, class F=row_major> class matrix;`
- The layout can optionally be changed from `row_major` to `column_major` (Fortran style) with the second argument
- A few useful functions, but slow numerical operations since it does not use BLAS:
  - `matrix<double> a(3,3);` // constructor for a 3x3 matrix
  - `a(1,2)=4.;` // sets matrix element $a_{23} = 4$
  - `std::cout << a;` // prints the matrix
  - `double* p = a.data();` // gets a pointer to the data for use in a BLAS call
  - `c=prod(a,b);` // calculates c=a*b
- Look at the documentation for more functions and examples

LAPACK wrappers

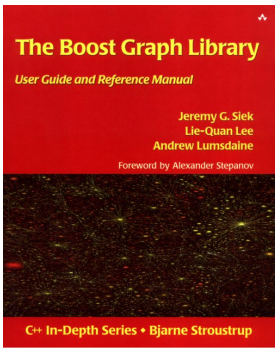
- The Boost uBlas contains dense matrices which have the Fortran data layout
- The uBlas wrapper project contains easier wrapper functions to some BLAS and LAPACK functions.
  - Your contributions to this open source project are appreciated
- uBlas wrapper:

```
template <class MatrixA, class MatrixB>
int gesv(MatrixA& a, MatrixB& b);
```

- Fortran LAPACK:
  - `void dgesv_(const int& n, const int& nrhs, double da[], const int& lda, int ipivot[], double db[], const int& ldb, int& info);`
  - `void sgesv_(const int& n, const int& nrhs, float da[], const int& lda, int ipivot[], float db[], const int& ldb, int& info);`
  - ...

Graph problems: The Boost Graph Library (BGL)

- The BGL is a generic library for graph problems, modeled after the standard template library
- Is a part of Boost, available from http://boost.org/
- A very nice manual exists as a book



ALPS: Monte Carlo simulations

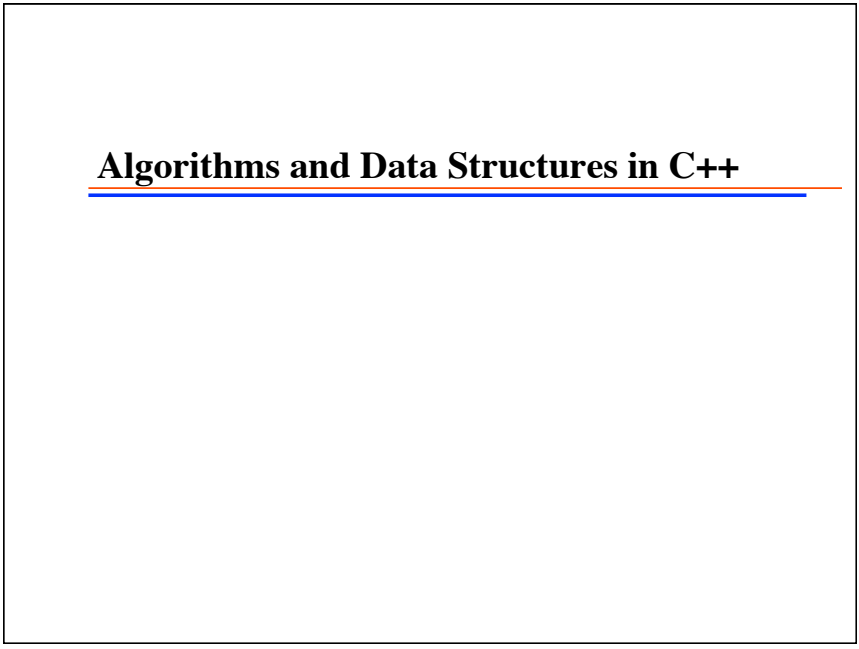
- is a library for the automatic parallelization of Monte Carlo simulations
- performs
  - parameter input
  - checkpoints and result files in hardware independent format
  - parallelization of MC simulations
  - load balancing
  - collection and evaluation of results
  - reliable error estimates
- develop and debug your code on a workstation
- link it on a parallel machine and it will run in parallel
- Available from http://alps.comp-phys.org/

Libraries you should install for your future work

- BLAS or ATLAS from http://netlib.org/
- LAPACK from http://netlib.org/
- Boost from http://boost.org/
- Blitz+ http://sourceforge.net/project/showfiles.php?group_id=63961
- FFTW from http://www.fftw.org/
- If you need a specific library, look at
  - Numerical libraries in Fortran or C: Netlib http://netlib.org/
  - Numerical libraries in C++: http://oonumerics.org/
  - General C++: Boost http://boost.org/

Chapter 12

An Introduction to Parallel Computing



Complexity analysis

- Answers the question "How does the time needed for an algorithm scale with the problem size *N*?"
  - Worst case analysis: maximum time needed over all possible inputs
  - Best case analysis: minimum time needed
  - Average case analysis: average time needed
  - Amortized analysis: average over a sequence of operations
- Usually only worst-case information is given since average case is much harder to estimate.

The O notation

- Is used for worst case analysis:

  An algorithm is $O(f(N))$ if there are constants c and N_0, such that for $N \geq N_0$ the time to perform the algorithm for an input size N is bounded by $t(N) < c\, f(N)$

- Consequences
  - $O(f(N))$ is identically the same as $O(a\, f(N))$
  - $O(a\, N^x + b\, N^y)$ is identically the same as $O(N^{\max(x,y)})$
  - $O(N^x)$ implies $O(N^y)$ for all $y \geq x$

Notations

- Ω is used for best case analysis:

  An algorithm is $\Omega(f(N))$ if there are constants c and N_0, such that for $N \geq N_0$ the time to perform the algorithm for an input size N is bounded by $t(N) > c\,f(N)$

- Θ is used if worst and best case scale the same

  An algorithm is $\Theta(f(N))$ if it is $\Omega(f(N))$ and $O(f(N))$

Time assuming 1 billion operations per second (1Gop)

Complexity	N=10	10^2	10^3	10^4	10^5	10^6
1	1 ns	1 ns	1 ns	1 ns	1 ns	1ns
ln N	3 ns	7 ns	10 ns	13 ns	17 ns	20 ns
N	10 ns	100 ns	1 μs	10 μs	100 μs	1 ms
N log N	33 ns	664 ns	10 μs	133 μs	1.7 ms	20 ms
N^2	100 ns	10 μs	1 ms	100 ms	10 s	17 min
N^3	1 μs	1 ms	1 s	17 min	11.5 d	31 a
2^N	1 μs	10^{14} a	10^{285} a	10^{2996} a	10^{30086} a	10^{301013} a

Time assuming 10 petaoperations per second (10 Pop/s)

Assume a parallel machine with 10 peta operations per second and perfect parallelization but one operation still needs at least 1ns

Complexity	N=10	10^2	10^3	**10^6**	**10^9**	**10^{12}**
1	1 ns	1 ns	1 ns	1 ns	1 ns	1 ns
ln N	1 ns	1 ns	1 ns	1 ns	1 ns	1 ns
N	1 ns	1 ns	1 ns	1ns	100 ns	100 μs
N log N	1 ns	1 ns	1 μs	1.33 ns	177 s	200 μs
N^2	1 ns	1 ns	1 ns	100 μs	100 s	3a
N^3	1 ns	1 ns	100 ns	100 s	3000 a	10^{12} a
2^N	1 ns	10^7 a	10^{278} a			

Which algorithm do you prefer?

- When do you pick algorithm A, when algorithm B? The complexities are listed below

Algorithm A	Algorithm B	Which do you pick?
O(ln N)	O(N)	
O(ln N)	N	
O(ln N)	1000 N	
ln N	O(N)	
1000 ln N	O(N)	
ln N	N	
ln N	1000 N	
1000 ln N	N	

Complexity: example 1

- What is the O, Ω and Θ complexity of the following code?

```
double x;
std::cin >> x;
std::cout << std::sqrt(x);
```

Complexity: example 2

- What is the O, Ω and Θ complexity of the following code?

```
unsigned int n;
std::cin >> n;
for (int i=0; i<n; ++i)
  std::cout << i*i << "\n";
```

Complexity: example 3

- What is the O, Ω and Θ complexity of the following code?

```
unsigned int n;
std::cin >> n;
for (int i=0; i<n; ++i) {
  unsigned int sum=0;
  for (int j=0; j<i; ++j)
    sum += j;
  std::cout << sum << "\n";
}
```

Complexity: example 4

- What is the O, Ω and Θ complexity of the following two segments?
  - Part 1:

```
unsigned int n;
std::cin >> n;
double* x=new double[n]; // allocate array of n numbers
for (int i=0; i<n; ++i)
  std::cin >> x[i];
```

  - Part 2:

```
double y;
std::cin >> y;
for (int i=0; i<n; ++i)
  if (x[i]==y) {
    std::cout << i << "\n";
    break;
  }
```

Complexity: adding to an array (simple way)

- What is the complexity of adding an element to the end of an array?
  - allocate a new array with N+1 entries
  - copy N old entries
  - delete old arrray
  - write (N+1)-st element
- The complexity is O(N)

Complexity: adding to an array (clever way)

- What is the complexity of adding an element to the end of an array?
  - allocate a new array with 2N entries, but mark only N+1 as used
  - copy N old entries
  - delete old arrray
  - write (N+1)-st element
- The complexity is O(N), but let's look at the next elements added:
  - mark one more element as used
  - write additional element
- The complexity here is O(1)
- The amortized (averaged) complexity for N elements added is

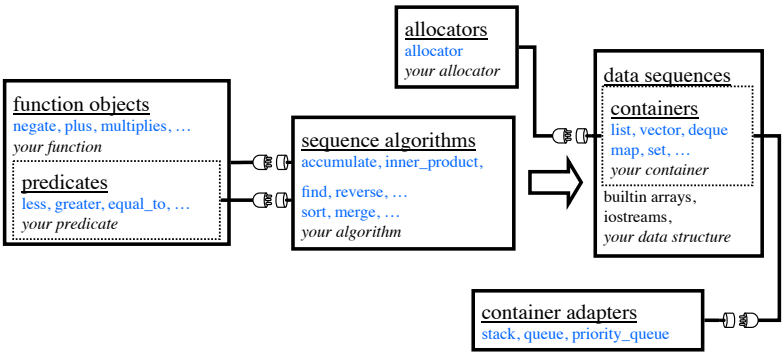
$$\frac{1}{N}\left(O(N)+(N-1)O(1)\right)=O(1)$$

STL: Standard Template Library

- Most notable example of generic programming
- Widely used in practice
- Theory: Stepanov, Musser; Implementation: Stepanov, Lee


- *Standard* Template Library
  - Proposed to the ANSI/ISO C++ Standards Committee in 1994.
  - After small revisions, part of the official C++ standard in 1997.

The standard C++ library



The `string` and `wstring` classes

- are very useful class to manipulate strings
  - `string` for standard ASCII strings (e.g. “English”)
  - `wstring` for wide character strings (e.g. “日本語”)
- Contains many useful functions for string manipulation
  - Adding strings
  - Counting and searching of characters
  - Finding substrings
  - Erasing substrings
  - …
- Since this is not very important for numerical simulations I will not go into details. Please read your C++ book

The `pair` template

```
template <class T1, class T2> class pair {
public:
  T1 first;
  T2 second;
  pair(const T1& f, const T2& s)
   : first(f), second(s)
  {}
};
```

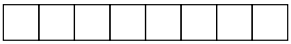
- will be useful in a number of places

Data structures in C++

- We will discuss a number of data structures and their implementation in C++:
- Arrays:
  - C array
  - `vector`
  - `valarray`
  - `deque`
- Linked lists:
  - `list`
- Trees
  - `map`
  - `set`
  - `multimap`
  - `multiset`
- Queues and stacks
  - `queue`
  - `priority_queue`
  - `stack`

The array or vector data structure

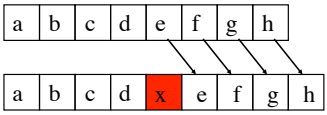
- An array/vector is a consecutive range in memory



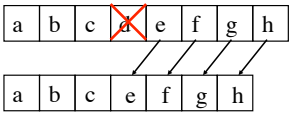
- Advantages
  - Fast $O(1)$ access to arbitrary elements: `a[i]` is `*(a+i)`
  - Profits from cache effects
  - Insertion or removal at the end is $O(1)$
  - Searching in a sorted array is $O(\ln N)$
- Disadvantage
  - Insertion and removal at arbitrary positions is $O(N)$

Slow O(*N*) insertion and removal in an array

- Inserting an element
  - Need to copy O(N) elements

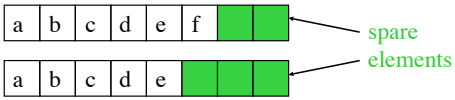


- Removing an element
  - Also need to copy O(N) elements

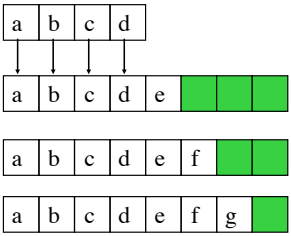


Fast O(1) removal and insertion at the end of an array

- Removing the last element
  - Just change the size
    - Capacity 8, size 6:
    - Capacity 8, size 5:



- Inserting elements at the end
  - Is amortized O(1)
    - first double the size and copy in O(N):
    - then just change the size:

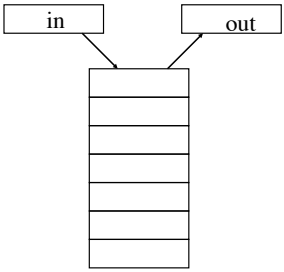


The deque data structure (double ended queue)

- Is a variant of an array, more complicated to implement
  - See a data structures book for details
- In addition to the array operations also the insertion and removal at beginning is $O(1)$
- Is needed to implement queues

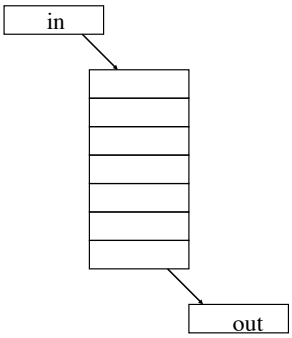
The stack data structure

- Is like a pile of books
  - LIFO (last in first out): the last one in is the first one out
- Allows in $O(1)$
  - Pushing an element to the top of the stack
  - Accessing the top-most element
  - Removing the top-most element



The queue data structure

- Is like a queue in the Mensa
  - FIFO (first in first out): the first one in is the first one out
- Allows in $O(1)$
  - Pushing an element to the end of the queue
  - Accessing the first and last element
  - Removing the first element

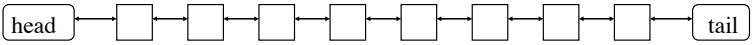


The priority queue data structure

- Is like a queue in the Mensa, but professors are allowed to go to the head of the queue (not passing other professors though)
  - The element with highest priority (as given by the < relation) is the first one out
  - If there are elements with equal priority, the first one in the queue is the first one out
- There are a number of possible implementations, look at a data structure book for details

The linked list data structure

- An linked list is a collection of objects linked by pointers into a one-dimensional sequence



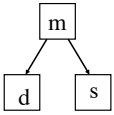
- Advantages
  - Fast $O(1)$ insertion and removal anywhere
    - Just reconnect the pointers
- Disadvantage
  - Does not profit from cache effects
  - Access to an arbitrary element is $O(N)$
  - Searching in a list is $O(N)$

The tree data structures

- An array needs
  - $O(N)$ operations for arbitrary insertions and removals
  - $O(1)$ operations for random access
  - $O(N)$ operations for searches
  - $O(\ln N)$ operations for searches in a sorted array
- A list needs
  - $O(1)$ operations for arbitrary insertions and removals
  - $O(N)$ operations for random access and searches
- What if both need to be fast? Use a tree data structure:
  - $O(\ln N)$ operations for arbitrary insertions and removals
  - $O(\ln N)$ operations for random access and searches

A node in a binary tree

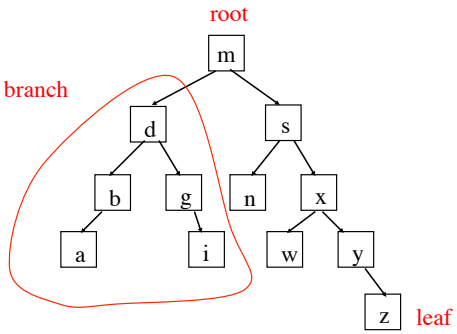
- Each node is always linked to two child nodes
  - The left child is always smaller
  - The right child node is always larger



A binary tree

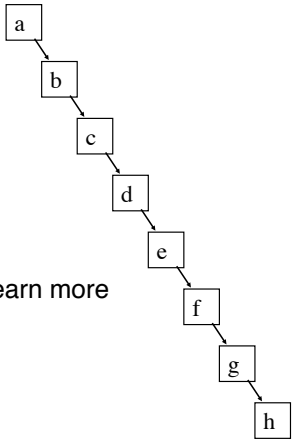
- Can store $N=2^n-1$ nodes in a tree of height n
- Any access needs at most $n = O(\ln N)$ steps

- Example: a tree of height 5 with 12 nodes



Unbalanced trees

- Trees can become unbalanced
  - Height is no longer O(ln N) but O(N)
  - All operations become O(N)
- Solutions
  - Rebalance the tree
  - Use self-balancing trees
- Look into a data structures book to learn more



Tree data structures in the C++ standard

- Fortunately the C++ standard contains a number of self-balancing tree data structures suitable for most purposes:
  - `set`
  - `multiset`
  - `map`
  - `multimap`
- But be aware that computer scientists know a large number of other types of trees and data structures
  - Read the books
  - Ask the experts

The container concept in the C++ standard

- Containers are sequences of data, in any of the data structures
  - `vector<T>` is an array of elements of type T
  - `list<T>` is a doubly linked list of elements of type T
  - `set<T>` is a tree of elements of type T
  - …
- The standard assumes the following requirements for the element T of a container:
  - default constructor `T()`
  - assignment `T& operator=(const T&)`
  - copy constructor `T(const T&)`
  - Note once again that assignment and copy have to produce identical copy: in the Penna model the copy constructor should not mutate!

Connecting Algorithms to Sequences

```
find( s, x ) :=
    pos ← start of s
    while pos not at end of s
        if element at pos in s == x
            return pos
        pos ← next position
    return pos
```

```
int find( char const(&s)[4], char x )
{
    int pos = 0;
    while (pos != sizeof(s))
    {
        if ( s[pos] == x )
            return pos;
        ++pos;
    }
    return pos;
}
```

```
struct node
{
    char value;
    node* next;
};
```

```
node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

Connecting Algorithms to Sequences

```
find( s, x ) :=
    pos ← start of s
    while pos not at end of s
        if element at pos in s == x
            return pos
        pos ← next position
    return pos
```

```
char* find(char const(&s)[4], char x)
{
    char* pos = s;
    while (pos != s + sizeof(s))
    {
        if ( *pos == x )
            return pos;
        ++pos;
    }
    return pos;
}
```

```
struct node
{
    char value;
    node* next;
};
```

```
node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

Connecting Algorithms to Sequences

```
find( s, x ) :=
    pos ← start of s
    while pos not at end of s
        if element at pos in s == x
            return pos
        pos ← next position
    return pos
```

```
char* find(char const(&s)[4], char x)
{
    char* pos = s;
    while (pos != s + sizeof(s))
    {
        if ( *pos == x )
            return pos;
        ++pos;
    }
    return pos;
}
```

```
struct node
{
    char value;
    node* next;
};
```

```
node* find( node* const s, char x )
{
    node* pos = s;
    while (pos != 0)
    {
        if ( pos->value == x )
            return pos;
        pos = pos->next;
    }
    return pos;
}
```

F. T. S. E.

Fundamental Theorem of Software Engineering

"We can solve any problem by introducing an extra level of indirection"

--Butler Lampson

Andrew Koenig

Iterators to the Rescue

- Define a common interface for
  - traversal
  - access
  - positional comparison
- Containers provide iterators
- Algorithms operate on pairs of iterators

```
template <class Iter, class T>
Iter find( Iter start, Iter finish, T x )
{
   Iter pos = start;
   for (; pos != finish; ++pos)
   {
      if ( *pos == x )
         return pos;
   }
   return pos;
}
```

```
struct node_iterator
{
   // ...
   char& operator*() const
   { return n->value; }

   node_iterator& operator++()
   { n = n->next; return *this; }
private:
   node* n;
};
```

Describe Concepts for std::find

```
template <class Iter, class T>
Iter find(Iter start, Iter finish, T x)
{
    Iter pos = start;
    for (; pos != finish; ++pos)
    {
        if ( *pos == x )
            return pos;
    }
    return pos;
}
```

- Concept Name?
- Valid expressions?
- Preconditions?
- Postconditions?
- Complexity guarantees?
- Associated types?

Traversing an array and a linked list

- Two ways for traversing an array
  - Using an index:

```
T* a = new T[size];
for (int n=0;n<size;++n)
  cout << a[n];
```

  - Using pointers:

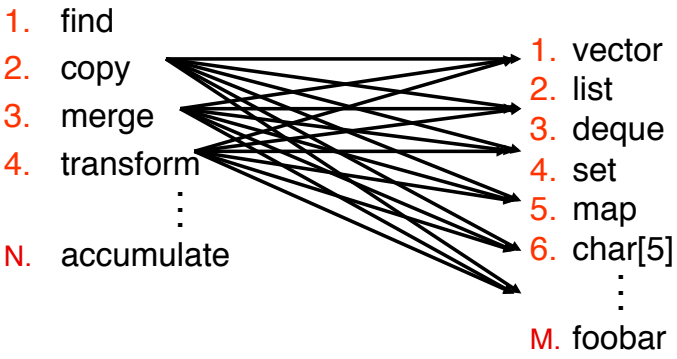
```
for (T* p = a;
     p !=a+size;
     ++p)
  cout << *p;
```

- Traversing a linked list

```
template <class T> struct node
{
  T value; // the element
  node<T>* next; // the next Node
};

template<class T> struct list
{
  node<T>* first;
};
list<T> l;
…
for (mode<T>* p=l.first;
       p!=0;
       p=p->next)
    cout << p->value;
```

NxM Algorithm Implementations?



Generic traversal

- Can we traverse a vector and a list in the same way?

- Instead of

```
for (T* p = a;
     p !=a+size;
     ++p)
  cout << *p;
```

- We want to write

```
for (iterator p = a.begin();
     p !=a.end();
     ++p)
  cout << *p;
```

- Instead of

```
for (node<T>* p=l.first;
     p!=0;
     p=p->next)
  cout << p->value;
```

- We want to write

```
for (iterator p = l.begin();
     p !=l.end();
     ++p)
  cout << *p;
```

Implementing iterators for the array

```
template<class T>
  class Array {
  public:
    typedef T* iterator;
    typedef unsigned size_type;
    Array();
    Array(size_type);

    iterator begin()
    { return p_;}
    iterator end()
    { return p_+sz_;}

  private:
    T* p_;
    size_type sz_;
  };
```

- Now allows the desired syntax:

```
for (Array<T>::iterator p =
  a.begin();
     p !=a.end();
     ++p)
  cout << *p;
```

- Instead of

```
for (T* p = a.p_;
     p !=a.p_+a.sz_;
     ++p)
  cout << *p;
```

Implementing iterators for the linked list

```
template <class T>
  struct node_iterator {
  Node<T>* p;
  node_iterator(Node<T>* q)
   : p(q) {}

  node_iterator<T>& operator++()
  { p=p->next;}

  T* operator ->()
  { return &(p->value);}

  T& operator*()
  { return p->value;}

  bool operator!=(const
        node_iterator<T>& x)
  { return p!=x.p;}

  // more operators missing ...
  };
```

```
template<class T>
  class list {
   Node<T>* first;
  public:
    typedef node_iterator<T> iterator;

    iterator begin()
    { return iterator(first);}

    iterator end()
    { return iterator(0);}
  };
```

- Now also allows the desired syntax:

```
for (List<T>::iterator p = l.begin();
     p !=l.end();
     ++p)
  cout << *p;
```

Iterators

- have the same functionality as pointers
- including pointer arithmetic!
  - `iterator a,b; cout << b-a;` // # of elements in [a,b[
- exist in several versions
  - forward iterators … move forward through sequence
  - backward iterators … move backwards through sequence
  - bidirectional iterators … can move any direction
  - input iterators … can be read: `x=*p;`
  - output iterators … can be written: `*p=x;`
- and all these in const versions (except output iterators)

Container requirements

- There are a number of requirements on a container that we will now discuss based on the handouts

Containers and sequences

- A container is a collection of elements in a data structure
- A sequence is a container with a linear ordering (not a tree)
  - vector
  - deque
  - list
- An associative container is based on a tree, finds element by a key
  - map
  - multimap
  - set
  - multiset
- The properties are defined on the handouts from the standard
  - A few special points mentioned on the slides

Sequence constructors

- A sequence is a linear container (vector, deque, list,…)
- Constructors
  - `container()` … empty container
  - `container(n)` … n elements with default value
  - `container(n,x)` … n elements with value x
  - `container(c)` … copy of container c
  - `container(first,last)` … first and last are iterators
    - container with elements from the range [first,last[
- Example:

```
std::list<double> l;
// fill the list
...
// copy list to a vector
std::vector<double> v(l.begin(),l.end());
```

Direct element access in deque and vector

- Optional element access (not implemented for all containers)
  - `T& container[k]` … k-th element, no range check
  - `T& container.at(k)` … k-th element, with range check
  - `T& container.front()` … first element
  - `T& container.back()` … last element

Inserting and removing at the beginning and end

- For all sequences: inserting/removing at end
  - `container.push_back(T x)` // add another element at end
  - `container.pop_back()` // remove last element
- For list and deque (stack, queue)
  - `container.push_front(T x)` // insert element at start
  - `container.pop_front()` // remove first element

Inserting and erasing anywhere in a sequence

- List operations (slow for vectors, deque etc.!)
  - `insert (p,x)` // insert x before p
  - `insert(p,n,x)` // insert n copies of x before p
  - `insert(p,first,last)` // insert [first,last[before p
  - `erase(p)` // erase element at p
  - `erase(first,last)` // erase range[first,last[
  - `clear()` // erase all

Vector specific operations

- Changing the size
  - `void resize(size_type)`
  - `void reserve(size_type)`
  - `size_type capacity()`
- Note:
  - `reserve` and `capacity` regard memory `allocated` for vector!
  - `resize` and `size` regard memory currently used for vector data
- Assignments
  - `container = c` … copy of container c
  - `container.assign(n)` …assign n elements the default value
  - `container.assign(n,x)` … assign n elements the value x
  - `container.assign(first,last)` … assign values from the range [first,last[
- Watch out: assignment does not allocate, do a resize before!

The `valarray` template

- acts like a vector but with additional (mis)features:
  - No iterators
  - No reserve
  - Resize is fast but erases contents
- Many numeric operations are defined:

```
std::valarray<double> x(100), y(100), z(100);
x=y+exp(z);
```

  - Be careful: it is not the fastest library!
  - We will learn about faster libraries later

Sequence adapters: `queue` and `stack`

- are based on deques, but can also use vectors and lists
  - `stack` is first in-last out
  - `queue` is first in-first out
  - `priority_queue` prioritizes with < operator
- stack functions
  - `void push(const T& x)` … insert at top
  - `void pop()` … removes top
  - `T& top()`
  - `const T& top() const`
- queue functions
  - `void push(const T& x)` … inserts at end
  - `void pop()` … removes front
  - `T& front()`, `T& back()`, `const T& front()`, `const T& back()`

Index of C++

Index of theory