JÜLICH FORSCHUNGSZENTRUM

Programming in C++

Introduction to the language standard C++14 (and a preview of C++17)

29 May – 01 June 2017 | Sandipan Mohanty | Jülich Supercomputing Centre

JÜLICH FORSCHUNGSZENTRUM

Chapter 1

Introduction

C++

Express ideas in code *

- Specify actions to be executed by the machine
- Concepts to use when thinking about what can be done
- Direct mappings of built in operations and types to hardware
- Affordable and flexible abstraction mechanisms

C++ is a language for developing and using elegant and efficient abstractions

* Chapter 1, The C++ Programming Language, 4^{th} Edition, Bjarne Stroustrup

C++

Goals

- General purpose: no specialization to specific usage areas
- No over simplification that precludes a direct expert level use of hardware resources
- Leave no room for a lower level language
- What you don't use, you don't pay for

Goals

- Express ideas directly in code
- Express independent ideas independently in code
- Represent relationships among ideas directly in code
- Combine ideas expressed in code freely
- Express simple ideas with simple code

Programming Styles

A programming solution should be...

- Small
- Fast
- Correct
- Readable
- Maintainable

Programming styles in C++

- Procedural programming
- Data abstraction
- Object oriented programming
- Generic programming
- Functional programming

C++ is not restricted to any specific programming style. Choose any combination of the above that fits best with the problem you are trying to solve.

C++ in 2017

- Current standard, C++14. Follows C++11, precedes C++17, to be finalised later this year.
- Easier, cleaner and more efficient language

```
list<pair<string,size_t>> F;

for (auto entry : freq)
  F.push_back(entry);

F.sort([](auto i, auto j) {
  return i.second < j.second;
});
// with C++17 ...
for (auto planet : solar_system) {
  auto [Decln, RA] =
      planet.calc_coordinates(ut);
      // ...
}
```


Compiler support for C++11/C++14 (May 2017)

- **LLVM/clang++** : Complete (Version 3.4 and above). Useful error messages. Fast compilation. Generated binaries competitive with others. Own STL : libc++.
- **GCC/g++** : Complete (Version 5.0 and above). Mature code base. Own STL : libstdc++. In GCC 6.1.0, C++14 became the default language standard, and using any older standard requires `--std=c++98` or `--std=c++11` options.
- **Intel Compiler** : Practically complete (Version 2017.1). Great optimizer. Uses system STL. Not free.
- **IBM XLC Compiler** : Usable with version 14.1. System/external STL. Not free.

Course plan

In this course you will ...

- study illustrative code and do programming exercises
- learn C++ according to the current language standard
  - Essential C++ concepts
  - Classes and class hierarchies
  - Templates and generic programming
- learn to use the C++ standard library
  - STL containers and algorithms
  - Smart pointers, time library, random numbers, multi-threading ...
- learn to make graphical user interfaces for your C++ programs using Qt Quick

Resources

- Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14, Scott Meyers, **ISBN** 978-1491903995
- 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

Online Resources

- http://www.cplusplus.com/reference/
- http://en.cppreference.com/
- http://www.qt.io
- http://www.threadingbuildingblocks.org
- https://github.com/isocpp/CppCoreGuidelines

```
git clone https://github.com/isocpp/CppCoreGuidelines.git
```


Elementary Constructs

Getting started

The first step: Hello World!

- Set up environment for using the a recent version of `g++`. For example, using environment modules:
  `module load gcc/7.1.0`
- Find your favourite text editor and type in the simple "hello world" program in this slide
- Compile and run the program using the following commands :

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

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

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


Getting started

Useful aliases

We will use thin wrappers `G` and `A` (in `$course_home/bin`) for `g++` and `clang++`, which fill in frequently used options. They are inserted into your `PATH` when you type
`module load course`

- `G` is (roughly) a shorthand for

```
g++ --std=c++14 -pedantic $*
```

- `A` is (roughly) a shorthand for

```
clang++ -std=c++14 -pedantic -stdlib=libc++ $*
```

Whichever compiler you choose to use, load the corresponding module : `module load gcc/7.1.0` or `module load gcc/6.3.0` or `module load llvm/4.0.0`

Getting started

Comments on the hello world program

- **#include** <header> tells the preprocessor to include the contents of header in the current "translation unit"
- **#include** "somefile" searches for a file called somefile and imports its contents
- The search for headers is performed in an implementation defined way.

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

- Comments in C++ start with // and run to the end of the line
- Think of std::cout as a sink which prints what you throw at it on the screen

Getting started

Comments on the hello world program

- All C++ programs must contain a unique main() function
- All executable code is contained either in main(), or in functions invoked directly or indirectly from main()
- The return value for main() is canonically an integer. A value 0 means successful completion, any other value means errors. UNIX based operating systems make use of this.
- In a C++ main function, the **return** 0; at the end of main() can be omitted.

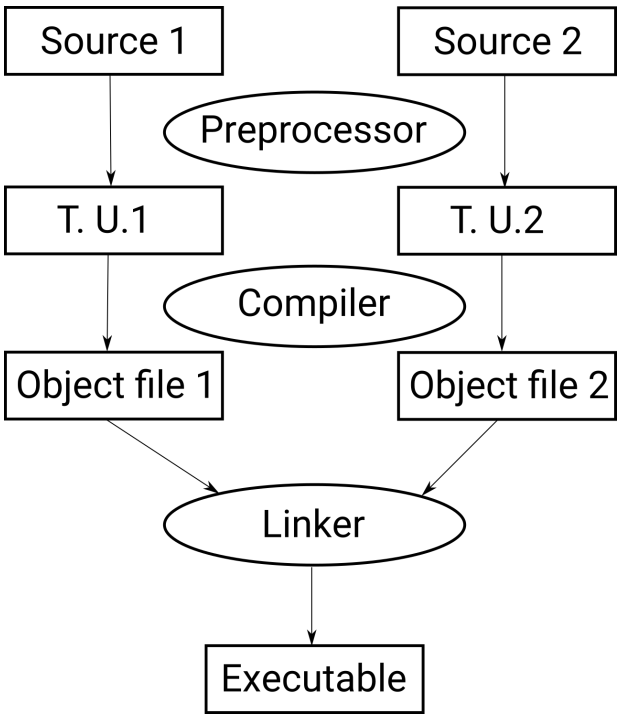
Getting started

Command line arguments

- Main can be declared as `int main (int argc, char *argv[])` to accept parameters from the command line
- The entire command line is broken into a sequence of character strings (by unquoted and unescaped white space) and passed as the array `argv`
- The name of the program is the first string in this list, `argv[0]`. Therefore `argc` is never 0.

```
// Hello xyz
#include <iostream>
int main(int argc, char *argv[])
{
  std::cout<<"Hello, ";
  if (argc > 1)
    std::cout <<argv[1]<< "!\n";
  else
    std::cout<<"world!\n";
}

G hello_xyz.cc
a.out bird
```


The compilation process



The compilation process

Exercise 1.1:

By passing the additional compiler option `-v` to `g++`, recompile the hello world program and observe the logs from different stages of the compilation process.

- Where does the `g++` search for headers ?
- Where does `clang++` look for headers ? Try `clang++ -v hello.cc` and `clang++ -stdlib=libc++ -v hello.cc`, and observe the differences.

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
- Recommendation: 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!

Types, variables and declarations

```
long n_particles{1000}; // 64 bits interpreted with rules for integers
const double G{6.67408e-11}; // 64 bits interpreted as a floating point number
char user_choice='y';
bool received_ngb_updates=false;
```

- A "type" defines the possible values and operations for an object
- An "object" is some memory holding a value of a certain type
- A "value" is bits interpreted according to a certain type
- A "variable" is a named object
- Every expression has a "type"
- A "declaration" is a statement introducing a name into the program
- C++ is a statically typed language

Types, variables and declarations

- Types like **`char`**, **`int`**, **`float`**, **`double`** are known as fundamental types
- Fundamental types can be inter-converted when possible
- Arithmetic operations $+$, $-$, $*$, $/$, as well as comparisons $<$, $>$, $<=$, $>=$, $==$, $!=$ are defined for the fundamental types, and mapped in an obvious way to low level instructions
- As in many languages, $=$ is assignment where as $==$ is equality comparison
- Note how variables are "initialized" to sensible values when they are declared

Initialization

- Both **int** `i=23` and **int** `i{23}` are valid initializations
- The newer curly bracket form should be preferred, as it does not allow "narrowing" initializations:
  int `i{2.3};` *//Compiler error*
- The curly bracket form can also be used to initialise C++ collections:

```
std::list<double> masses{0.511, 938.28, 939.57};
std::vector<int> scores{667,1}; // Vector of two elements, 667 and 1
std::vector<int> lows(250,0); // vector of 250 zeros
```

- The old initialisation notation with `()` is needed in some cases
- Variables can be declared anywhere in the program. So, avoid declaring a variable until you have something meaningful to store in it

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, vectors etc.
double m=0.5; // Initialising with '='
// is ok for simple variables, but ...
int k=5.3; // Allowed, although the
// integer k stores 5, and not 5.3
int j{5.3}; // Helpful compiler error.
int i{}; // i=0
vector<int> u{4,0};// u={4, 0}
vector<int> v(4,0);// v={0, 0, 0, 0}
```

- Variables can be initialised at declaration with a suitable value enclosed in `{}`
- Pre-C++11, only the `=` and `()` notations (also demonstrated in the left panel) were available. Initialising non trivial collections was not allowed.

- Recommendation: Use `{}` initialisation syntax as your default. A few exceptional situations requiring the `()` or `=` syntax can be seen in the left panel.

The keywords auto and decltype

```
int sqr(int x)
{
  return x*x;
}
int main()
{
  char oldchoice{'u'},choice{'y'};
  size_t i=20'000'000;//grouping in C++14
  double electron_mass{0.511};
  int mes[6]{33,22,34,0,89,3};
  bool flag{true};
  decltype(i) j{9};
  auto positron_mass = electron_mass;
  auto f = sqr; // Without "auto", f can
  // be declared like this:
  //int (*f)(int)=&sqr;
  std::cout << f(j) << '\n';
  auto muon_mass{105.6583745};
  // In C++17, if somefunc() returns
  // tuple<string, int, double>
  auto [name, nspinstates, lifetime]
       = somefunc(serno);
}
```

- 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"
- In C++17, several variables can be declared and initialised together from a suitable tuple, as shown

Scope of variable names

```
double find_root()
{
  if (not initialized()) init_arrays();
  for (int i=0;i<N;++i) {
    // loop counter i defined only
    // for this "for" loop.
  }
  double newval=0; // This is ok.
  for (int i=0;i<N;++i) {
    // The counter i here is a
    // different entity
    if (newval < 5) {
        string fl{"small.dat"};
        // do something
    }
    newval=...;
    cout << fl << '\n'; // Error!
  }
  int fl=42; // ok
  // C++17
  if (auto fl=filename; val < 5) {
    // fl is available here
  } else {
    // fl is also available here
  }
}
```

- Variable declarations are allowed throughout the code
- A scope is :
  - A block of code, bounded by { and }
  - A loop or a function body
  - (C++17) Both **`if`** and **`else`** parts of an **`if`** statement
- Variables defined in a scope exist from the point of declaration till the end of the scope. After that, the name may be reused.

Example 1.1:

The programs `examples/vardecl.cc`, `examples/vardecl2.cc` and `vardecl_in_if.cc` demonstrate variable declarations in C++. Use

```
g++ program.cc -o program
```

to compile. For the last one, you will need the option `-std=c++1z`.

C++ namespaces

```
// Somewhere in the header iostream
namespace std {
  ostream cout;
}

// In your program ...
#include <iostream>

int main()
{
  for (int cout=0;cout<5;++cout)
    std::cout<<"Counter="<<cout<<'\n';

  // Above, plain cout is an integer,
  // but std::cout is an output stream
  // The syntax to refer to a name
  // defined inside a namespace is:
  // namespace_name::identifier_name
}
```

- A **namespace** is a named context in which variables, functions etc. are defined.
- The symbol `::` is called the scope resolution operator.
- **using namespace** `blah` imports all names declared inside the **namespace** `blah` to the current scope.

C++ namespaces

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

- No name clash due to the same name in different namespaces
- Functions defined inside namespaces need to be accessed using the same scope rules as variables

C++ namespaces

```
// examples/multdef/somehdr.hh
extern int D;
bool f(int);
bool g(int);
// examples/multdef/file1.cc
#include "somehdr.hh"
bool f(int x)
{
  return x>D;
}
// examples/multdef/file2.cc
#include "somehdr.hh"
bool g(int x)
{
  return x*x<=D;
}
// examples/multdef/main.cc
#include "somehdr.hh"
int D=56;
int main()
{
  if (f(23) or g(33)) return 1;
  return 0;
}
```

- Variables declared outside any function are global variables and live for the entire duration of the program
- For non-constant variables declared in a namespace (global or not), the **`extern`** keyword can be used to avoid multiple definitions

Example 1.2:

The files in the directory `examples/multdef` illustrate the need and use of the **`extern`** keyword. The global variable `D` is used in functions `f(int)` and `g(int)`, defined in `file1.cc` and `file2.cc`. It is only given a value in `main.cc`, where it is defined without the **`extern`** keyword. To compile, you need to do the following steps:

```
G -c file1.cc
G -c file2.cc
G -c main.cc
G file1.o file2.o main.o -o multdef
```

Check what happens if you remove the **`extern`** in the header. In C++17, you can also declare `D` to be "inline", and assign a value in the header itself. There is also need to define it explicitly in any source file. Check this!

unnamed namespaces

```
// examples/unnamednsp/somehdr.hh
bool f(int);
bool g(int);
// examples/unnamednsp/file1.cc
// namespace {
int D=42;
// }
bool f(int x)
{
  return x>D;
}
// examples/unnamednsp/file2.cc
// namespace {}
int D=23;
// }
bool g(int x)
{
  return x*x<=D;
}
// examples/unnamednsp/main.cc
int main()
{
```

- Global variables such as `D` in `file1.cc` and `file2.cc` will clash during linking
- They can be enclosed in anonymous namespaces as shown
- The unnamed namespace is a uniquely named namespace with internal linkage

Unnamed namespaces

Example 1.3:

The directory `examples/unnamednsp` contains the same files as `examples/multdef`, with unnamed namespaces enclosing the variables `D` in files `file(1|2).cc`. Comment/uncomment the namespace boundaries and try to build. With the unnamed namespaces in place, the variables in the two files are independent of each other.

Inline namespaces and versioning

```
namespace myl {
  inline namespace V200 {
    double solve();
  }
  namespace V150 {
    double solve();
  }
  namespace V100 {
    double solve();
  }
}
void elsewhere(){
  myl::solve(); //myl::V200::solve()
  myl::V150::solve(); //as it says.
}
```

- Identifiers declared inside an **`inline`** namespace are automatically exported to the surrounding namespace, as if there was an implicit **`using namespace ...`**
- Useful tool for vending a library where you must support multiple versions of something

C++ namespaces: Final comments

```
//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"};
  }
}
// With C++17 ...
namespace mylibrary::onefeature {
    double solve(int i);
}
int main()
{
  namespace USOH=UnitedStates::OH;
  std::cout<<"London is"
           <<USOH::London<<'\n';
}
```

- **namespaces** can be nested. In C++17, direct nested declarations are allowed.
- Long **namespace** names can be given aliases
- Tip1: Don't indiscriminately put **using namespace ...** tags, especially in headers.
- Tip2: The purpose of **namespaces** is to avoid name clashes. Not taxonomy!

Preprocessor directives

Example 1.4:

The program `examples/bits_in_int.cc` counts the number of 1 bits in the binary representation of the numbers you enter. Compile and run to check. Then use the C preprocessor `cpp` to process the file. The output is what the compiler receives.

Preprocessor directives

```
#include <iostream>
unsigned char n_on_bits(int b)
{
    #define B2(n) n, n+1, n+1, n+2
    #define B4(n) B2(n), B2(n+1), \
     B2(n+1), B2(n+2)
    #define B6(n) B4(n), B4(n+1), \
     B4(n+1), B4(n+2)
    const unsigned char tbl[256]
    { B6(0), B6(1), B6(1), B6(2) };
    #undef B6

    return tbl[b       & 0xff] +
           tbl[(b>> 8) & 0xff] +
           tbl[(b>>16) & 0xff] +
           tbl[(b>>24) & 0xff];
}
int main()
{
    #ifdef B6
    #error B6 should not be defined!
    #endif
...
```

- Processed before the compiler checks language syntax
- **#define** `pat sub` substitutes any occurrence of `pat` in the source with `sub`
- Definitions valid until **#undef**. Scoping rules of the language do not apply.
- Uses: reduce code text, tailored compiler errors, hints to the compilers (e.g. **#pragma**)

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
- In C++17, you can leave out the message parameter

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

C-style enumerations

```
enum color { red, green, blue };
using rgbtype = array<float, 3>;
rgbtype adjust(rgbtype rgb)
{
  float factors[3]{0.8,1.0,0.93};
  rgb[red] *= factors[red];
  rgb[green] *= factors[green];
  rgb[blue] *= factors[blue];
  return rgb;
}
```

- A type whose instances can take a few different values (e.g., directions on the screen, colours, supported output modes ...)
- Less error prone than using integers with ad hoc rules like, "1 means red, 2 means green ..."
- Internally represented as (and convertible to) an integer
- All type information is lost upon conversion into an integer

Scoped enumerations

- Defined with **`enum class`**
- Must always be fully qualified when used: `traffic_light::red` etc.
- No automatic conversion to **`int`**.
- Possible to use the same name, e.g., `green`, in two different scoped enums.

```
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) ...;
```


Pointers and references

```
int i{5};
int * iptr{&i}; // iptr points at i
i+=1;
std::cout << *iptr ; // 6
(*iptr)=0;
std::cout << i ; // 0
int & iref{i}; // iref "refers" to i
iref=4;
std::cout << i ; // 4
```

- A pointer is an integral type to store the memory address of objects
- If `iptr` is a pointer, `*iptr` is the object it is pointing at
- For a variable `X`, its memory address can be obtained using `&X`
- A reference is a restricted pointer which must always point at the same object
- When in use, a reference appears as if it were a regular variable

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!

C++ strings

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
  char name[100], *bkpname;
  int nmsize,i;
  printf("What is your name ? ");
  if ( fgets(name,100,stdin) ) {
    nmsize = strlen(name);
    bkpname = (char *) malloc((nmsize+1)*
          sizeof(char));
    bkpname = strdup(name);
    for (i=0;i<nmsize;++i) {
      name[i]=toupper(name[i]);
    }
    printf("%s <--> %s\n",bkpname,name);
  }
  if (bkpname) free(bkpname);
  return 0;
}
```

```
#include <iostream>
#include <string>
int main()
{
  std::string name, bkpname;
  std::cout << "What is your name ? ";
  std::getline(std::cin,name);
  if (not name.empty()) {
    bkpname = name;
    for (int i=0;i<name.size();++i) {
      name[i]=toupper(name[i]);
    }
    std::cout<<bkpname<<" <--> "
      <<name<<"\n";
  }
}
```

- Cleaner, easier to maintain code using C++ strings.

Did you notice the memory leak in the C code ?

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 1.5:

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

JÜLICH FORSCHUNGSZENTRUM

Exercise 1.2:

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 1.6:

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

Arrays

```
double A[10]; // C-style array
int sz;
std::cin >> sz;
int M[sz]; // Not allowed!
// Since C++11 ...
#include <array>
...
std::array<double,10> A; // On stack
// Like the C-style array, but obeys
// C++ standard library conventions.
for (size_t i=0;i<A.size();++i) {
  P*=A[i];
}
std::vector<double> B(sz,3.0);
```

- Fixed length C-style arrays exist, but do not know their own size
- Variable length arrays of C99 are illegal in C++, although tolerated with warnings in many compilers

- `std::array<type,size>` is a compile-time fixed length array obeying STL conventions
- `std::vector<type>` is a dynamically allocated resizeable array type

Branches/Selections

```
if (condition) {
  // code
} else if (another condition) {
  // code
} else {
  //code
}
switch (enumarable) {
case 1:
  // code
  break;
case 2:
  // code
  break;
default:
  // code
};
x = N>10 ? 1.0 : 0.0;
```

- The **`if`** and **`switch`** constructs can be used to select between different alternatives at execution time.
- Conditional assignments are frequently written with the ternary operator as shown

Loops

```
for (initialisation;condition;increment){
  // Loop body
}
for (int i=0;i<N;++i) s+=a[i];
while (condition) {}
while (T>t0) {}
do {} while (condition);
do {
} while (ch=='y');
for (variable : collection) {}
for (int i : {1,2,3}) f(i);
for (int i=0;i<N;++i) {
  if (a[i]<cutoff) s+=a[i];
  else break;
}
for (std::string s : names) {
  if (s.size()>10) {
    longnames.push_back(s);
    continue;
  }
  // process other names
}
```

- Execute a block of code repeatedly
- Loop counter for the **`for`** loop can and should usually be declared in the loop head
- The **`break`** keyword in a loop immediately stops the loop and jumps to the code following it
- The **`continue`** keyword skips all remaining statements in the current iteration, and continues in the loop

Exercise 1.3:

What is the largest number in the Fibonacci sequence which can be represented as a 64 bit integer ? How many numbers of the sequence can be represented in 64 bits or less ? Write a C++ program to find out. Use only concepts presented so far.

Exercise 1.4:

Finish the program `exercises/gcd.cc` so that it computes and prints the greatest common divisor of two integers. The following algorithm (attributed to Euclid!) achieves it :

1. Input numbers : smaller , larger
2. remainder = larger mod smaller
3. larger = smaller
4. smaller = remainder
5. if smaller is not 0, go back to 2.
6. larger is the answer you are looking for

Range based for loops

```
// 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

```
// 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

Example 1.7:

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

Functions

```
return_type function_name(parameters)
{
  // function body
}
double sin(double x)
{
  // Somehow calculate sin of x
  return answer;
}
int main()
{
  constexpr double pi{3.141592653589793};
  for (int i=0;i<100;++i) {
    std::cout << i*pi/100
       << sin(i*pi/100) <<"\n";
  }
  std::cout << sin("pi") <<"\n"; //Error!
}
```

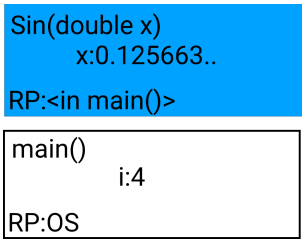
- All executable code is in functions
- Logically connected reusable blocks of code
- A function must be called with values called arguments.
- The type of the arguments must match or be implicitly convertible to the corresponding type in the function parameter list

Functions: Syntax

```
// A function prototype
bool pythag(int i, int j, int k);
// and a function definition
int hola(int i, int j)
{
  int ans{0};
  if (pythag(i,j,23)) {
  // A prototype or definition must be
  // visible in the translation unit
  // at the point of usage
    ans=42;
  }
  return ans;
}
// Definition of pythag
bool pythag(int i, int j, int k)
{
  // code
}
// Trailing return type (since C++11)
auto f(double x, double y)->double
{
  // function body
}
```

- A function prototype introduces a name as a function, its return type as well as its parameters
- Functions can live freely
- C++11 introduced an alternative syntax with the return type coming after the parameters (last example in the left panel)

Functions at run time




```
double sin(double x)
{
  // Somehow calculate sin of x
  return answer;
}
int main()
{
  constexpr double pi{3.141592653589793};
  for (int i=0;i<100;++i) {
    std::cout << i*pi/100
       << sin(i*pi/100) <<"\n";
  }
  std::cout << sin("pi") <<"\n"; //Error!
}
```

- When a function is called, it is given a "workbook" in memory called a stack frame
- Argument values are copied to the stack frame, and a return address is stored.
- (Non-static) local variables are allocated in the stack frame
- When the function concludes, execution continues at the stored return address, and the stack frame is destroyed

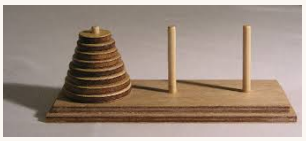
Recursion

- SP=<in factorial()> n=1 u=1 RP=<4>
- SP=<in factorial()> n=2 u=2 RP=<4>
- SP=<in factorial()> n=3 u=3 RP=<4>
- SP=<in factorial()> n=4 u=4 RP=<9>
- SP=<in someother()> RP=<...>

```
1  unsigned int factorial(unsigned int n)
2  {
3    int u=n; // u: Unnecessary
4    if (n>1) return n*factorial(n-1);
5    else return 1;
6  }
7  int someother()
8  {
9    factorial(4);
10 }
```

- A function calling itself
- Each level of "recursion" has its own stack frame
- Function parameters are copied to the stack frame
- Local variables at different levels of recursion live in their own stack frames, and do not interfere

Example 1.8:



The tower of Hanoi is a mathematical puzzle with three towers and a set of disks of increasing sizes. At the beginning, all the disks are at one tower. In each step, a disk can be moved from one tower to another, with the rule that a larger disk must never be placed over a smaller one. The example `examples/honoi.cc` solves the puzzle for a given input number of disks, using a recursive algorithm. Test the code and verify the solution.

static variables in functions

```
void somefunc()
{
  static int ncalls=0;
  ++ncalls;
  // code --> something unexpected
  std::cerr << "Encountered unexpected"
    << "situation in the " << ncalls
    << "th call to " << __func__ << "\n";
}
```

- Private to the function, but survive from call to call.
- Initialisation only done on first call.
- Aside: The built in macro `__func__` always stores the name of the function

Function overloading

```
int power(int x, unsigned int n)
{
  ans=1;
  for (;n>0;--n) ans*=x;
  return ans;
}
double power(double x, double y)
{
  return exp(y*log(x));
}
```

```
double someother(double mu,
                 double alpha,
                 int rank)
{
  double st=power(mu,alpha)*exp(-mu);
  if (n_on_bits(power(rank,5))<8)
    st=0;

  return st;
}
```

- The same function name can be used for different functions if the parameter list is different
- Function name and the types of its parameters are combined to create an "internal" name for a function. That name must be unique
- It is not allowed for two functions to have the same name and parameters and differ only in the return value

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!
- Different popular use: define the entire function (even if it is large) in the header file, as identical inline objects in multiple translation units are allowed. (E.g. header only libraries)

Exercise 1.5:

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.

Function return values as auto

- Automatic type deduction methods of C++14 can be used for the return values in function definitions
- Return type ambiguity will be a compiler error in such situations
- `decltype(auto)` can also be used like **`auto`** for the return type, along with the different type deduction rules which apply for `decltype(auto)`

```
auto greet(std::string nm)
{
    for (auto & c: nm) c=std::toupper(c);
    std::cout << nm << std::endl;
    return nm.size()>10;
}
```

Exercise 1.6:

The program `exercises/autoret.cc` illustrates the use of **`auto`** as the function return type. What happens if you have multiple return statements ?

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 C-style arrays on the stack, if obviously 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`**
- Integers of type **`constexpr`** can be array sizes

Exercise 1.7: constexpr

The program `exercises/constexpr0.cc` demonstrates the use of constexpr functions. How would you verify that it is really evaluated at compile time ? Try to compile in the following cases:

- The code as is.
- After uncommenting the line with a **`static_assert`**
- Keeping the static assert, but giving it the correct value to check against
- Keeping the static assert with the correct value,
  - removing the **`constexpr`** from the function
  - replacing the **`constexpr`** with a **`const`** in the function

Restrictions on functions declared constexpr

- Must be "pure" functions. Can not alter global state.
- For C++11:
  - Must consist of a single return statement
  - Can not contain loops or temporary local variables, loops or **`if`** statements
- C++14 introduced multi-line **`constexpr`** functions with loops, local variables and branches.

Exercise 1.8: constexpr

The file `exercises/constexpr1.cc` has essentially the same code as in the previous exercise, but replaces the **`constexpr`** function with one that uses a loop and a local variable. Check that it behaves as expected in C++14, but not with C++11.

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 1.9: Strings and I/O

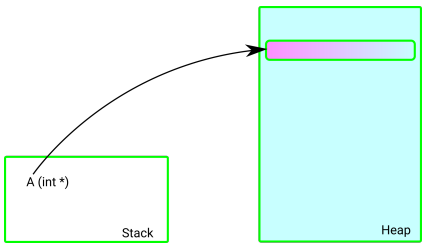
Write a simple program to find the largest word in a plain text document.

Heap vs stack

```
double f(double x)
{
  int i=(int) x; // stack variable
  double M[1000][1000][1000]; // Oops!
  M[123][344][24]=x;
  return x-M[i][555][1];
}
int main()
{
  std::cout << f(5) << "\n";
  // Immediate SEGFAULT
}
```

- Variables in a function are allocated on the stack, but sometimes we need more space than what the stack permits
- We do not know how much space we should reserve for a variable (e.g. a `string`)
- We need a way to allocate from the "free store"

Heap vs stack




```
void f()
{
  int *A = new int[1000000];
  // use A
  delete [] A;
}
```

Note: Heap allocation and deallocation are slower than those on the stack! Don't put everything on the heap.

- The pointer `A` is still on the stack. But it holds the address of memory allocated by the **`new`** operator on the "heap"
- Memory allocated from the heap stays with your program until you free it, using **`delete`**
- If you forget the address of the allocated memory, you have a memory leak

Object lifetime management with smart pointers

Smart pointers in C++

- 3 kinds of smart pointers were introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`
- `unique_ptr` claims exclusive ownership of the allocated array. When it runs out of its scope, it calls delete on the allocated resource. It is impossible to "forget" to delete the memory owned by `unique_ptr`
- Several instances of `shared_ptr` may refer to the same block of memory. When the last of them expires, it cleans up.
- Helper functions `make_unique` and `make_shared` can be used to allocate on heap and retrieve a smart pointer to the allocated memory

Dynamic memory with smart pointers

```
using big = std::array<int, 1000000>;
int f()
{
  auto u1=std::make_unique<big>();
  // use u1
} // u1 expires, and frees the allocated memory
```

- Current recommendation: avoid free **new**/**delete** calls in normal user code
- Use them to implement memory management components
- Use `unique_ptr` and `shared_ptr` to manage resources
- You can then assume that an ordinary pointer in your code is a "non-owning" pointer, and let it expire without leaking memory

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 (first choice):

```
auto c = make_unique<complex_number>(1.2,4.2); // on the heap
int asize=100; // on the stack
auto darray = make_unique<double[]>(asize);
// The stack frame contains the unique_ptr variables c and darray.
// The memory locations they point to on the other hand, are not
// on the stack, but on the heap. But, you don't need to worry about
// releasing that memory explicitly. If you don't have any way of
// accessing the resource (the pointers expire), the memory will be
// freed for you.
//
```

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 (second choice):

```
complex_number *c = new complex_number{1.2,4.2}; // on the heap
int asize=100; // on the stack
double *darray = new double[asize];
// The stack frame contains the pointer variables c and darray
// The memory locations they point to on the other hand, are not
// on the stack, but on the heap. Unless you release that memory
// explicitly, before the stack variables expire, there will be
// a memory leak.
delete c;
delete [] darray;
```


Evaluation order

Sub-expression evaluation

Guess the output!

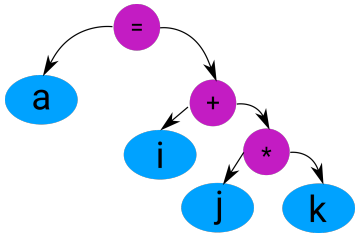
```
int i=0;
std::cout << ++i <<'\t'<< i <<'\t' << i++ <<"\n";
```

- 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");
```


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

Sub-expression evaluation II

4. Value calculations of built-in post increment (`i++`) and post decrement (`i--`) operators are sequenced before their side effects
5. Side effect calculations of built-in pre-increment (`++i`) and pre-decrement (`--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`

Sub-expression evaluation III

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
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 1.9: 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 ill structured code like this.

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

```
#include <stdexcept>

double f(double x)
{
  double answer=1;
  if (x>=0 and x<10) {
    while (x>0) {
      answer*=x;
      x-=1;
    }
  } else {
    throw std::domain_error("Value "+
      std::string(x)+" is out of range");
  }
  return answer;
}
```

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

Assertions

```
#include <cassert>
bool check_things()
{
    // false if something is wrong
    // true otherwise
}
double somewhere()
{
   // if I did everything right,
   // val should be non-negative
   assert(val>=0);
   assert(check_things());
}
```

- After we are satisfied that the program is correctly implemented, we can pass `-DNDEBUG` to the compiler, and skip all assertions.

- **`assert`**`(condition)` aborts if `condition` is false
- Used for non-trivial checks in code during development. The errors we are trying to catch are logic errors in implementation.
- If the macro `NDEBUG` is defined before including `<cassert>` **`assert`**`(condition)` reduces to nothing

Exercise 1.10:

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.

Exercise 1.11:

Handle invalid inputs in your `gcd.cc` program so that if we call it as `gcd apple orange` it quits with an understandable error message. Valid inputs should produce the result as before.

Chapter 2

C++ abstraction mechanisms

C++ classes

```
struct Vector3 {
  double x,y,z;
  inline double dot(Vector3 other)
  {
    return x*other.x +
           y*other.y +
           z*other.z;
  }
};

void somefunc()
{
  int a, b, c; // On the stack
  Vector3 d,e,f; // On the stack
  // ...
  if (d.dot(e)<d.dot(f)) doX();
}
```

- Use defined data types
- Identifies a concept
- Encapsulates the data needed to describe it
- Specifies how that data may be manipulated
- Objects of user defined types can live on the stack

C++ classes

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

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`"

Defining member functions outside the struct

```
struct complex_number
{
  complex_number add(complex_number b);
};

complex_number 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.

Member functions' promise to not change object

```
struct complex {
  double m_real, m_imag;
  double modulus();
  complex sub(const complex b);
};

void somewhere_else()
{
  complex z1,z2;
  z1.sub(z2);
  // We know z2 didn't change.
  // But did z1 ?
}
```

- Member functions have an implicit argument: the calling instance, through the **this** pointer. Where do we put a **const** qualifier, if we want to express that the calling instance must not change ?

Member functions' promise to not change object

```
struct complex {
  double m_real, m_imag;
  double modulus();
  complex sub(const complex b) const;
};

void somewhere_else()
{
  complex z1,z2;
  z1.sub(z2);
  // We know z2 didn't change.
  // We know z1 didn't change,
  // as we called a const member
}
```

- Member functions have an implicit argument: the calling instance, through the **this** pointer Where do we put a **const** qualifier, if we want to express that the calling instance must not change ?
- Answer: After the closing parentheses of the function signature.

Overloading member functions

```
struct complex_number
{
  complex_number multiply(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};
complex *b=new complex{3,4}
double c;
...
a=b->multiply(c);
```

- Member functions can and often are overloaded
- Member functions can be accessed from a pointer to an object using `->` instead of `.` : `b->multiply(c)` means `(*b).multiply(c)`

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);
```

- Operators like $+$, $-$, $*$, $/$, $<<$, $>>$, $|$, ... are functions with a special calling syntax

Exercise 2.1:

The file `complex_numbers.cc` contains a short implementation of a complex number type, with the syntax discussed up to this point. Change the member functions doing the mathematical operations into operators, and modify their usage in the main function.

Some example classes

```
class Angle {
  double rd = 0;
public:
  enum unit {
      radian,
      degree
  };
  Angle operator-(Angle a) const ;
  Angle operator+(Angle a) const ;
  Angle operator-=(Angle a) ;
```

```
class Vector3
{
public:
  enum crdtype {cartesian=0,polar=1};
  inline double x() const {return dx;}
  inline void x(double gx) {dx=gx;}
  double dot(const Vector3 &p) const;
  Vector3 cross(const Vector3 &p) const;
```

```
class IsingLattice {
public:
  using update_type = std::pair<size_t,
        size_t>;
  IsingLattice();
  IsingLattice(size_t Nx, double JJ);
  void setLatticeSize(size_t ns);
```

```
class KMer {
public:
  Nucleotide at(size_t i);
  bool operator==(const KMer &);
```

```
class SimulationManager {
public:
  void loadSettings(std::string file);
  bool checkConfig();
  void start();
```

Datatypes

Type	Bits	Value
Float	0100 0000 0100 1001 0000 1111 1101 1011	3.1415927
Int	0100 0000 0100 1001 0000 1111 1101 1011	1078530011

- Same bits, different rules $\Longrightarrow$ different type

From arbitrary collection of members to a new "data type"

```
class Date {
  int m_day, m_month, m_year;
public:
  static Date today();
  Date operator+(int n) const;
  Date operator-(int n) const;
  int operator-(const Date &) const;
};
```

- Make sure every way to create an object results in a valid state
- Provide only those operations on the data which keep the essential properties intact

C++ classes: user defined data types

```
struct Vector3 {
  double x,y,z;
  inline double dot(Vector3 other)
  {
    return x*other.x +
           y*other.y +
           z*other.z;
  }
};
struct complex_number {
  double realpart, imagpart;
  double modulus();
  // etc.
};
// But also,
struct Cat {
}; // If it helps solve your problems
struct HonoiTower {
  // To represent disks on one tower
};
```

- "Nouns" important in expressing ideas about the problem being solved are candidates to become classes
- Need not be profound and universally usable. If it helps solving a problem clearly and efficiently, write it.
- Use `namespaces` and local classes to avoid name conflicts

Object creation: constructors

- In C++, initialisation functions for a struct have the same name as the struct. They are called `constructors`.

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

- Alternative syntax to initialise variables in constructors

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

- A structure can have as many constructors as it needs.

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

- Constructors may be (and normally are) overloaded.
- When a variable is declared, a constructor with the appropriate number of arguments is implicitly called
- 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; // C++11 or later!
  double imaginary = 0;
};
...
complex_number a(4.3,23.09),b;
```

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

Freeing memory for user defined types

```
struct darray
{
  double *data=nullptr;
  size_t sz=0;
  darray(size_t N) : sz{N} {
    data = new double[sz];
  }
};

double tempfunc(double phasediff)
{
  // find number of elements
  darray A{large_number};
  // do some great calculations
  return answer;
}
```

What happens to the memory ?

The struct `darray` has a pointer member, which points to dynamically allocated memory

- When the life of the variable `A` ends the member variables (e.g. the pointer `data`) go out of scope.
- How does one free the dynamically allocated memory attached to the member `data` ?

Freeing memory for user defined types

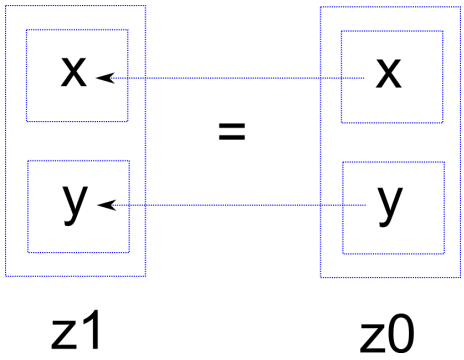
```
struct darray
{
  double *data=nullptr;
  size_t sz=0;
  darray(size_t N) : sz{N} {
    data = new double[sz];
  }
  ~darray() {
    if (data) delete [] data;
  }
};

double tempfunc(double phasediff)
{
  // find number of elements
  darray A{large_number};
  // do some great calculations
  return answer;
}
```

For any struct which explicitly allocates dynamic memory

- We need a function that cleans up all explicitly allocated memory in use. Such functions are called destructors, and have the name ~ followed by the struct name.
- Destructors take no arguments, and there is exactly one for each struct
- The destructor is automatically called when the variable expires.

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
```

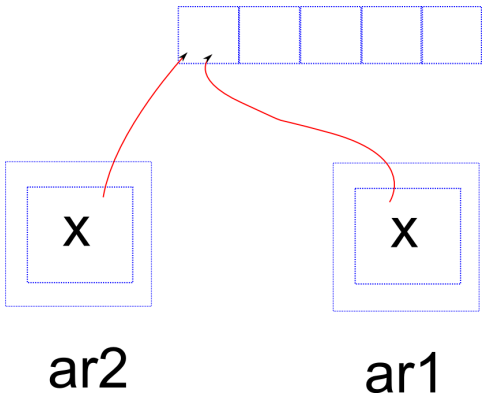


What values should the members get ?

- 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;
};
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
```

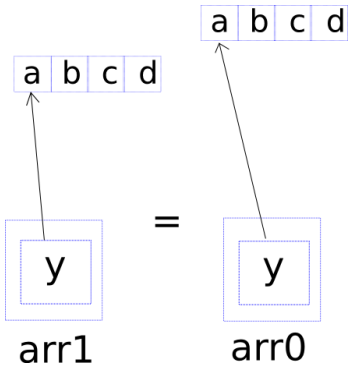


Copying pointers with dynamically allocated memory

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

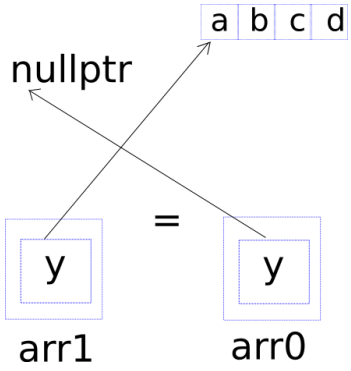
Copying and assignments

```
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 semantics 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 (or zero)

- 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. "Rule of zero"

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

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

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 implementer of the class then loses the freedom to change internal organisation of the class for efficiency or other reasons

C++ classes

```
class complex_number
{
public:
  complex_number(double re, double im)
  : m_real(re), m_imag(im) {}
  complex_number() = default;
  double real() const {return m_real;}
  double imag() const {return m_imag;}
...
private:
    double m_real=0,m_imag=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
- Make accessor functions **`const`** when possible

Exercise 2.2:

The program `exercises/complex_number_class.cc` contains a 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.

Constructor/destructor calls

Exercise 2.3:

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 2.4:

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

Making std::cout recognize 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';
```


Class invariants

- A class is supposed to represent a concept: a complex number, a date, a dynamic array.
- It will often contain data members of other types, with assumed constraints on those values:
  - A dynamic array is supposed to have a pointer that is either `nullptr` or a valid block of allocated memory, with the correct size also stored in the structure.
  - A Date structure could have 3 integers for day, month and year, but they can not be, for example, 0,-1,1
- Using **`private`** data members and well designed **`public`** interfaces, we can ensure that assumptions behind a concept are always true.

Class invariants

```
class darray {
private:
  double * dataptr=nullptr;
  size_t sz=0;
public:
// initialize with N elements
  darray(size_t N);
  ~darray();
// resize to N elements
  void resize(size_t N);
// other members who don't change
// dataptr or sz
};
```

- Construct ensuring class Invariants
- Maintain Invariants in every member
- $\rightarrow$ a structure which always has sensible values

C++ classes

Exercise 2.5:

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 member functions do not have an implicit **this** pointer argument. They can be invoked as `ClassName::function()`.

- 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

Example 2.1:

A singleton class is a type designed in such a way that there is at most one object of that type. Every attempt to get an object of that type results in a pointer or reference to the same object. One way to implement this, is by making all constructors of the class private, and providing a **static** `Singleton * getInstance()` member function. This function, being a member of the class, can access the private constructors. It can create a **static** object and return its address. From outside, the only way to get an object of this type is then to call `getInstance()`, which will return a pointer to the same static object. The singleton programming pattern is demonstrated in `examples/singleton.cc`.

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 ...

Templates

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);
```

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.

JÜLICH FORSCHUNGSZENTRUM

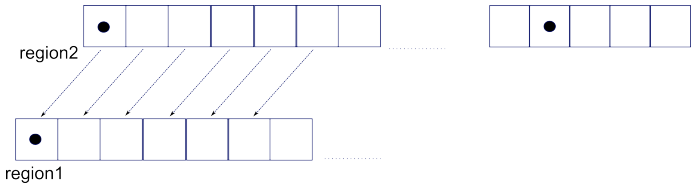
Introduction to C++ templates

Example 2.2:

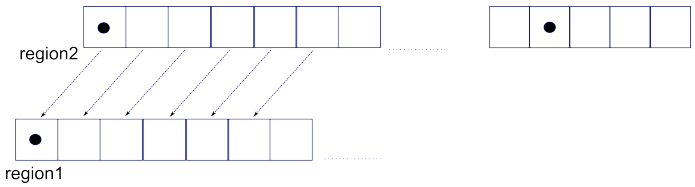
The example code `examples/template_intro.cc` contains the above templated copy function. Compile and check that it works. Read through the code and understand it. Although we seemingly call a function we wrote only once, first with an array of doubles and then with an array of strings, the compiler sees these as calls to two different functions. No runtime decision needs to be made according to the incomming types in this example.

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;
}
```



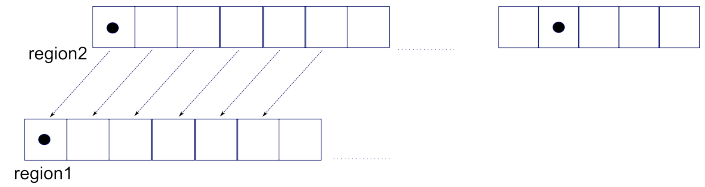
memcpy



Generic code

- The use of **`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 corresponding element to the output iterator position
- 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 had iterators to two kinds of sequences, you could substitute them in the above template and have a nice copy function!
- Better still, you don't even need to do it yourself! The compiler will do it for you.

Test generic copy function

Exercise 2.6:

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.

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
- Template substitution happens at compile time. Type information is retained and used to generate efficient code.

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 templates 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.

Inheritance and 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

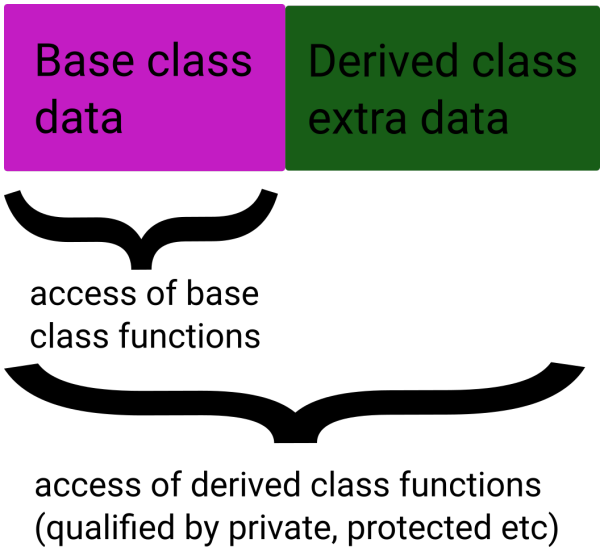
  in our system.
- A loop over a darray of them will be nice. But `darray< ??? >`
- Object oriented languages like C++, Java, Python ... have a concept of "inheritance" for the classes, to describe such conceptual relations between different types.

Inheritance: basic syntax

```
class SomeBase {
public:
    double f();
protected:
    int i;
private:
    int j;
};
class Derived : public SomeBase {
    void haha() {
    // can access f() and i
    // can not access j
    }
};
void elsewhere()
{
    SomeBase a;
    Derived b;
    // Can call a.f(),
    // but e.g., a.i=0; is not allowed
}
```

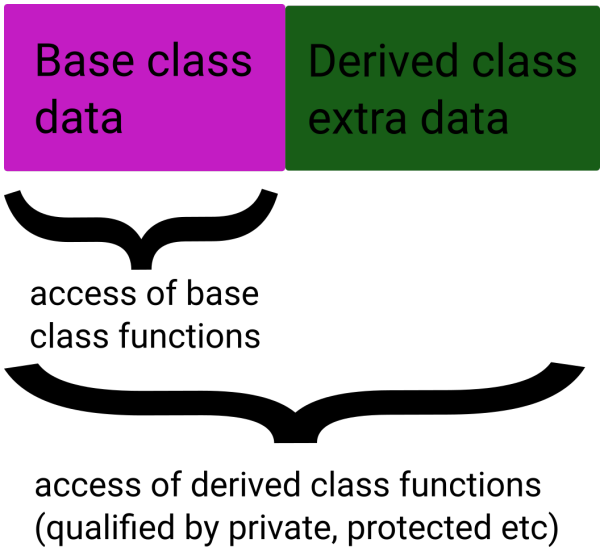
- Class members can be **`private`**, **`protected`** or **`public`**
- **`public`** members are accessible from everywhere
- **`private`** members are for internal use in one class
- **`protected`** members can be seen by derived classes

Inheritance



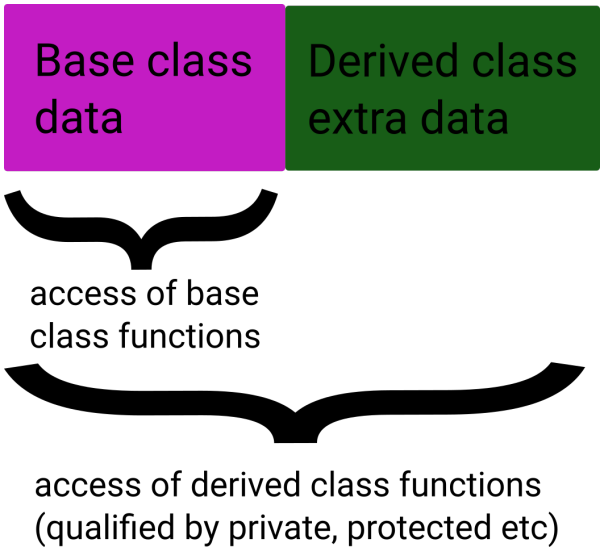
- Inheriting class may add more data, but it retains all the data of the base
- The base class functions, if invoked, will see a base class object
- The derived class object *is a* base class object, but with additional properties

Inheritance



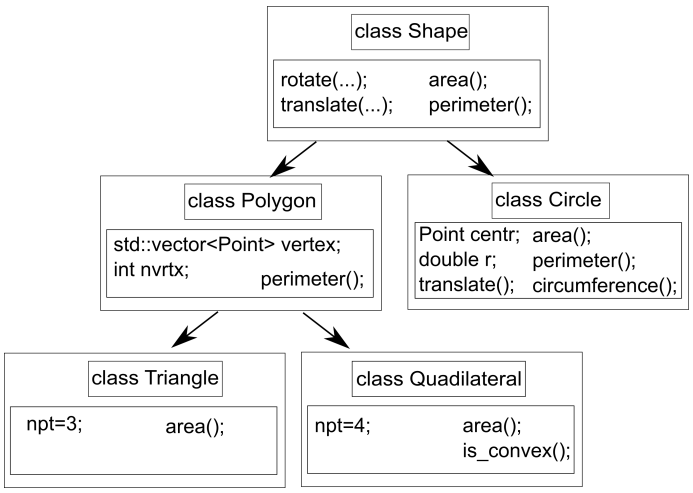
- A pointer to a derived class always points to an address which also contains a valid base class object.
- `baseptr=derivedptr` is called "upcasting". Always allowed.
- Implicit downcasting is not allowed. Explicit downcasting is possible with **`static_cast`** and **`dynamic_cast`**

Inheritance




```
class Base {
public:
  void f(){std::cout<<"Base::f()\n";}
protected:
  int i{4};
};
class Derived : public Base {
  int k{0};
public:
  void g(){std::cout<<"Derived::g()\n";}
};
int main()
{
  Derived b;
  Base *ptr=&b;
  ptr->g(); // Error!
  static_cast<Derived *>(ptr)->g(); //OK
}
```

Inheritance with virtual functions



- 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 with virtual functions

```
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;
};
```

- 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`** in the base class *can be re-implemented* in a derived class.

Note: In C++, member functions are not virtual by default.

Class inheritance with virtual functions

```
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.
- **`virtual`** re-implemented in a derived class are said to be "overriden", and ought to be marked with **`override`**
- 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 with virtual functions

```
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 with virtual functions

```
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))
  }
};
```

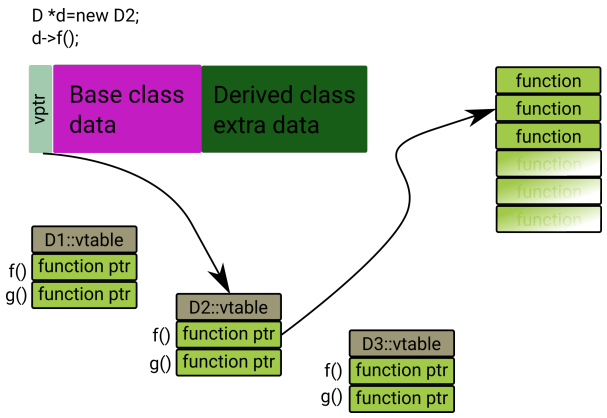
- The keyword **`override`** ensures that the compiler checks there is a corresponding base class function to override.
- Virtual functions can be re-implemented without this keyword, but an accidental omission of a **`const`** or an `&` can lead to really obscure runtime errors.

Class inheritance with virtual functions

```
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()`

Calling virtual functions: how it works



- For classes with virtual functions, the compiler inserts an invisible pointer member to the data and additional book keeping code
- There is a table of virtual functions for each derived class, with entries pointing to function code somewhere
- The `vptr` pointer points to the *vtable* of that particular class

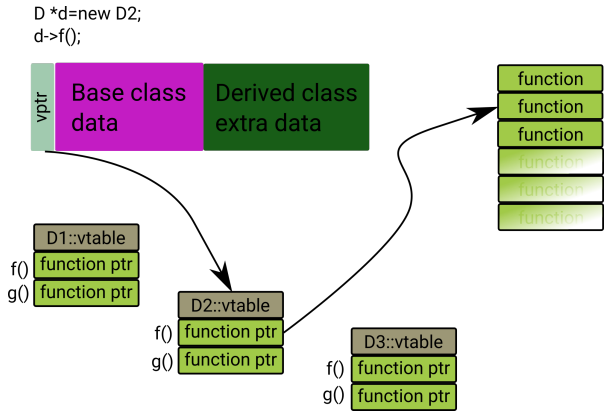
vptr

Example 2.3:

The program `examples/vptr.cc` gives you a tiny class with two **`double`** data members. The `main` function simply creates an object of this kind and prints its size. It prints 16 (bytes) as expected. Uncomment a line containing a virtual destructor function for this class, and recompile and re-run. It will now print 24 as the size on a typical 64 bit machine. Compiling with `g++ -O0 -g3 --no-`**`inline`** and running it in the debugger, you can see the layout of the data structure in memory, and verify that the extra data member is indeed a pointer to a *vtable* for the class.

Calling virtual functions: how it works

- Virtual function call proceeds by first finding the right *vtable*, then the correct entry for the called function, dereferencing that function pointer and then executing the correct function body
- Don't make everything virtual! The overhead, with modern machines and compilers, is not huge. But abusing this feature will hurt performance



- But if virtual functions offer the cleanest solution with acceptable performance, don't invent weird things to avoid them!

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

More control over classes

Class design improvements

- Possible to initialise data in class declaration
- Initialiser list constructors
- Delegating constructors allowed
- Inheriting 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 2.9:

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++, 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

Curiously Recurring Template Pattern

- You need types A and B which have some properties in common, which can be calculated using similar data
- There are a few polymorphic functions, but conceptually A and B are so different that you don't expect to store them in a single pointer container
- The penalty of using virtual functions seems to matter
- Option 1: implement as totally different classes, just copy and paste the common functions
- Option 2: try the CRTP

Curiously Recurring Template Pattern

```
template <class D> struct Named {
  inline string get_name() const
  {
    return static_cast<D const *>(this)
                ->get_name_impl();
  }
  inline int version() const
  {
    return 42;
  }
};
struct Acetyl : public Named<Acetyl> {
  inline string get_name_impl() const
  {
    return "Acetyl";
  }
};
struct Car : public Named<Car> {
  inline string get_name_impl() const
  {
    return get_brand()+get_model()+
        get_year();
  }
};
```

```
int main()
{
  Acetyl a;
  Car 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 2.4: 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 Car. `get_name()` is inherited, but behaves differently for the two types. All this is without using virtual functions.

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;
}
```


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 read-only 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

Example 2.5:

The file `examples/darray_complete.cc` has a completed and extensively commented dynamic array class. 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`.

Chapter 3

Lambda functions

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

Example 3.1:

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
`[a=move(b)]`	Move capture
`[&,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

Example 3.2:

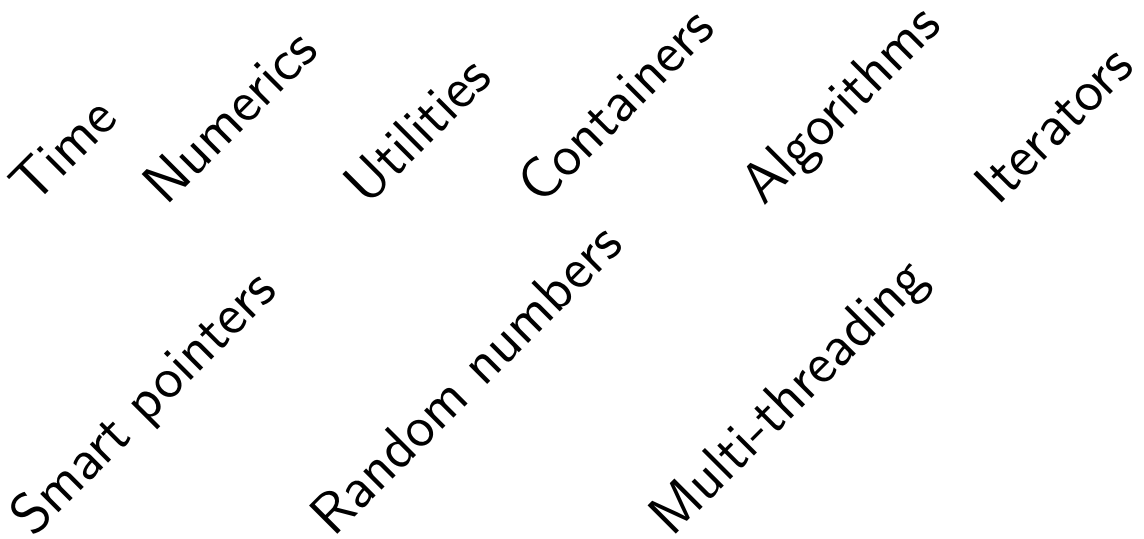
The program `captures.cc` demonstrates capturing variables in the surrounding scope of a lambda function. Compile and run. Observe the differences between capturing by value with and without **`mutable`**, and capturing by reference.

Exercise 3.1:

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.

Chapter 4

Standard Template Library



The time library

- **namespace** `std::chrono` defines many time related functions and classes (include file: `chrono`)
- `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

Example 4.1: examples/chrono_demo.cc

The time library

```
// examples/chrono_demo.cc
#include <iostream>
#include <chrono>
#include <vector>
#include <thread>
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';
  auto d = steady_clock::now() -t;
  std::cout<<"Prime search took "<<duration<double>(d).count()<<" seconds"<<'\n';
  std::cout << "Sleeping for 3 seconds ...\n";
  std::this_thread::sleep_for(3s);
  // h, min, s, ms, us and ns are known time units.
  std::cout << "finished.\n";
}
```

Unique pointer

```
// examples/uniqueptr.cc
int main()
{
  auto u1=std::make_unique<MyStruct>(1);
  //auto u2 = u1; //won't compile
  auto u3 = std::move(u1);
  std::cout << "Data value for u3 is u3->vl = " << u3->vl <<'\n';
  auto u4 = std::make_unique<MyStruct[]>(4);
}
```

- Exclusive access to resource
- The data pointed to is freed when the pointer expires
- Can not be copied (deleted copy constructor and assignment operator)
- Data ownership can be transferred with `std::move`
- Can create single instances as well as arrays through `make_unique`

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 4.2: 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.

std::bind

```
//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';
  auto dda = std::bind(add, _2, _1);
  // new functional, passing arguments to add in the reverse order
  dda(1000000, 99999999);
}
```

Example 4.3: binddemo.cc

Random number generation

- Old `rand()` , `erand()`, `drand()` etc have poor properties with regard to correlations
- Random number classes in the standard library replace and vastly supercede these functions

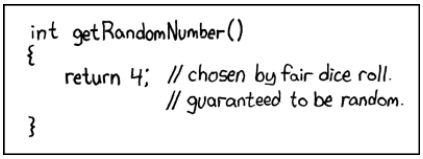


Figure: Source XKCD: `http://xkcd.com`

Random number generation

- Share a common structure
- Uniform random generator engine with (hopefully) well tested properties
- Distribution generator which adapts its input to a required distribution

$$p(n)=\frac{m^n e^{-m}}{n!}$$

Random distribution

Randomness engine

std::bind

```
std::mt19937_64 engine;
std::poisson_distribution<> dist{8.5};
auto gen = std::bind(dist, engine);
// Better still ...
// auto gen = [&dist, &engine]{
//     return dist(engine);
// };
r = gen();
```


Random number generators

```
#include <random>
#include <functional>
#include <iostream>
#include <map>
int main()
{
    std::poisson_distribution<> distribution{8.5};
    std::mt19937_64 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_64` is a 64 bit 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 4.1:

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

Exercise 4.2:

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<>`

Exercises

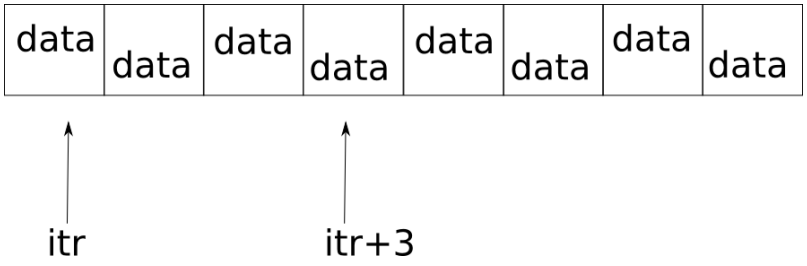
Exercise 4.3:

For a real valued random variable X with normal distribution of a given mean μ and standard deviation σ, calculate the following quantity:

$$K[X] = \frac{\langle (X-\mu)^4 \rangle}{(\langle (X-\mu)^2 \rangle)^2}$$

Fill in the random number generation parts of the program `exercises/K.cc`. Run the program a few times varying the mean and standard deviation. What do you observe about the quantity in the equation above ?

vector: templated dynamic array type

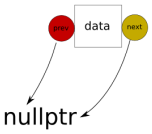


A vector

- Element type is a template parameter
- Consecutive elements in memory
- Can be accessed using an "iterator"

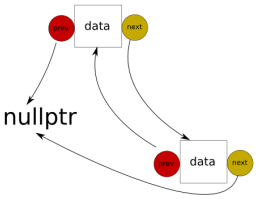
A linked list

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`



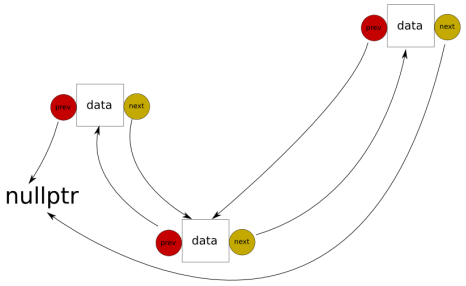
A linked list

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.



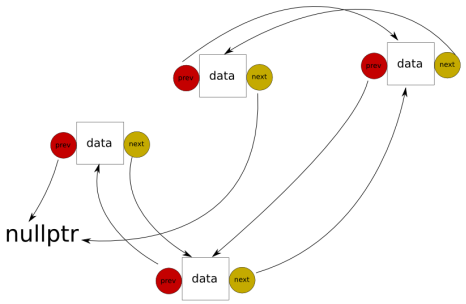
A linked list

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



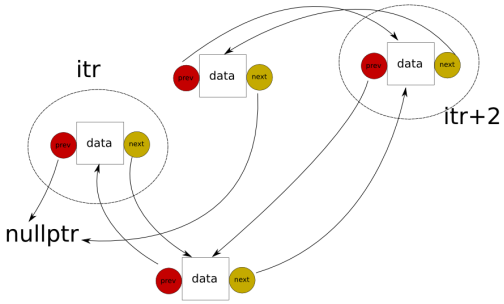
A linked list

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.



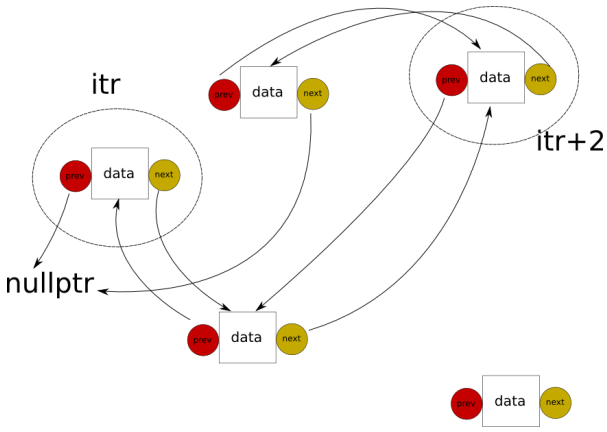
A linked list

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



A linked list

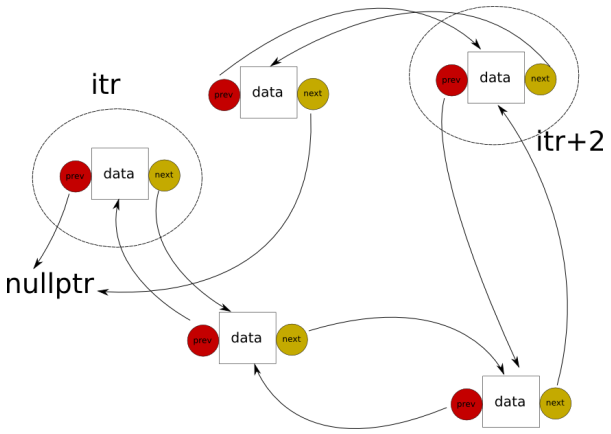
Inserting a new element in the middle of the list does not require moving the existing nodes in memory.



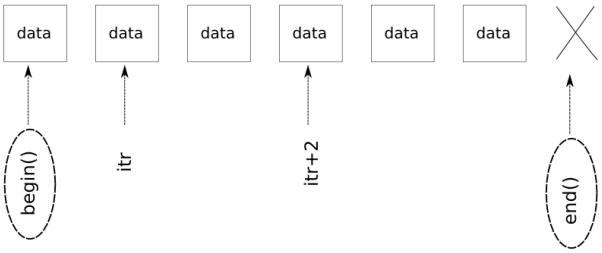
29 May – 01 June 2017 Sandipan Mohanty | Jülich Supercomputing Centre 207 | 330

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 corresponding iterator classes
- If `iter` is an iterator, `*iter` is data.
- All of them provide `begin()` and `end()` functions

std::list

Example 4.4: std::list demo

We need a program to manage a list of names and ages. 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. A small program to do this is `examples/list0.cc`

```
struct userinfo {int age; string 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([](auto a, auto b){
    return a.age < b.age or
      a.age == b.age and a.name < b.name;
  });
  for (auto & user : ulist)
    std::cout<<user.name<<" : "
        <<user.age<<'\n';
}
```

Linked lists

```
struct userinfo {int age; string 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([](auto a, auto b){
    return a.age < b.age or
      a.age == b.age and a.name < b.name;
  });
  for (auto & user : ulist)
    std::cout<<user.name<<" : "
        <<user.age<<'\n';
}
```

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
- `std::list<userinfo>` is a linked list of our hand made structure `userinfo` !
- Notice how we used a lambda function to specify sorting order

std::list

```
#include <list>
...
  std::list<userinfo> ulist;
  userinfo tmp;
...
  ulist.push_back(tmp);
...
  ulist.sort([](auto a, auto b){
    return a.age < b.age or
      a.age == b.age and a.name < b.
          name;
  });
...
  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
- `ulist.sort(comp)` can use a user supplied binary predicate (`comp`) while comparing elements

std::list

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

Iterator

- `iterator` is something that iterates over a sequence and gives access to an element of the sequence like a pointer
- In this example, it is the `current` pointer for our linked list, with additional code to represent the concept of an iterator

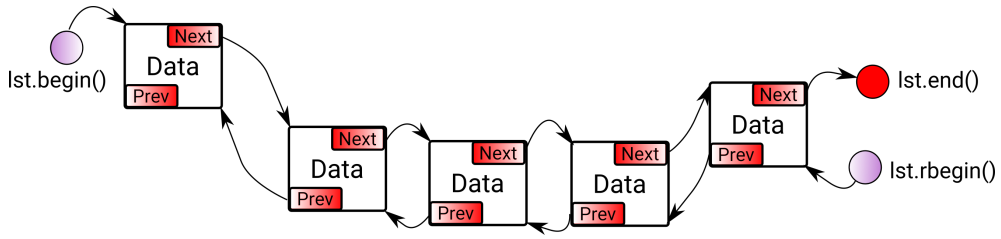
std::list

```
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 (one past the) end of the sequence

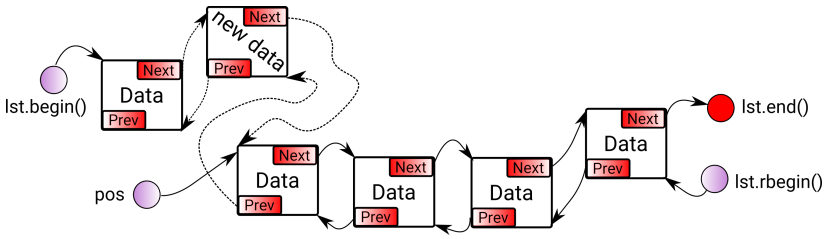
std::list



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



- `ulist.insert(pos,newdata)` inserts the value `newdata` before `pos`
- `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()` and `ulist.pop_front()` erase the terminal elements

Familiarise yourself with std::list

Exercise 4.4:

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.
Consult the online documentation for the syntax and usage of the STL containers. Look up `std::list` at

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

or

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

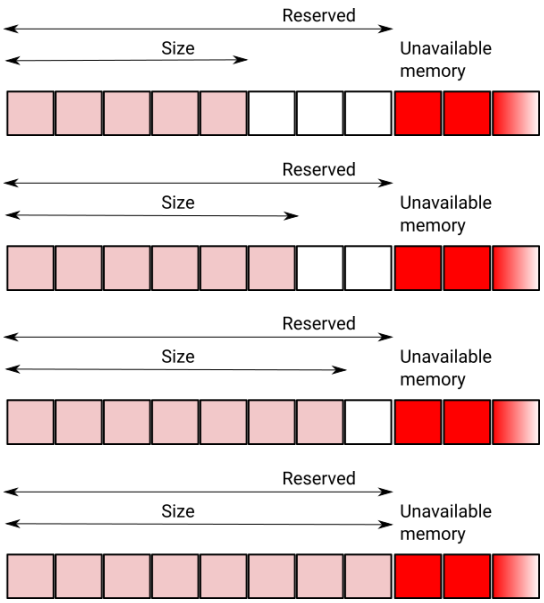
Other STL containers

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

Using std::vector

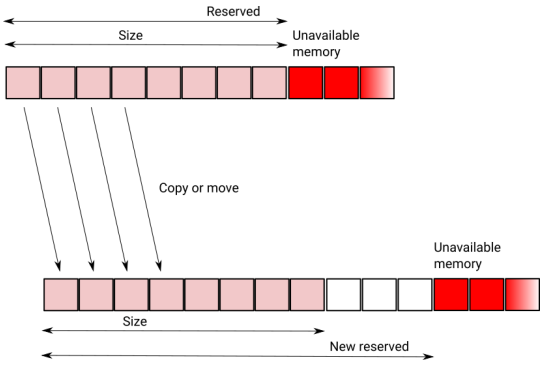
- `std::vector<int> v(10);` : array of 10 integers
- Efficient indexing operator `[]`, for unchecked element access
- `v.at(i)` provides range checked access. An exception is thrown if `at(i)` is called with an out-of-range `i`
- `std::vector<std::list<userinfo>> vu(10);` array of 10 linked lists!
- Supports `push_back` and insert operations, but sometimes has to relocate the all the elements because of one `push_back` operation (next slide)

std::vector



- `std::vector` may reserve a few extra memory blocks to allow a few quick `push_back` operations.
- New items are simply placed in the previously reserved but unused memory and the size member adjusted.
- With repeated `push_back` calls, `std::vector` will run out of its pre-allocated capacity

std::vector



- When `push_back` is no longer possible, a new larger memory block is reserved, and all previous content is moved or copied to it.
- A few more quick `push_back` operations are again possible.

Exercise 4.5:

Construct a list and a vector of 3 elements of the `Verbose` class from your earlier exercise. Add new elements one by one and pause to examine the output.

`std::vector<T>`

- Fast element access, since the elements are stored in contiguous memory locations
- Insertion and deletion can involve copy/move operations of a subset of elements
- For each element, the minimal amount of memory consistent with the element's alignment requirement is stored

`std::list<T>`

- To reach the i'th element from the first element, one must access the second element, and then the next one, and the next one, and so on until the i'th element is found
- Insertion and deletion are very fast, as none of the existing elements need to be copied or moved item For each element, two additional pointers are stored, increasing the memory foot print if we need billions of elements of tiny size

std::array : arrays with fixed compile time size

- `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, integer value>` 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

Associative containers: 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

A word counter program

Example 4.5:

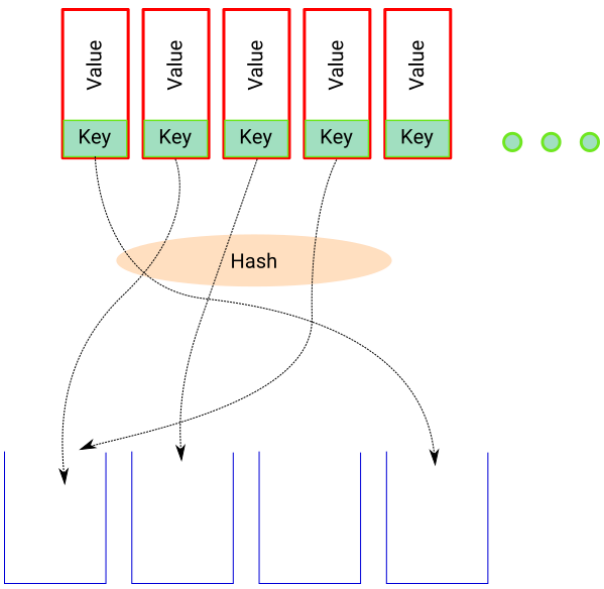
Fake exercise: Write a program that counts all different words in a text file and prints the statistics.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <map>
int main(int argc, char *argv[])
{
  std::ifstream fin(argv[1]);
  std::map<std::string,unsigned> freq;
  std::string s;
  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 container which stores an integer, for each unique `std::string` key.
- The iterator for `std::map` "points to" a `pair<key,value>`

std::unordered_map and std::unordered_set



Unordered map

- Like `std::map<k,v>` and `std::set<v>`, but do not sort the elements
- Internally, these are hash tables, 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
...
std::transform(inp.begin(), inp.end(),
               out.begin(), rotate);
//apply rotate() to each input element,
//and store results in output sequence
```


STL algorithms

Exercise 4.6:

- The standard library 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;
}
```


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;
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(),
       [](int i, int j){
    return i*i < j*j;
  });
  //Sort using custom comparison
  for (auto el: v) cout << el << endl;
}
```

- Lambda functions obviate the need for many adhoc out-of-context classes.

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 be previously 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))`

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`

Exercise 4.7:

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.

Exercise 4.8:

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.

Example 4.6: Probabilities with playing cards

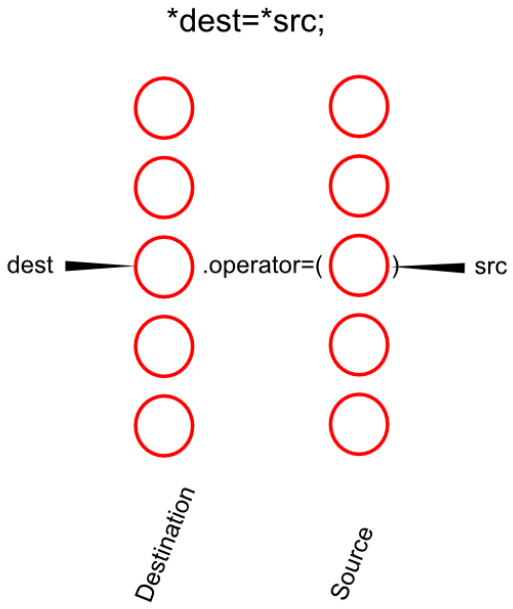
The program `examples/cards_problem.cc` demonstrates many topics discussed during this course. It has a **`constexpr`** funtion to create a fixed length array to store results, several standard library containers and algorithms as well as the use of the random number machinery for a Monte Carlo simulation. It has extensive comments explaining the use of various features. Read the code and identify the different techniques used, and run it to solve a probability question regarding playing cards.

Iterator adaptors

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

The copy algorithm

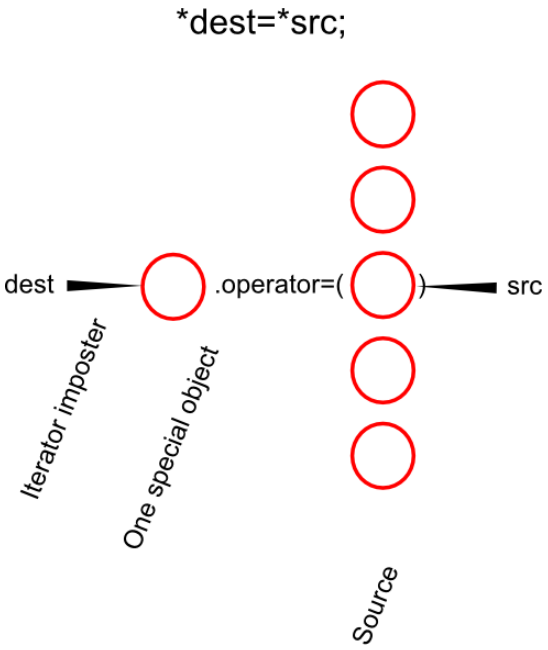
- Move iterators along source and destination sequences
- Get objects at the iterators
- Call **`operator`**`=()` on LHS with RHS as argument



Iterator adaptors

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

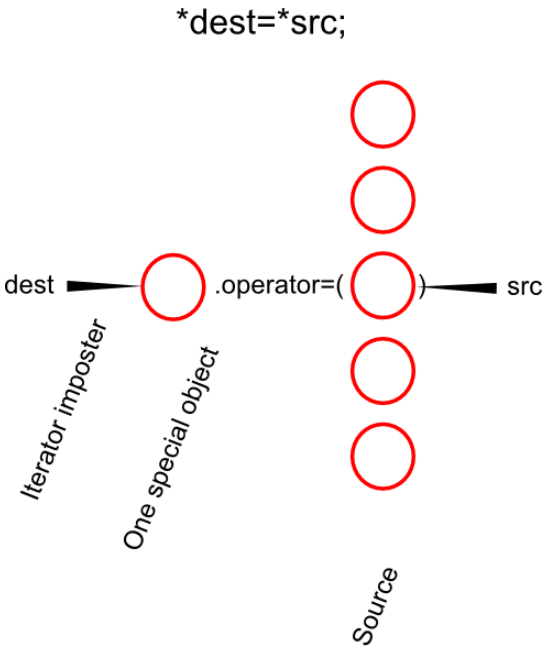
- What if, the destination "iterator" does not really browse any sequence, but simply returns one special object with unary **operator***() ?
- ... and that special object has **operator**=() defined with RHS as argument ?



Iterator adaptors

```
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!



... 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 4.9:

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.

Example 4.7: Reading in a file as a list of strings

The example `examples/line_read.cc` demonstrates how iterator adaptors can be used to read a file as a list of strings (one for each line), using `std::copy` and an appropriately designed iterator imposter and a line proxy class.

Chapter 5

Metaprogramming in C++

Metaprogramming in C++14

Template specialisation

```
template <class T>
class vector {
  // implementation of a general
  // vector for any type T
};
template <>
class vector<bool> {
  // Store the true false values
  // in a compressed way, i.e.,
  // 32 of them in a single int
};
void somewhere_else()
{
  vector<bool> A;
  // Uses the special implementation
}
```

- Templates are defined to work with generic template parameters
- But special values of those parameters, which should be treated differently, can be specified using "template specialisations" as shown
- If a matching specialisation is found, it is preferred over the general template

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++

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: static_assert2.cc

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

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 5.2: 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.

Fold expressions in C++17

```
#include <iostream>
template <typename ... Args>
auto addup(Args ... args)
{
  return (1 + ... + args);
}
int main()
{
  std::cout << addup(1,2,3,4,5)
            << "\n";
}
```

Example 5.3: examples/foldex.cc

- `... op ppack` translates to reduce from the left with operator `op`
- `ppack op ...` means, reduce from the right with `op`
- `init op ... op ppack` reduces from the left, with initial value `init`
- `pack op ... op init` reduces from the right ...

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

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
- Wrapper for a clean overall look

Simplification using constexpr if

```
// examples/print_tuple_cxx17.cc
template <int idx, int MAX, typename...
     Args>
struct PRINT_TUPLE {
  static void print(std::ostream & strm,
          const std::tuple<Args...> & t)
  {
    if constexpr (idx < MAX) {
      strm << std::get<idx>(t);
      if constexpr ((idx+1)<MAX)
          strm << ", ";
      PRINT_TUPLE<idx+1,MAX,Args...>
        ::print(strm,t);
    }
  }
};
```

- C++17 introduces a **`if constexpr`** statement that can be used for compile time conditionals.
- Using **`if constexpr`** as shown, we can shorten the code a bit, by avoiding the final template specialisation case

Example 5.4:

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

Example 5.5:

Printing a tuple is demonstrated in `print_tuple.cc`, `print_tuple_cxx17.cc` and `print_tuple_foldex.cc`. The last two will need compiler flags to enable C++17.

VoT to ToV

Exercise 5.1:

Write a function to convert a vector of tuples into a tuple of vectors. The following main program should print out a vector of doubles extracted from the vector of tuples.

```
int main()
{
    std::vector<std::tuple<int,double,int>> Vt;
    Vt.emplace_back<>(std::make_tuple(1,4.3,0));
    Vt.emplace_back<>(std::make_tuple(3,8.3,3));
    Vt.emplace_back<>(std::make_tuple(2,0.3,2));
    auto Tv=vot_tov(Vt); //supply the function vot_tov()
    for (auto vl : std::get<1>(Tv)) std::cout << vl << "\n";
}
```

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 (C++17: will be) able to write :

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

- 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 5.6: 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.

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

Exercise 5.2:

The program `exercises/cexprprime.cc` demonstrates compile time verification of a number being a prime number, using the old C++11 style constexpr functions. Rewrite for C++14 so that larger values, like the one in the commented out line, can be handled without increasing recursion limits.

Chapter 6

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 → The Qt Company
- `https://www.qt.io`
  - Downloads, tutorials, documentation
  - Link to other courses
- Books: `https://wiki.qt.io/Books`

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

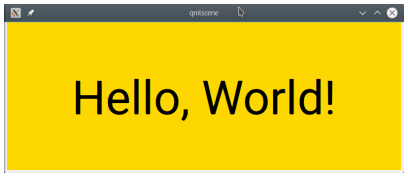
- **Qt Widgets:** For more traditional classic desktop GUI (not discussed here)
- **Qt Quick:** Declarative language for fluid and animated user interfaces.

Environment set-up for Qt5

- `module load qt` will set the environment variables you need
- We will be using `CMake` to build all our Qt5 based programs. A generic `CMakeLists.txt` file suitable for starting new projects can be found under `$COURSE_HOME/share/Qt5`

Hello, World! Qt Quick style

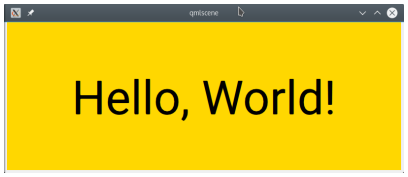
```
// examples/hello.qml
import QtQuick 2.5
Rectangle {
  width: 800
  height: 300
  color: "gold"
  Text {
    anchors.centerIn: parent
    text: "Hello, World!"
    font.pointSize : 48
    font.family: "Lucida"
  }
}
```



- If you just need to show some simple visual element on the screen, the size of the C++ program you need to write is 0 characters.
- Nothing to compile.
- View the content with `qmlscene hello.qml`

Hello, World! Qt Quick style

```
// examples/hello.qml
import QtQuick 2.5
Rectangle {
  width: 800
  height: 300
  color: "gold"
  Text {
    anchors.centerIn: parent
    text: "Hello, World!"
    font.pointSize : 48
    font.family: "Lucida"
  }
}
```



- The QML file is "data" used by the QML engine to generate the visual components
- It is possible to write interface side logic in a convenient Javascript syntax
- In a typical Qt Quick C++ application, you pass on the main qml file to a Qt C++ class, which renders it on the screen

QML

```
// examples/hello.qml
import QtQuick 2.5
Rectangle {
  width: 800
  height: 300
  color: "gold"
  Text {
    anchors.centerIn: parent
    text: "Hello, World!"
    font.pointSize : 48
    font.family: "Lucida"
```

- JSON style format to say: "I want a rectangle with width W and height H ... "

- The fundamental elements in QML, using which you construct your interface, are called `Item`s. `Rectangle` and `Text` in this example are `Item`s.
- An `Item` can contain other `Item`s
- Items can be composed to create new `Item`types in QML. But the fundamental ones are implemented in C++ inheriting from `QQuickItem`. (later!)

QML

```
// examples/hello.qml
import QtQuick 2.5
Rectangle {
  width: 800
  height: 300
  color: "gold"
  Text {
    anchors.centerIn: parent
    text: "Hello, World!"
    font.pointSize : 48
    font.family: "Lucida"
```

- JSON style format to say: "I want a rectangle with width W and height H ... "

- An `Item` can have "properties". A property can be assigned a value, or "bound" to another property of possibly another `Item`, so that if the RHS property changes, the change will be reflected in the LHS property as well.

QML

```
// examples/hello.qml
import QtQuick 2.5
Rectangle {
  width: 800
  height: 300
  color: "gold"
  Text {
    anchors.centerIn: parent
    text: "Hello, World!"
    font.pointSize : 48
    font.family: "Lucida"
  }
}
```

Positioning using anchors

- Try to resize the "Hello, World!" window you created, by dragging the edge or cornor of the window. What happens to the text ? Does it deform ? Does it move closer to or farther from the top ?

- An `Item` has anchors, which can be bound to anchors in another `Item`
- Some other possibilities :
  `anchors.left : parent.left`,
  `anchors.verticalCenter : other.verticalCenter`,
  `anchors.fill: parent` ...

Basic GUI development with Qt Quick

- Use a simple and intuitive markup language to create visual components and check that they look right
- Put them together in a suitable GUI shell for the application
- Write the business end of the application with as little coupling with Graphical Interfaces as possible
- Write a top level controller class which will receive user input from the GUI and take appropriate action
- Implement any new visual element necessary in C++, by inheriting from `QQuickItem`
- In the `main.cc`
  - register any C++ types you wrote which shall be instantiated in the QML files
  - create the UI represented by your QML, and enter event loop

QML Hello, World! through a C++ main function

Example 6.1:

- The folder `examples/qmlhello` contains the `hello.qml` file along with a `main.cc` file, and a generic `CMakeLists.txt` file for Qt5 projects. Build and run :

```
cd /path/to/qmlhello
mkdir build; cd build
cmake ..
make
cd ..
build/qmlhello
```

- What happens if your current working directory does not have the `hello.qml` file ? What happens if you are in a directory containing a different file which happens to be named `hello.qml` ?

The C++ main() for a program with a QML GUI

```
// examples/qmlhello/main.cc
#include <QGuiApplication>
#include <QtQuick/QQuickView>
int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQuickView gui;
    gui.setSource(QUrl("hello.qml"));
    gui.show();
    return app.exec();
}
```

- `QGuiApplication` manages global resources
- Create one in main and 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 functionality and graphical elements

Qt resource system

```
// examples/qmlhello/main.cc
#include <QGuiApplication>
#include <QtQuick/QQuickView>
int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    QQuickView gui;

    gui.setSource(QUrl("hello.qml"));
gui.setSource(QUrl("qrc:/hello.qml"));
    gui.show();
    return app.exec();
}
```

```
// examples/qmlhello/ui.qrc
// You create this!!
<RCC>
    <qresource prefix="/">
        <file>hello.qml</file>
    </qresource>
</RCC>
```

Example 6.2:

- In your `main.cc` file in `examples/qmlhello`, replace the red line with the blue line in the top panel
- Create a file with the content as shown in the bottom panel, with a suffix `.qrc`
- Build as before, and run in different working directories. What change do you notice ?

Qt resource system

- Resource files have the extension `.qrc`, and specify what auxiliary files will be required at run time, e.g., icons, images, GUI description QML files ...
- Resources identified in the `.qrc` files are incorporated into the binary
- Paths in the resource file are relative to that file
- The prefix `qrc:/` in a `QUrl` object tells Qt to look for the path as given in the resource file.

Rectangle

```
// examples/demos/rectangle.qml
Rectangle {
    id: box
    width: 600
    height: 400
    rotation: 45
    radius: 30 // Rounded corners!
    border.color: "red"
    border.width: 3
    antialiasing: true
    // color: "green"
    // or,
    gradient : Gradient {
      GradientStop {
        position: 0.0
        color : "red"
      }
      GradientStop {
        position: 1.0
        color : "green"
      }
    }
}
```

Rectangle items

- Can be used to create circular items as well, by choosing an appropriate `radius`
- If `gradient` is specified, `color` will be ignored
- `gradient` and `antialiasing` are relatively expensive, and should be used for static items only

MouseArea

```
// examples/demos/mousearea.qml
import QtQuick 2.7
Rectangle {
    width: 600
    height: 400
    Rectangle {
        id: box
        width: 300
        height: 200
        anchors.centerIn: parent
        rotation: 45
        radius: 100 // Rounded corners!
        border.color: "red"
        border.width: 3
        antialiasing: true
        color: "green"
    }
    MouseArea {
        anchors.fill: box
        onClicked: {
          console.log("Click detected!")
        }
    }
}
```

Mouse area

- Fill a named visual object with a mouse area to capture mouse events
- You can then take action based on “signals” emitted when a mouse event happens
- Q: Does the mouse click correspond to the rotated or unrotated rectangle ? Does it follow the curved boundary ?

Text

```
// examples/demos/text.qml
import QtQuick 2.3
Rectangle {
    width: 600
    height: 400
    Text {
        anchors.fill: parent
        text: "<a href=\"http://www.fz-juelich.de\">Forschungszentrum</a>
               <a href=\"http://www.juelich.de\">Juelich</a>"
        color: "green"
        wrapMode: Text.Wrap
        font {
          family: "Times";
          pointSize: 32;
          bold:true
        }
        onLinkActivated: console.log("Activated link " + link)
    }
}
```

- You can choose the color, font, wrapping mode for the text
- Rich-text/HTML style formatting is supported
- Can tell you when an embedded link is activated

Image

```
// examples/demos/image.qml
import QtQuick 2.0

Image {
    source: "start.svg"
    fillMode: Image.PreserveAspectFit
}
```

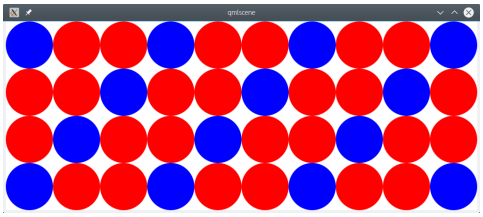
- Bitmap formats like PNG, JPEG as well as vector graphics formats like SVG
- `fillMode` can be used to specify how it should behave when resized
- For large images, set `asynchronous` to true.

Rows, columns and grids

```
// examples/demos/row.qml
import QtQuick 2.7
Row { // or Column, or Grid
    Rectangle {
        width: 100
        height:100
        color: "red"
    }
    Rectangle {
        width: 100
        height:100
        color: "green"
    }
    Rectangle {
        width: 100
        height:100
        color: "blue"
    }
}
```

- Arrange items in rows, columns or grids
- For grids, the property `columns` can be used to specify grid shape
- If the contained items are of different sizes, properties like `horizontalItemAlignment` can be used to set the desired alignment.

Repeater



```
// examples/demos/repeater.qml
import QtQuick 2.7
Grid {
  columns: 10
  Repeater {
    model: 40
    // or,
    // model: ["red","green","blue"]
    Rectangle {
      width: 100
      height:100
      radius:50
      color: index%3?"red":"blue"
      // color: modelData
    }
  }
}
```

- Repeats an item according to a “model”
- If model is an integer, creates as many copies
- If model is a list, repeats according to list size
- Each copy has access to an `index` inside the repeater
- For list models, there is also `modelData` referring to the corresponding list element

Item

- Base type for all visual items in Qt Quick
- Not visible itself, but implements common visual element attributes like width, height, anchoring and key handling support
- Convenient to bundle a few elements into a new QML type

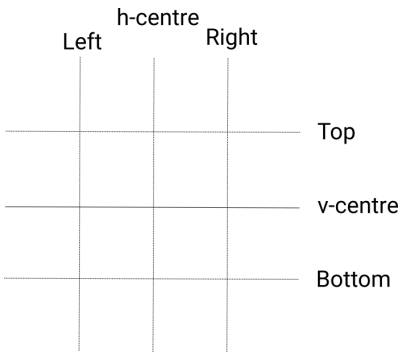
Item

```
// examples/demos/MyToolButton.qml
import QtQuick 2.7
Item {
    id: root
    property alias icon : theimg.source
    property alias label : thetext.text
    width: theimg.implicitWidth + thetext.implicitWidth
    Row {
        Image {
            id: theimg
            height: root.height
            fillMode: Image.PreserveAspectFit
        }
        Text {
            id: thetext
            padding:20
            anchors.verticalCenter: theimg.verticalCenter
        }
    }
}
// examples/demos/toolbuttons.qml
import QtQuick 2.7
Row {
    MyToolButton { height: 50; icon: "start.svg"; label: "Start" }
    MyToolButton { height: 50; icon: "start.svg"; label: "Also, start!" }
}
```


- When we load `toolbuttons.qml` in `qmlscene`, it looks for a definition of the apparent typename `MyToolButton`.
- It finds the file `MyToolButton.qml` in its search path, and finds an item definition, which it accepts as the definition of the type `MyToolButton`

- `MyToolButton.qml` does not specify the source of the image or the actual text.
- We create top level aliases for properties we want to be modifiable from the outside.
- We arrange the items relative to each other
- Now we can reuse this new QML element as an image-text combo whenever we need one

Positioning with anchors




```
Rectangle {
//haha
    Text {
        anchors.top : parent.top
        anchors.right: parent.right
        // ...
    }
}
```

- Each `Item` has 7 invisible anchor lines, called left, horizontalCenter, right, top, verticalCenter, bottom and baseline
- You can use the anchor lines to position the elements relative to each other

Layouts

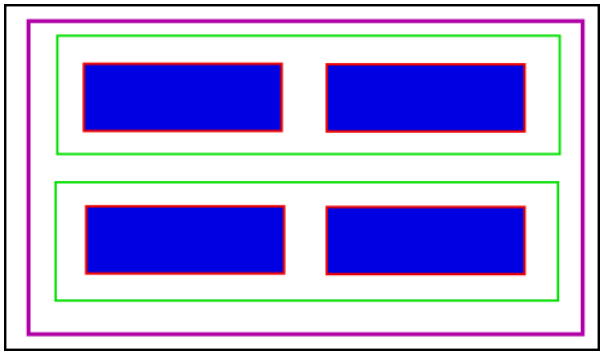
- Invisible entities which take care of arranging their children in a well defined manner
- Layouts are Items. So, a Layout can have other Layouts or Items as children
- `ColumnLayout` arranges its children vertically
- `RowLayout` arranges its children horizontally
- `GridLayout` arranges its children in a grid
- Layouts *can resize* the `Item`s they arrange

Layouts

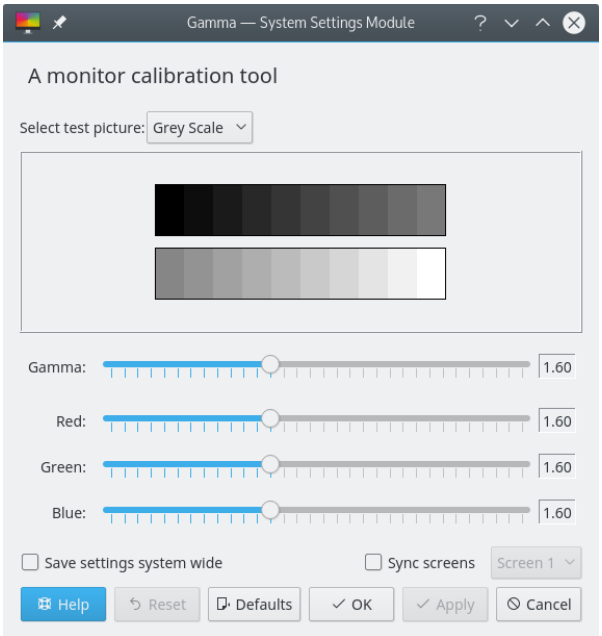
- Since the layouts can resize their contents, they need to know the minimum, maximum, preferred, heights and widths of their contents.
- Each `Item` placed in a Layout gets additional properties, `Layout.minimumWidth`, `Layout.preferredWidth`, `Layout.maximumWidth` etc. To specify hints to the layout
- If `Layout.fillWidth` is set to true for an `Item`, it will stretch to fill horizontal space. If not, it will stay at the preferred width, even if the containing window is enlarged.
- If `Layout.fillHeight` is ...

Layouts

```
// examples/demos/layouts_grid.qml
Rectangle {
  property int margin: 10
  width: g.implicitWidth+ 2*margin
  height: g.implicitHeight+ 2*margin
  GridLayout {
    id: g
    anchors.fill: parent
    columns : 2
    Rectangle {
        color: "blue";
        border.color: "red" ;
        Layout.preferredWidth: 100;
        Layout.preferredHeight: 50
        Layout.fillWidth: true
        Layout.fillHeight: true
    }
    Rectangle {
        color: "blue";
        border.color: "red" ;
        Layout.preferredWidth: 100;
        Layout.preferredHeight: 50
        Layout.fillWidth: true
        Layout.fillHeight: true
    }
}
```



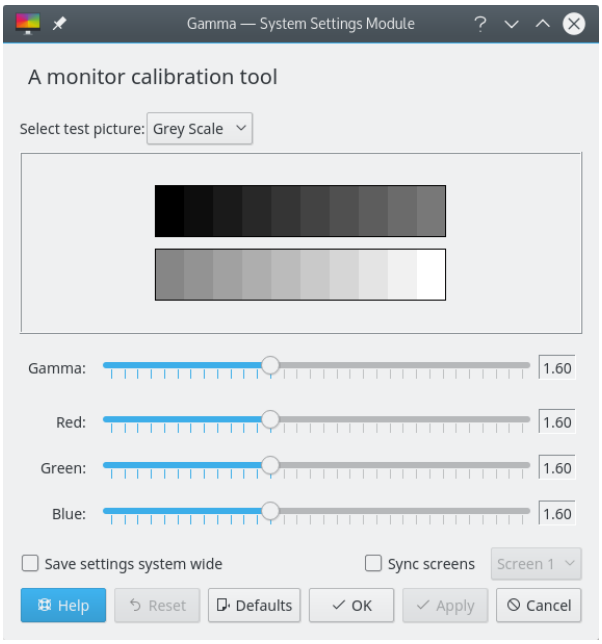
Qt Quick Controls



Frequently needed compounds

- With `Items`, `Text` etc, we can create more complex building blocks for a GUI
- But for frequently used components, it is desirable to have some prepackaged entities available through the library. $\Longrightarrow$ Qt Quick Controls library

Qt Quick Controls



Frequently needed compounds

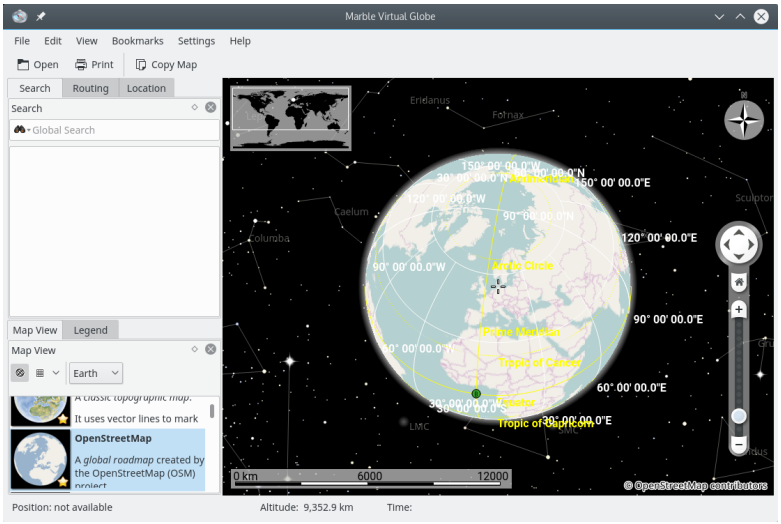
- Provides Buttons, Sliders, Labels, ComboBoxes, CheckBoxes, RadioButtons, SplitView, TabView, ScrollView ...

Qt Quick Controls

Exercise 6.1: Develop using Qt Quick Controls

In the folder `exercises/gamma` you will find the QML tree for an interface similar to the one of the KDE Plasma program called kgamma, to make gamma adjustments of the monitors. It's missing a few buttons. Try to finish it, so that it looks close to the interface you see in the previous slide.

Creating the main window



- Has menu bars, tool bars, status bars etc
- Has a central item for the main functionality
- Can be created in QML using `ApplicationWindow`

Creating the main window

```
ApplicationWindow {
  title: "My great program"
  visible: true
  menuBar: MenuBar {
    Menu {
      title: "&File"
      MenuItem {
        text: "E&xit"
        shortcut: StandardKey.Quit
        onTriggered: Qt.quit()
      }
    }
    Menu {
      title: "&Help"
      MenuItem {
        text: "About..."
        onTriggered: aboutDialog.open()
      }
    }
  }
  toolBar: ToolBar { ... }
  statusBar: StatusBar { RowLayout { ... } }
}
```

- Menu bar, tool bar and status bar sub-elements are already present in the main window, but need to be bound to items of the right type. They know where they should live.
- The top menus can be filled with items as shown. Each menu item when selected, emits a “triggered” signal.

Creating the main window

```
Action {
    id: startAction
    text: "&Run"
    shortcut: StandardKey.Forward
    iconName: "file-run"
    onTriggered: prog.start()
}
Action {
    id: stopAction
    text: "&Stop"
    shortcut: StandardKey.Close
    iconName: "file-stop"
    onTriggered: prog.stop()
}
```

- Actions are named invisible entities which can be “triggered”
- An action can be added to a menu as an item
- Actions have a signal ”triggered“, which could be connected to something

Creating the main window

```
toolBar: ToolBar {
  RowLayout {
    ToolButton {
      iconSource: "icons/start.svg"
      onClicked:startAction.trigger(this)
      tooltip:"Start"
    }
    ToolButton {
      iconSource: "icons/stop.svg"
      onClicked:stopAction.trigger(this)
      tooltip: "Stop"
    }
  statusBar: StatusBar {
    RowLayout {
      Label {
        text: "Ready"
      }
      ProgressBar {
        id: progbar
        value: prog.cycleCount
      }
    }
```

- `toolBar` can to be filled with `ToolButtons`, with a text and an icon
- `statusBar` can similarly be given some content as shown
- Between the tool bar and the status bar, some visual item takes the stage as the main item in the program

Properties

```
Shades { gamma : gammachannel.value }
...

text: theslider.value.toFixed(2)
...
```

- A property of a qml item can be bound to another property of that or another item
- The lhs property then "follows" the rhs property
- Provided through the Qt meta object system

Example 6.3:

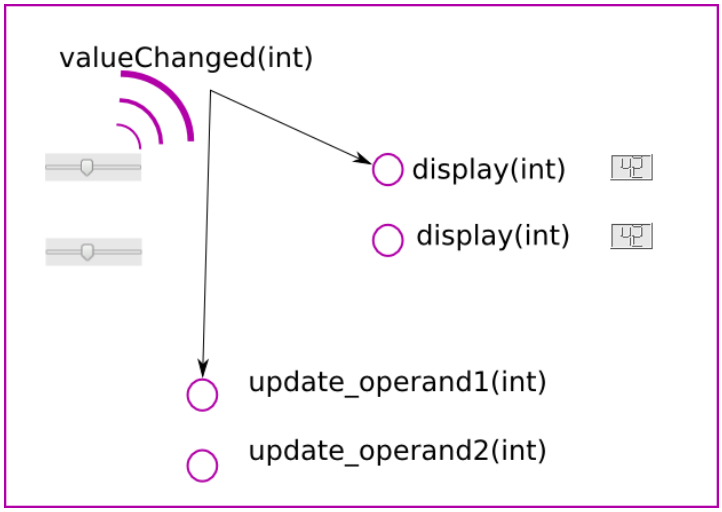
The qml file `examples/demos/binding.qml` shows how the one QML item can be set up to follow changes in another QML item using properties. Observe the behaviour using `qmlscene` and then look at the QML file to see the code that does the work.

Exercise 6.2:

Insert the right property bindings (two total lines) in `exercises/gamma` so that the gray scale bar as well as the right side numeric values for the sliders start to respond to the sliders.

Communication between objects

Qt signals and slots mechanism



When a signal is emitted, all slots it is connected to are called.

Qt signals and slots

- Signals are declared, but not implemented by the user
- 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
- A signal called somethingHappened is exposed to the QML system as onSomethingHappened

Qt signals and slots

```
MouseArea {
  anchors.fill: box
  onClicked: {
    console.log("Click detected!")
  }
}
Action {
  id: quitAction
  text: "&Quit"
  shortcut: StandardKey.Quit
  iconName: "file-exit"
  onTriggered: Qt.quit()
}
```

- The response to the signal in QML can be written as shown here.
- C++ functions can be called here if they have been made available to QML through the `Q_INVOKABLE` macro

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 setting the `Q_OBJECT` macro for the class.

Calling C++ functions

```
    QGuiApplication a(argc, argv);
    DoSomething interf;
    QQuickView gui;
    gui.rootContext()->setContextProperty("interf", &interf);
    gui.setSource(QUrl("qrc:///ui.qml"));
    gui.show();
    return a.exec();
```

- In the main program, you can declare an object of that type
- Create a `QQuickView` as usual and pass the object of your own class as an argument to the `setContextProperty()` function, along with a name which should be used for this object inside QML.
- You can now access member functions of your class which have been marked with `Q_INVOKABLE` directly from QML

Calling C++ functions

```
Rectangle {
    id: rect
    function updateUI() {
        text.text = interf.sum();
    }
    Rectangle {
        id: button1
        MouseArea {
            anchors.fill: parent
            onClicked: {
              button1.pressed = !button1.pressed;
              text.text = interf.greet();
            }
        }
    }
}
```

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

Example 6.4: Calling C++ functions from Qml

The folder `examples/bigredbutton` contains a QML file to create a big round red button. There is a class with some simple functions in it, and we call one of them when the red button is pressed. This program is a minimal demo, where you make something happen by interacting with the GUI. Break it a little. Remove the `Q_INVOKABLE` macros, infront of the functions.

Exercise 6.3: A GUI for one of your own simpler programs

Choose one of the simpler programs you have written in a previous exercise, e.g., the Poisson distribution demonstration, and make it start using a button.

Information exchange : GUI and your C++ classes

Example 6.5: infoexchange

The example program `examples/infoexchange` demonstrates a few major components in any C++ program with a Qt Quick GUI. It modifies the Kurtosis problem in yesterday's exercise to give it a GUI. The sliders in the GUI can now change the mean and standard deviation of the distribution. The start button triggers the calculation. The result of the calculation is then displayed in a big label in the GUI.

Information exchange : GUI and your C++ classes

- We need to create a controller class to mediate between our GUI and the calculations
- This class, `KurtosisProblem`, inherits from `QQuickItem`, and contains the `Q_OBJECT` macro
- The `start()` member function performs the calculation
- Mean and standard deviation of the normal distribution are member variables

```
 class KurtosisProblem : public QQuickItem {
    Q_OBJECT
public:
    Q_INVOKABLE void start();
    Q_PROPERTY(qreal mean MEMBER m_mean NOTIFY meanChanged)
    Q_PROPERTY(qreal sigma READ sigma WRITE setSigma NOTIFY sigmaChanged)
    Q_PROPERTY(qreal result MEMBER m_result NOTIFY resultChanged)
    KurtosisProblem(QQuickItem *parent = 0);
    inline auto sigma() const { return m_sigma; }
    inline void setSigma(qreal sg) {
        std::cout << "Called setSigma with argument " << sg << "\n";
        auto old_sig = m_sigma;
        if (fabs(sg - old_sig)>eps) {
            emit sigmaChanged();
            m_sigma = sg;
        }
    }
signals:
    void meanChanged();
    void sigmaChanged();
    void resultChanged();
private:
    qreal m_mean = 10.0, m_sigma = 35.8, m_result=0;
};
```


Information exchange : GUI and your C++ classes

- To make mean and standard deviation visible in QML as "properties", we make them `Q_PROPERTY` in our C++ class.
- For each property, we decide how it should be accessed from QML
- We make the `start()` function `Q_INVOKABLE` so that we can call it from QML.

Information exchange : GUI and your C++ classes

- The implementation of the class simply performs the same calculation as the `K.cc` program
- But, since `KurtosisProblem` inherits from `QQuickItem`, and takes a QQuickItem parent, we write that constructor.

Information exchange : GUI and your C++ classes

```
int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);
    qmlRegisterType<KurtosisProblem>("KurtosisProblem", 1, 0, "KurtosisProblem");
    QQuickView gui;
    gui.setSource(QUrl("qrc:///main.qml"));
    gui.show();
    return app.exec();
}
```

- In the main program, we register the `KurtosisProblem` class with QML
- Rest of the code remains like in our QML hello world programs.

Information exchange : GUI and your C++ classes

- In the `main.qml`, we instantiate our `KurtosisProblem` Qt Quick Item, and give it a name
- We bind the mean and sigma "properties", to the values of the respective control sliders
- The start button's `onClicked` signal handler is set to call the `start()` function for the instantiated object

```
import KurtosisProblem 1.0
Rectangle {
  KurtosisProblem {
    id : prog
    mean : meanctrl.value
    sigma: sigctrl.value
  }
...
  Button {
    id: startbutton
    anchors.centerIn: parent
    text: "Start"
    onClicked: prog.start()
  }
...
```

- Now let's see if that works!

Custom visual Items I

Introduction to Qt Scene Graph

- `QQuickItem`s in a QML scene populate a tree structure of `QSGNode` objects
- To make a new visible type to use in QML, you need to inherit the `QQuickItem` class, and in the constructor, set the flag `QQuickItem::ItemHasContents`.
- Override the `updatePaintNode` member function where you insert and manipulate any visual items you want to create
- This function is called by the rendering engine when your node needs to be redrawn
- The argument passed is a pointer to a `QSGNode`, meant to represent the geometry and texture of your visual item.

Custom visual Items II

Introduction to Qt Scene Graph

- If the node is empty, you should allocate memory for it, but once you pass it over to the rendering engine, it handles the clean up.
- Each time the function is called for an object, it is called with the same `QSGNode` pointer, so that allocation and initialization does not have to be repeated, just the changes recorded
- When the the rendered content has changed, you have to call `update()`. This schedules a call to `updatePaintNode` if the item is visible
- Example: `LatticeDisplay` in the Ising model example

Square Lattice Ising Model Simulator I

- Toy physics problem well suited as a simple but non-trivial example
- Square lattice with spins, with only nearest neighbour interactions
- Properties easily approximated using a simple Metropolis Monte Carlo simulation
- One can vary the temperature and the strength of the external magnetic field
- The lattice reacts : at temperatures below about 2.27, the very long spin correlation lengths are favoured. At higher temperatures, spins are more random, and the magnetisation is small

Square Lattice Ising Model Simulator I

- The lattice and the Monte Carlo engine can be implemented without any thought about the GUI
- The lattice display has to be a visual Qt Quick Item, and must override `updatePaintNode`
- A top level handler class `MCRunner` handles GUI inputs and passes them on to the MC engine
- Results are passed back to the GUI using the properties system and displayed.
- Implementation : `examples/slimsim`

Square Lattice Ising Model Simulator