JÜLICH FORSCHUNGSZENTRUM

Programming in C++(C++11) for C programmers

2–4 and 10–12 June 2014 | Sandipan Mohanty (s.mohanty@fz-juelich.de) Jülich Supercomputing Centre

JÜLICH FORSCHUNGSZENTRUM

Day 1

Course plan

In this course you will ...

- learn the C++ syntax according to the latest standard
- learn to use the C++ standard library
- get a brief introduction to
  - Intel (R) **T**hread **B**uilding **B**locks (TBB)
  - graphical user interface design with Qt

Resources

Books

- The C++ Programming Language(Fourth Edition), Bjarne Stroustrup, ISBN: 978-0321563842
- The C++ Standard Library: A Tutorial and Reference, Nicolai M. Josuttis, **ISBN** 978-0-321-62321-8
- C++11: Der Leitfaden für Programmierer zum neuen Standard, Rainer Grimm, **ISBN** 978-3-8273-3088-8
- Structured Parallel Programming, Michael McCool, Arch D. Robinson, James Reinders, **ISBN** 978-0-12-415993-8
- C++ GUI Programming with Qt4, Jasmin Blanchette and Mark Summerfield, **ISBN** 978-0132354165

Resources

There are also good online resources

- `http://www.cplusplus.com/reference/`
- `http://en.cppreference.com/`
- `http://www.qt-project.org`
- `http://www.threadingbuildingblocks.org`

Compiler support for C++11 (June 2014)

- **LLVM/clang++** : Feature complete (Version 3.4). Useful error messages. Fast compilation. Generated binaries competitive with others. Own STL : libc++.
- **GCC/g++** : Complete (Version 4.9). Mature code base. Own STL : libstdc++.
- **Intel Compiler** : Incomplete but improving. Great optimizer. Uses system STL. Not free.
- **IBM XLC Compiler** : Not quite usable with C++11 yet. Not free.

Getting started

The first step: Hello World!

- Log in to your work station and to `judge.fz-juelich.de`
- Set up environment for g++ 4.9 by typing `module load gcc/4.9.0`
- Find your favourite text editor and type in the simple "hello world" program
- Compile and run the program using the following commands :

```
g++ helloworld.cc -o hello
./hello
```

```
// Hello World!
#include <iostream>

using namespace std;

int main()
{
    cout<<"Hello, world!\n";
}
```


We are going to use thin wrappers `G` and `A` on compilers `g++` and `clang++`. The following are defined at `/homeb/zam/sandipan/C++2014/bin`

- `G` is a shorthand for

```
g++ --std=c++11 -pedantic \
-I/homeb/zam/sandipan/local/gcc/4.9.0/include/c++/4.9.0 \
-I/homeb/zam/sandipan/local/gcc/4.9.0/include \
-L/homeb/zam/sandipan/local/gcc/4.9.0/lib64 \
-L/homeb/zam/sandipan/local/gcc/4.9.0/lib
```

- `A` is a shorthand for

```
clang++ -std=c++11 -stdlib=libc++ -pedantic \
-I/homeb/zam/sandipan/local/llvm/3.4.1/include/c++/v1 \
-I/homeb/zam/sandipan/local/llvm/3.4.1/include \
-L/homeb/zam/sandipan/local/llvm/3.4.1/lib
```

Whichever compiler you choose to use, load the corresponding module : `module load gcc/4.9.0` or `module load llvm/3.4.1`

Getting started

Comments on the hello world program

- Standard include files normally don't have any extensions (e.g. `iostream`)
- Think of `cout` as some sort of sink into which we are sending data
- A **namespace** is a context for identifiers (next slide)
- You can omit the **return** `0` at the end of `main()` in C++

```
// Hello World
#include <iostream>

using namespace std;

int main()
{
    cout<<"Hello, world!\n";
}
```


C++ namespaces

```
// examples/namespaces.cc
#include <iostream>
using namespace std;
namespace UnitedKingdom
{
  string London{"Big city"};
}
namespace UnitedStates
{
  string London{"Small town in Kentucky"};
}
int main()
{
  using namespace UnitedKingdom;
  cout<<London<<'\n';
  cout<<UnitedStates::London<<'\n';
}
```

- There is no name conflict in having two variables called `London` in two different **namespaces**
- The symbol `::` is called the scope resolution operator. You will see that often!
- **using namespace** `blah` imports all names declared inside the **namespace** `blah`.

C++ namespaces

```
//examples/namespaces2.cc
#include <iostream>
namespace UnitedKingdom
{
  std::string London{"Big city"};
}
namespace UnitedStates
{
  namespace KY {
    std::string London{" in Kentucky"};
  }
  namespace OH {
    std::string London{" in Ohio"};
  }
}
int main()
{
  namespace USOH=UnitedStates::OH;
  std::cout<<"London is"
           <<USOH::London<<'\n';
}
```

- Note how **namespaces** can be nested
- Note how long **namespace** names can be given aliases
- Tip: Use **namespaces** to organise your code
- Tip2: Don't indiscreminately put **using namespace ...** tags, especially in header files.

C++ comments

```
int mesh::get_neighbours(int ipart)
{
    // Get neighbours for particle ipart
    int j=cell_of(ipart); //Locate ipart
    // if (debug) check(j); // uncomment to debug
}
```

- // starts a comment that runs till the end of that line
- // comments can be nested
- /* ... */ comments can also be used.

Declare variables where you need them

```
double find_root()
{
  if (not initialized()) init_arrays();
  for (int i=0;i<N;++i) {
    //stuff
  }
  double newval=0; // This is ok.
  for (int i=0;i<N;++i) {
    if (newval < 5) {
        string fl{"small.dat"};
        // do something
    }
    newval=...;
    cout << fl << '\n'; // Error!
  }
}
```

- Variable declarations are allowed throughout the code
- Place them just before you need them.
- Be mindful of the scope of the variables

The C++11 uniform initialisation syntax

```
  int I{20};
  // define integer I and set it to 20
  string nat{"Germany"};
  // define and initialise a string
  double a[4]{1.,22.1,19.3,14.1};
  // arrays have the same syntax
  tuple<int,int,double> x{0,0,3.14};
  // So do tuples
  list<string> L{"abc","def","ghi"};
  // and lists, and ...
  double m=0.5;
  // this is also fine for simple
  // variables
  int i{}; // i=0
  int j{5.0}; // Error! Narrowing!
  int k=5.3; // OK. But k=5!
```

- Variables can be initialised when declared with an appropriate value enclosed in `{}`
- Pre-C++11, parantheses `()` were used in some cases. Initialising non trivial collections was not allowed.
- Use this `{}` initialisation syntax when possible. Exceptions will be pointed out when appropriate

Use auto and decltype when possible

```
int sqr(int x)
{
  return x*x;
}
int main()
{
  char oldchoice,choice='y';
  int i=20;
  double electron_mass = 0.511;
  int mes[6]{33,22,34,0,89,3};
  bool flag = true;
  double *p = nullptr;
  decltype(i) j=9;
  auto positron_mass = electron_mass;
  auto f = sqr;
  //Sure, we could have written...
  //int (*f)(int)=&sqr;
  //but you see the point!
  std::cout << f(j) << '\n';
}
```

- If a variable is initialised when it is declared, in such a way that its type is unambiguous, the keyword **auto** can be used to declare its type
- The keyword `decltype` can be used to say "same type as that one"

Example 1.1:

The programs `examples/vardecl.cc` and `examples/vardecl2.cc` demonstrate the use of **auto** and `decltype`. Understand what these keywords are for. Use

```
G program.cc -o program
```

to compile.

C++ standard library strings

Character strings

- String of characters
- Knows its size (see example)
- Allocates and frees memory as needed
- No need to worry about \0
- Can contain \0 in the middle
- Simple syntax for assignment (=), concatenation(+), comparison (<, ==, >)

```
#include <string>
...
std::string fullname;
std::string name{"Albert"};
std::string surname{"Einstein"};

//Concatenation and assignment
fullname=name+" "+surname;

//Comparison
if (name=="Godzilla") run();

std::cout<<fullname<<'\n';

for (size_t i=0;i<fullname.size();++i) {
  if (fullname[i]>'j') blah+=fullname[i];
}
```

Don't use C style strings!

Raw string literals

```
// Instead of ...
 string message{"The tag \"\\maketitle\" is unexpected here."};
// You can write ...
 string message{R"(The tag "\maketitle" is unexpected here.)"};
```

- Can contain line breaks, '\' characters without escaping them
- Very useful with regular expressions
- Starts with `R"(` and ends with `)"`
- More general form `R"delim(` text `)delim"`

Example 1.2:

The file `examples/rawstring.cc` illustrates raw strings in use.

Exercise 1.1:

The file `exercises/raw1.cc` has a small program printing a message about using the continuation character '\' at the end of the line to continue the input. Modify using raw string literals.

Converting to and from strings

```
std::cout << "integer : "<<std::to_string(i) << '\n';
tot+=std::stod(line); // String-to-double
```

- The standard library `string` class provides functions to inter-convert with variables of type **`int`**, **`double`**
- Akin to `atoi`,`strtod` and using `sprintf`

Example 1.3:

Test example usage of string↔number conversions in `examples/to_string.cc` and `examples/stoX.cc`

Range based for loops

"Syntactic sugar" from C++11

```
// Instead of ...
//for (size_t i=0;i<fullname.size();++i) {
//   if (fullname[i]>'j') blah+=fullname[i];
//}
// you could write ...
for (auto c : fullname) if (c>'j') blah+=c;

// Loop over a linked list ...
std::list<double> L{0.5,0.633,0.389,0.34,0.01};
for (auto d : L ) {
  std::cout << d << '\n';
}
```

Anything that has a begin and an end

- Iteration over elements of a collection
- Use on strings, arrays, STL lists, maps ...

Range based for loops

"Syntactic sugar" from C++11

```
// Loop over a small list of names ...
for (auto day : { "Monday", "Tuesday", "Wednesday","Thursday","Friday"}) {
  std::cout << day << '\n';
}
// or a list of non contiguous integers ...
for (auto i : { 1,1,2,3,5,8,13,21 }) {
  std::cout << Recs[i] << '\n';
}
```

Anything that has a begin and an end

- collections which provide a `begin()` and `end()` functions
- ... or which work well with global `begin()` and `end()` functions

Exercise 1.2:

You will find two tiny example programs `loop.cc` and `loop2.cc` to demonstrate the range based for loops. Modify the code to write out an array of strings.

Input and output

std::cout and std::cin

- To read user input into variable `x`, simply write `std::cin>>x;`
- To read into variables `x`,`y`,`z`,`name` and `count`

```
std::cin >> x >> y >> z >> name >> count;
```

`std::cin` will infer the type of input from the type of variable being read.

- For printing things on screen the direction for the arrows is towards `std::cout`:

```
std::cout << x << y << z << name << count << '\n';
```

Reading and writing files

- Include `fstream`
- Declare your own source/sink objects, which will have properties like `std::cout` or `std::cin`

```
#include <fstream>
...
  std::ifstream fin;
  std::ofstream fout;
```

- Connect them to file names

```
  fin.open("inputfile");
  fout.open("outputfile");
```

- Use them like `std::cout` or `std::cin`

```
  double x,y,z;
  int i;
  std::string s;
  fin >> x >> y >> z >> i >> s;
  fout << x << y << z << i << s << '\n';
```


Exercise 1.3: Strings and I/O

Write a simple program to find the largest word in a plain text document.

Memory allocation/deallocation

- You don't need it often:
  - `std::string` takes care of itself
  - Using standard library containers like `vector`, `list`, `map`, `deque` even rather complicated structures can be created without explicit memory allocation and de-allocation.
- When you nevertheless must:

```
complex_number *c = new complex_number{1.2,4.2};
int asize=100;
double *my_array = new double[asize];
...
delete c;
delete [] my_array;
```


inline functions

```
double sqr(double x)
{
  return x*x;
}
```

```
inline double sqr(double x)
{
  return x*x;
}
```

When function call overhead matters

- To eliminate overhead when a function is called, request the compiler to insert the entire function body where it is called
- Very small functions which are called very frequently
- Only a request to the compiler!

Exercise 1.4:

Why not simply use macros rather than bothering with inline functions ? Check the program `exercises/inlining.cc` and compare the inlined version of `sqr` and the corresponding macro.

The type bool

```
bool debugging=false;
...
if (debugging) {
  std::cout<<"additional messages.\n";
}
```

- Possible values **`true`** or **`false`**
- Can be converted back and forth from integer types

The type qualifier const

```
const double pi=3.141592653589793;
const unsigned int n_3d_ngb=26;

int cell[n_3d_ngb];

const double BL=optimal_length(); // OK
BL=0.8*BL; //Compiler error
```

- Can not be changed after initialisation
- Compilers and debuggers know their type, unlike in the case of macros
- They can be used as the size of static arrays, if known at compile time

The constexpr keyword

```
constexpr double b=13.2;
constexpr double r=1.08;
constexpr double a=6*r*r*r*r-5*r*2*b;
constexpr unsigned fact(unsigned N)
{
  return N<2?1:N*fact(N-1);
}
int f()
{
  int indexes[fact(4)];
```

- **constexpr** is used to declare that something is possible to evaluate at compile time
- Compiler can optimize more, because of compile time evaluations
- Non-trivial calculations can be done at compile time using **constexpr**
- Things of type **constexpr** can be array sizes

Example 1.4: constexpr

The program `examples/constexpr0.cc` demostrates the use of constexpr functions. How would you verify that it is really evaluated at compile time ?

Enumerations

- C++11 supports scoped enumerators called **`enum class`**
- Must always be fully qualified when used: `traffic_light::red` etc.
- No automatic conversion to int.
- Plain enumerations **`enum`** are also supported

```
enum class color { red, green, blue };
enum class traffic_light {
  red,yellow,green
};
bool should_brake(traffic_light c);

if (should_brake(blue)) apply_brakes();
//Syntax error!
if (state==traffic_light::yellow) ...;
```

The reference type

Motivation

- Return more than one value from a function
- Pass big bulky data structures to functions without copying

```
void get_bounds(int i1, int i2,
                data_type data)
{
  //calculations
  i1=45;
  i2=103;
}
...
int setup_plot()
{
  int r0=0,r1=0;
  data_type mydata;
  read_data(mydata);
  get_bounds(r0,r1,mydata);
  //oops! r0 and r1 are still 0!
}
```


The reference type

Motivation

- Return more than one value from a function
- Pass big bulky data structures to functions without copying
- Pass by address, but use normal syntax!

```
void get_bounds(int & i1, int & i2,
                data_type &data)
{
  //calculations
  i1=45;
  i2=103;
}
...
int setup_plot()
{
  int r0=0,r1=0;
  data_type mydata;
  read_data(mydata);
  get_bounds(r0,r1,mydata);
  //r0 is 45 and r1 is 103
}
```

The reference type

```
// Argument passing by value
double find_arsenic_tolerance(Rat R)
{
  double qnty=0,dqnty=1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty+=dqnty;
  }
  return qnty;
}
...
int lab()
{
  Rat r;
  double t=find_arsenic_tolerance(r);
  // r is still alive!
}
```

Arguments are passed by value

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat`.
- So you send a copy or clone of `r`
- The clone gets injections and is eventually destroyed.

Information about the clone is information about the original!

The reference type

```
// Argument passing by reference
double find_arsenic_tolerance(Rat & R)
{
  double qnty=0,dqnty=1.0e-5;
  while (not R.dead()) {
      R.inject(dqnty);
      qnty+=dqnty;
  }
  return qnty;
}
...
int lab()
{
  Rat r;
  double t=find_arsenic_tolerance(r);
  // r is no more!
}
```

Arguments are passed by value

- The function `find_arsenic_tolerance` needs, as the argument, an object of type `Rat &`, i.e., a reference to which Rat.
- So you send a copy of the Id tag on `r` to the function.
- The function acts on the `Rat` object which was referenced.

Information about the original rat, but the rat was modified.

The reference type

We want to change an object

- When we want our object to be modified in some way by a function, it is no good to pass only a copy.
- In this example, a clone of the wounded leg will be bandaged

```
void bandage_leg(Leg l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
int main()
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //No benefits to h.
}
```


The reference type

We want to change an object

- Modifying a copy of our object is useless
- But a copy of a reference is good enough.
- In this example, the function works on the leg that was referred to.

```
void bandage_leg(Leg & l)
{
  //Select right bandage
  //Wrap bandage around l
}
...
int main()
{
  Human h;
  ...
  // h got a wounded left leg
  bandage_leg(h.left_leg());
  //Intended benefits to h
}
```

We can use a function working with a reference when we want it to change our original object.

The reference type

Cloning is expensive

- Sometimes, the data structures are very large, and copying them is expensive
- Functions taking that kind of classes will implicitly perform big cloning operations, slowing the program down.

```
int count_bad_tires(Truck t)
{
  int n=0;
  for (int i=0;i<t.n_wheels();++i) {
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
int main()
{
  Truck mytruck;
  ...
  nbad=count_bad_tires(mytruck);
  // Unnecessary cloning of mytruck
}
```


The reference type

Cloning is expensive

- Using a reference in the function argument, we get away with cloning only the reference to a truck
- The same effect can be achieved by a pointer, but the syntax with references is cleaner

```
int count_bad_tires(Truck & t)
{
  int n=0;
  for (int i=0;i<t.n_wheels();++i) {
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
int main()
{
  Truck mytruck;
  ...
  nbad=count_bad_tires(mytruck);
  // Clone of reference to truck, not
  // clone of truck
}
```

The constant reference type

Cloning is expensive

- We want to use a reference as the argument only because it is efficient
- How do we ensure that the original object would not be allowed to change ?

```
int count_bad_tires(Truck & t)
{
  int n=0;
  for (int i=0;i<t.n_wheels();++i) {
      check_pressure(t.wheel(i));
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
int main()
{
  Truck mytruck;
  ...
  nbad=count_bad_tires(mytruck);
  // Was there any change to mytruck ?
}
```


The constant reference type

Cloning is expensive

- We want to use a reference as the argument only because it is efficient
- How do we ensure that the original object would not be allowed to change ?
- Using a **`const`** `reference`

```
int count_bad_tires(const Truck & t)
{
  int n=0;
  for (int i=0;i<t.n_wheels();++i) {
      check_pressure(t.wheel(i));
      if (not t.wheel(i).good()) ++n;
  }
  return n;
}
...
int main()
{
  Truck mytruck;
  ...
  nbad=count_bad_tires(mytruck);
  // Was there any change to mytruck ?
  // Not if this compiled!
}
```

The reference type

Under the hood

- The reference type is a restricted pointer
- It must always point to an object, not to `nullptr` like a pointer
- It can not be created uninitialised.

Summary: reference

- A fixed pointer with nicer syntax
- Use to pass arguments to functions which are supposed to modify them
- Use references when copying the type is expensive
- Use **`const`** references to ensure that the function does not modify objects it accesses by reference

Example: references

Example 1.5:

The reference type The example code `examples/references.cc` demonstrates these many aspects of references. Study the code, compile and run. Try to understand the values printed.

R-value references and move symantics

```
void add_to_list(string s);
...
string name,lname;
name="James";
lname="Bond";

add_to_list(name);
add_to_list(name+lname);

add_to_list(lname);
```

- Sometimes we want to be able to use references on "nameless" objects
- Avoid copy operations if no harm can come through stealing resources
- It is possible in C++11 with a new kind of references
- Later!

Introduction to C++ templates

Same operations on different types

- Exactly the same high level code
- Differences only due to properties of the arguments

```
void copy_int(int *start, int *end, int *start2)
{
  for (;start!=end; ++start,++start2) {
    *start2=*start;
  }
}
void copy_string(string *start, string *end, string *
     start2)
{
  for (;start!=end; ++start,++start2) {
    *start2=*start;
  }
}
void copy_double(double *start, double *end, double *
     start2)
{
  for (;start!=end; ++start,++start2) {
    *start2=*start;
  }
}
...
double a[10],b[10];
...
copy_double(a,a+10,b);
```


JÜLICH FORSCHUNGSZENTRUM

Introduction to C++ templates

Same operations on different types

- Exactly the same high level code
- Differences only due to properties of the arguments
- Let the compiler do the horse work!

```
template <class itr>
void mycopy(itr start, itr end, itr start2)
{
  for (;start!=end; ++start,++start2) {
    *start2=*start;
  }
}
...
double a[10],b[10];
string anames[5],bnames[5];
...
mycopy(a,a+10,b);
mycopy(anames,anames+5,bnames);
```

Internally the compiler translates the two calls to

```
mycopy<double *>(a,a+10,b);
mycopy<string *>(anames,anames+5,bnames);
```

... so that there is no name conflict.

Introduction to C++ templates

Example 1.6:

The example code `examples/template_intro.cc` contains the above templated copy function. Compile and check that it works. Read through the code and understand it.

Code legibility

```
double foo(double x, int i)
{
double y=1;
if (i>0) {
for (int j=0;j<i;++j) {
y *= x;
}
} else if (i<0) {
for (int j=0;j>i;--j) {
y /= x;
}
return y;
}
```

Code indentation

- Human brains are not made for searching { and } in dense text

Style

```
double foo(double x, int i)
{
  double y=1;
  if (i>0) {
    for (int j=0;j<i;++j) {
      y *= x;
    }
  } else if (i<0) {
    for (int j=0;j>i;--j) {
      y /= x;
    }
  }
  return y;
}
```

Code indentation

- Indenting code clarifies the logic
- Misplaced brackets, braces etc are easier to detect
- 4-5 levels of nesting is sometimes unavoidable
- Recommentation: indent with 2-4 spaces and be consistent!)

Style

```
double foo(double x, int i)
{
  double y=1;
  if (i>0) {
    for (int j=0;j<i;++j) {
      y *= x;
    }
  } else if (i<0) {
    for (int j=0;j>i;--j) {
      y /= x;
    }
  }
  return y;
}
```

Code indentation

- Set up your editor to indent automatically!
- Use a consistent convention for braces ({ and }).
- These are for the human reader (most often, yourself!). Be nice to yourself, and write code that is easy on the eye!

C++ classes

A complex number struct

Imagine we deal very frequently with complex numbers in a program. C has no elementary complex number type. Bad way: simply keep track of the real and imaginary parts of all numbers in separate arrays, `real[]` and `imaginary[]` and make sure the correct indices are accessed when needed:

```
double complex_mult_real(int i,int j)
{
  return real[i]*real[j]-imaginary[i]*imaginary[j];
}
```

This is rarely ever done! Because one can easily create a suitable **`struct`** to store the numbers.

C++ classes

```
typedef struct complex_number
{
  double real, imaginary;
} complex_number;
...
complex_number c_mult(complex_number a,
                      complex_number b)
{
  complex_number c;
  c.real=a.real*b.real-
         a.imaginary*b.imaginary;
  c.imaginary=a.real*b.imaginary
             +a.imaginary*b.real;
  return c;
}
int main()
{
  complex_number z1={1.3,5.3},
  complex_number z2={3.2,0.23};
  complex_number z3=c_mult(z1,z2);
  printf("%f+%f i\n",
         z3.real,z3.imaginary);
}
```

- Much more elegant
- Easier to manage
- Logically related data in one **`struct`**
- A set of functions to manage common operations on this kind of data can be written once
- the structure can then be used whenever complex numbers are needed.

Very close to the concept of a C++ **`class`**!

C++ classes

```
typedef struct complex_number
{
  double real, imaginary;
} complex_number;
...
complex_number c_mult(complex_number a,
                      complex_number b)
{
  complex_number c;
  c.real=a.real*b.real-
         a.imaginary*b.imaginary;
  c.imaginary=a.real*b.imaginary
             +a.imaginary*b.real;
  return c;
}
int main()
{
  complex_number z1={1.3,5.3},
  complex_number z2={3.2,0.23};
  complex_number z3=c_mult(z1,z2);
  printf("%f+%f i\n",
         z3.real,z3.imaginary);
}
```

- Identifies a concept
- Encapsulates the data needed to describe it
- Specifies how that data may be manipulated
- C++ develops this idea of programming concepts further.

C++ classes

Modifying our complex number structure using C++ syntax

- ```
  typedef struct complex_number
  {
    double real, imaginary;
  } complex_number;
  ```

  The **typedef** is no longer needed!
- Once you have written,

```
struct complex_number
{
  double real, imaginary;
};
```

- the name `complex_number` can already be used as a type name!

```
complex_number a, b, c;
```


C++ classes

Modifying our complex number structure using C++ syntax

Just like data (`real` or `imaginary`), functions, relevant for the concept, can be declared inside the **struct** :

```
struct complex_number
{
  double real, imaginary;
  double modulus()
  {
      return sqrt(real*real+
                  imaginary*imaginary);
  }
};
...
complex_number a{1,2},b{3,4};
double c,d;
...
c=a.modulus();//1*1+2*2
d=b.modulus();//3*3+4*4
```

- Data and function members
- A (non-static) member function is invoked on an instance of our structure.
- `a.real` is the real part of `a`. `a.modulus()` is the modulus of a.
- Inside a member function, member variables correspond to the invoking instance.
```

C++ classes

Modifying our complex number structure using C++ syntax

A member function can take arguments like any other function.

```
struct complex_number
{
  complex_number add(complex_number b)
  {
    return complex_number(real+b.real,
                imaginary+b.imaginary);
  }
  // with C++11 ...
  complex_number subt(complex_number b)
  {
    return {real-b.real,
            imaginary-b.imaginary};
  }
};
...
complex_number a(0,0),b(0,0),c(1,1);
...
c=a.add(b);
```

- Data members of the function arguments need to be addressed with the "." operator
- Probably a better way to "pronounce" the `a.add(b)` is "sum of `a` with `b`"

Member functions

```
struct Cat
{
    int tail;
    int head;
    int leg[4];
    Cat();
    void move_tail();
...
};
...
Cat c;
c.move_tail();
```

Member functions

- Relative to an object of that type
- They represent
  - properties of that object e.g. `z.modulus();`
  - actions that make sense for objects of that type e.g. `Cat c;c.move_tail();`
- Think about it! `move_tail(c)` has a very different feel to it than `c.move_tail();`

Member functions

```
struct Cat
{
    int thetail;
    int thehead;
    int theleg[4];
};
int Cat::tail()
{
    return thetail;
}
int Cat::head()
{
    return thehead;
}
```

Member functions

- Relative to an object of that type
- When a member function accesses a data member, there is no ambiguity about which object we are referring to.

Member functions

```
struct Cat
{
    int thetail;
    int thehead;
    int theleg[4];
};
int Cat::head()
{
    return thehead;
}
Cat * Cat::which_cat_is_best()
{
    return this;
}
```

Member functions

- How would a member function to refer to the object as a whole ?
- There is an implicit pointer **`this`** available inside each (non-static) member function!

Function overloading

Modifying our complex number structure using C++ syntax

```
struct complex_number
{
  complex_number mult(complex_number b)
  {
    return {real*b.real-imaginary*b.imaginary,
            real*b.imaginary+imaginary*b.real};
  }
  complex_number multiply(double d)
  {
    return {d*real, d*imaginary};
  }
};
...
complex_number a{0.33,0.434},b{3,4};
double c;
...
b=a.multiply(c);
```

- It is ok to have several functions with the same name, if they have different signatures. This is called "overloading".
- Use overloading when it makes sense.

Constructors

```
complex_number make_complex(double re, double im)
{
  complex_number c;
  c.real=re;
  c.imaginary=im;
  return c;
}
```

We need this, whenever we create structures like the complex number. In C++, they are called `constructor`s. We don't need to be creative about their names: they have the same name as the structures we are trying to create.

```
struct complex_number
{
  complex_number(double re, double im)
  {
    real=re;
    imaginary=im;
  }
};
```

Constructors

- Alternative syntax to initialise variables in constructors

```
struct complex_number
{
  complex_number(double re, double im) : real{re}, imaginary{im} {}
};
```

- When a variable is declared, a constructor with the appropriate number of arguments is implicitly called

```
complex_number a(3.2,9.3); // C++11 and older
complex_number b{4.3,1.9}; // C++11
```


Constructors

```
struct complex_number
{
  complex_number(double re, double im)
  {
    real=re;
    imaginary=im;
  }
  complex_number()
  {
    real=imaginary=0;
  }
  double real,imaginary;
};
...
complex_number a(4.3,23.09),b;
```

- Constructors may be (and normally are) overloaded.
- The default constructor is the one without any arguments. That is the one invoked when no arguments are given while creating the object.

Constructors

```
struct complex_number
{
  complex_number(double re, double im)
  {
    real=re;
    imaginary=im;
  }
  complex_number() {}
  double real = 0; // Only C++11!
  double imaginary = 0;
};
...
complex_number a(4.3,23.09),b;
```

- In C++11, member variables can be initilised to "default values" at the point of declaration
- Member variables not touched by the constructor stay at their default values

Defining member functions outside the struct

```
struct complex_number
{
  double add(complex_number b);
};
double complex_number::add(complex_number b)
{
  return {real+b.real,imaginary+b.imaginary};
}
```

- If the function body is more than a line, for readability, it is better to only declare the function in the **`struct`**
- In such a case, when the function is defined, the scope resolution operator `::` has to be used to specify that the function belongs with the structure.

Operator overloading

Defining new actions for the operators

```
struct complex_number
{
  complex_number operator+(complex_number b);
  // instead of complex_number add(complex_number b);
};
complex_number complex_number::operator+(complex_number b)
{
  return {real+b.real, imaginary+b.imaginary};
}
...
complex_number a,b,c;
...
c=a+b; // means a.operator+(b);
```


Redefining operators for a class

Teaching cout how to print your construction

```
std::ostream & operator<<(std::ostream &os, complex_number &a)
{
  os<<a.real;
  if (a.imaginary<0) os<<a.imaginary<<" i ";
  else os<<" +"<<a.imaginary<<" i ";
  return os;
}
complex_number a;
...
std::cout<<"The roots are "<<a<<" and "<<a.conjugate()<<'\n';
```

Exercise 1.5:

Complete the complex number class in `exercises/complex_numbers.cc`. Implement the rest of the arithmatic operations. What happens if you change the keyword **`struct`** to a **`class`** ?

Public and private members

Separating interface and implementation

```
int foo(complex_number a, int p, truck c)
{
  complex_number z1,z2,z3=a;
...
  z1=z1.argument()*z2.modulus()*z3.conjugate();
  c.start(z1.imaginary*p);
}
```

Imagine that ...

- We have used our complex number structure in a lot of places
- Then one day, it becomes evident that it is more efficient to define the complex numbers in terms of the modulus and argument, instead of the real and imaginary parts.
- We have to change a lot of code.

Public and private members

Separating interface and implementation

```
int foo(complex_number a, int p, truck c)
{
   complex_number z1,z2,z3=a;
...
   z1=z1.argument()*z2.modulus()*z3.conjugate();
   c.start(z1.imaginary*p);
}
```

Imagine that ...

- External code calling only member functions can survive
- Direct use of member variables while using a class is often messy, the implementor of the class then loses the freedom to change internal organisation of the class for efficiency or other reasons

C++ classes

Modifying our complex number structure using C++ syntax

```
class complex_number
{
public:
  complex_number(double re, double im)
  : realpt(re), imagpt(im) {}
  complex_number() = default;
  double real() { return realpt; }
  double imaginary() { return imagpt; }
...
private:
    double realpt=0,imagpt=0;
};
```

- Members declared under the keyword **`private`** can not be accessed from outside
- Public members (data or function) can be accessed
- Provide a consistent and useful interface through public functions
- Keep data members hidden

Freeing memory for user defined types

```
class expt
{
...
  double *mydata;
};

void expt::allocate(unsigned int n)
{
  mydata=new double[n];
}

double tempfunc(double phasediff)
{
  expt A;
  // find number of elements
  A.allocate(nele);
  // do some great calculations
  return answer;
}
```

What happens to the memory ?

The class `expt` has a pointer member, which points to dynamically allocated memory

- When the life of the variable `A` ends the memory corresponding to the member variables (e.g. the pointer `mydata` is freed.
- But who frees the dynamically allocated memory ?

Freeing memory for user defined types

```
class expt
{
...
  double *mydata;
};

void expt::allocate(unsigned int n)
{
  mydata=new double[n];
}

double tempfunc(double phasediff)
{
  expt A;
  // find number of elements
  A.allocate(nele);
  // do some great calculations
  return answer;
}
```

What happens to the memory ?

For classes which explicitly allocate dynamic memory

- We need a function that cleans up all explicitly allocated memory in use
- It would be useful if that function is automatically called when the variable is about to expire.

Freeing memory for user defined types

```
class expt
{
  expt();
  ~expt();
// ...
  double *mydata;
};
expt::~expt()
{
  if (mydata) delete [] mydata;
}
void expt::allocate(unsigned int n)
{
  mydata=new double[n];
}
double tempfunc(double phasediff)
{
  expt A;
  // find number of elements
  A.allocate(nele);
  // do some great calculations
  return answer;
}
```

Destructors

- There is an automatically called `destructor` function
- Called when a variable expires
- Fixed name suggesting it is the opposite of a constructor.
- No function arguments or return value

Freeing memory for user defined types

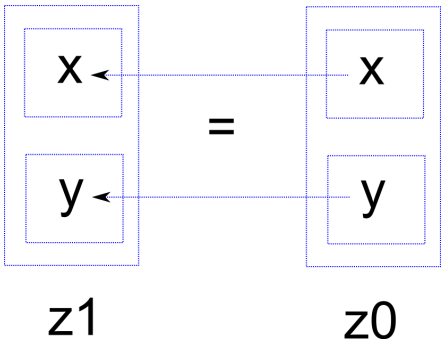
```
class expt
{
  expt();
  ~expt();
//...
    double *mydata=nullptr;
};
expt::~expt()
{
  if (mydata) delete [] mydata;
}
void expt::allocate(unsigned int n)
{
  mydata=new double[n];
}
double tempfunc(double phasediff)
{
  expt A;
  // find number of elements
  A.allocate(nele);
  // do some great calculations
  return answer;
}
```

Destructors

- If the user does not define a `destructor`, the compiler automatically generates one, which trivially frees all member variables.
- If you explicitly allocate memory in your class, make sure you implement a `desctuctor` function to cleanup.

Copying and assignments

```
struct complex_number
{
  double x, y;
};
//...
complex_number z0(2.0,3.0),z1;
z1=z0; // assignment operator
complex_number z2(z0); //copy constructor
```

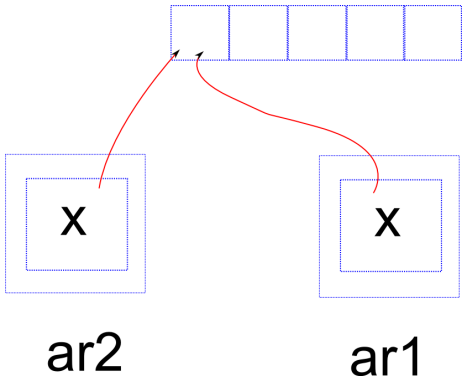


What values should the members get ?

- In most cases, we want to assign the data members to the corresponding members
- This happens automatically, but using special functions for these copy operations
- You can redefine them for your class
- Why would you want to ?

Copying and assignments

```
class darray {
  double *x;
};
void darray::darray(unsigned n)
{
  x=new double[n];
}
void foo()
{
  darray ar1(5);
  darray ar2(ar1); //copy constructor
  ar2[3]=2.1;
  //oops! ar1[3] is also 2.1 now!
} //trouble
```

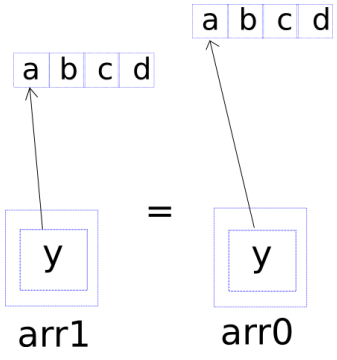


Copying pointers with dynamically allocated memory

- May not be what we want
- Leads to "double free" errors when the objects are destroyed

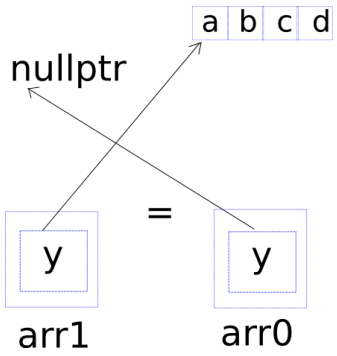
Copying and assignments

```
class darray {
  double *x=nullptr;
  unsigned int len=0;
public:
  // Copy constructor
  darray(const darray &);
  //assignment operator
  darray & operator=(const darray &);
};
darray::darray(const darray & other)
{
  if (other.len!=0) {
    len=other.len;
    x = new double[len];
    for (unsigned i=0;i<len;++i) {
      x[i]=other.x[i];
    }
  }
}
darray & darray::operator=(const darray &
      other)
{
  if (this!=&other) {
    if (len!=other.len) {
      len=other.len;
      if (x) delete [] x;
      x= new double[len];
    }
    for (unsigned i=0;i<len;++i) {
      x[i]=other.x[i];
    }
  }
  return *this;
}
```



Move constructor/assignment operator

```
class darray {
  darray(darray &&); //Move constructor
  darray & operator=(darray &&);
  //Move assignment operator
};
darray::darray(darray && other)
{
  len=other.len;
  x=other.x;
  other.x=nullptr;
}
darray & darray::operator=(darray &&
      other) {
  len=other.len;
  x=other.x;
  other.x=nullptr;
  return *this;
}
darray d1(3);
init_array(d1);//d1={1.0,2.0,3.0}
darray d2{d1};//Copy construction
// d1 and d2  are {1.,2.,3.}
darray d3{std::move(d1)}; //Move
// d3 is {1.,2.,3.}, but d1 is empty!
```



- Construct or assign from an R-value reference (`darray &&`)
- Steal resources from RHS
- Put disposable content in RHS

Move constructor/assignment operator

- You can enable move symantics for your class by writing a constructor or assignment operator using an R-value reference
- Usually you will not be using it explicitly
- You can invoke the move constructor by casting the function argument to an R-value reference, e.g.
  `darray d3{std::move(d1)}`

Big five

- Default constructor
- Copy constructor
- Move constructor
- Assignment operator
- Move assignment operator

- How many of these do you have to write for each and every class you make ?
- Answer: None! If you don't have bare pointers in your class, and don't want anything fancy happening, the compiler will auto-generate reasonable defaults.

Big five

```
class cnumber {
public:
  cnumber(double x, double y) : re{x}, im{y} {}
  cnumber() = default;
  cnumber(const cnumber &) = default;
  cnumber(cnumber &&) = default;
  cnumber & operator=(const cnumber &) = default;
  cnumber & operator=(cnumber &) = default;
};
```

- If you have to write any constructor yourself, auto-generation is disabled
- But you can request default versions of the rest of these functions as shown

Big five

```
class darray {
  darray() = delete;
  darray(const cnumber &) = delete;
  darray(cnumber &&) = default;
  darray & operator=(const cnumber &) = delete;
  darray & operator=(cnumber &) = default;
};
```

- You can also explicitly request that one or more of these are not auto-generated
- In the example shown here, it will not be possible to copy objects of the class, but they can be moved

Constructor/destructor calls

Exercise 1.6:

The file `exercises/verbose_ctordtor.cc` demonstrates the automatic calls to constructors and destructors. The simple class `Vbose` has one `string` member. All its constructors and destructors print messages to the screen when they are called. The `main()` function creates and uses some objects of this class. Follow the messages printed on the screen and link them to the statements in the program. Does it make sense (i) When the copy constructor is called ? (ii) When is the move constructor invoked ? (iii) When the objects are destroyed ?

Suggested reading: `http://www.informit.com/articles/printerfriendly/2216986`

Exercise 1.7:

The example program `complex_number_class.cc` contains a more complete version of the complex number class, with all syntax elements we discussed in the class. It is heavily commented with explanations for every subsection. Please read it to revise all the syntax relating to classes. Write a `main` program to use and test the class.

Exercise 1.8:

Write a class to represent dynamic arrays of complex numbers.

- You should be able to write :

```
complex_array q;
std::cout<<"Number of points: ";
unsigned int npt;
std::cin>>npt;

q.resize(npt);
for (unsigned j=0;j<npt;++j) {
   q[j].set(j*pi/npt,0);
}
```

- Define an inner product of two complex arrays

Exercises

Exercise 1.9:

A histogram class Write a histogram class. It should be possible to set the range, and the number of bins. A function `put` should take a **`double`** and put it in the right bin. It should be possible to ask the histogram to output its data to a file in a normalised form.

Exercise 1.10:

A 3D vector class Write a class to represent the concept of vectors in 3D space. Make sure you have operators or functions for addition, subtraction, scalar and vector product, translation and rotation of the vector as well as input and output.

Day 2

C++ classes

- Constructors and destructors of classes are special functions which do not "return" anything
- In C++11, constructors can invoke other constructors to get the work done

```
complex_number::complex_number(double x,double y) : re{x}, im{y} {}
complex_number::complex_number(double x) : complex_number(x,0) {}
```

- A (non-static) member function of class `A` implicitly takes a reference to an object of type `A` as an argument. To demand that it does not modify its implicit argument, we need to put the **`const`** keyword after the function signature :

```
double f(const string &s,const string &t);
// So that we know f(a,b) will not change a or b
class complex_number {
  complex_number comp(const complex_number &other) const
  // so that we know c.comp(d) will not change c or d
};
```


C++ classes

Exercise 2.1:

The file `exercises/angles.cc` contains a lot of elements from what we learned about classes. There are also a few new tips, and suggested experiments. Read the code. Run it. Try to understand the output. Do the suggested code changes, and make sure you understand the compiler errors or the change in results.

Static members

```
class Triangle {
public:
  static unsigned counter;
  Triangle() : ...
  {
    ++counter;
  }
  ~Triangle() { --counter; }
  static unsigned instanceCount() {
    return counter;
  }
};
... Triangle.cc ...
unsigned Triangle::counter=0;
```

- Static variables exist only once for all objects of the class.
- Can be used to keep track of the number of objects of one type created in the whole application
- Must be initialised in a source file somewhere, or else you get an "unresolved symbol" error

Exercise 2.2:

Use a **`static`** variable to make a "singleton" class. In such a class, there will always be only one object. Even when you try to create another object, you should effectively get the already created object rather than a new one.

Exercise 2.3:

Write a dynamic array class for double precision members. Demostrate with a program where one can read in the desired size from standard input, and create an array of that size. It must free any memory it has allocated when its scope ends. Implement a copy and move constructors and assignment operators. Define the `[]` operator to yield the i'th element of the array just like a C-style array would.

Exercise 2.4:

Write a small program which accepts an unsigned integer N as input, and then reads N numbers from the standard input, stores them in dynamic array of the type you created in the previous exercise. After N numbers have been read, it should calculate the mean and standard deviation of the values.

Error handling

Run-time error handling

When there is nothing reasonable to return

```
double f(double x)
{
  double answer=1;
  if (x>=0 and x<10) {
    while (x>0) {
      answer*=x;
      x-=1;
    }
  } else {
    // the function is undefined
  }
  return answer;
  // should we really return anything
  // if the function went into
  // the "else"?
}
```

Exceptions

- A function may be called with arguments which don't make sense
- An illegal mathematical operation
- Too much memory might have been requested.

When there is nothing reasonable to return

```
class myexception : public std::exception {
  double x;
  public:
  myexception(double vl) : x(vl) {}
  const char * what() const noexcept {
    std::string msg=("bad parameter value ")+
      std::to_string(x);
    return msg.c_str();
  }
};
double f(double x)
{
  double answer=1;
  if (x>=0 and x<10) {
    while (x>0) {
      answer*=x;
      x-=1;
    }
  } else {
    throw(myexception(x));
  }
  return answer;
}
```

```
try {
  std::cout<<"Enter start point : ";
  std::cin >> x;
  std::cout<<"The result is "
           <<f(x)<<'\n';
} catch (myexception ex) {
  std::cerr<<ex.what()<<'\n';
}
```

- Enclose the area where an exception might be thrown in a **`try`** block
- In case the error happens, control shifts to the **`catch`** block

Exercise 2.5:

Test exceptions The program `exercises/exception.cc` demonstrates the use of exceptions. Rewrite the loop so that the user is asked for a new value until a reasonable value for the function input parameter is given.

Compile time assertions

```
double advance(unsigned long L)
{
  static_assert(sizeof(L)>=8,"long type must be at least 8 bytes)");
  //Bit manipulation assuming "long" is at least 8 bytes
}
```

- Prints the second argument as an error message if the first argument evaluates to false.
- Express assumptions clearly, so that the compiler notifies you when they are violated

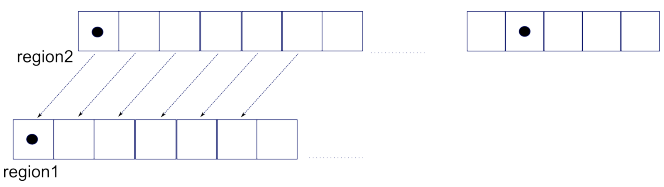
Example 2.1:

In `examples/static_assert0.cc` you have a demo of using static assert to warn you when an important assumption in your code is violated on the system. It does not produce any error, but you can change the condition to something else which is violated in your system, to see how this mechanism works.

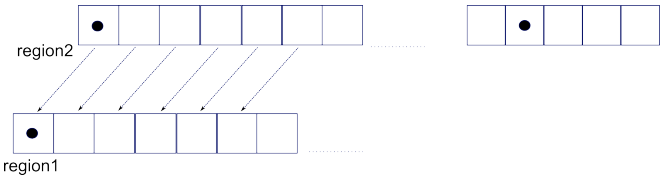
C++ templates

memcpy

```
void* memcpy(void* region1, const void* region2, size_t n)
{
  const char* first = (const char*)region2;
  const char* last = ((const char*)region2) + n;
  char* result = (char*)region1;
  while (first != last)
    *result++ = *first++;
  return result;
}
```



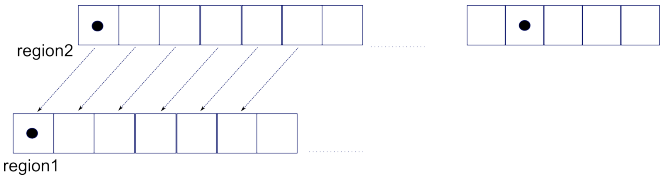
memcpy



Generic code

- The **`void`** `*` is a trick: pointer to any class can be cast to a **`void`** `*`
- So, the memcpy code can copy arrays of int, float, double etc.
- But what if the collection we want to copy is in a linked list, instead of an array ?
- ... or bare copy is not enough for your type (Remember why you needed copy constructors ?)

memcpy



Generic code

The logic of the copy operation is quite simple. Given an iterator range `first` to `last` in an input sequence, and a target location `result` in an output sequence, we want to:

- Loop over the input sequence
- For each position of the input iterator, copy the element to the output iterator.
- Increment the input and output iterators
- Stop if the input iterator has reached `last`

A template for a generic copy operation

```
template <typename InputIterator, typename OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
{
  while (first != last) *result++ = *first++;
  return result;
}
```

C++ template notation

- A template with which to generate code!
- If you know iterators to two kinds of sequences, you can substitute them in the above template and have a nice copy function!
- You don't even have to do it! The compiler will do the work.
- Let's see that in action!

Test generic copy function

Exercise 2.6:

exercises/copy0.cc The file `exercises/copy0.cc` contains the above generic copy function, and a test code in which we copy a `std::list` into a `std::vector`.

- Compile and run the program
- At the end of the main program, declare two strings `s1="AAAA AAAA"; s2="BBBB BBBB";` Check that the same copy function can copy s1 to s2!

C++ templates

Exercise 2.7:

Write a generic inner product function, that takes two sequences of "stuff", multiplies them term by term and adds them. Show that you can use it to multiply lists of doubles, integers, complex numbers. Also, check that you can use the same function after changing one or both lists to vectors.

JÜLICH FORSCHUNGSZENTRUM

Writing generic code

- Identify similar operations performed on different kinds of data
- Think about the essential logic of those operations, not involving peculiar properties of one data type.
- Write code using templates, so that when the templates are instantiated, correct code results for the different data types

Ordered pairs

```
struct double_pair
{
    double first,second;
};
...
double_pair coords[100];
...
struct int_pair
{
    int first,second;
};
...
int_pair line_ranges[100];
...
struct int_double_pair
{
    // wait!
    // can I make a template out of it?
};
```

Template classes

- Classes can be templated too
- Generated when the template is "instantiated"

```
template <class T, class U>
struct pair
{
    T first;
    U second;
};
```


Ordered pairs

```
pair<double,double> coords[100];
pair<int,int> line_ranges[100];
pair<int,double> whatever;
```

Template classes

- Classes can be templated too
- Generated when the template is "instantiated"

```
template <class T, class U>
struct pair
{
    T first;
    U second;
};
```

Exercise: Write a template class of your own

```
template <typename Datatype>
class darray
{
private:
    Datatype *data=nullptr;
    unsigned len=0;
public:
    Datatype operator[] const;
    Datatype & operator[];
    etc.
};
...
darray<string> A(24);
darray<complex_number> C(32);
```

Exercise 2.8: Template classes

The dynamic array from your earlier exercise is a prime candidate for a template implementation Rewrite your code using templates so that you can use it for an array of strings as well as an array of dynamic array of doubles.

Some fun: overloading the () operator

```
class swave
{
private:
  double a=1.0, omega=1.0;
public:
  swave()=default;
  swave(double x, double w) :
     a{x}, omega{w} {}
  double operator()(double t) const
  {
    return a*sin(omega*t);
  }
};
```

```
const double pi=acos(-1);

int N=100;
swave f{2.0,0.4};
swave g{2.3,1.2};

for (int i=0;i<N;++i) {
  double ar=2*i*pi/N;
  std::cout<<i<<" "<<f(ar)
           <<" "<<g(ar)
           <<'\n';
}
```

Functionals

- Function like objects, i.e., classes which define a `()` operator
- If they return a **`bool`** value, they are called predicates

Functionals

Using function like objects

- They are like other variables. But they can be used as if they were functions!
- You can make vectors or lists of functionals, pass them as arguments ...

Write your own functional!

Exercise 2.9:

Write a functional class where the return value of f(x) is given by a user specified piece-wise continuous linear function. You should write a class PieceWise. It should have a function to read a vector of x_i,y_i values from a file. Sort them according to x values. Then implement an **`operator()`** function, so that when you write

```
PieceWise f;
f.read_file("somefile.dat");
double x = whatever;
double y = f(x);
```

you get the correct piecewise linear function evaluated.

Day 3

A dynamic array template class

```
template <typename T>
class darray {
public:
  darray() = default;
  darray(const darray<T> &oth);
  darray(darray<T> &&oth);
  darray(size_t N);
  darray<T> &operator=(const darray<T> &);
  darray<T> &operator=(darray<T> &&);
  ~darray();
  inline T operator[](size_t i) const { return arr[i]; }
  inline T &operator[](size_t i) { return arr[i]; }
  T sum() const;
  template <typename U>
  friend ostream &operator<<(ostream &os,const darray<U> &);
private:
  void swap(darray &oth) {
    std::swap(arr,oth.arr);
    std::swap(sz,oth.sz);
  }
  T *arr=nullptr;
  size_t sz=0;
};
```

- Two versions of the `[]` operator for readonly and read/write access
- Use **const** qualifier in any member function which does not change the object
- Note how the **friend** function is declared

A dynamic array template class

```
template <typename T> T darray<T>::sum() const {
  T a{};
  for (size_t i=0;i<sz;++i) a+=arr[i];
  return a;
}
template <typename U>
ostream &operator<<(ostream &os,const darray<U> &da) {
...
}
template <typename T> darray<T>::darray(size_t N) {
  if (N!=0) {
    arr=new T [N];
    sz=N;
  }
}
template <typename T> darray<T>::~darray() {
  if (arr) delete [] arr;
}
template <typename T> darray<T>::darray(darray<T> &&oth) {
  swap(oth);
}
```

- The template keyword must be used before all member functions defined outside class declaration
- The class name before the :: must be the fully qualified name e.g. darray<T>

A dynamic array template class

```
template <typename T>
darray<T>::darray(const darray<T> &oth) {
  if (oth.sz!=0) {
    sz=oth.sz;
    arr=new T[sz];
  }
  for (size_t i=0;i<sz;++i) arr[i]=oth.arr[i];
}
template <typename T>
darray<T> &operator=(const darray<T> &oth) {
  if (this!=&oth) {
    if (arr && sz!=oth.sz) {
      sz=oth.sz;
      delete [] arr;
      arr=new T[sz];
    }
    for (size_t i=0;i<sz;++i) arr[i]=oth.arr[i];
  }
  return *this;
}
template <typename T> darray<T> &operator=(darray<T> &&oth) {
  swap(oth);
  return *this;
}
```

- All function definitions, not just the declarations must be visible at the point where templates are instantiated.
- Often, template functions are defined inside the class declarations.

A dynamic array template class

```
template <typename T>
class darray {
public:
  inline T operator[](size_t i) const { return arr[i]; }
  inline T &operator[](size_t i) { return arr[i]; }
  T sum() const {
    T a{};
    for (size_t i=0;i<sz;++i) a+=arr[i];
    return a;
  }
  darray() = default;
  darray(size_t N) {
    if (N!=0) {
      arr=new T [N];
      sz=N;
    }
  }
  ~darray() { if (arr) delete [] arr; }
  darray(darray<T> &&oth) { swap(oth); }
  darray &operator=(darray &&oth) {
    swap(oth);
    return *this;
  }
```

- We escape having to write the **template** < **typename** T>before every function
- Class declaration longer, so that it is harder to quickly glance at all available functions

A dynamic array template class

```
  darray(const darray<T> &oth) {
    if (oth.sz!=0) {
      sz=oth.sz;
      arr=new T[sz];
    }
    for (size_t i=0;i<sz;++i) arr[i]=oth.arr[i];
  }
  darray &operator=(const darray &oth) {
    if (this!=&oth) {
      if (arr && sz!=oth.sz) {
        sz=oth.sz;
        delete [] arr;
        arr=new T[sz];
      }
      for (size_t i=0;i<sz;++i) arr[i]=oth.arr[i];
    }
    return *this;
  }
private:
  void swap(darray &oth) {
    std::swap(arr,oth.arr);
    std::swap(sz,oth.sz);
  }
  T *arr=nullptr;
  size_t sz=0;
};
```

- Note: assignment operators must return a reference to the calling object (i.e., ***this**)
- Notice how the copy constructor and the assignment operators have redundant code

Copy and swap

- We want to reuse the code in the copy constructor and destructor to do memory management
- Pass argument to the assignment operator by value instead of reference
- Use the class member function `swap` to swap the data with the newly created copy

```
darray & operator=(darray d) {
  swap(d);
  return *this;
}
// No further move assignment operator!
```

- Neat trick that works in most cases
- Reduces the big five to big four

Initialiser list constructors

- We want to initialise our `darray<T>` like this:

```
darray<string> S{"A","B","C"};
darray<int> I{1,2,3,4,5};
```

- We need a new type of constructor: the initializer_list constructor

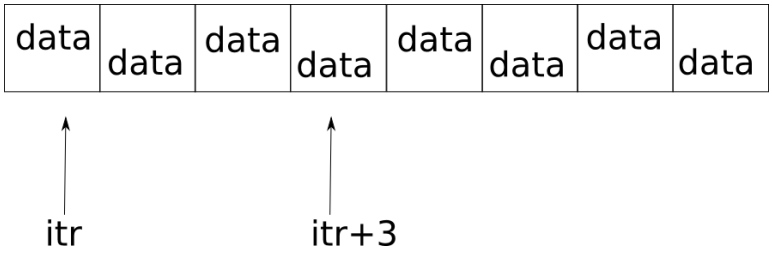
```
darray(initializer_list<T> l) {
  arr=new T[l.size()];
  size_t i=0;
  for (auto el : l) arr[i++]=el;
}
```

Example 3.1:

The file `examples/darray_complete.cc` has a completed dynamic array class with all the above modifications. It is extensively commented. Notice the use of the new function syntax in C++11. See how to enable range based for loop for your classes. See how the output operator was written entirely outside the class. Check the subtle difference in this case between using {10} and (10) as arguments while constructing a darray.

C++ Standard Template Library

Abstract dynamic array

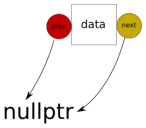


A vector

- Consecutive elements in memory
- Can be accessed using an "iterator"

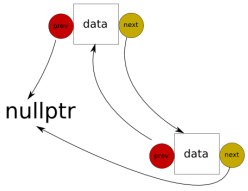
A linked list

A linked list is a collection of connected nodes. Each node has some data, and one or two pointers to other nodes. They are the "next" and "previous" nodes in the linked list. When "next" or "previous" does not exist, the pointer is set to `nullptr`



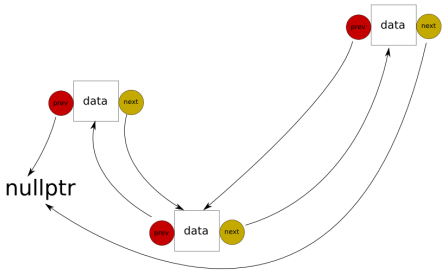
A linked list

When a new element is added to the end of a list, its "previous" pointer is set to the previous end of chain, and it becomes the target of the "next" pointer of the previous end.



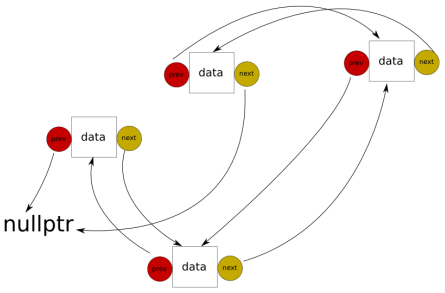
A linked list

New elements can be added to the front or back of the list with only a few pointers needing rearrangement.



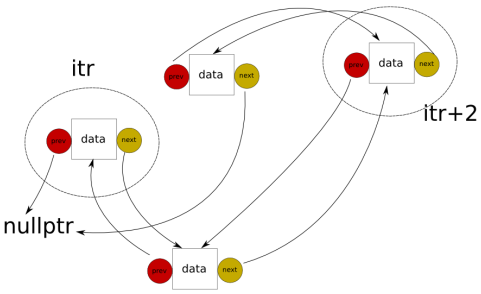
A linked list

Any element in the list can be reached, if one kept track of the beginning or end of the list, and followed the "next" and "previous" pointers.



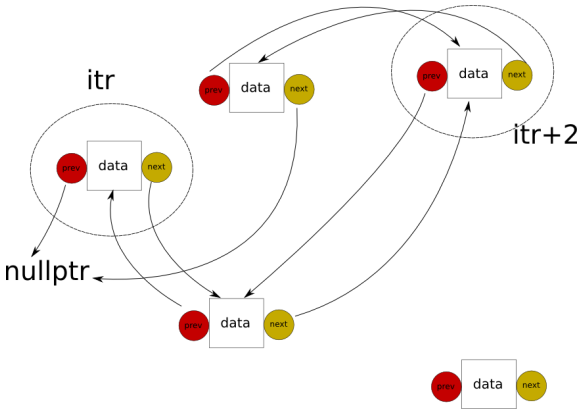
A linked list

A concept of an "iterator" can be devised, where the ++ and −− operators move to the next and previous nodes.



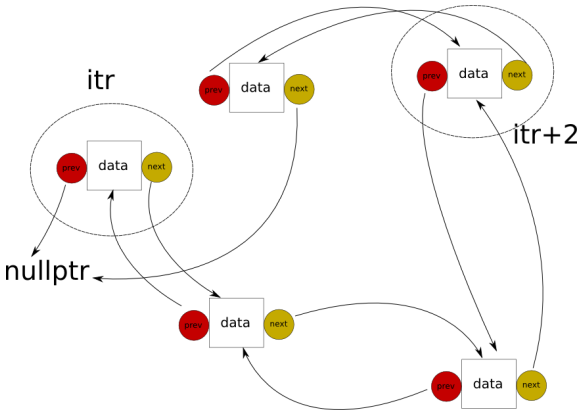
A linked list

Inserting a new element in the middle of the list does not require moving the existing nodes in memory.

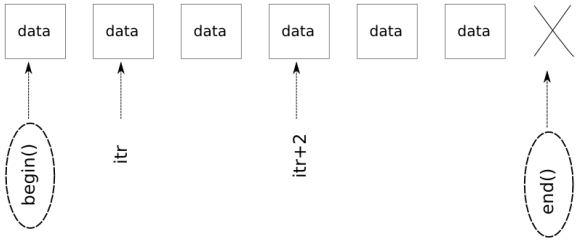


A linked list

Just rearranging the next and previous pointers of the elements between which the new element must go, is enough. This gives efficient $O(1)$ insertions and deletions.



Generic "containers"



- Generic data holding constructions
- Can be accessed through a suitably designed "iterator"
- The data type does not affect the design => template
- Similarity of interface is by choice

STL containers

- `std::vector<>` : dynamic arrays
- `std::list<>` : linked lists
- `std::queue<>` : queue
- `std::deque<>` : double ended queue
- `std::map<A,B>` : associative container

Structures to organise data

- Include file names correspond to class names
- All of them provide an iterator class
- If `iter` is an iterator, `*iter` is data.
- All of them provide `begin()` and `end()` functions

Linked lists

Exercise 3.1:

Write a small C progam to manage a list of names and ages. The program should simply ask the user to enter new information or quit. The user can keep adding information on names and ages, as long as required. When the user quits, the program should print the entered names sorted according to age.

Note: This is *not* a real exercise! Just think about doing it, but we will do this using the standard template library.

Linked lists

Solution using STL lists

- Program `list0.cc`
- Let's first learn using STL!

```
#include <iostream>
#include <list>
#include <string>
struct userinfo
{
  unsigned int age;
  std::string name;
  bool operator<(const userinfo &u)
  {
    return age < u.age or
      age == u.age and name < u.name;
  }
};
```

```
int main()
{
  std::list<userinfo> ulist;
  char c='y';
  do {
    std::cout << "Add more entries ?";
    std::cin>>c;
    if (c!='y') break;
    userinfo tmp;
    std::cout << "Name :";
    std::cin >> std::ws;
    getline(std::cin,tmp.name);
    std::cout << "Age :";
    std::cin >> tmp.age;
    ulist.push_back(tmp);
  } while (c=='y');
  ulist.sort();
  for (auto & user : ulist) {
    std::cout<<user.name<<" : "
        <<user.age<<'\n';
  }
}
```

Linked lists

```
#include <iostream>
#include <list>
#include <string>

struct userinfo
{
  unsigned int age;
  std::string name;
  bool operator<(const userinfo &u)
  {
    return age<u.age or
      age==u.age and name < u.name;
  }
};
int main()
{
  std::list<userinfo> ulist;
  char c='y';
  do {
```

Solution using STL lists

- We don't worry about how long a name would be
- We don't need to keep track of `next`, `previous` pointers
- How did it know how to sort our data ?
- How does it work ?!

Linked lists

```
#include <iostream>
#include <list>
#include <string>

struct userinfo
{
  unsigned int age;
  std::string name;
  bool operator<(const userinfo &u)
  {
    return age<u.age or
      age==u.age and name < u.name;
  }
};
int main()
{
  std::list<userinfo> ulist;
  char c='y';
  do {
```

Solution using STL lists

- `std::list` is a template for a linked list.
- `std::list<userinfo>` is a linked list of our hand made structure `userinfo` !

Our solution to the exercise with std::list

```
#include <list>
...
  std::list<userinfo> ulist;
  userinfo tmp;
...
  ulist.push_back(tmp);
...
  ulist.sort();
...
  for (auto & user : ulist) {
    std::cout<<user.name<<" : "
             <<user.age<<std::endl;
  }
```

- `ulist.push_back(tmp)` appends (a copy of) the object `tmp` at the end of the list
- `ulist.sort()` sorts the elements in the list using the < operator for its elements

Our solution to the exercise with std::list

```
#include <list>
...
  std::list<userinfo> ulist;
  userinfo tmp;
...
  ulist.push_back(tmp);
...
  ulist.sort();
...
  for (auto & user : ulist) {
    std::cout << user.name << " : "
        << user.age << std::endl;
  }
```

- `ulist.push_back(tmp)` appends (a copy of) the object `tmp` at the end of the list
- `ulist.sort()` sorts the elements in the list using the < operator for its elements
- The range based for loops can be used on `lists`

Our solution to the exercise with std::list

```
std::list<userinfo>::iterator it;
for (it=ulist.begin(); it!=ulist.end();++it) {
  std::cout << it->name << " : "
      << it->age << std::endl;
}
```

Iterator

- `std::list<userinfo>::iterator` : is something that iterates over a sequence
- In this example, it mimics a `current` pointer for our linked list.
- Access to the element at the iterator is just like it would be for a pointer to that element.

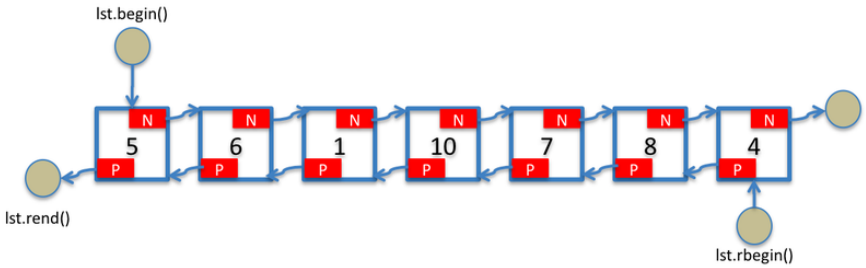
Our solution to the exercise with std::list

```
for (auto it=ulist.begin(); it!=ulist.end();++it) {
  std::cout << it->name << " : "
      << it->age << std::endl;
}
```

Iterator

- An `iterator` has the `++` operations defined on it, which mean "move to the next element in the sequence".
- Similarly there are `--` operators defined
- `ulist.begin()` and `ulist.end()` return `iterator`s at the start and end of the sequence

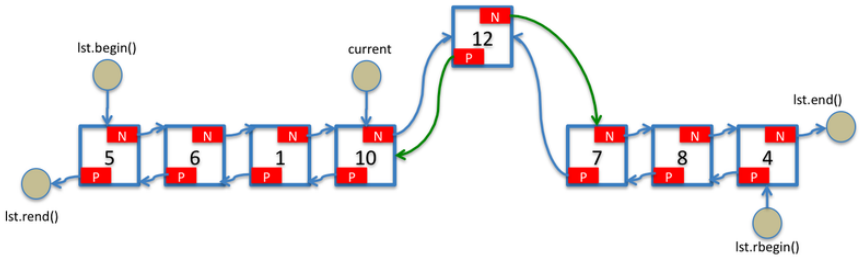
Our solution to the exercise with std::list



Iterator

- `ulist.end()` actually points to a fictitious "one-past-the-last" element. This enables the familiar C-style loop iterations.
- `ulist.rbegin()` and `ulist.rend()` return `reverse_iterator`s which browse the sequence in the opposite direction

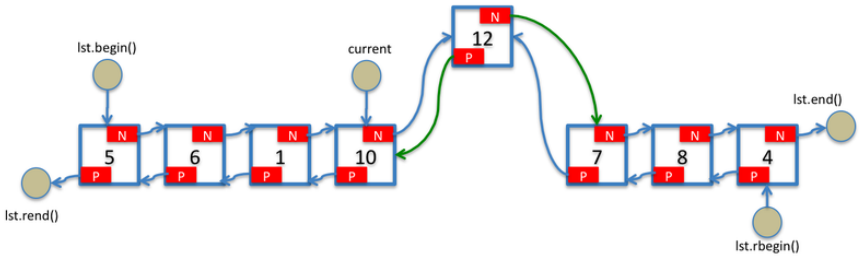
List operations



Iterator

- `std::list` supports efficient $O(1)$ insertions and deletions anywhere
- `ulist.insert(pos, tmp)` inserts a new element `tmp` before the location pointed to be the iterator `pos`

List operations



Iterator

- `ulist.erase(pos)` erases the element at `pos`
- `ulist.erase(pos1, pos2)` erases the elements starting at `pos1` until but excluding the element at `pos2`
- `ulist.pop_back()` erases the last element
- `ulist.pop_front()` erases the first element

Familiarise yourself with std::list

Exercise 3.2:

The program `exercises/list1.cc` demonstrates the syntax of STL lists. Study the program. Change it so that the list contents are printed in reverse order. Also, create two lists and learn how to merge them.

Tip: The internet is a good place to look for syntax and usage examples. Look up `std::list` at

`http://www.cplusplus.com/reference/stl/list/`

or

`http://en.cppreference.com/w/cpp/container/list`

Get familiar with std::list

Exercise 3.3:

Write a program that reads a text file and then prints all words beginning with 's' or 'S', in alphabetical order. Only one instance of each word should be printed.

Other STL containers

- `std::vector<stuff>` : a dynamic array of `stuff`
- `std::queue<stuff>` : a queue
- `std::map<key, stuff>` : an associative array
- `std::valarray<stuff>` : a simple fast dynamic array
- `std::bitset<N>` : a special container with single "bits" as elements
- `std::set<stuff>` : for the concept of a set

Using std::vector

Exercise 3.4:

The class `std::vector` has many similarities with `std::list`, although it represents a dynamic array of stuff, like your own dynamic array class. Write a program in which you declare an vector of complex numbers. Fill the vector with some values. The program then should iterate over the vector and add the elements. To loop over the vector, try using an iterator, range based for loops as well as the usual notation for arrray, i.e., v[i] where i is an integer type.

Using std::vector

- `std::vector<int> v(10);` : array of 10 integers
- Efficient indexing operator `[]`, for unchecked element access
- `v.at(i)` provides range checked access. An exception is thrown if `at(i)` is called with an out-of-range `i`
- `std::vector<std::list<userinfo> > vu(10);` array of 10 linked lists!

More exercise with std::vector

Exercise 3.5:

Write a simple class `StatisticsCollector`. It should have a member function `put(double)` to receive a value. You pass a series of values you get from some experiment to an object of this type. It should then be able to return statistical properties like mean, standard deviation and auto-correlation for up to an adjustible number of inputs.

std::map

```
std::map<std::string, int> flsize;
flsize["S.dat"]=123164;
flsize["D.dat"]=423222;
flsize["A.dat"]=1024;
```

- Think of it as a special kind of "vector" where you can have things other than integers as indices.
- Template arguments specify the key and data types
- Could be thought of as a container storing (key,value) pairs : $\{("S.dat", 123164), ("D.dat", 423222), ("A.dat", 1024)\}$
- The less than comparison operation must be defined on the key type
- Implemented as a tree, which keeps its elements sorted

Exercise 3.6:

Write a program that counts all different words in a text file and prints the statistics.

A word counter program

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>

int main()
{
  std::string filename;
  std::cout << "Enter filename:";
  std::cin >> filename;

  std::ifstream fin(filename.c_str());

  std::string s;
  std::map<std::string,unsigned> freq;

  while (fin >> s) freq[s]++;

  for (auto i : freq)
    std::cout<<std::setw(12)<< i.first
             <<std::setw(4) << ':'
             <<std::setw(12)<< i.second
             <<std::endl;
}
```

A quick histogram!

- `std::map<string,int>` is a funny container which stores integer values, but accesses them through `std::string` keys.
- The iterator for `std::map` "points to" a `pair<key,value>`
- `std::setw(12)` is to set the width of the next output operation to 12

New container class in C++11: std::array

- `std::array<T,N>` is a fixed length array of size N holding elements of type `T`
- It implements functions like `begin()` and `end()` and is therefore usable with STL algorithms like `transform`, `generate` etc.
- The array size is a template parameter, and hence a compile time constant.
- `std::array<std::string,7> week{"Mon","Tue","Wed","Thu","Fri","Sat","Sun"};`
- `std::array<DataType,int>` is therefore equivalent to the raw C-style fixed length arrays, and should be used when fixed length arrays are needed. Depending on the application, they may be a bit faster than a vector

std::unordered_map and std::unordered_set

- Like `std::map<k,v>` and `std::set<v>`, but do not sort the elements
- Internally, these are hash based containers, providing faster element access than `std::map` and `std::set`
- Additional template arguments to specify hash functions

STL algorithms

The similarity of the interface, e.g. `begin()`, `end()` etc., among STL containers allows generic algorithms to be written as template functions, performing common tasks on collections

```
...
std::vector<YourClass> vc(inp.size());
std::copy(inp.begin(),inp.end(),
          vc.begin());
//Copy contents of list to a vector
auto pos = std::find(vc.begin(),vc.end(),
                     elm);
//Find an element in vc which equals elm
std::sort(vc.begin(),vc.end());
//Sort the vector vc. The operator "<"
//must be defined
...
for (auto & el : inp) {
  std::cout << el << std::endl;
}
// Do something "for each" element of a
// container
```


JÜLICH FORSCHUNGSZENTRUM

STL algorithms

Exercise 3.7:

- The standard libary provides a large number of template functions to work with containers
- Look them up in `www.cplusplus.com` or `en.cppreference.com`
- In a program, define a vector of integers 1..9
- Use the suitable STL algorithms to generate successive permutations of the vector

STL algorithms: sorting

- `std::sort(iter_1, iter_2)` sorts the elements between iterators `iter_1` and `iter_2`
- `std::sort(iter_1, iter_2, less_than)` sorts the elements between iterators `iter_1` and `iter_2` using a custom comparison method `less_than`, which could be any callable object

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
struct myless
{
  bool operator()(int i, int j)
  {
    return (i*i < j*j);
  }
};
int main()
{
  vector<int> v{2,-3,7,4,-1,9,0};
  sort(v.begin(),v.end());
  //Sort using "<" operator
  for (auto el : v) cout << el << endl;
  sort(v.begin(),v.end(),myless());
  //Sort using custom comparison
  for (auto el: v) cout << el << endl;
}
```


std::transform

- `std::transform(begin_1 , end_1, begin_res, unary_function);`
- `std::transform(begin_1 , end_1, begin_2, begin_res, binary_function);`
- Apply callable object to the sequence and write result starting at a given iterator location
- The container holding result must previously been resized so that it has the right number of elements
- The "result" container can be (one of the) input container(s)

```
std::vector<double> v{0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9};
std::list<double> l1(v.size(),0),l2(v.size(),0);
std::transform(v.begin(),v.end(),l1.begin(),sin);
std::transform(v.begin(),v.end(),l1.begin(),l2.begin(),std::max);
```

Result: `l1` contains `sin(x)` for each `x` in `v`, and `l2` contains the `greater(x,sin(x))`

New algorithms in C++11: all_of and any_of

```
bool valid(std::string name)
{
  return all_of(name.begin(),name.end(),
                [](char c) {return (isalpha(c))||isspace(c);});
}
```

- `std::all_of(begin_ , end_ , condition)` checks if all elements in a given range satisfy `condition`
- `condition` is a callable object
- `std::any_of(begin_ , end_ , condition)` checks if any single element in a given range satisfies `condition`

Lambda Functions

- Purpose
- Basic syntax
- Uses
  - Effective use of STL
  - Initialisation of const
  - Concurrency
  - New loop styles

Herb Sutter: Get used to }); : you will be seeing it a lot!

Motivation

```
struct Person {string firstname,lastname;
       size_t age;};
class Community {
    std::vector<Person> thepeople;
...
    template <class F>
    vector<Person> select(F f)
    {
        vector<Person> ans;
        for (auto p : thepeople) {
            if (f(p)) ans.push_back(p);
        }
        return ans;
    }
};
...
void display_list(Community &c)
{
   size_t ma;
   std::cout<<"Minimum age: ";
   std::cin >> ma;
   // We want to select all Persons with
        age >= ma
   auto v = c.select(???);
}
```

What should we pass as an argument to `c.select()` ?

Motivation

```
struct Person {string firstname,lastname;
       size_t age;};
class Community {
  std::vector<Person> thepeople;
  template <class F>
  vector<Person> select(F f)
  {
    vector<Person> ans;
    for (auto p : thepeople) {
      if (f(p)) ans.push_back(p);
    }
    return ans;
  }
};
```

```
void display_list(Community &c)
{
  size_t ma;
  std::cout<<"Minimum age: ";
  std::cin >> ma;
  // We want to select all Persons with
  // age >= ma. Perhaps ...
  bool filter(Person p) {
  // Error! Nested function!
    return p.age>=ma;
  }
  auto v = c.select(filter);
}
```

A locally defined function aware of the variable `ma` would be perfect. But a nested function definition is **not allowed** in C++.

Motivation

```
void display_list(Community &c)
{
   size_t ma;
   std::cout<<"Minimum age: ";
   std::cin >> ma;
   // perhaps ...
   bool filter(Person p) { // Error!
       return p.age>=ma;
   }
   // Besides,...
   ma=std::max(legal_minimum_age(),ma);
   auto v = c.select(filter);
   //What value of ma should filter see ?
}
```

Besides, should the locally defined function use the value of `ma` when the function was defined, or when it was passed to `c.select()`?

We want something like the local function here, but with more control over how it sees its environment.

Motivation

```
class Filter {
    public:
    Filter(size_t &co) : cutoff(co){}
    bool operator()(Person p) {
        return p.age>=cutoff;
    }
    private:
    cutoff=0;
};
void display_list(Community &c)
{
   size_t ma;
   std::cout<<"Minimum age: ";
   std::cin >> ma;
   ma=std::max(legal_minimum_age(),ma);
   Filter filter{ma};
   auto v = c.select(filter);
}
```

A function object class, or a functor can certainly solve the problem. But,

- it is cumbersome to write a new functor for every new kind of filter.
- it takes the logic of what is done in the filter out of the local context
- it wastes possibly useful names

Enter lambda functions

```
void display_list(Community &c)
{
   size_t ma;
   std::cout<<"Minimum age: ";
   std::cin >> ma;
   ma=std::max(legal_minimum_age(),ma);
   auto v = c.select([ma](Person p){
       return p.age>=ma;
   });
}
```

- "Captures" `ma` by value and uses it in a locally defined function-like-thing
- We control how it sees variables in its context
- Keeps the logic of the filter in the local context

Enter lambda functions

Example 3.2:

The program `community.cc` has exactly the code used in this explanation. Check that it works! Modify so that the selection is based on the age and a maximum number of characters in the name.

Lambda Functions: Syntax

[capture](arguments)mutable->return_type{body}

Examples

- `[ ](int a, int b)->bool{return a>b;}`
- `[=](int a)->bool{return a>somevar;}`
- `[&](int a){somevar += a;}`
- `[=,&somevar](int a){somevar+=max(a,othervar);}`
- `[a,&b]{f(a,b);}`

- The optional keyword **mutable** allows variables captured by value to be changed inside the lambda function
- The return type is optional if there is one return statement
- Function arguments field is optional if empty

Lambda Functions: captures

- Imagine there is a variable **int** p=5 defined previously
- We can "capture" p by value and use it inside our lambda

```
auto L=[p](int i){std::cout << i*3+p;};
L(3); // result : prints out 14
auto M=[p](int i){p=i*3;}; // syntax error! p is read-only!
```

- We can capture p by value (make a copy), but use the **mutable** keyword, to let the lambda function change its local copy of p

```
auto M=[p](int i)mutable{return p+=i*3;};
std::cout<<M(1)<<" ";std::cout<<M(2)<<" ";std::cout<<p<<"\n";
// result : prints out "8 14 5"
```

- We can capture p by reference and modify it

```
auto M=[&p](int i){return p+=i*3;};
std::cout<<M(1)<<" ";std::cout<<M(2)<<" ";std::cout<<p<<"\n";
// result : prints out "8 14 14"
```

No default capture!

`[]`	Capture nothing
`[=]`	Capture all by value (copy)
`[=,&x]`	Capture all by value, except x by reference
`[&]`	Capture all by reference
`[&,x]`	Capture all by reference, except x by value

- A lambda with empty capture brackets is like a local function, and can be assigned to a regular function pointer. It is not aware of identifiers defined previously in its context
- When you use a variable defined outside the lambda in the lambda, you have to capture it

Exercises

Exercise 3.8:

The program `captures.cc` demonstrates capturing variables in the surrounding scope of a lambda function. Compile and run. Observe the differences between capturing by value with and without **`mutable`**, and capturing by reference.

Exercise 3.9:

The program `lambda_1.cc` shows one way to initialise a list of integers to {1,2,3,4,5... }. Modify to initialise the list to the Fibonacci sequence.

Exercises

Exercise 3.10:

The program `sort_various.cc` defines an array of strings and prints them sorted (using `std::sort`) in various ways:

- alphabetical order using the default string comparison
- according to the lengths of the strings, by passing a lambda function to `std::sort`
- alphabetical order of back-to-front strings.

The third part is left for you to fill in. Use the algorithm `std::reverse` to reverse the strings locally in your lambda function.

Exercises

Exercise 3.11:

The program `sort_complex.cc` defines an array of complex numbers, and demonstrates how to sort it by their real parts using a lambda function as the comparison function. Add code to subsequently print the list sorted according their absolute difference from a given complex number.

The type of a lambda function

- ```
  auto comp=[](int a, int b){return a>b;};
  std::map<int,int,???> m{comp};
  ```

  What type is our comparison function above ? What should one write as the third template parameter ?

- Use `decltype` :

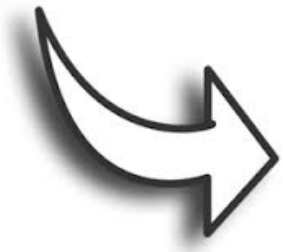
  ```
  std::map<int,int,decltype(comp)> m{comp};
  ```

  Tell the compiler: "You know the type of the above mentioned variable `comp` ? I don't want to see it, but just use that type here as the third template argument".

Lambda Functions and Functors

Lamdas are automatically generated functors

[capture](arguments)->return_type{body}



```
class __local_functor {
private:
    CaptureTypes __captures;
public:
...
    auto operator()(arguments)->return_type {body}
};
```
```

STL algorithms with Lambda Functions

- Many STL algorithms implement general patterns which occur frequently in different kinds of programs: `for_each`, `find_if`, `copy_if` etc.
- Code using them can be clear in its intent, less error prone, and sometimes more efficient.
- But for good reasons many programmers found it far more convenient to write hand written loops

Annoyance while using algorithms

```
#include <iostream>
#include <algorithm>
using namespace std;
struct myless
{
  bool operator()(double x,double y)
  {
    return (sin(x) < sin(y));
  }
};
struct is_valid
{
  bool operator()(int x)
  {
    return (x>=-2.718281828 &&
            x<=2.718281828);
  }
};
```

```
void print(double x)
{
  cout << x << " ";
}
int main()
{
  vector<double> v{0.11,0.07,0.34,
                   0.81,1.28,0.98,2.72,
                   8.45,0.65,11.9};
  auto it=max_element(v.begin(),v.end());
  for_each(v.begin(),it,print);
  cout << "\n";
  sort(v.begin(),v.end(),myless);
  vector<double> sel;
  copy_if(v.begin(),v.end(),
          back_inserter(sel),is_valid);
  for_each(sel.begin(),sel.end(),print);
}
```

- Waste of useful names for ad-hoc tasks which may be used only once
- "Algorithms take logic of these ad-hoc tasks out of context"

Using algorithms with lambdas

```
int main()
{
  vector<double> v{0.11,0.07,0.34,
                   0.81,1.28,0.98,2.72,
                   8.45,0.65,11.9};
  auto it=max_element(v.begin(),v.end());
  auto print = [](double x) {
      cout << x << " ";
  };
  for_each(v.begin(),it,print);
  cout << "\n";
  sort(v.begin(),v.end(),
       [](double x, double y){
      return sin(x)<sin(y);
  });
  vector<double> sel;
  copy_if(v.begin(),v.end(),
        back_inserter(sel),
        [](double x) {
        return x>=-2.718281828 &&
               x<=2.718281828;
  });
  for_each(v.begin(),it,print);
}
```

- ~~"Algorithms take logic of these ad-hoc tasks out of context"~~
- Using lambdas, we can keep the logic local
- No names need to be wasted
- Did you notice the }); ?

Using algorithms with lambdas

Another example from Sutter's talk ... Find the first element in v in the interval (x,y) and do something with it...

- Hand written loop ...

```
auto i = v.begin(); // need it later
for (;i!=v.end();++i) if (*i>x && *i<y) break;
```

- STL C++03 without writing your own functor

```
auto i = std::find_if(v.begin(),v.end(),
                      compose2(logical_and<bool>(), // non-standard!
                      bind2nd(greater<int>(),x), bind2nd(less<int>(),y)));
```

- STL C++11 without lambdas

```
auto i = std::find_if(v.begin(),v.end(), bind(logical_and<bool>(),
                      bind(greater<int>(),_1,x), bind(less<int>(),_1,y)));
```

- STL C++11 with lambdas

```
  auto i = std::find_if(v.begin(),v.end(),[x,y](int i) {return i>x && i<y;});
```

Annoyance while using algorithms

Most loop bodies are not one-liners

- Loop bodies often contain many lines of complex code and operate on variables defined outside the loop

```
for (auto i=v.begin();i!=v.end();++i) {
...
}
```

- It was awkward to change such code using algorithms

```
// somewhere else, may be in another file...
class somefunctor {
public:
    somefunctor(VarTypesINeed varsINeed) : ... {}
    return_type operator()(int) {
    ...
    }
private:
    VarTypesINeed locVarsINeed;
};
// only then can you write this when you need ...
std::for_each(v.begin(),v.end(),somefunctor);
```


Annoyance while using algorithms

Most loop bodies are not one-liners

- Loop bodies often contain many lines of complex code and operate on variables defined outside the loop

```
for (auto i=v.begin();i!=v.end();++i) {
...
...
...
}
```

- It was awkward to change such code using algorithms
- Not any more!

```
std::for_each(v.begin(),v.end(),[whatINeed](int i){
...
...
...
});
```

- A lambda function can contain arbitrary amount of code, and has a sophisticated way to refer to its environment.

Using algorithms with lambdas

```
void show_freqs(std::vector<string> sv)
{
    vector<size_t> v;
    transform(sv.cbegin(),sv.cend(),back_inserter(v),
              [](string s) {
        s = s.substr(4);//ignore "val=" at the start
        return stoul(s);
    });
    ...
    map<size_t,size_t> H;
    using btype = decltype(H)::value_type;
    for (auto i : v) H[i]++;
    auto max_elem = max_element(H.begin(),H.end(),
                                [](btype a,btype b) {
        return a.second < b.second;
    });
    for_each(H.begin(),H.end(),
             [max_elem](pair<size_t,size_t> i) {
        size_t nchars=(i.second*100.0/
                       max_elem->second);
        cout << setw(10) << i.first << "| "
             << string (nchars,'#')<<"\n";
    });
}
```

- Algorithms like `transform` and `for_each` look much more like the loops they really are.
- Notice the formatting trick used for lambdas
- Did you notice the }); ?

Algorithms with lambdas

- Prefer algorithms to hand-written loops because of:
  - Performance
  - Correctness
  - Clarity

Your own new loop structures

- Since lambdas work with any length of function bodies, algorithms you write can be used as new "loop structures". Suppose you wrote a functional to do something on every nth element in a range...

```
template<typename I,typename F>
void for_each_nth(I & begin, I & end, unsigned s, F &f)
{
...
}
```

- With lambda functions, this becomes a new loop structure for your programs!

```
// For every other element in v
for_each_nth(v.begin(),v.end(),2,[](int i){
    //whatever you want to do with every other element of v!
});
```


Asynchronous tasks

```
double foo(double x)
{
    // hard calculation
}
double bar(double x)
{
    // another hard calculation
}
vector<double> haha(vector<double> v)
{
    for (size_t i=0;i<v.size();++i) {
        v[i]=max(foo(v[i]),bar(v[i]));
    }
    return v;
}
```

- Example: independent operations to be performed on each element of an array

Asynchronous tasks

```
double foo(double x)
{
    // hard calculation
}
double bar(double x)
{
    // another hard calculation
}
vector<double> haha(vector<double> v)
{
    tbb::parallel_for(0,v.size(),1,
                      [&v](size_t i){
        v[i]=max(foo(v[i]),bar(v[i]));
    });
    return v;
}
```

- Example: independent operations to be performed on each element of an array
- With minimal changes, one can wrap the required range of code lines in a lambda function, and use a `parallel_for` loop.

A lot more like this in the TBB session!

Initialisation of a const

```
vector<double> V;
ifstream fin{"atoms.dat"};
double par=0;
while (fin>>par) V.push_back(par);
// But V should be an array of constant parameters!
...
for (size_t i=0;i<V.size();++i) {
    a[i]=6*V[i]*V[i]-5*(V[i]+=1); // We want to prevent such errors!
}
```

How does one initialise a variable with arbitrarily many lines of code, and then have it behave like a **`const`** for the rest of the program ?

Initialisation of a const

```
vector<double> Vtmp;
ifstream fin{"atoms.dat"};
double par=0;
while (fin>>par) Vtmp.push_back(par);
const vector<double> V=Vtmp;
```

Works, but wastes a couple of names.

Initialisation of a const

```
const vector<double> V=[=]{
    ifstream fin{"atoms.dat"};
    vector<double> ans;
    double par=0;
    while (fin>>par) ans.push_back(par);
    return ans;
}();
// V is an array of constant parameters!
for (auto d : V) cout << d << "\n";
//V[2]+=0.3;      // wont compile!
//V.push_back(2); // wont compile!
```

Elegant and leaves the namespace outside the initialisation block unpoluted.

Exercise 3.12:

Use the `generate_n` algorithm and the `back_inserter` convenience function to initialise an `std::vector<double>` of N random values distributed uniformly in the range $-\pi$ to π. For this exercise, you can use the `rand()` function to generate the random numbers.

Exercise 3.13:

Write functions to calculate the mean and median of a container of **`double`** data, using the `accumulate` and the `nth_element` algorithms respectively.

Exercise 3.14: Monoalphabetic substitution cipher

Monoalphabetic substitution cipher is an ancient encryption method. In this method, a map is constructed of the letters of the alphabet with a scrambled version of the alphabet:

Original	a b c d e f g h i j k l m n o p q r s t u v w x y z
Cipher	d m t w g f x v j z r n e l k u o q b s c p a i h y

For instance, with the above substitution table, the word "ancient" will be enciphered as "dltjgls". (continued on next slide ...)

Exercise 3.14: (continued ...)

For convenience, the scrambled order can be generated by taking a passphrase, deleting repetitions, and giving the remaining characters the first positions. The rest of the positions can be given to the remaining characters in their own order. For instance, if the passphrase is "apples are tasty", we would start by removing spaces and repetitions to get "aplesrty". Our scrambled alphabet will then be, "aplesrtybcdfghijkmnoquvwxz". (continued on next slide ...)

JÜLICH FORSCHUNGSZENTRUM

Exercise 3.14: (continued ...)

Write a program which

- takes the name of a text file as a command line parameter
- asks the user to enter a passphrase
- produces a scrambled alphabet (using both capital and small letters) with the passphrase according to the above scheme
- enciphers/deciphers the text according to the above method
- saves the result in a new file

For this exercise, you can leave punctuation and white space untouched.

Note: Monoalphabetic substition cipher is very easily broken by frequency analysis without any need of the key.

Day 4

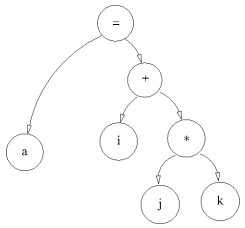
Sub-expression evaluation

```
int i=0;
std::cout << ++i <<'\t'<< i <<'\t' << i++ <<"\n";
// clang++ prints 1 1 1, g++ prints 2 2 0 although both parse it as
// cout.operator<<(exp1).operator<<('\t').operator<<(i).operator<<('\t')
// .operator(exp2).operator("\n");
```

- We stumbled upon it in our lambda example, but the issue is more elementary

Sub-expression evaluation

```
a = i+j*k;
```



- A statement or expression in C++ ends with a semi-colon ";"
- An expression may be parsed as a tree of subexpressions
- Subexpression evaluation order is governed by well-defined, but non-trivial rules [1]

[1] http://en.cppreference.com/w/cpp/language/eval_order

Sub-expression evaluation I

```
std::cout << ++i <<'\t'<< i <<'\t' << i++ <<"\n";
// clang++ prints 1 1 1, g++ prints 2 2 0 although both parse it as
// cout.op<<(exp1).op<<('\t').op<<(i).op<<('\t').op(exp2).op("\n");
```

1. Each value and side-effect of a full expression is sequenced before those of the next full expression
2. Value computations (but not side effects!) of operands to an operator are sequenced before the value computation of the result of the operator
3. Every value and side-effect evaluation of all arguments of a function are sequenced before the function body
4. Value calculations of built-in postincrement (`i++`) and postdecrement (`i--`) operators are sequenced before their side effects

Sub-expression evaluation II

5. Side effect calculations of built-in preincrement (`++i`) and predecrement (`--i`) operators are sequenced before their values
6. For built-in logical AND and OR operators (`&&` and `||`) the value as well as side effects of the LHS are sequenced before those of the RHS
7. For `cond?then:otherwise` ternary operator, value computation of `cond` is sequenced before both value and side effects of the `then` and `otherwise`
8. Modification of the LHS in an assignment operation is sequenced after the value calculation of both LHS and RHS, but ahead of the value calculation of the whole assignment expression

Sub-expression evaluation III

9. Every value and side effect calculation of the LHS of a built-in comma operator, such as in
   `for (i=L.begin(),j=i;i!=L.end();++i,++j)`
   is sequenced before those of the RHS
10. In comma separated initializer lists, value and side effects of each initializer is sequenced before those of the initializers to its right

Evaluation order: examples

```
i = ++i + i++; // undefined behaviour
i = i++ + 1; // undefined behaviour, although i= ++i +1 is well defined
f(++i,++i); // undefined behaviour
std::cout << i <<' '<<i++<<'\n'; // undefined behaviour!
a[i]=i++; //  undefined behaviour!
```

Example 4.1: Beware of unsequenced operations!

The file `examples/unsequenced.cc` illustrates all the above examples and comma separated initializer lists. Compilers go give warnings in this kind of simple situations. Use both the GNU and clang compilers to compile the program, and compare the behaviour. Keep this example in mind and avoid bugs like the one in our lambda example.

Default arguments

```
std::vector<std::string> split(std::string S, char d=' ');
// Split strings by a given character d. Use ' ' for d if d is not given
std::string st{"It is quite warm, although not sunny"};
auto v = split(st,',');
auto w = split(st);
// v=["It is quite warm","although not sunny"]
// w=["It","is","quite","warm,","although","not","sunny"]
```

- Functions and templates can have "default" values for one or more arguments
- Default values are assigned at the point of declaration
- Parameters with default values must follow mandatory parameters

Nested templates

```
template <template <typename,typename> class Container, typename Type>
std::ostream & operator<<(std::ostream & os,
                         const Container<Type,std::allocator<Type>> & v)
{
  for (auto & el : v) os << el << '\n';
  return os;
}
int main()
{
  std::vector<int> V{0,1,2,2,1};
  std::cout <<"vector<int>\n"<< V << '\n';
  std::list<double> D{0.0,3.1415926,2.71828};
  std::cout <<"list<double>\n"<< D << '\n';
}
```

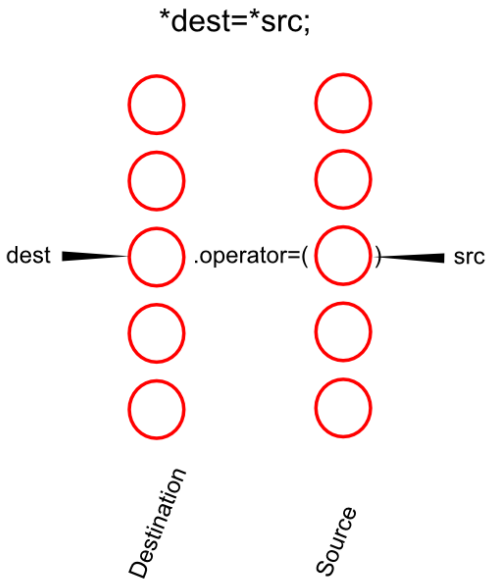
- The first template parameter is itself a template
- Possible to use **auto** etc in the template function or class code
- Example: `examples/temptemp.cc`

Iterator adaptors

The copy algorithm

```
template <typename InItr,typename OutItr>
void copy(InItr src, InItr src_end,
          OutItr dest)
{
  while (src!=src_end) {
    *dest = *src;
    ++src,++dest;
  }
}
```

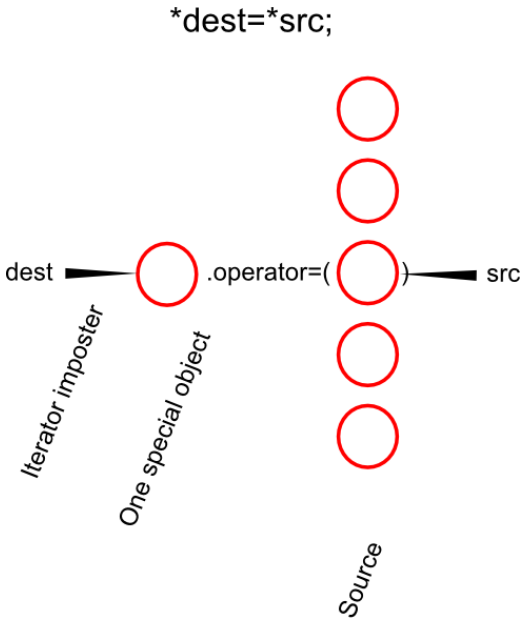
- Move iterators along source and destination sequences
- Get objects at the iterators
- Call **operator**=() on LHS with RHS as argument



The copy algorithm

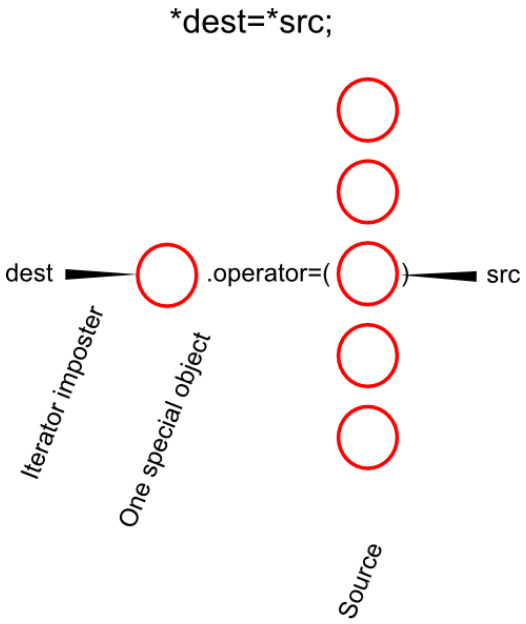
```
struct FakeItr {
  FakeItr &operator++(int){return *this;}
  FakeItr &operator++(){return *this;}
  Proxy operator*() {return prx;}
};
```

- What if, the destination "iterator" does not really browse any sequence, but simply returns one special object with unary **operator***() ?
- ... and that special object has **operator**=() defined with RHS as argument ?



The copy algorithm

```
struct Proxy {
  template <typename T>
  OutputProxy& operator=(const T &t) {
    std::cout<<"Output by proxy: "
             << t <<"\n";
    return *this;
  }
};
```



- The `copy` algorithm then becomes a way to call `prx.operator=(obj)` for each object in the source sequence
- ... and that function need not actually assign anything!

Iterator imposter

Example 4.2: Our iterator imposter

Our very own iterator imposter is in the file `examples/copy_misuse.cc`. See for yourself that this works!

... but, WHY ?

- Our `copy` function assumes that the destination container has enough space
- You must resize them ahead of using them
- ... or, perhaps you could create an imposter, which calls `push_back` on the destination container in its **`operator=`** ?

```
template <typename InItr,
          typename OutItr>
void copy(InItr src, InItr src_end,
          OutItr dest)
{
  while (src!=src_end) {
    *dest++ = *src++;
  }
}
...
std::list<int> l{1,2,3,4,5};
std::list<int> m;
copy(l.begin(),l.end(),m.begin());
//segfault
```


Iterator adaptors

```
#include <iterator>
...
list<int> l{1,2,3,4,5};
list<int> m;
copy(l.begin(),l.end(), back_inserter(m));
copy(m.begin(),m.end(), ostream_iterator<int>(cout," "));
```

- `std::back_insert_iterator` is an "iterator adaptor", which uses a container's own `push_back` function to insert elements
- The convenience function `std::back_inserter` simplifies handling
- Similarly `std::ostream_iterator` is an iterator adaptor, which simulates an output sequence on an output stream

Exercise 4.1:

Modify the program `exercises/transform.cc` to use an iterator adaptor to feed the results of `std::transform` into the results array, instead of setting the size in the beginning.

Exercise 4.2:

Read a file into a list of strings (one for each line), using `std::copy` and an appropriately designed iterator imposter and a line proxy class.

Bind

Binders and adaptors

```
//examples/binddemo.cc
#include <iostream>
#include <functional>
using namespace std::placeholders;
int add(int i, int j)
{
  std::cout << "add: received arguments "<<i<<" and "<<j<<'\n';
  return i+j;
}
int main()
{
  auto a10 = std::bind(add,10,_1); // new functional a10, which calls add(10,_1)
  // where _1 is the argument passed to a10
  std::cout << a10(1) << '\n'; // call add(10,1)
  std::cout << a10(2) << '\n'; // call add(10,2)
  std::cout << a10(3) << '\n';
}
```

- `std::bind(C,...)` creates a new function object with modified properties

Example 4.3: std::bind

Experiment with the program `binddemo.cc` to understand how it is used. How would you modify the code so that the newly created functional calls a specific member function of a class istead of a free standing function ?

Random numbers

Random number generation

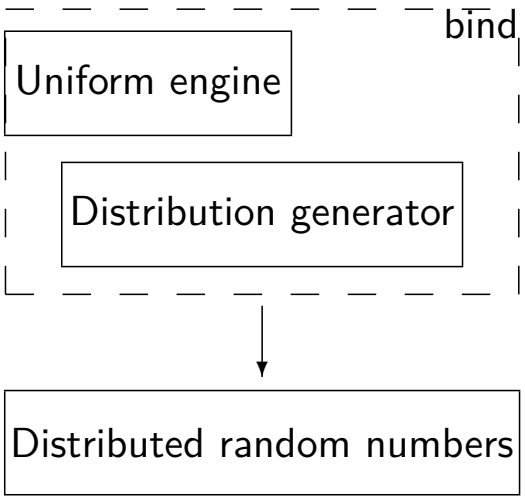
- Old `rand()` , `erand()`, `drand()` etc had poor properties with regard to correlations
- New random number classes in the standard library replace and vastly supersede the old functions

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Figure : Source XKCD: `http://xkcd.com`

Random number generation

- Share a common structure
- There is an underlying simple uniform random generator engine with (hopefully) well tested properties
- There is a distribution generator which produces random numbers of a required distribution, using the underlying uniform generator engine



Random number generators

```
#include <random>
#include <functional>
#include <iostream>
#include <map>
int main()
{
    std::poisson_distribution<> distribution(8.5);
    std::mt19937 engine;
    auto generator = std::bind(distribution,engine);
    std::map<int,unsigned> H;
    for (unsigned i=0;i<5000000;++i) H[generator()]++;
    for (auto & i : H) std::cout<<i.first<<" "<<i.second<<'\n';
}
```

- `std::mt19937` is an implementation of Mersenne Twister 19937
- The template `std::poisson_distribution` is a functional implementing the Poission distribution
- `std::bind` "binds" the first argument of `distribution` to `engine` and returns a new function object

Random number generators

```
std::normal_distribution<> G{3.5,1.2}; // Gaussian mu=3.5, sig=1.2
std::uniform_real_distribution<> U{3.141,6.282};
std::binomial_distribution<> B{13};
std::discrete_distribution<> dist{0.3,0.2,0.2,0.1,0.1,0.1};
// The following is an engine like std::mt19937, but is non-deterministic
std::random_device seed; // int i= seed() will be a random integer
```

- Lots of useful distributions available in the standard
- With one or two lines of code, it is possible to create a high quality generator with good properties and the desired distribution
- `std::random_device` is a non-deterministic random number generator.
  - It is good for setting seeds for the used random number engine
  - It is slower than the pseudo-random number generators

Random number generator: Exercises

Exercise 4.3:

Make a program to generate normal distributed random numbers with user specified mean and variance, and make a histogram to demonstrate that the correct distribution is produced.

Exercise 4.4:

Make a program to implement a "biased die", i.e., with user specified non-uniform probability for different faces. You will need `std::discrete_distribution<>`

Exercise 4.5: Random Shuffle

`std::random_shuffle` implements Knuth's shuffle algorithm: (to shuffle an array `A` of `n` elements with indexes `0..n-1`

```
for i = n-1 down to 1 do
  j=random integer in range 0<=j<=i
  exchange A[j] and A[i]
```

- Use this algorithm along with a suitable lambda function to implement all permutations of the string "abcdefg". Use an std :: map<string,**unsigned**> object to make a histogram of the permutations, and verify that indeed the algorithm produces all permutations with equal probability.
- Instead of using the STL algorithm, write your own template function to do this. What happens when you replace 0<=j<=i with 0<=j<i ?

Discussion: Random permutations

- Works for an arbitrary vector like container of arbitrary elements
- Takes a range of iterators, and a callable object taking an unsigned integer type and returning a random smaller integer.
- We use a lambda function for the job
- We use a `map<string,unsigned>` to make a histogram

```
// File examples/shuffle.cc
#include <iostream>
#include <random>
#include <algorithm>
#include <map>
int main()
{
  std::random_device seed;
  std::mt19937 mt(seed());
  std::uniform_real_distribution<> dist;
  auto gen = std::bind(dist,mt);
  std::string v{"abcdefg"};
  std::map<std::string,unsigned> H;
  for (int j=0;j<1000000;++j) {
    std::random_shuffle(v.begin(),
                        v.end(),
                    [&gen](size_t i)
      {return (size_t)(gen()*i);});
    H[v]++;
  }
  for (auto val : H)
     std::cout << val.first << " "
               << val.second << '\n';
}
```

Chrono

The time library

Measure time intervals in the program

- **namespace** `std::chrono` defines many time related functions and classes
- `system_clock`: System clock
- `steady_clock`: Steady monotonic clock
- `high_resolution_clock`: To the precision of your computer's clock
- `steady_clock::now()` : nanoseconds since 1.1.1970
- `duration<`**`double`**`>`: Abstraction for a time duration. Uses `std::ratio<>`internally

The time library

Example code

```
#include <iostream>
#include <chrono>
#include <vector>

using namespace std::chrono;
bool is_prime(unsigned n);

int main()
{
  std::vector<unsigned> primes;
  auto t = steady_clock::now();
  for (unsigned i=0;i<10000;++i) {
    if (is_prime(i)) primes.push_back(i);
  }
  std::cout << "Primes till 10000 are ... " << '\n';
  for (unsigned i : primes) std::cout << i << '\n';
  nanoseconds d = steady_clock::now() -t;
  std::cout<<"Prime search took "<<duration<double>(d).count()<<" seconds"<<'\n';
  std::cout<<"Prime search took "<<d<<" nanoseconds"<<'\n';
}
```


Regular expressions

Regular expressions

examples/regex1.cc

```
//examples/regex1.cc
#include <iostream>
#include <string>
#include <regex>
int main()
{
  std::string regexstr
              {R"(\b(?:\d[ -]*?){13,16}\b)"};
  std::regex rex{regexstr};
  std::string crd;
  bool valid=false;
  do {
    std::cout <<"Credit card number: ";
    getline(std::cin, crd);
    if (not (valid=std::regex_match(crd,rex))) {
      std::cerr << "Invalid format.\n";
    }
  } while (not valid);
}
```

- `std::regex` represents regular expressions, created from an ordinary string describing it
- Use raw strings as shown to avoid multiple \ symbols
- Functions like `std::regex_match`, `std::regex_search` provide very powerful means of string processing

Regular expressions

examples/regex2.cc

```
// There is a typedef smatch which specializes
// match_results for string

int main()
{
  std::string line;
  std::regex pat
              {"^Subject: (Re: |Aw: )*(.*)"};

  while (std::cin) {
    std::getline(std::cin, line);
    std::smatch matches;
    if (std::regex_match(line, matches, pat))
      std::cout << matches[2] << '\n';
  }
}
```

- `regex_match` with an `smatch` argument stores marked subexpressions
- Index 0 stores the entire match. Index 1.. store subexpressions identified by ()
- Test on the sample mail or one of your mails

Regular expressions

A note on sub-expressions

- Parentheses () in a regular expression are used to create sub-fields within a match
- The `match_results` template stores the sub-fields like this:
  - `m.prefix()`: the text before the match
  - `m.suffix()`: the text following the match
  - `m[0]`: all of the match
  - `m[1]..m[N]` : sub-fields 1 to N

Example 4.4: Regular expression

Example codes `regex1.cc`, `regex2.cc` and `regex_match.cc` demostrate the use of the new STL regex library.

Smart pointers

Smart pointers

Figure : Source: XKCD (`http://xkcd.com`)

- 3 kinds of smart pointers were introduced in C++11
- They allow memory managed programs to be written
- Ordinary pointers are also allowed, but cost a lot of time in terms of debugging.

Unique pointer

```
// examples/uniqueptr.cc
std::unique_ptr<MyStruct> get_my_struct(int p)
{
  return std::unique_ptr<MyStruct> {new MyStruct(p)};
}
int main()
{
  auto u1=get_my_struct(1);
  //std::unique_ptr<MyStruct> u2 = u1; //won't compile
  std::unique_ptr<MyStruct> u3 = std::move(u1);
  std::cout << "Data value for u3 is u3->vl = "
            << u3->vl <<'\n';
}
```

- Exclusive access to resource
- The data pointed to is freed when the pointer expires
- Can not be copied
- Data ownership can be transferred with `std::move`

Shared pointer

```
// examples/sharedptr.cc
int main()
{
  auto u1=std::make_shared<MyStruct>(1);
  std::shared_ptr<MyStruct> u2 = u1; // Copy is ok
  std::shared_ptr<MyStruct> u3 = std::move(u1);
  std::cout << "Reference count of u3 is "
            << u3.use_count() << '\n';
}
```

- Can share resource with other shared/weak pointers
- The data pointed to is freed when the pointer expires
- Can be copy assigned/constructed
- Maintains a reference count `ptr.use_count()`

Weak pointer

```
// examples/weakptr.cc
int main()
{
  auto s1=std::make_shared<MyStruct>(1);
  std::weak_ptr<MyStruct> w1(s1);
  std::cout << "Ref count of s1 = "
            <<s1.use_count()<<'\n';
  std::shared_ptr<MyStruct> s3(s1);
  std::cout << "Ref count of s1 = "
            <<s1.use_count()<<'\n';
}
```

- Does not own resource
- Can "kind of" share data with shared pointers, but does not change reference count

Smart pointers: examples

Example 4.5: uniqueptr.cc, sharedptr.cc, weakptr.cc

Read the 3 smart pointer example files, and try to understand the output. Observe when the constructors and destructors for the data objects are being called.

Custom deleters

```
shared_ptr<string> p{new string{"First"},[](string *p){
    cout << "delete "
    << *p << endl;
    delete p;
}};
p2=p;
p=shared_ptr<string>{new string{"Second"}};
```

- Called when the object pointed to is destroyed
- Attaches deleter to the object created

examples/sharedptr_custom_deleter.cc

Custom deleters: Arrays

```
shared_ptr<int> p{new int[10]}; //bad
shared_ptr<int> p{new int[10], [](int *p) {delete [] p;}};
```

- By default, when reference count becomes 0, a `shared_ptr` deletes the object pointed to, but it uses **`delete`** `p` rather than **`delete`** `[] p`
- In case you wish to handle an array with a `shared_ptr`, provide your own deleter

examples/sharedptr_custom_deleter.cc

Smart pointers: summary

- `unique_ptr<T>` is "smart" only in not allowing copying. It does not need additional data members to do any book keeping. It can be implemented very efficiently.
- `shared_ptr<T>` and `weak_ptr<T>` need additional data members like reference counters. The consequent small performance penalty should be kept in mind.
- `auto_ptr<T>` from C++98 is deprecated in C++11 and should not be used.

Variadic templates

Variadic templates

```
template <typename ... Args>
int countArgs(Args ... args)
{
  return (sizeof ...args);
}

std::cout<<"Num args="<<countArgs(1,"one","ein","uno",3.232)<<'\n';
```

- Templates with arbitrary number of arguments
- Typical use: template meta-programming
- Recursion, partial specialisation
- The ... is actual code! Not blanks for you to fill in!

Parameter pack

- The ellipsis (...) template argument is called a parameter pack [2]
- It represents 0 or more arguments which could be type names, integers or other templates :

```
template <typename ... Args> class mytuple;
// The above can be instantiated with :
mytuple<int,int,double,string> t1;
mytuple<int> t2;
mytuple<> t3;
```

- Definition: A template with at least one parameter pack is called a variadic template

[2] http://en.cppreference.com/w/cpp/language/parameter_pack

Parameter pack

```
//examples/variadic_1.cc
template <typename ... Types> void f(Types ... args);
template <typename Type1, typename ... Types> void f(Type1 arg1, Types ... rest) {
  std::cout <<typeid(arg1).name()<<": "<< arg1 << "\n";
  f(rest ...);
}
template <> void f() {}
int main()
{
  int i{3},j{};
  const char * cst{"abc"};
  std::string cppst{"def"};
  f(i,j,true,k,l,cst,cppst);
}
```

- Divide argument list into first and rest
- Do something with first and recursively call template with rest
- Specialise for the case with 1 or 0 arguments

Parameter pack expansion

- `pattern ...` is called a parameter pack expansion
- It applies a pattern to a comma separated list of instantiations of the pattern
- If we are in a function :

```
template <typename ... Types> void g(Types ... args)
```

- `args...` means the list of arguments used for the function.
- Calling `f(args ...)` in `g` will call `f` with same arguments
- Calling `f(h(args)...)` in `g` will call `f` with an argument list generated by applying function `h` to each argument of `g`
- In `g(true,"abc",1)`, `f(h(args)...)` means `f(h(true),h("abc"),h(1))`

Parameter pack expansion

```
template <typename ... Types> void f(Types ... args);
template <typename Type1, typename ... Types> void f(Type1 arg1, Types ... rest) {
    std::cout <<" The first argument is "<<arg1
              <<". Remainder argument list has "<<sizeof...(Types)<<" elements.\n";
    f(rest ...);
}
template <> void f() {}
template <typename ... Types> void g(Types ... args) {
    std::cout << "Inside g: going to call function f with the sizes of "
              << "my arguments\n";
    f(sizeof(args)...);
}
```

- **`sizeof...`**`(Types)` retrieves the number of arguments in the parameter pack
- In `g` above, we call `f` with the sizes of each of the parameters passed to `g`
- Similarly, one can generate all addresses as `&args...`, increment all with `++args...` (examples `variadic_2.cc` and `variadic_3.cc`)

Parameter pack expansion: where

```
template <typename ... Types> void f(Types & ... args) {}
template <typename ... Types> void h(Types ... args) {
    f(std::cout<<args<<"\t"...);
    [=,&args ...]{ return g(args...); }();
    int t[sizeof...(args)]={args ...};
    int s=0;
    for (auto i : t) s+=i;
    std::cout << "\nsum = "<<s <<"\n";
}
```

- Parameter pack expansion can be done in function parameter list, function argument list, template parameter list or template argument list
- Braced initializer lists
- Base specifiers and member initializer lists in classes
- Lambda captures

Example 4.6: Parameter packs

Study the examples `variadic_1.cc`,`variadic_2.cc` and `variadic_3.cc`. See where parameter packs are begin expanded, and make yourself familiar with this syntax.

Tuple

Tuples

```
#include <tuple>
#include <iostream>
int main()
{
  std::tuple<int,int,std::string> name_i_j{0,1,"Uralic"};
  auto t3=std::make_tuple<int,bool>(2,false);
  auto t4=std::tuple_cat(name_i_j,t3);
  std::cout << std::get<2>(t4) <<'\n';
}
```

- Like `std::pair`, but with arbitrary number of members
- "Structure templates without names"
- Accessor "template functions" `std::get<index>` with `index` starting at 0
- Supports relational operators for lexicographical comparisons
- `tuple_cat(args ...)` concatenates tuples.

Tuples

```
std::tuple<int,int,string> f(); // elsewhere
int main()
{
  int i1;
  std::string name;
  std::tie(i1,std::ignore,name) = f();
}
```

- `tie(args ...)` "extracts a tuple" into named variables.
- Some fields may be ignored during extraction using `std::ignore` as shown

JÜLICH FORSCHUNGSZENTRUM

Example 4.7:

The file `examples/tuple0.cc` demonstates the use of tuples with some commonly used functions in their context.

JÜLICH FORSCHUNGSZENTRUM

Day 5

From yesterday: reading lines from a file to container

- Helper class `Line`
- Notice trick to provide implicit conversion to strings, so that we can store in a container of strings rather than a container of `Line`

```
// examples/line_read.cc
#include <list>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <iterator>
using namespace std;
struct Line {
  string data;
  operator string() const {return data;}
};
istream &operator>>(istream &is,Line &l)
{
  getline(is,l.data);
  return is;
}
int main(int argc, char *argv[])
{
  list<string> lines;
  ifstream fin{argv[1]};
  copy(istream_iterator<Line>(fin),
            istream_iterator<Line>(),
            back_inserter(lines));
  for (auto line :lines)cout<<line<<'\n';
```


Exercise 5.1:

The program `exercises/is_it_random.cc` produces two random sequences using two generators at two points in the code. Do you notice anything undesirable in the output ? How can you fix it ? Hint: You will need `std::ref`

User defined literals

```
int main()
{
  double N=6.023e23;
  Temperature T;
  T.value(293.0);
  Temperature T2 = 350_C;
  Temperature T3 = 900_K;
  complex_number c = 1+2_i;
  ...
}
```

Redefining the "" operator!

- Defining your own rules for how literals are interpreted for your class
- Desirable to enable clean and easily read initialisations

User defined literals

```
class Temperature
{
  public:
  Temperature() = default;
  etc.
};
Temperature operator "" _K(const char * d)
{
  return {std::stod(d)};
}
Temperature operator "" _C(double d)
{
  return d+273.15;
}
```

Redefining the "" operator!

- "Raw" form takes **`const char *`** as argument
- "Cooked" form takes interpreted value of the number

User defined literals

```
int main()
{
    double N=6.023e23;
    Temperature T;
    T.value(293.0);
    Temperature T2 = 350_C;
    Temperature T3 = 900_K;
    complex_number c = 1+2_i;
    ...
}
```

Exercise 5.2:

- The demo program `examples/literals.cc`, and `examples/cooked_literals.cc` show how this is done using a simple "temperature" class
- Make something similar for your complex number class!

Meta-programming in C++11

Template metaprogramming

An interesting property of C++ templates

C++ templates ...

- can take typenames and integral types as parameters
- can be used to specify compile time constant sizes
- but also give you a peculiar kind of "function" in effect

```
template <typename T, int N>
struct my_array {
    T data[N];
};
```

```
template <typename T,
          int nrows, int ncols>
struct my_matrix {
    T data[nrows*ncols];
};
```

```
template <int i, int j>
struct mult {
    static const int value=i*j;
};
...
my_array<mult<19,21>::value> vals;
```

Template meta-programming in C++11

Recursion and specialisation

- no "if", "for" or state variables like for run-time functions
- But, we have recursion and specialisation!
- Iteration can be emulated using tail recursion and specialisation
- Conditional branches can also be done with specialisation

```
template <int N> struct factorial {
  static const int value=N*
                  factorial<N-1>::value;
};
template <> struct factorial<0> {
  static const int value=1;
};
...
std::array<int,factorial<7>::value> P;
```

```
template <int i, bool flag>
struct cube_mag_impl { static const int value=i*i
      *i;};
template<int i, false> struct cube_mag_impl {
  static const int value=-i*i*i;
};
template <int i>
using cube_mag=cube_mag_impl<i,(i>0)>::value;
std::array<double,cube_mag<7>> A;
std::array<double,cube_mag<-3>> B;
```


Evaluate dependent types

- Suppose we want to implement a template function

```
template <typename T> U f(T a);
```

  such that when T is a non-pointer type, U should take the value T. But if T is itself a pointer, U is the type obtained by dereferencing the pointer
- We could use a template function to "compute" the type U like this:

```
template <typename T> struct remove_pointer { using type=T; };
template <typename T> struct remove_pointer<T*> { using type=T; };
```

- We can then declare the function as:

```
template <typename T> remove_pointer<T>::type f(T a);
```

Type functions

- Gives rise to a variety of "type functions"
- Compute properties of types
- Compute dependent types

```
template <typename T1, typename T2>
std::is_same<T1,T2>::value

template <typename T>
std::is_integral<T>::value

template <typename T>
std::make_signed<T>::type
```


static_assert with type traits

```
#include <iostream>
#include <type_traits>
template < class T >
class SomeCalc
{
  static_assert(std::is_arithmetic<T>::value,
  "argument T must be an arithmetic type");
};
int main()
{
  SomeCalc<string> mycalc; //Compiler error!
  ...
}
```

- Use **`static_assert`** and `type_traits` in combination with **`constexpr`**

Example 5.1:

The program `static_assert2.cc` demostrates the above.

Typetraits

Unary predicates

- `is_integral<T>` : `T` is an integer type
- `is_const<T>` : has a **`const`** qualifier
- `is_class<T>` : struct or class
- `is_pointer<T>` : Pointer type
- `is_abstract<T>` : Abstract class with at least one pure virtual function
- `is_copy_constructible<T>` : Class allows copy construction

Typetraits

Type relations

- `is_same<T1,T2>` : `T1` and `T2` are the same types
- `is_base_of<T,D>` : `T` is base class of `D`
- `is_convertible<T,T2>` : `T` is convertible to `T2`

Another example with typetraits

```
template <typename T>
void f_impl(T val, true_type);
// for integer T
template <typename T>
void f_impl(T val, false_type);
// for others
template <typename T>
void f(T val)
{
    f_impl(val,std::is_integral<T>());
}
```

Situation: You have two different ways to implement a function, for integers and for everything else.

- Implement two specializations using a `true_type` and a `false_type` argument
- Use `is_integral` trait to choose one or the other at compile time

Metaprogramming with constexpr

Metaprogramming with constexpr

```
constexpr bool is_prime_rec(size_t number, size_t c)
{
  return (c*c>number)?true:(number % c == 0)?false:is_prime_rec(number, c+1);
}
constexpr bool is_prime(size_t number)
{
  return (number<=1)?false:is_prime_rec(number,2);
}
int main()
{
  constexpr unsigned N=1000003;
  static_assert(is_prime(N),"Not a prime");
```

- New mechanism for compile time calculations
- Code can be clearer than template functions
- Floating point calculations possible
- Recommendation: use **`constexpr`** functions for numeric calculations at compile time
- The same functions are available at run time if the arguments are not compile time constants

Printing a tuple

```
template <int idx, int MAX, typename... Args> struct PRINT_TUPLE {
  static void print(std::ostream & strm,const std::tuple<Args...> & t)
  {
    strm << std::get<idx>(t)<<(idx+1==MAX ? "" : ", ");
    PRINT_TUPLE<idx+1,MAX,Args...>::print(strm,t);
  }
};
template <int MAX, typename... Args> struct PRINT_TUPLE<MAX,MAX,Args...> {
  static void print(std::ostream &strm, const std::tuple<Args...> &t){}
};
template <typename ... Args>
std::ostream & operator<<(std::ostream & strm,const std::tuple<Args...> &t)
{
  strm << "[";
  PRINT_TUPLE<0,sizeof...(Args),Args...>::print(strm,t);
  return strm <<"]";
```

- Tail recursion in place of a loop
- Template specialization is used to stop the recursion

Example 5.2:

Printing a tuple is demonstrated in `print_tuple.cc`

Zipping tuples together

- Suppose we have two tuples in our program:

```
std::tuple<int,int,std::string> t1;
std::tuple<unsigned,unsigned,unsigned> t2;
```

- We want to zip them together with a function that produces:

```
std::tuple<std::pair<int,unsigned>,
           std::pair<int,unsigned>,
           std::pair<std::string,unsigned>> t3=zip(t1,t2);
```

- What should be the return type of the general version of the function, for n-tuples ?

```
template <class ... Args1> struct zip_st {
  template <class ... Args2> struct with {
    using type = std::tuple<std::pair<Args1,Args2>...>;
  };
};
template <class ... Args1>
template <class ... Args2>
zip_st<Args1 ...>::with<Args2 ...>::type
    zip(const std::tuple<Args1 ...> &t1,const std::tuple<Args2 ...> &t2);
```


Range based for loops for multiple containers

- We have three containers A, B and C of the same size, and we want to print their corresponding elements like

```
for (size_t i=0;i<A.size();++i)
    std::cout << A[i]<<'\t'<<B[i]<<'\t'<<C[i]<<'\n';
```

- Not good, if the containers do not have the **`operator[]`** defined.
- Separate iterators for each is possible, but ungainly.
- We want to be able to write:

```
for (auto (el1,el2,el3) : zip(A,B,C))
    std::cout<<el1<<'\t'<<el2<<'\t'<<el3<<'\n';
```

- We can't. But we can do the following with a little work:

```
for (auto el : zip(A,B,C)) std::cout<<std::get<0>(el)<<'\t'<<std::get<1>(el)
                                <<'\t'<<std::get<2>(el)<<'\n';
```

Range based for loops for multiple containers

```
template <typename ... Args> class ContainerBundle {
public:
  using iterator = IteratorBundle<typename Args::iterator ...>;
  using iter_coll = std::tuple<typename Args::iterator ...>;
  ContainerBundle(typename std::add_pointer<Args>::type ... args)
                : dat{args ...}, bg{args->begin() ...}, nd{args->end() ...} {}
  ~ContainerBundle() = default;
  ContainerBundle(const ContainerBundle &) = delete;
  ContainerBundle(ContainerBundle &&) = default;

  inline iterator begin() { return bg; }
  inline iterator end() const { return nd; }
private:
  std::tuple<typename std::add_pointer<Args>::type ...> dat;
  iterator bg,nd;
};
```

- We need a container bundle class taking pointers to each container

Range based for loops for multiple containers

```
template <typename ... Itr> class IteratorBundle {
public:
    using value_type = std::tuple<typename Itr::value_type ...>;
    using internal_type = std::tuple<Itr ...>;
    IteratorBundle() = default; // etc.
    bool operator==(const IteratorBundle &it) const { return loc==it.loc; }
    // special implementation insisting that all
    // elements in the bundle compare unequal. This ensures proper
    // function for containers of different sizes.
    bool operator!=(const IteratorBundle &it) const;
    inline value_type operator*() { return deref(); }
    inline IteratorBundle & operator++() {
        advance_them<0,sizeof ...(Itr)>::eval(loc);
        return *this;
    }
```

- We need an iterator bundle class to represent combined iterators
- Handling containers of different sizes is tricky. We do it here with a special implementation of **`operator!=`**

Range based for loops for multiple containers

```
template <bool... b> struct static_all_of;
template <bool... tail>
struct static_all_of<true, tail...> : static_all_of<tail...> {};
//no need to look further if first argument is false
template <bool... tail>
struct static_all_of<false, tail...> : std::false_type {};
template <> struct static_all_of<> : std::true_type {};

template <typename ... Args>
ContainerBundle<typename std::remove_pointer<Args>::type ...> zip(Args ... args)
{
    static_assert(static_all_of<std::is_pointer<Args>::value ...>::value,
                  "Each argument to zip must be a pointer to a container! ");
    return {args ...};
}
```

- And some code to ensure we pass pointers when creating the zipped containers

Example 5.3: Range-for for multiple containers

The file `examples/multi_range_for.cc` demonstrates the use of multiple containers with range based for loops and the syntax just discussed. Check what happens when you forget to pass a pointer to the `zip` function! Notice that when the containers are of different sizes, the range for only iterates until the shortest container is exhausted.

Multi-threading

Native multi-threading in C++11

Motivation

- Speed : clock frequencies are not likely to rise as quickly as they did 20 years back
- Power consumption: $P \propto CV^2 f$, with $f \propto \frac{(V-V_{th})^{\alpha}}{V}$. "Performance per watt".
- Your code does not have to run 10 times faster on 10 CPU cores for it to be acceptable.

Native multi-threading in C++

- `std::async` is a high level interface to C++11 threading library
- `async(foo)` calls the "callable object" `foo`, possibly in a new thread
- `async(std::launch::async,foo)` launches `foo` in a new thread
- If `foo` returns `T`, `async(foo)` returns an object of type `future<T>`

```
int somefunc()
{
  auto f1=std::async(foo);
  // May start in a new thread,
  // or be deferred
  auto f2=std::async(std::launch::async,
                     foo);
  // Starts in a new thread
  std::cout << f1.get() << std::endl;
  // Start f1 if it was not started, wait
  // for it to finish, and get the result
}
int main()
{
  async(std::launch::async, foo);
  // The above must finish before
  // the code can continue. Why ?
  somefunc();
  // At this point, both async jobs
  // started by somefunc() must have
  // returned, since f2 is destroyed
}
```

Native multi-threading in C++

- If `foo` returns `T`, `async(foo)` returns an object of type `future<T>`
- A `future<T>` object can yield a `T` object when the function `get()` is called.
- `get()` and the destructor of `future<T>` block

```
int somefunc()
{
  auto f1=std::async(foo);
  // May start in a new thread,
  // or be deferred
  auto f2=std::async(std::launch::async,
                     foo);
  // Starts in a new thread
  std::cout << f1.get() << std::endl;
  // Start f1 if it was not started, wait
  // for it to finish, and get the result
}
int main()
{
  async(std::launch::async, foo);
  // The above must finish before
  // the code can continue. Why ?
  somefunc();
  // At this point, both async jobs
  // started by somefunc() must have
  // returned, since f2 is destroyed
}
```


Example 5.4:

`examples/async1.cc` has a trivial program demonstrating how to run callable objects in different threads. Observe the use of `future` and `async`. Compile with the extra option `-pthread` with `clang++` or `g++`

Example 5.5:

`examples/async2.cc` implements an integral of a function in a range in a serial manner and using as many threads as the hardware allows. How much faster is the parallel calculation ?

Native multi-threading in C++

- `std::thread` is a low level interface to start a thread
- `std::thread` takes a callable object and runs it in a new thread
- `threadx.join();` calls at the end is to make sure that the parent thread waits for the completion of the created threads

```
int main()
{
    std::thread t1{some_calc(10,20)};
    std::thread t2{some_calc(20,30)};
    std::thread t3{some_calc(30,40)};
    std::thread t4{some_calc(40,50)};

    t1.join();
    t2.join();
    t3.join();
    t4.join();
}
```


Native multi-threading in C++

- There is a variadic template constructor for `std::thread`
- This lets you pass arguments to the different callable objects in different threads

```
void sum_everything(int a,int b,
                    int c,int d)
{
  return a+b+c+d;
}
int main()
{
  std::thread t1{sum_everything,
                 1,2,3,4};
  t1.join();
}
```

Native multi-threading in C++

- `std::mutex` is a class providing a lockable "mutual exclusion" object
- The block of code between `lock()` and `unlock()` functions of a `mutex` is only accessible to one thread at a time
- `std::lock_guard<mutex>` acquires a lock and releases it when it expires

```
std::mutex m;
...
m.lock();
// Do something best done
// one thread at a time
m.unlock();
```


Threading Building Blocks

JÜLICH FORSCHUNGSZENTRUM

TBB: Threading Building Blocks I

- Parallel programming constructs for the end user
- Template library rather than language extensions
- Provides utilities like `parallel_for`, `parallel_reduce` to simplify the most commonly used structures in parallel programs
- Provides scalable concurrent containers such as vectors, hash tables and queues for use in multithreaded environments
- No direct support for vector parallelism. But can be combined with auto-parallelisation and **`#pragma`** `simd` etc from Cilk Plus
- Supports complex models such as pipelines, dataflow and unstructured task graphs

JÜLICH FORSCHUNGSZENTRUM

TBB: Threading Building Blocks II

- Scalable memory allocation, avoidance of false sharing, thread local storage
- Low level synchronisation tools like mutexes and atomics
- Work stealing task scheduler
- `http://www.threadingbuildingblocks.org`
- Structured Parallel Programming, Michael McCool, Arch D. Robinson, James Reinders

Using TBB

- Include `tbb/tbb.h` in your file
- Public names are available under the namespaces `tbb` and `tbb::flow`
- You indicate "available parallelism", scheduler may run it in parallel if resources are available
- Unnecessary parallelism will be ignored
- Load the TBB module in your environment with `module load tbb` to compile the examples

Parallel for loops

- Template function modeled after the **`for`** loops, like many STL algorithms
- Takes a callable object as the third argument
- Using lambda functions, you can expose parallelism in sections of your code

```
tbb::parallel_for(first,last,f);
// parallel equivalent of
// for (auto i=first;i<last;++i) f(i);

tbb::parallel_for(first,last,stride,f);
// parallel equivalent of
// for (auto i=first;i<last;i+=stride)
//   f(i);

tbb::parallel_for(first,last,
                  [captures](anything){
  //Code that can run in parallel
});
```

Parallel for with ranges

- Splits `range` into smaller ranges, and applies `f` to them in parallel
- Possible to optimize `f` for subranges rather than a single index
- Any type satisfying a few design conditions can be used as a range
- Multidimensional ranges possible

```
tbb::parallel_for(0,1000000,f);
// One parallel invocation for each i!
tbb::parallel_for(range,f);

// A type R can be a range if the
// following are available
R::R(const R &);
R::~R();
bool R::is_divisible() const;
bool R::empty() const;
R::R(R & r,split);//Split constructor
```


Parallel for with ranges

```
tbb::blocked_range<int> r{0,30,20};
assert(r.is_divisible());
blocked_range<int> s{r};
//Splitting constructor
assert(!r.is_divisible());
assert(!s.is_divisible());
```

- `tbb::blocked_range<int>(0,4)` represents an integer range 0..4
- `tbb::blocked_range<int>(0,50,30)` represents two ranges, 0..25 and 26..50
  - So long as the size of the range is bigger than the "grain size" (third argument), the range is split

Parallel for with ranges

```
void daxpcy_tbb(double a, std::vector<double> &x, std::vector<double> &y) {
  tbb::parallel_for(tbb::blocked_range<int>(0,x.size()),
                    [&](tbb::blocked_range<int> r){
    for (size_t i=r.begin();i!=r.end();++i) {
      y[i]=a*sin(x[i])+cos(y[i]);
    }
  });
}
```

- `parallel_for` with a range uses split constructor to split the range as far as possible, and then calls f(range), where `f` is the functional given to `parallel_for`
- It is unlikely that you wrote your useful functions with ranges compatible with `parallel_for` as arguments
- But with lambda functions, it is easy to fit the parts!

Example 5.6: TBB parallel for demo

The program `examples/daxpcy.cc` demonstrates the use of parallel for in TBB. It is a slightly modified version of the commonly used DAXPY demos. Instead of calculating $y = a * x + y$ for scalar a and large vectors x and y, we calculate $y = a * sin(x) + cos(y)$. To compile, you need to load your compiler and TBB modules, and use them like this:

```
module load gcc/4.9.0 tbb
G $TBB_INCLUDES $TBB_LIBRARIES daxpcy.cc -o daxpcy_gcc
```

Parallel reductions with ranges

```
T result = tbb::parallel_reduce(range, identity, subrange_reduction, combine);
```

- `range` : As with parallel for
- `identity` : Identity element of type T. The type determines the type used to accumulate the result
- `subrange_reduction` : Functor taking a "subrange" and an initial value, returning reduction
- `combine` : Functor taking two arguments of type T and returning reduction over them over the subrange. Must be associative, but not necessarily commutative.

Parallel reduce with ranges

```
double inner_prod_tbb(std::vector<double> & x, std::vector<double> & y) {
  return tbb::parallel_reduce(
      tbb::blocked_range<int>(0,n), // range
      double{}, // identity
      [&](tbb::blocked_range<int> &r,float in){
        return std::inner_product(x.begin()+r.begin(),x.begin()+r.end(),
                                  y.begin()+r.begin(),in);
      }, // subrange reduction
      std::plus<double>{} // combine
  );
}
```

- With TBB ranges, we can use blocked implementations with hopefully vectorisable calculations in subranges
- Two functors are required, either of which could be lambda functions
- Important to add the contribution of initial value in subrange reductions

Example 5.7: TBB parallel reduce

The program `tbbreduce.cc` rewrites the program used to calculate the integral using async, now using `tbb::parallel_reduce`. Check how lambda functions are used to do the integral. What kind of speed up do you see relative to the serial version ? Does it make sense considering the number of physical cores in your computer ?

TBB task groups

- Run an arbitrary number of function objects in parallel
- In case an exception is thrown, the task group is cancelled

```
struct Equation {
  void solve();
};

std::list<Equation> equations;
tbb::task_group g;
for (auto eq : equations)
  g.run([&]{eq.solve();});

g.wait();
```

Atomic variables

- "Instantaneous" updates
- Lock-free synchronization
- For `tbb::atomic<T>`, T can be integral, enum or pointer type
- If `index==k` simultaneous calls to `index++` by n threads will increase `index` to `k+n`. Each thread will use a distinct value between `k` and `k+n`

```
std::array<double,N> A;
tbb::atomic<int> index;

void append(double val)
{
  A[index++]=val;
}
```

But it is important that we use the return value of `index++` in the threads!

Enumerable thread specific

```
tbb::enumerable_thread_specific<double> E;
double Eglob=0;
double f(size_t i, size_t j);
tbb::blocked_range2d<size_t> r{0,N,0,N};
tbb::parallel_for(r,[&](tbb::blocked_range2d<size_t> r){
  auto & eloc=E.local();
  for (size_t i=r.rows().begin();i!=r.rows().end();++i) {
    for (size_t j=r.cols().begin();j!=r.cols().end();++j) {
      if (j>i) eloc += f(i,j);
    }
  }
});
Eglob=0;
for (auto & v : E) {Eglob+=v;v=0;}
```

- Thread local "views" of a variable
- behaves like an STL container of those views
- Member function `local()` gives a reference to the local view in the current thread
- Any thread can access all views by treating it as an STL container

Example 5.8: Reduction with enumerable thread specific

You can use the `enumerable_thread_specific` and `parallel_for` to implement reduction. The program `examples/tbbreduce1.cc` demonstrates this.

Concurrent containers

```
#include <tbb/concurrent_vector.h>

auto v=tbb::concurrent_vector<int>(N,0);

tbb::parallel_for(v.range(),[&](tbb::concurrent_vector::range_type r){
    //...
});
```

- Random access by index
- Multiple threads can grow container and add elements concurrently
- Growing the container does not invalidate any iterators or indexes
- Has a `range()` member function for use with `parallel_for` etc.

Exercise 5.3: N particle systems with pairwise interactions

Use the `enumerable_thread_specific` and `parallel_for` to calculate the pairwise interactions in an N-particle system.

Day 6

Class hierarchies

Class inheritance

Analogy from biology

Relationships among organisms

- Shared traits along a branch and its sub-branches
- Branches "inherit" traits from parent branch and introduce new distinguishing traits
- A horse *is* a mammal. A dog *is* a mammal. A monkey *is* a mammal.

Class inheritance

```
struct Point {double X, Y;};
class Triangle {
public:
  // Constructors etc., and then,
  void translate();
  void rotate(double byangle);
  double area() const;
  double perimeter() const;
private:
  Point vertex[3];
};
class Quadilateral {
public:
  void translate();
  void rotate(double byangle);
  double area() const;
  double perimeter() const;
  bool is_convex() const;
private:
  Point vertex[4];
};
```

Geometrical figures

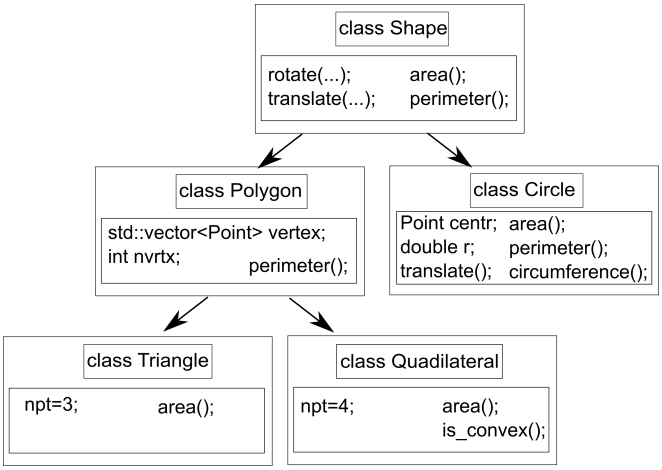
- Many actions (e.g. translate and rotate) will involve identical code
- Properties like `area` and `perimeter` make sense for all, but are better calculated differently for each type
- There may also be new properties (`is_convex`) introduced by a type

Class inheritance

- We want to write a program to
  - list the area of all the geometric objects
  - select the largest and smallest objects
  - draw

  in our system.
- A loop over a darray of them will be nice. But `darray< ??? >`
- Object oriented languages like C++, Java, Python ... have a concept of "inheritance" for the classes, to describe such conceptual relations between different types.

Class inheritance



- Abstract concept class "Shape"
- Inherited classes add/change some properties
- and inherit other properties from "base" class

A triangle *is* a polygon. A polygon *is* a shape. A circle *is* a shape.

Class inheritance

```
class Shape {
public:
  virtual ~Shape()=0;
  virtual void rotate(double)=0;
  virtual void translate(Point)=0;
  virtual double area() const =0;
  virtual double perimeter() const =0;
};
class Circle : public Shape {
public:
  Circle(); // and other constructors
  ~Circle();
  void rotate(double phi) {}
  double area() const override
  {
    return pi*r*r;
  }
private:
  double r;
};
```

Syntax for inheritance

- Circle is a derived class from base class Shape
- A derived class inherits from its base(s), which are indicated in the class declaration.
- Functions marked as **`virtual`** *can be* re-implemented in a derived class.

Class inheritance

```
class Shape {
public:
  virtual ~Shape()=0;
  virtual void rotate(double)=0;
  virtual void translate(Point)=0;
  virtual double area() const =0;
  virtual double perimeter() const
        =0;
};
class Circle : public Shape {
public:
  Circle(); // and other constructors
  ~Circle();
  void rotate(double phi) {}
  double area() const override
  {
    return pi*r*r;
  }
private:
  double r;
};
...
Shape a; // Error!
Circle b; // ok.
```

- A derived class inherits all member variables and functions from its base.
- Functions marked as **`virtual`** in the base class *can be* re-implemented in a derived class
- A class with a pure virtual function (with "=0" in the declaration) is an abstract class. Objects of that type can not be declared.

Class inheritance

```
class Polygon : public Shape {
public:
  double perimeter() const
  {
    // return sum over sides
  }
protected:
  darray<Point> vertex;
  int npt;
};
class Triangle : public Polygon {
public:
  Triangle() : npt(3)
  {
    vertex.resize(3); // ok
  }
  double area() const override
  {
    // return sqrt(s*(s-a)*(s-b)*(s-c))
  }
};
```

Syntax for inheritance

- Variables or functions declared under the keyword private in the base class can not be accessed directly in the derived classes.
- But if they are declared protected, they can be used in the derived classes, while being unavailable to the outside world.

Class inheritance

```
class Polygon : public Shape {
public:
  double perimeter() const final
  {
    // return sum over sides
  }
protected:
  darray<Point> vertex;
  int npt;
};
class Triangle : public Polygon {
public:
  Triangle() : npt(3)
  {
    vertex.resize(3); // ok
  }
  double area() const override
  {
    // return sqrt(s*(s-a)*(s-b)*(s-c))
  }
};
```

Syntax for inheritance

- Triangle implements its own `area()` function, but can not implement a `perimeter()`, as that is declared as **`final`** in `Polygon`. This is done if the implementation from the base class is good enough for intended inheriting classes.

Class inheritance

```
class Polygon : public Shape {
public:
  double perimeter() const final
  {
    // return sum over sides
  }
protected:
  darray<Point> vertex;
  int npt;
};
class Triangle : public Polygon {
public:
  Triangle() : npt(3)
  {
    vertex.resize(3); // ok
  }
  double area() override  // Error!!
  {
    // return sqrt(s*(s-a)*(s-b)*(s-c))
  }
};
```

- The keyword **`override`** ensures that the compiler checks there is a corresponding base class function to override.
- Virtual functions can be re-implemented without this keyword, but an accidental omission of a **`const`** or an `&` can lead to really obscure runtime errors.

Class inheritance

```
int main()
{
  darray<Shape *> shape;
  shape.push_back(new Circle(0.5, Point(3,7)));
  shape.push_back(new Triangle(Point(1,2),Point(3,3),Point(2.5,0)));
...
  for (size_t i=0;i<shape.size();++i) {
    std::cout<<shape[i]->area()<<'\n';
  }
}
```

- A pointer to a base class is allowed to point to an object of a derived class
- Here, `shape[0]->area()` will call `Circle::area()`, `shape[1]->area()` will call `Triangle::area()`

Class inheritance

Inherit or include as data member ?

```
class DNA {
...
  std::valarray<char> seq;
};
class Cell : public DNA ???
or
class Cell {
...
  DNA mydna;
};
```

- A derived class extends the concept represented by its base class in some way.
- Although this extension might mean addition of new data members,

```
Concept B = Concept A + new data
```

does not necessarily mean the class for B should inherit from the class for A

Class inheritance

Inherit or include as data member ?

```
class DNA {
...
  std::valarray<char> seq;
};

class Cell : public DNA ???

or

class Cell {
...
  DNA mydna;
};
```

is vs *has*

- A good guide to decide whether to inherit or include is to ask whether the concept B contains an object A, or whether any object of type B is also an object of type A, like a monkey is a mammal, and a triangle is a polygon.
- is $\Longrightarrow$ inherit . has $\Longrightarrow$ include

Class inheritance

Inheritance summary

- Base classes to represent common properties of related types : e.g. all proteins are molecules, but all molecules are not proteins. All triangles are polygons, but not all polygons are triangles.
- Less code: often, only one or two properties need to be changed in an inherited class
- Helps create reusable code
- A base class may or may not be constructable (`Polygon` as opposed to `Shape`)

Class decorations in C++11

More control over classes

Class design improvements

- Possible to initialise data in class declaration
- Initialiser list constructors
- Delegating constructors allowed
- Inheritting constructors possible

```
class A {
  int v[]{1,-1,-1,1};
public:
  A() = default;
  A(std::initializer_list<int> &);
  A(int i,int j,int k,int l)
  {
    v[0]=i;
    v[1]=j;
    v[2]=k;
    v[3]=l;
  }
  // Delegate work to another
       constructor
  A(int i,int j) : A(i,j,0,0) {}
};
class B : public A {
public:
// Inherit all constructors from A
  using A::A;
  B(string s);
};
```


More control over classes

Class design improvements

- Explicit **`default`**, **`delete`**, **`override`** and **`final`**
- "Explicit is better than implicit"
- More control over what the compiler does with the class
- Compiler errors better than hard to trace run-time errors due to implicitly generated functions

```
class A {
  // Automatically generated is ok
  A() = default;
  // Don't want to allow copy
  A(const A &) = delete;
  A & operator=(const A &) = delete;
  // Instead, allow a move constructor
  A(const A &&);
  // Don't try to override this!
  void getDrawPrimitives() final;
  virtual void show(int i);
};
class B : public A
{
  B() = default;
  void show() override; // will be an
       error!
};
```

Exercise 6.1:

The directory `exercises/geometry` contains a set of files for the classes Point, Shape, Polygon, Circle, Triangle, and Quadrilateral. In addition, there is a `main.cc` and a `Makefile`. The implementation is old style. Using what you have learned about classes in C++11, improve the different classes using keywords like **`default`**,**`override`**, **`final`** etc. Compile by typing `make`. Familiarise yourself with

- Implementation of inherited classes
- Compiling multi-file projects
- The use of base class pointers arrays to work with heterogeneous types of objects

Exercise 6.2:

Implement a `LorentzVector` class in `exercises/LorentzVector` by extending the 3D vectors you wrote to include a time coordinate as the 4-vectors in special relativity. Should it inherit from `Vector3D` or contain a `Vector3D` data member ?

Curiously Recurring Template Pattern

```
template <class ClassWithName> struct Named {
  inline string get_name() const
  {
    return static_cast<ClassWithName const *>
           (this)->get_name_impl();
  }
  inline int version() const
  {
    return 1.0;
  }
};
struct Acetyl : public Named<Acetyl> {
  inline string get_name_impl() const
  {
    return "Acetyl";
  }
};
struct HBMM : public Named<HBMM> {
  inline string get_name_impl() const
  {
    return "HBMM";
  }
};
```

```
int main()
{
  Acetyl a;
  HBMM b;
  cout << "get_name on a returns : "
       << a.get_name() << '\n';
  cout << "get_name on b returns : "
       << b.get_name() << '\n';
  cout << "Their versions are "
       << a.version()<< " and "
       << b.version()<<'\n';
}
```

CRTP

- Polymorphism without virtual functions
- Faster in many cases

Example 6.1: CRTP

The file `examples/crtp1.cc` demonstrates the use of CRTP to implement a form of polymorphic behaviour. The function `version()` is inherited without changes in Acetyl and HBMM. `get_name()` is inherited, but behaves differently for the two types. All this is without using virtual functions.

Graphical user interfaces with Qt5

Introduction to GUI design with Qt

- Cross platform framework for application and UI development
- Easily readable C++ code for GUI programming
- "Principle of least surprises"
- Trolltech → Nokia → Digia
- https://qt-project.org
  - Downloads, tutorials, documentation
  - Link to other courses
- C++ GUI programming with Qt 4 Prentice Hall

Introduction to GUI design with Qt

- Extends the C++ language with new excellent container classes
- Java style iterators
- Copy-on-write
- Signals and slots mechanisms
- I/O classes with Unicode I/O, many character encodings, and platform independent abstraction for binary data.
- Neat framework for GUI programming
- For you: GUI programming is a good training ground to get used to "object oriented programming"

Qt5: Two ways of creating GUI

- **Qt Quick:** Declarative language for fluid and animated user interfaces
- **Qt Widgets:** For more traditional desktop GUI

Hello Qt

Hello, World! Qt style

```
// examples/qthello.cc
#include <QtGui/QApplication>
#include <QtGui/QLabel>
int main(int argc, char *argv[]) {
  QApplication app(argc,argv);
  QLabel *label =
          new QLabel("Hello world!");
  label->show();
  return app.exec();
}
```

- Compile and run :

```
 g++ qthello.cc -lQtGui -lQtCore
./a.out
```

- You can format the text in a QLabel with html to test this.

The first Widget-based Qt program

- QApplication
  - Manages global resources
  - Create one in main
  - Pass the argc and argv to it
  - At the end of main, pass control to it with app.exec()
  - Forget about it and concentrate on setting up the graphical elements
- `QLabel`
  - a label. One of numerous kinds of Widgets in Qt

Hello, World! Qt style

```
// examples/qt5hello.cc
#include <QtWidgets/QApplication>
#include <QtWidgets/QLabel>
int main(int argc, char *argv[]) {
  QApplication app(argc,argv);
  QLabel *label =
          new QLabel("Hello world!");
  label->show();
  return app.exec();
}
```

- Compile and run :

```
 g++ qthello.cc -lQt5Widgets -lQt5Gui -lQt5Core
./a.out
```

- In Qt5, the headers for all Widget classes are in `QtWidgets/` rather than `QtGui/`
- Obscure compiler/linker errors result if you do not change the include paths

Qt Widgets

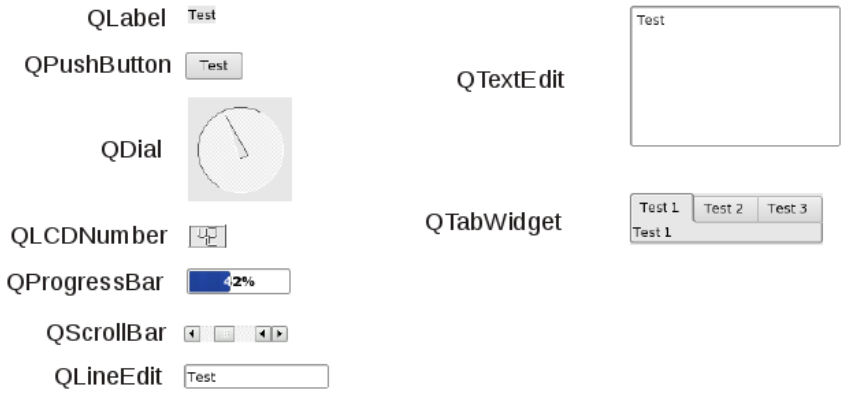
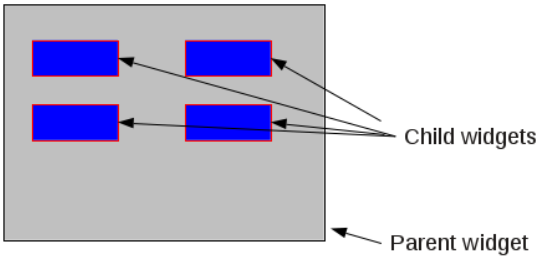


Figure : Widgets are graphical building blocks. They can communicate with each other by sending and receiving "signals". A widget can contain other widgets.




```
class MyWidget :public QWidget
{
    QLabel *child1;
    MyScoreDisplay *child2;
    MyInputWidget *child3;
    QPushButton *child4;
    ...
}
int main(argc, argv)
{
    QApplication qapp(argc, argv);
    MyWidget *wgt = new MyWidget;
    wgt->show();
    return app.exec();
}
```

- Widgets in Qt are implemented as regular C++ classes, which inherit from `QWidget` directly or indirectly.
- A Qt GUI application consists of a main widget defining the appearance and interaction of the GUI. The `main()` function is then just a call to the `show()` function of this widget as in the hello example.

Qt Widgets

- All widget class constructors take a pointer to a "parent" widget as one of the arguments
- If parent pointer is set to nullptr, the widget is shown in a window of its own

```
class MyWidget :public QWidget
{
  MyWidget(QWidget *parent=0);
  ...
}
MyWidget::MyWidget(QWidget *parent):
QWidget(parent)
{
  child1=new QLabel("Hello world",this);
  child2=new MyScoreDisplay(this);
  ...
}
```

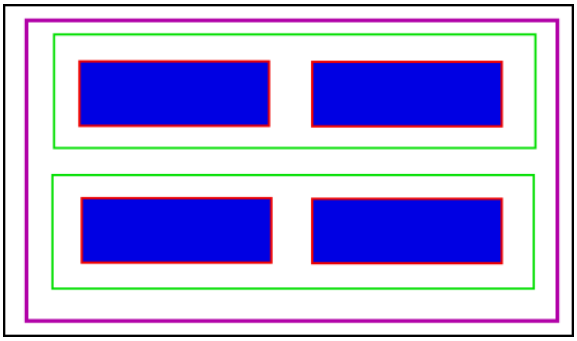
- A parent widget constructs its children by passing itself (through the “this” pointer) to their constructors. This is done more elegantly using layout managers (next slide)
- The destructor of a widget takes care of destroying all child widgets

Layouts

- Invisible entities which take care of arranging their children in a well defined manner
- A layout can have other layouts or widgets as children
- `QVBoxLayout` arranges its children vertically
- `QHBoxLayout` arranges its children horizontally
- When a layout is given to a widget with the `setLayout()` function, its contents become children of the widget.

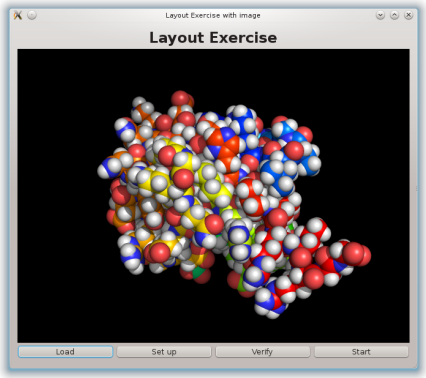
Layouts

```
MyWidget::MyWidget(QWidget *parent):
QWidget(parent)
{
  child1=new QLabel("Hello world");
  child2=new MyScoreDisplay;
  ...
  QHBoxLayout *h1=new QHBoxLayout;
  QHBoxLayout *h2=new QHBoxLayout;
  h1->addWidget(child1);
  h1->addWidget(child2);
  h2->addWidget(child3);
  h2->addWidget(child4);
  QVBoxLayout *v=new QVBoxLayout;
  v->addLayout(h1);
  v->addLayout(h2);
  setLayout(v);
}
```



Layouts exercise 1

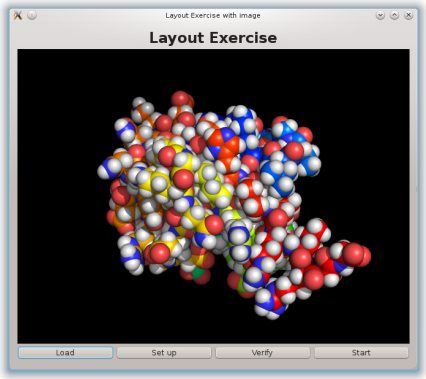
The directory `layouts` contains a set of files to build the widget shown to the right. But you have to finish it by using Qt layout managers to arrange the contained widgets (push buttons, labels, line editors and the image).



- Compile using the `make` command
- Run and feel the need of layouts
- Insert the appropriate layout code
- Run the corrected program. See what happens if you resize the window.

Layouts exercise 2

- In the directory containing the files for the Layout Exercise 1, there is an image file for the molecule "ubiq_spheres.png". What happens when you rename that file, or move your executable to another directory ? Why does this happen ?



Qt Resources

- Create a text file "myresources.qrc" in the layout exercise directory with the following contents

```
<!DOCTYPE RCC><RCC version="1.0">
 <qresource>
     <file>ubiq_spheres.png</file>
 </qresource>
 </RCC>
```

- Initialize the `QPixmap` in `MyWidget` like this:

```
   QPixmap imgdata(":/ubiq_spheres.png");
```

- Remove the `Makefile`, and compile in this way:

```
   qmake -project
   qmake
   make
```

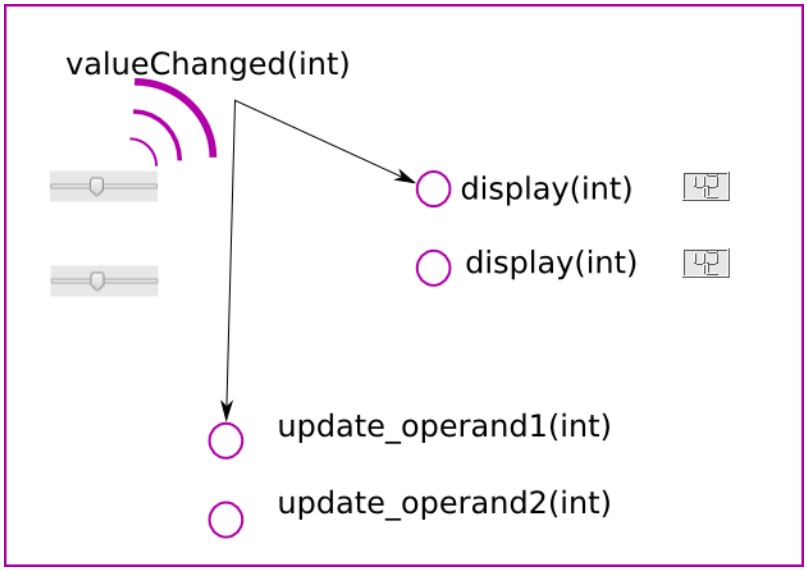
- Move the image and the executable to different directories now. The program should now still show the image even if it is not in the same path.

Qt Resources

- Icons and other data files which are needed for your GUI but are not meant to change frequently, can be compiled in to the executable
- Create a .qrc file as in the example
- Use qmake to generate a make file. It adds the `RESOURCES += abc.qrc` so that the application is compiled with the resources it needs
- Refer to resources with paths starting with `qrc:/` or just `:/`

Communication between widgets

Qt signals and slots mechanism



When a signal is emitted, all slots it is connected to (in any widget) are called.

Qt signals and slots

- Slots are ordinary functions
- Signals are declared, but not implemented by the user
- Standard Qt widget classes have a lot of signals and slots defined
- We can connect signals to slots as we like using the connect() function
- If a class has its own signals or slots, the `Q_OBJECT` macro needs to be used as shown

```
#ifndef MyBigProg_HH
#define MyBigProg_HH
#include <QtGui/QWidget>
class QLabel;
class QPushButton;
class QLineEdit;
class QString;
class MyBigProg : public QWidget {
  Q_OBJECT
public:
  MyBigProg(QWidget *parent=0);
private slots:
  void updateLastname(QString s);
  void updateFirstname(QString s);
  signals:
  void something_changed();
private:
  QString fullname, firstname,lastname;
  QLabel *namedisp, *flabel1, *flabel2;
  QLineEdit *field1, *field2;
  QPushButton *quitbutton;
};
#endif
```

Qt signals and slots

- Qt signals have nothing to do with UNIX signals! They are callbacks with multiple targets
- The `connect()` function is inheritted from `QObject`. An object does not need to be a widget to use signals and slots

```
MyBigProg::MyBigProg(QWidget *parent) :
QWidget(parent)
{
// define all widgets
// . . .
  connect(quitbutton,SIGNAL(clicked()),
         this,SLOT(close()));
  connect(field1,
      SIGNAL(textEdited(QString)),this,
      SLOT(updateFirstname(QString)));
  connect(field2,
      SIGNAL(textEdited(QString)),this,
      SLOT(updateLastname(QString)));
// The syntax is ...
// connect(sender,SIGNAL(signal_),
//           receiver,SLOT(slot_));
// connect(sender,&SenderType::signal_,
              receiver, &ReceiverType::
                   slot_);
// where signal_ and slot_ are function
// signatures without argument names
// Alternatively, (In Qt5 and C++11)
// connect(sender,&SenderType::signal_,
//         [](){lambda function code;});
```


Qt signals and slots

- You can connect a signal with many slots, many signals with a slot and signals with other signals
- When a signal is emitted all connected slots are automatically called

```
MyBigProg::MyBigProg(QWidget *parent) :
QWidget(parent)
{
// define all widgets
// put them in layouts
// . . .

  connect(quitbutton,SIGNAL(clicked()),
         this,SLOT(close()));
  connect(field1,SIGNAL(textEdited(
        QString)),
          this,SLOT(updateFirstname(
                QString)));
  connect(field2,SIGNAL(textEdited(
        QString)),
            this,SLOT(updateLastname(
                  QString)));

// The syntax is ...
// connect(sender,SIGNAL(signal_),
//           receiver,SLOT(slot_));
// where signal_ and slot_ are function
// signatures without argument names
```

Exercises: Qt signals and slots

Exercise 6.3:

The folder `widgets0` contains a simple widget class which demonstrates the use of signals and slots. Compile and run. Check that the widgets are now functional. Modify the program so that the "Quit" button is disabled until you click the check box.

Exercise 6.4:

The folder `customslot0` contains another simple widget demonstrating how to implement your own functionality using a class with its own slots. First compile and run to see what the program does, and then read the code to understand how.

Exercises: Qt signals and slots

Exercise 6.5:

The folder `browser0` contains a program that is almost identical to the layout example earlier. Instead of an image, it contains a `QWebView` object. Compile and run. It should work and create a simple we browser. But, the buttons or the URL input fields don't work. Insert appropriate signal-slot connections to make them work!

QString

- A string class from Qt, often passed around between Qt classes
- 2 bytes per character. Good for unicode etc.
- Provides many methods to ease common tasks on strings
- Can be converted to `std::string` and C-style strings

```
QString s{"Jülich.txt"};
if (s.endsWith(".png")) do_something();
QString s1{"Research"};
QString s4{s1.toLower()};
// s4="research"
QString s5{s1.replace(2,4, "a")};
//s5="reach"
str.replace("&","&amp;");
// replace all & with &amp;
str=" this   is an input\n";
str2=str.trimmed();
//str2="this   is an input";
str3=str.simplified();
//str3="this is an input";
QString st="Artificial intelligence is
     no "
+"match for natural stupidity";
QStringList words = st.split(" ");
words.sort();
words.join();
std::string ststr=st.toStdString();
```


Creating the main window

- Has menu bars, tool bars, status bars etc
- Has a "central widget" for the main functionality
- Created by inheriting `QMainWindow`

Creating the main window

```
#ifndef MAINWINDOW_HH
#define MAINWINDOW_HH
#include <QMainWindow>
class QAction;
class QMenu;
class MyApplication;
class MainWindow : public QMainWindow {
    Q_OBJECT
    MyApplication *g;
    QAction *aboutaction,*quitaction;
    QMenu *filemenu,*modemenu;
public:
    explicit MainWindow(QWidget *parent=0);
signals:
public slots:
private slots:
    void updateStatusBar();
    void showAboutBox();
private:
    void createActions();
    void createMenus();
    void createStatusBar();
};
```

- Different menus inside the menu bar are `QMenu` widgets
- The top menu bar, status bar etc already belong to `QMainWindow`. But they need to be filled and initialized.
- `QActions` are helper objects for menu bars and tool bars

Creating the main window

```
#include "MainWindow.hh"
#include "MyApplication.hh"
#include <QMenu>
#include <QMenuBar>
#include <QAction>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
  g = new MyApplication;
  setCentralWidget(g);
  createActions();
  createMenus();
  createStatusBar();
  setWindowIcon(QIcon(":/fig/icon.png"));
}
```

The constructor of a main window ...

- creates the central and any auxilliary widgets
- calls different functions to initialize menus etc
- optionally sets a window icon

Creating the main window

```
void MainWindow::createActions()
{
    aboutaction = new QAction(tr("About"),this);
    aboutaction->setStatusTip(tr("Show about box for this application"));
    connect(aboutaction,SIGNAL(triggered()), this, SLOT(showAboutBox()));
    quitaction = new QAction(tr("E&xit"),this);
    quitaction->setStatusTip(tr("Quit!"));
    connect(quitaction,SIGNAL(triggered()),this,SLOT(close()));
    ...
}
```

- An action can have a name and a status tip.
- Actions have a signal "triggered", which should be connected to something

Creating the main window

```
void MainWindow::createMenus()
{
    filemenu = menuBar()->addMenu(tr("&File"));
    filemenu->addAction(aboutaction);
    filemenu->addAction(quitaction);
}
void MainWindow::createStatusBar()
{
    statusBar();
    updateStatusBar();
}
```

- Fill the menu bar with some menus
- Fill each menu you add with some "actions"
- Initialize the status bar

Example/Exercise Qt and OpenGL

- The directory `examples/glscene` contains the files for a simple scene rendered with Qt openGL.
- The class `Entity` is a generic object with a few vertexes which can draw itself.
- The class `Tetrahedron` inherits from `Entity` and specifies particular properties for tetrahedra.
- The `MyGLScene` class sets up the openGL particulars and handles mouse events.
- Compile (make sure there is a line `QT += opengl` in the project file)with qmake and make, and run. Experiment by writing a `Cube` class and insert a few to the scene!

Hello Qml

Designing interfaces with QtQuick

```
import QtQuick 2.1
Rectangle {
  id: rect
  color: "green"

  Text {
    id: text
    anchors.centerIn: parent
    color: "red"
    text : "Hello"
  }
}
```

- Descriptive language to rapidly describe what you want to appear on the screen
- The above lines can be put in a file (`qmlhello.qml`) and previewed with `qmlscene qmlhello.qml`
- Interface logic can be implemented in Javascript
- Can easily talk to your C++ classes
- `http://www.qt-project.org`

Functionality through Qt C++ classes

Calling C++ functions

```
class DoSomething : public QObject{
    Q_OBJECT
public:
    Q_INVOKABLE double sum() const;
    Q_INVOKABLE QString greet() const;
private:
    std::array<double,10> dat{{3,2,4,3,5,4,6,5,7,6}};
};
```

- If you want to call a member function from within the qml interface, you have to put the macro `Q_INVOKABLE` in front of it, in addition to the `Q_OBJECT` for the class.

Calling C++ functions

```
    QGuiApplication a(argc, argv);
    DoSomething interf;
    QQuickView view;
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.rootContext()->setContextProperty("interf", &interf);
    view.setSource(QUrl("qrc:///ui.qml"));
    view.setGeometry(100, 100, 800, 480);
    view.show();
    return a.exec();
```

- In the main program, you can just declare an object of that type
- create a `QQuickView` object and pass it an object of the type with a name as shown
- give a `qml` file to describe the user interface
- and access the C++ functions directly from the `qml`

Calling C++ functions

```
Rectangle {
    id: rect
    function updateUI() {
        //interf.toggleEcho(button.pressed);
        text.text = interf.sum();
    }
    Rectangle {
        id: buttonl
        MouseArea {
            anchors.fill: parent
            onClicked: { buttonl.pressed = !buttonl.pressed; text.text = interf.
                greet(); }
        }
    }
}
```

- ... and access the C++ functions directly from the `qml`

Example 6.2: Calling C++ functions from Qml

The folder `examples/plugin_hello` contains the files necessary to make a small test program creating two buttons and some central text. Pressing the buttons calls two member functions from a C++ class and puts the results in the text in the user interface. This program is a minimal demo, and illustrates the techniques to invoke sophisticated functionality implemented in C++ from the Qml based user interface.