An introduction to 'Modern' C++

April 21, 2022

John Bulava
Zeuthen Data Science Seminar (ZDSS)

What is C++?

- Compiled and ‘Close to the metal’: efficient use of memory and CPU resources
- High-level design concepts: Classes, Templates, Lambda-expressions
- Newer C++ standards (C++14, C++17, C++20) support concurrent programming and scientific computing

Photo taken near Ostkreuz S-bahn station.

A bad omen for C++?!?

Advantages of C++:

- Portability: same code on laptops on supercomputers
- Control: strongly checked typing, safer pointers, access control

- Additional Functionality compared to C, Fortran:
  - Object-Oriented Programming (OOP): manipulate 'classes', which are new user defined types.
  - Generic Programming: use 'templates', which enable a single code unit to be used for several types.
  - (almost) no additional performance cost.

Disadvantages of C++:

- Compilation time and executable size will balloon!
- Inner workings of classes are hidden from the end user, easy to do stupid things.
- Additional functionality enables a new level of obfuscated code.

"With great power comes great responsibility"
-Uncle Ben, Spiderman

```cpp
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;

  return 0;
}
```

- cout is an **object** of **class** ‘ostream’, defined in the **standard library**
- cout is declared in the standard library header `iostream.h`, within the `std` **namespace**
- ‘<<’ is an **operator** to send to an ostream object.
- `endl` is an end of line character and a buffer flush.

- Similar to cout, there is cerr, clog.
- ‘<<’ can be **overloaded** to take objects of many different types.

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

int main() {

    MyClass mc;
    cout << "Hello World! " << 12345 << " "
        << 5.6 << " " << mc << endl;

    return 0;
}
```

```cpp
#include<iostream>
class MyClass {
  private:
    int i;
  public:
    MyClass(int j) : i{j} {} //constructor

    void printout() {
      std::cout << "i = " << i << std::endl;
    }
};

int main() {
  //MyClass mc1; //will not compile
  MyClass mc2(4);
  mc2.printout();
  //std::cout << mc2.i; //will not compile
  return 0;
}
```

- Class objects are instantiated by calling the **constructor**. The constructor which takes no arguments is the **default constructor**.
- Since no default constructor is declared, objects of type `MyClass` can't be constructed with no arguments.
- Using the curly brackets to initialize `i` is only possible in C++11 (**initializer list**) and above.
- Members are accessed using the '`.`'.
- Only public members may be accessed! Since `i` is declared as private, it cannot be directly accessed outside the class.
- Unless otherwise specified, class members default to private access.
- Note: C++ also has the `struct` keyword. Only difference between structs and classes is that struct members default to public access.

Preview of general class ideas in C++:

- The interface to classes is tightly controlled! This (hopefully) reduces errors. Always consider the interface, use minimal access.
- Classes can **inherit** from other classes, to gain some of thier members. **Multiple inheritance** is also possible.
- We say that the **derived class** inherits from the **base class**.
- **Polymorphism**: Code can be written to interact with a pointer to a base class object. The pointer could point to a derived class object!

```
class Vector {
  private:
    double* elem;
    int sz;
  public:
  //constructor
  Vector(int s) :elem{new double[s]}, sz{s} {
    for (int i=0; i<s; ++i)
      elem[i]=0;
  }

  //destructor
  ~Vector() { delete[] elem; }

    double& operator[](int i) {
      return elem[i];
    }
    int size() const {return sz;}
};
```

- The `operator[]` and `size` members are declared but not defined.
- Class members are typically declared in header (`.h`, `.hpp`) files and defined in (`.cc`,`.cpp`) files.
- The above code belongs in `vector.h`. Read the header to see the interface.
- The **destructor** defines what must be done to (safely) destroy an object.
- A default destructor is automatically generated which destructs each member variable (data member). This would result in a memory leak for `Vector`!

- `new` is the C++ equivalent of `malloc` while `delete` is the C++ equivalent of `free`.
- Note that deleting an array requires '`[]`'.
- '`[]`' is **overloaded** to behave like a C-style array. `v[5]=6` is equivalent to `v.operator[](5)=6`. More about this later.
- Calling the `size` member does not change the **state** of the object, i.e. the values of the data members. The `const` modifier indicates this explicitly.
- There is NO access to the underlying pointer and `sz` cannot be changed. These are **private** members.
- It is (as far as I can see) impossible to cause a memory leak using `Vector`, as `new` and `delete` are called automatically.

`Vector` is an example of **RAII (Resource Allocation Is Initialization)**:

- Classes which manage resources (memory, files, input/ouput, etc.) are called **containers** or **handles**.
- Resources are allocated during initialization by the constructor and released when the destructor is called, i.e. when the object passes out of scope.
- No 'naked' `new` or `delete` calls, files left open, etc.

Pointers in C++:

```
//pointer to a single double
double *d = new double;
delete d;
//pointer to an array of doubles
double *darr = new double[5];
delete[] darr;
//'new' calls constructor
MyClass *mcp = new MyClass();
mcp->member();
//'delete' calls destructor.
delete mcp;
//default constructor called for each.
MyClass *mca = new MyClass[10];
//Call the destructor for each one.
delete[] mca;
```

References in C++:

```
MyClass mc;
MyClass& mcr = mc;
...
mcr.member();

//and const. refs.
const MyClass& mccr = mc;
...
mccr.const_member();
```

C++ **references** are very safe.

```
double& h1;    //will not compile
const double& h2; //will not compile

double d1=5.6;
const double& r1 = d1; //fine
double& r2 = r1;       //will not compile
const double& r2 = r1; //fine
```

- A reference must be initialized to an existing object.
- A reference can never be moved to another object.
- Thus, a ref. is confined to the scope of the object it references.

- When `new` is used w/ a class, the constructor is called.
- Just like C, every `new` must be accompanied by a `delete`, which calls the destructor.
- Dynamically allocated C-style arrays work the same way with classes.
- References must be initialized when declared.
- Since references do not allocate new resources, they do not have to be ‘deleted’, and just fall out of scope.

More about access control. Consider the following function declarations:

```
//Pass by value, return a value
double func(double arg);

//When used with assignment, a temporary
//is formed and a copy occurs
double a,b;
a = func(b);

//Pass a reference, it can be changed
double changer(double& arg);

//Pass a ref., it cannot be changed
double no_changer(const double& arg);
```

Functions can also take and return pointers:

```
//I hope it doesn't delete my pointer....
double danger1(double*);

//I hope it doesn't grab any resources....
double* danger2(double *);
```

When in doubt (i.e. by default) pass by constant reference (except for small objects). Don't use pointers unless absolutely necessary.

Besides the constructor and destructor, there are other special member functions:

- the **copy constructor**: Constructs a class object from another object.

- the overloaded **assignment operator** $(=)$: Assigns the object to an existing object.

```
public:
   //copy constructor
   Vector( const Vector& vec_in);


   //overloaded assignment operator
   Vector& operator=(const Vector& vec_in);
```

If you don't explicitly define the destructor, copy constructor, and assignment operators, they will be defined for you.

```
public:
   //default copy constructor
   Vector(const Vector& vec_in): sz(vec_in.sz),
       elem(vec_in.elem) {}

   //default assignment operator
   Vector& operator=(const Vector& vec_in) {
       sz = vec_in.sz;
       elem = vec_in.elem;
    }

   //default destructor
   ~Vector() {}
```

The default copy constructor and assignment operators perform **shallow copies**.

Since `Vector` is a resource handle, we need to implement our own copy constructor and assignment operator, which perform **deep copies**.

```
//deep copy constructor
Vector(const Vector& vec_in): sz(vec_in.sz) {
    if (vec_in.elem != 0) {
      elem = new double[sz];
      for (int i = 0;i<sz;i++)
        elem[i]=vec_in[i];
    } else
        elem = 0;
}
```

```
//deep assignment operator
Vector& operator=(const Vector& vec_in) {
   if (this == &vec_in)
     return *this;

   delete[] elem; //erase existing elements!
   sz = vec_in.sz;

   if (vec_in.elem != 0) {
     elem = new double[sz];
     for (int i = 0;i<sz;i++)
       elem[i]=vec_in[i];
   } else
       elem = 0;

   return *this;
}
```

- Recall that for resource handles, a non-trivial destructor is also required.
- **The Rule of Three**: If a class requires a non-trivial implementation of one of the following, it requires a non-trivial implementation of all of the following:
  - Destructor
  - Copy Constructor
  - Overloaded Assignment Operator
- Resource handles typically require all three.
- If shallow copying will suffice, let the compiler do it!

Copy construction and assignment can be disabled:

```
private:

    Vector(const Vector& vec_in);

    Vector& operator=(const Vector& vec_in);
```

Now the following won't compile:

```
Vector v1(5);  //OK
Vector v2(v1);  //ERROR!

Vector v2(3);  //OK
v2=v1;          //ERROR!
```

What about the default constructor?

- The compiler will automatically generate a constructor which default constructs all members.
- But that won't grab any resources, or initialize them!
- The default constructor can be similarly disabled:

```
private:
    Vector();
```

So that the following won't compile:

```
//ERROR if default constructor
//is disabled.
Vector v1;
```

- 'Rule of Four' with disabling of default constructor.

An important aspect of C++ object-oriented programming is **inheritance**:

- A class that inherits from another class gains its members.
- The class inherited from: **base class**
- The inheriting class: **derived class**
- Members of the derived class have access to the `public` and `protected` members of the base class.

```
class MyBase {
    public:
      void member();
  };

class MyDerived : public MyBase {
    public:
     void other_member();
  };
```

```cpp
int main() {
  MyDerived md;

  md.other_member();
  md.member();

  return 0;
}
```

- Only `public` and `protected` members of `MyBase` are inherited by `MyDerived`.
- This is an example of **public inheritance**: no additional restrictions are placed on the inherited members.
- `protected`, `private` inheritance is also possible. Default is `private` for classes and `public` for structs.

- Objects are constructed ‘bottom up’: base before derived
- Objects are detroyed ‘top down’: derived before base
- **IMPORTANT**: virtual base class members can be overidden by derived class.
- **pure virtual** base class members MUST be overridden by derived classes.
- Any class with pure virtual members is called **abstract**.

```
class MyBase {
 public:
  //Virtual member: can be overridden
  virtual void member1();

  //Pure virtual member: must be overridden
  virtual void member2()=0;

  //Ordinary member: cannot be overridden
  void member3();
};
```

```cpp
class MyDerived : public MyBase {
  public:
    //Allowed, but not required
    void member1();

    //Required
    void member2();

    //Forbidden
    void member3();
};
```

Overriding base class members enables **polymorphism**:

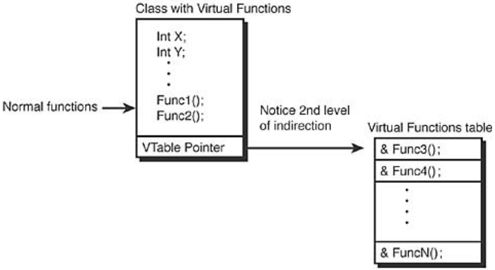
- Derived classes can be dropped into *base class* pointers or references.
- These pointers or references can be manipulated *without knowing which class they point to*.
- The particular derived class can be chosen at compile-time (**compile-time polymorphism**) or at run-time (**run-time polymorphism**).

```
class MyBase {
 public:
  virtual void member();
};

class MyDerived1 : public MyBase {
 public:
  void member();
};
```

```
class MyDerived2 : public MyBase {
 public:
  void member();
};
void func(MyBase& b) { b.member(); }
int main() {
  //Compile-time polymorphism
  MyDerived1 d1; MyBase& r1 = d1;
  func(r1);
  //Run-time polymorphism
  MyBase *bp;
  int one_or_two; cin << one_or_two;
  if (one_or_two==1)
    bp = new MyDerived1();
  else if (one_or_two==2)
    bp = new MyDerived2();
  else bp = new MyBase();
  func(*bp); delete bp;
  return 0;
}
```

- **WARNING**: Classes which are used as polymorphic base classes should have `virtual` destructors!
- There is a computational and storage overhead due to virtual functions.
- In each object of a class w/ virtual functions, a pointer to a entry in the **vtable** (virtual function table) ensures that the correct overridden function is called.
- Vtable functions cannot be inlined, are (20-25%) more inefficient to call, and require an extra pointer to the vtable.



Last major functionality of C++: **templates**

- Recall `Vector`: a container/resource handle for `double`'s
- An analogous container for other types would be very similar.
- Using **templates**, the same source code can be reused for many types.

```
template<typename DataType>
class Vector {
   private:
      DataType* elem;
      int sz;

   public:
   //constructor
   Vector(int s):elem{new DataType[s]}, sz{s} {}

   //destructor
   ~Vector() { delete[] elem; }

   DataType& operator[](int i) {return elem[i];}
   const DataType& operator[](int i) const {
     return elem[i];
   }
   int size() const;
}
```

Now, `Vector` can be used to store *any* type!

```
#include "vector.h"
int main() {
  Vector<double> vecd(10);
  Vector<int> veci(10);
  Vector<bool> vecb(10);

  ....

  vecd[5]=4.5;
  cout << "size = " << vecd.size() << endl;

  return 0;
}
```

Remarks about templates:

- Templates are a mechanism for generating code.
- Template code generation occurs at compile time.
- Only the code which is needed/used is generated.
- In order to generate code, compiler must have templated code functions/members defined and present.

Template arguments can also be variables as well as types!

- Since all template-related code is generated at compile-time, provides a mechanism to specify parameters and/or evaluation expressions at compile-time.
- In principle, any operation can be implemented using ‘template manipulations’ and performed at compile-time.
- Compilation time, compilation errors, and executable size grow significantly with templated code.

A compile-time implementation of the factorial function:

```
//General definition
template<int n>
struct Factorial {
  static const int val = Factorial::val<n-1>*n;
}

//Specialization for the base case
template<>
struct Factorial<0> {
   static const int val = 1;
}

int main() {
  cout << "4! = " << Factorial<4>::val << endl;
  return 0;
}
```

Templates enable a very useful design pattern: **policy**-based classes

- **Policies** are small classes that perform tasks needed for a larger class in a particular way.
- different choices for the policy classes, i.e. different policies, are chosen by passing the classes as template parameters to the larger class.
- Syntax trick: make the larger class inherit from the policy class(es).
- Another syntax trick: send the larger class to the policy class(es) as template parameters. = Curiously Recurring Template Pattern (CRTP)

Simple example:

```
struct Policy1 {
  void func() { /*do policy 1*/}
};
struct Policy2 {
  void func() { /*do policy 2*/}
};
template<typename Policy>
class MyClass : public Policy {
  //func() is a member!
  void member() { func(); }
}
int main() {
  MyClass<Policy1> mc;
  mc.func(); //policy 1 gets called
  return 0;
}
```

Templated version:

```
template<typename T>
struct Policy1 {
  void func() { /*do policy 1*/}
};
template<typename T>
struct Policy2 {
  void func() { /*do policy 2*/}
};
template<typename T,
   template<typename U> class Policy>
class MyClass : public Policy<T> {
  void member() { func(); }
}
int main() {
  MyClass<float,Policy2> mc;
  mc.func();
  return 0;
}
```

The Standard Template Library (STL):

- A collection of (mostly) templated container classes and algorithms. Very useful!
- Sequential Containers: `vector`, `deque`, `list`, ...
- Associative Containers: `set`, `map`, `multiset`, `multimap`, ...
- All containers have **iterators**.
- Generic algorithms: `find`, `copy`, `fill`, `replace`, `sort`, ...

Example: STL vector

```
#include<vector>
using namespace std;
int main() {
  vector<int, allocator> vec(10);
  vec[5] = 3;
  vec.resize(3); //destructive resize
  vec[0]=4; vec[1]=2; vec[2]=10;

  sort(vec.begin(), vec.end());
  //increase size by 1, last element is 3
  vec.push_back(3);

  //decrease size by 1, delete last element
  vec.pop_back();

  return 0;
}
```

Remarks:

- Notice that `sort` took `vec.begin()` and `vec.end()`.
- These are **iterators**, and are essentially pointers to the first and last elements of the vector.
- I could have sorted only the first three elements of the vector with

```
sort(vec.begin(), vec.begin()+3);
```

- All containers provide the same interface using iterators.

Iterators behave just like pointers:

```
#include<list>
#include<set>
using namespace std;
int main() {
  list<int> lcoll(10,1);
  set<int> scoll(10,1);
  for (list<int>::iterator it=lcol.begin();
    it!=lcol.end(); ++it)
      *it = 5;

  for (set<int>::const_iterator it=scol.begin();
    it!=scol.end(); ++it)
      cout << *it << endl;

  return 0;
}
```

Sequential Container guidelines:

- `vector`:
  - (typically) quick to append (`push_back`)
  - generic insertions (`insert`) always require reallocation.
  - memory is contiguous
- `deque`:
  - (typically) quick to prepend (`push_front`) and append.
  - generic insertions always require reallocation.
  - memory is contiguous
- `list`:
  - prepend, append, and generic insertion are typically fast.
  - memory is *not* contiguous.
- Just use `vector`, unless you have a good reason.

Associative Containers:

- data must be comparable, i.e. have a `less` or < implemented.
- implemented as binary trees, searching is very fast.
- order is well defined, iteration proceeds in the same order.
- `set`: collection of unique objects
- `multi_set`: duplicates allowed.
- `map`: set of (key, value) pairs, keys are unique.
- `multi_map`: set of (key, value) pairs, multiple entries with the same key are allowed.
- C++11 has `unordered_` versions which are implemented with hash functions.

STL Algorithms (some of them):

- Sorting:
  - `sort(vec.begin(), vec.end())`: quicksort implementation, at best $\mathrm{O}(n \log n)$, at worst $\mathrm{O}(n^2)$.
  - `partial_sort(vec.begin(), vec.end())`: heapsort implementation. At worst $\mathrm{O}(n \log n)$.
  - `stable_sort(vec.begin(), vec.end())`: mergesort implementation, $\mathrm{O}(n \log n)$ or $\mathrm{O}(n \log n \log n)$ depending on memory allocated.
  - NOTE: implementation is not set by the standard, but complexity is guaranteed. It's best to do some testing, and test results might not be portable.

- `for_each(vec.begin(), vec.end(), square)`, where square is a function that squares a double. Will perform the operation for each element in the range.
- `fill(vec.begin(), vec.end(), value)`: set all elements in the range to value.
- `count(vec.begin, vec.end(), value)`: count all elements in the range equal to value.
- `min_element(vec.begin(), vec.end())`: return the smallest element in the range. Similarly `max_element`.
- `find(vec.begin(), vec.end(), value)`: return the first element equal to value in the range. Similarly, `rfind` returns the last element equal to value.

Heterogeneous Containers:

- Each element can have a different type.
- `std::pair` is the simplest example, can hold 2 elements.
- `std::tuple` (C++11) can hold an arbitrary number of elements.
- Why are they useful? For example, functions that take a variable number of arguments.

pair example:

```
#include<utility>
#include<iostream>

int main() {

   std::pair<int, char> p1;
   p1 = std::make_pair(10, 'A');

   p1::first_type f = p1.first;
   p1::second_type s = p1.second;

   std::cout << f << s << std::endl;
   return 0;
}
```

tuple example (C++11):

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

int main() {
   tuple<int, char> t1(5, 'a');
   typedef typename T tuple<int, double, char>;
   T t2 = make_tuple(1, 3.1, '4');
   auto t3 = make_tuple(6, 5, "yeah", 1.23);
   cout << "size = " <<
      tuple_size<T>::value << endl;
   tuple_elements<3, T>:type e = get<3>(t2);
   cout << "3rd element = " << e;

   return 0;
}
```

More about pointers in C++: **Smart Pointers**

- Recall that `Vector` stored a pointer and handled its `new` and `delete`. (RAII)
- Also recall that the copy construtor and assignment operator performed 'deep' copies, in accordance with the 'Rule of Three'.
- It would be nice to have a 'smart' pointer class that also correctly deleted the pointer. Several questions arise:
  - How do we treat the copy constructor and assignment operator? Don't want a pointer to call `new` all the time.
  - How do we make it safe?

What's dangerous about this code? (Stroustrup)

```
void f(int i, int j) {

   X* p = new X;
   //...

   if (j<77) return;

   p->do_something();
   //...

   delete p;
}
```

- It would be nice to have a pointer that automatically cleaned up, using its destructor.
- Given such a 'smart' pointer, what about pointers to other pointers?

```
void f(int i) {

   SmartPointer<X> p1(new X(i));

   if (i<99) {
      SmartPointer<X> p2(new X(i+1));
      SmartPointer<X> p3(p1);
      p2 = p1; //what happens to newed mem.?

   } //Should ~p2 and ~p3 be called?

   p1->do_something(); //is p1 still ok?
}
```

- The problem lies in copy construction and assignment.
- In such cases, we need to designate an ‘owner’; only one of the pointers is responsible for cleanup.
- There are few options to make this safe(r):
  - Disable copy construction and assignment. A great choice if possible!
  - Transfer ownership on copy construction and assignment. This is in the spirit of C++11 move construction and move assignment.
  - If multiple simultaneous pointers to the same resource are required, only call `delete` *when the last one* goes out of scope.

The STL provides functionality to handle these options (C++11):

- `unique_ptr`:
  - Copy construction and assignment are disabled.
  - However, move construction and move assignment is enabled.

- `shared_ptr`:
  - Allows copy construction and assignment.
  - Move construction and assignment are also enabled.
  - Tracks the number of pointers to a particular memory location and only `delete`'s when the last one is destroyed.

- `weak_ptr`: can be constructed/assigned with a `shared_pointer`, but does not count towards number of pointers.

- `auto_ptr`: C++03 version of `unique_ptr`. Now deprecated.

Concurrency in C++11:

- No support whatsoever in C++03.
- C++11 has a clearly defined memory model (what the compiler may/not do when accessing memory) that supports threading.
- C++11 does NOT provide any support for multiple processes. Basic MPI functionality: BoostMPI.
- Additional reference for C++11 concurrency: 'C++ Concurrency in Action', A. Williams, (Manning, 2012)

Say hello on a new thread (Williams):

```cpp
#include<iostream>
#include<thread>

using namespace std;

void say_hello() {
    cout <<"Hello from thread 1"<<endl;
}

int main() {

    thread t1(say_hello);
    t1.join();

    return 0;
}
```

Comments:

- The spawning of additional threads is managed by the Standard Library `thread` class.
- Additional threads are constructed with a function (or function object) which begins the thread of execution.
- `main` is executed on the intial thread, which could finish and terminate the program before other threads are finished.
- The `join` member instructs other threads to wait until the completion of a particular thread.

- The thread exits when its intialized function completes.
- If initalized with a function object, it is copied (via the copy constructor) into local storage.
  - You don't want the function object to be large!
  - Watch out if the function object holds references.
- `thread::detach` can be called instead of `thread::join`.
  - This instructs the program to keep going.
  - Computation could continue after `detach` is called and possibly after the thread object is destroyed (i.e. falls out of scope).
  - If niether `join` nor `detach` is called before the thread object is destroyed, the program terminates.

A cautionary tale about detach: (Williams)

```
struct MyStruct {
    int& i;
    MyStruct(int& j) : i{j} {}

    void operator()() {
        for (int j = 0 ; j < 1e8 ; ++j)
            do_something(i);
    }
};

void problem() {
    int local_var = 0;
    MyStruct ms(local_var);

    thread t1(my_func);
    t1.detach();
}
```

Handy design Pattern: Use RAII to ensure that all threads are join'ed

```
class joined_thread {
  std::thread t;
  public:
    joined_thread(std::thread tin):
      t(std::move(tin)) {}

    ~joined_thread() {
      if (t.joinable())
        t.join();
    }

    joined_thread(
      const joined_thread&)=delete;
    joined_thread& opreator=(
      const joined_thread&)=delete;
};
```

Comments:

- Note that a `std::thread` object is stored.
  - `thread` copy construction and assignment are disabled.
  - However, move construction and move assignment are enabled.
  - Move construction and move assignement allow the transfer of ownership, and are performed using `std::move()`.
  - `std::move()` casts its argument to an rvalue reference (`&&`).
- A `thread` can only be `join`'ed or `detach`'ed once.
- `t.joinable()` checks if it is still possible to `join` this thread.
- As with other RAII classes, putting `join` in the destructor ensures it is called as the thread goes out of scope.

Test: will the program make it to the return?

```
#include<thread>
using namespace std;
void f() {
   \\...
}
void g() {
   \\...
}
int main() {
   thread t1(f);
   thread t2=move(t1);
   t1=thread(g);
   thread t3=move(t2);
   t1=move(t3);
   t1.join();
   return 0;
}
```

Test answer: NO!

```
#include<thread>
using namespace std;
void f() {
    \\...
}
void g() {
    \\...
}
int main() {
    thread t1(f);
    thread t2=move(t1);
    t1=thread(g);
    thread t3=move(t2);
    t1=move(t3); //t1 previously owned a thread
    t1.join();    //which is destroyed before join
    return 0;
}
```

Passing arguments to the thread function:

```
#include<thread>
using namespace std;
void f(int i, double d) {
   \\...
}
int main() {
   thread t(f, 3, 6.3);
   t.join();
   return 0;
}
```

Warning: conversions happen only when the thread function is called.

```
#include<thread>
#include<string>
using namespace std;
void f(int i, const string& s) {
    \\...
}
int main() {
    thread t(f, 3, "hello");
    t.join();
    return 0;
}
```

Note: `f` will convert `const char*` to `string`, but the thread `t` stores `"hello"` as a `const char*`.

What's the (potential) problem with this?:

```
#include<thread>
#include<string>

using namespace std;

void f(int i, const string& s) {
    \\...
}

void g() {
    char* buffer = "hello";
    thread t(f, 3, buffer);
    t.detach();
}
```

- Only the pointer is copied and stored in the local memory of the thread.
- That pointer is converted to a `string` when the thread function (`f`) is called.
- What if the pointer goes out of scope before the conversion happens and `f` is called? undefined behavior!
- Fix: convert when the thread is intialized. This ensures a new string is created and stored.

```
void g() {
   char* buffer = "hello";
   thread t(f, 3, string(buffer));
   t.detach();
}
```

Another warning: arguments are copied locally to thread, even if passed by ref. to thread function.

```
#include<thread>
using namespace std;
void f(const BigData& bd) {
   //...
}
int main() {
   BigData d;

   //data is copied to thread storage
   thread t(f,d);
   t.join();

   //d is unchanged by thread.

   return 0;
}
```

If you want to copy only a reference to the thread, use `std::ref`:

```
int main() {
    BigData d;

    //a reference to data is copied.
    thread t(f,ref(d));
    t.join();

    //d is modified by thread.

    return 0;
}
```

Sharing data amoung threads:

- if shared data is read-only, no problem.
- Race condition: result depends on execution order of thread.
  - could be benign: threaded accumulate.
  - or problematic: adding/removing items to/from a linked list.
  - a **data race** is a race condition due to many threads modifying the same data.
- Simplest mechanism for sharing data between threads in C++11: `mutex`

Simple mutex example:

```
#include<list>
#include<algorithm>
#include<mutex>
using namespace std;
list<int> lst;
mutex mtx;
void add_to_list(int val) {
    mtx.lock();
    lst.push_back(val);
    mtx.unlock();
}
bool list_contains(int val) {
    mtx.lock();
    bool ret = (find(lst.begin(), lst.end(), val)
        != lst.end());
    mtx.unlock();
    return ret;
}
```

- If each function is called on a different thread, locks prevent code from being executed simultaneously.
- each `lock()` call must have a corresponding `unlock()`!
- Situation where one or more threads wait forever: **deadlock**.
- A simple way to manage locks using RAII: `lock_guard`
  - constructed from a `mutex`
  - constructor calls `mutex::lock`.
  - descructor calls `mutex::unlock`.
  - usual RAII security. Deadlock is prevented if exception is thrown, early return, etc.

Simple `mutex` example with `lock_guard`:

```
#include<list>
#include<algorithm>
#include<mutex>
using namespace std;
list<int> lst;
mutex mtx;
void add_to_list(int val) {
   lock_guard<mutex> grd(mtx);
   lst.push_back(val);
}
bool list_contains(int val) {
   lock_guard<mutex> grd(mtx);
   return (find(lst.begin(), lst.end(), val)
      != lst.end());
}
```

WARNING: unlocked access to protected data is still possible!

```
struct Data {
    int a;
};

class Wrapper {
private:
    Data dat;
    mutex mut;
public:
    template<typname T>
    void safe_access(T f) {
        lock_guard<mutex> lck(mut);
        f(dat);
    }
};
```

```
Data* undat;

void problem(Data& d_in) {
   undat=&d_in;
}

Wrapper wrp;

void thread_func() {
   //data is locked and released
   wrp.saf_access(problem);

   //Access possible without locking!
   undat->a = 5;
}
```

Comments:

- Encapsulating shared data in a wrapper (with locks!) is a good idea.
- Unlocked access is still possible, e.g. by passing data externally.
- In order to be completely safe, don't pass pointers/references to data outside scope of the lock:
  - don't pass data as an argument to an external function.
  - don't assign data to external pointer/reference.
  - don't return pointer/reference to data from a member function.

Deadlock!

- Consider two resources (A and B), each with its own mutex.
- Consider two threads (1 and 2), which need to use both resources.
- Both locks must be aquired by both threads. What happens if they aquire locks in a different order?

- If locks are ALWAYS aquired in the same order, no problem.
- Not always so simple. What about e.g. locking two instances of the same class?
- Example: Swap members of two class instances

```
class Wrapper {
   Data dat;
   mutex mut;
   friend void swap_w(Wrapper&,Wrapper&);
};

void swap_w(Wrapper& w1, Wrapper& w2) {
   if (&w1==&w2) { return; }
   lock_guard<mutex> g1(w1.mut);
   lock_guard<mutex> g2(w2.mut);
   swap(w1.dat, w2.dat);
}
```

- 'Order' is fixed, always lock left argument first.
- What if `swap_w` is called simultaneaously on two threads, with the same arguments, in different orders? DEADLOCK!
- Fortunately, `std::lock(m1,m2,...)` can lock an arbitrary number `mutex`'s *without* any deadlock!
- Deadlock avoidance algorithms are non trivial. Example: Banker's Algorithm (Dijkstra '65)
- C++11 standard does not specify *how* `std::lock` works, so it's implementation dependent.

Revised code which uses `std::lock` to prevent deadlocks:

```
class Wrapper {
    Data dat;
    mutex mut;
    friend void swap_w(Wrapper&,Wrapper&);
};

void swap_w(Wrapper& w1, Wrapper& w2) {
    if (&w1==&w2) { return; }
    lock(w1.mut, w2.mut);
    lock_guard<mutex> g1(w1.mut, adopt_lock);
    lock_guard<mutex> g2(w2.mut, adopt_lock);
    swap(w1.dat, w2.dat);
}
```

We still use `lock_guard`, but pass the enum `std::adopt_lock` to transfer ownership.

- `std::lock` only handles locks aquired together.
- Deadlocks are still possible when multiple locks are aquired seperately.
- Deadlock prevention isn't easy! General rule (Williams): don't wait for another thread if it might be waiting for you.

Deadlocks can occur without locks or shared data!

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

void f() {//...}
void g(thread& t) {
    //...
    t.join();
}

int main() {
    thread t1;
    thread t2(g,t1);
    t1 = thread(g,t2);

    return 0;
}
```

General guidelines to avoid deadlocks:

- Don't aquire multiple locks (if possible). This prevents deadlocks due to `mutex` objects.
- Don't call external functions when holding a lock.
- Aquire multiple locks simultaneously if possible. If not, acquire them in a fixed order on each thread.
- If multiple locks are required, and cannot be aquired simultaneously, enforcing a fixed order aquisition defines a **lock heirarchy**.

```
HierarchicalMutex high_mut(100), med_mut(50),
    low_mut(25);

void med() {
    lock_guard<HierarchicalMutex> g(med_mut);
    //...do some medium level stuff
}

void threadfunc1() {
    lock_guard<HierarchicalMutex> g(high_mut);
    //...do some high level stuff
    med(); //Ok to lock a lower level.
}

void threadfunc2() {
    lock_guard<HierarchicalMutex> g(low_mut);
    //...do some low level stuff
    med(); //NOT OK.
}
```

`unique_lock`:

- `lock_guard` is nice, but only two options are possible:
  - lock the mutex on construction.
  - assume ownership of a locked mutex with `adopt_lock`.
- What if we want the RAII benefits, but don't want to lock when the guard is constructed?
- `std::unique_lock` can be constructed with a mutex and `std::defer_lock` which allows the mutex to be locked at a later date, but still retains the nice RAII safety.

Simple std::unique_lock example:

```
  class Wrapper {
   Data dat;
   mutex mut;
   friend void swap_w(Wrapper&,Wrapper&);
};

void swap_w(Wrapper& w1, Wrapper& w2) {
   if (&w1==&w2) { return; }
   unique_lock<mutex> g1(w1.mut, defer_lock);
   unique_lock<mutex> g2(w2.mut, defer_lock);
   lock(g1, g2);
   swap(w1.dat, w2.dat);
}
```

- `unique_lock` objects may be passed to `std::lock`.
- `lock_guard` cannot be move constructed or assigned.
- In analagy with `unique_ptr`, `unique_lock` allows ownership to be transferred using move contruction/assignment.
- `mutex` locks can now be transferred out of the immediate scope!

Synchronizing operations:

- So far, we have protected shared data by coordinating access with mutex and lock.
- Synchronizing operations is also important. We might want to:
    - wait for a specific event to happen: **futures**
    - wait for a condition to be true: **condition variables**

Condition variable example:

```
#include<condition_variable>
#include<mutex>

mutex mut;
vector<Data> data_vec;
condition_variable data_cond;

void data_get_thread() {
   while(more_data_to_get()) {
      Data temp = get_data();
      lock_guard<mutex> g(mut);
      data_vec.push_back(data);
      data_cond.notify_one();
   }
}
```

```
bool not_empty() { return !data_vec.empty();}

void data_proc_thread() {
   while(true) {
      unique_lock<mutex> lk(mut);

      data_cond.wait(lk, not_empty);
      Data temp = data_vec.back();
      data_vec.pop_back();
      lk.unlock();

      process(temp);

      if (done())
        break;
   }
}
```

Comments:

- `notify_one` member tells the waiting thread to 'wake-up' and check its condition.
- `wait` member takes a lockable object and a boolean function.
- When the waiting thread is notified, it aquires the lock and tests the condition.
- If the condition is false, the waiting thread releases the lock and continues to wait.
- If the condition is true, the waiting thread stops waiting and proceeds (with the lock aquired).

(Williams) Find the answer to Life, the Universe, and Everything using future

```
#include<future>
#include<iostream>

using namespace std;

int answer_to_ltuae();

int main() {
    future<int> the_answer =
        async(answer_to_ltuae);
    //...
    cout << "The answer is " <<
        the_answer.get()<<endl;

    return 0;
}
```

Comments:

- `async` launches a new thread to do the computation.
- constructor is just like `thread` constructor.
- `thread` did not have an easy way to get a result back from the computation.
- The (templated) `future` forces the program to wait at either the `get`, or the `wait` call.

Some syntatic sugar from C++20: the spaceship operator <=>

```
#include<iostream>
using namespace std;
class MyClass {
  int b;
  bool operator<=>(const MyClass& rhs) const = de
};
int main() {
  MyClass m1{5}, m2{6};
  //All the following compile:
  cout << "(m1 < m2) = "<< (m1<m2) << endl;
  cout << "(m1 > m2) = "<< (m1>m2) << endl;
  cout << "(m1 == m2) = "<< (m1==m2) << endl;
  cout << "(m1 != m2) = "<< (m1!=m2) << endl;
  cout << "(m1 <= m2) = "<< (m1<=m2) << endl;
  return 0;
}
```

In conclustion:

- C++ balances 'close to the metal' and abstraction
- 'Close to the metal aspects': pointers+references, strong typing, control over computation/copying
- Abstract concepts: overloaded operators, classes, templates
- Particular advantages for scientific computing: portability, large standard library, efficient compilers, concurrency
- Use C++ if:
  - Speed, efficiency, portability is important
  - Project is large, or there's a potential for growth
  - Multiple collaborators to 'divide and conquer'