Some features of modern C++
Writing readable code

Programming Concepts in Scientific Computing

EPFL, Master class

November 1, 2023

C++ versions

- C++98

C++ versions

- C++98
- C++11
- C++14
- C++17

C++ versions

- C++98
- C++11
- C++14
- C++17
- C++20

C++ versions

- C++98
- C++11
- C++14
- C++17
- C++20
- C++23 (to come)

Writing **readable** code: **auto**

```
std::vector<double> vec(10);
std::vector<double>::iterator it = vec.begin();
```

Writing **readable** code: **auto**

```
std::vector<double> vec(10);
std::vector<double>::iterator it = vec.begin();
```

It is not convenient:

Writing **readable** code: **auto**

```
std::vector<double> vec(10);
std::vector<double>::iterator it = vec.begin();
```

It is not convenient:

```
std::vector<double> vec(10);
auto it = vec.begin();
```

Writing **readable** code: range loops

```
std::vector<double> vec(10);
auto it = vec.begin();
auto end = vec.end();

for (; it != end; ++it) {
  std::cout << *it;
}
```

Writing **readable** code: range loops

So common that it is possible to write

```
std::vector<double> vec(10);

for (double &p : vec) {
  std::cout << p;
}
```

Writing **readable** code: range loops

So common that it is possible to write

```
std::vector<double> vec(10);

for (double &p : vec) {
  std::cout << p;
}
```

Combined with the auto

```
std::vector<double> vec(10);

for (auto &p : vec) {
  std::cout << p;
}
```

Writing **readable** code: range loops

Using double reference (&&)

```
for (auto &&p : vec) {
  std::cout << p;
}
```

iterator can return reference or copy with the same code.

Writing **readable** code: range loops

Using maps

```
std::map<std::string, double> m;

for (std::pair<std::string, double> p : m) {
  auto &k = p.first;
  auto &v = p.second;
  std::cout << "key: " << k << ", value: " << v;
}
```

Writing **readable** code: range loops

Using maps

```
std::map<std::string, double> m;

for (std::pair<std::string, double> p : m) {
  auto &k = p.first;
  auto &v = p.second;
  std::cout << "key: " << k << ", value: " << v;
}
```

Equivalent to

```
for (auto &&[k, v] : m) {
  std::cout << "key: " << k << ", value: " << v;
}
```

Smart Pointers

```
double *get_vector(int n) {

  double *v = new double[n];
  return v;
}
```

Smart Pointers

```
double *get_vector(int n) {

  double *v = new double[n];
  return v;
}
```

To use it beware to free/delete it:

Smart Pointers

```
double *get_vector(int n) {

  double *v = new double[n];
  return v;
}
```

To use it beware to free/delete it:

```
  double *vector = get_vector(10);
  // ... do what I need
  delete[] vector;
```

Smart Pointers

- Memory allocated on the heap needs to be freed
- Forgetting is prone to memory leaks
- Accessing freed memory: unknown result (*Segmentation Fault* usually)

Smart Pointers

- Memory allocated on the heap needs to be freed
- Forgetting is prone to memory leaks
- Accessing freed memory: unknown result (*Segmentation Fault* usually)
- std::shared_ptr are pointers meant to be shared
- std::unique_ptr are pointers guarantied to be unique

Smart Pointers

```
#include <iostream>
#include <memory>

std::unique_ptr<double> get_scalar() {
  // create a unique pointer
  return std::make_unique<double>(3);
}

int main() {

  std::unique_ptr<double> ptr = get_scalar();
  // ... do what I need like...
  std::cout << *ptr;
  // no need to delete scalar (will be automatically)

  // cannot be copied => compilation error
  // std::unique_ptr<double> ptr_copy = ptr;
}
```

Smart Pointers

```
#include <iostream>
#include <memory>

std::shared_ptr<double> get_vector(int n) {
  return std::shared_ptr<double>(new double[n]);
}

int main() {

  std::shared_ptr<double> ptr1 = get_vector(10);
  std::shared_ptr<double> ptr2 = ptr1;

  // memory of pointer freed when
  // ptr1 and ptr2 are out of scope
}
```

Writing **readable** code: Functors

```
struct MyFunctor {
  int operator()() { return 2; }
};
```

Writing **readable** code: Functors

```
struct MyFunctor {
  int operator()() { return 2; }
};
```

```
int main() {
  auto f = MyFunctor();
  std::cout << f() << std::endl;
}
```

Writing **readable** code: Functors

```
struct MyFunctor {
  int operator()(double v) { return v * 2; }
};
```

Writing **readable** code: Functors

```
struct MyFunctor {
  int operator()(double v) { return v * 2; }
};
```

```
  auto f = MyFunctor();

  std::vector<double> vec;
  for (auto d : vec) {
    auto res = f(d);
  }
```

Writing **readable** code: Functors

```
struct MyFunctor {
  int operator()(double v) { return v * 2; }
};

template <typename VecType, typename T> // _
void for_each(VecType &vec, T f) {
  for (auto d : vec) {
    auto res = f(d);
  }
}

int main() {
  auto f = MyFunctor();
  std::vector<double> vec(10);
  for_each(vec, f);
}
```

Writing **readable** code: Lambda functors

http://en.cppreference.com/w/cpp/language/lambda

```
struct MyFunctor {
  MyFunctor(double a) : a(a) {}

  int operator()(double v) { return v * a; }
  double a;
};
```

Writing **readable** code: Lambda functors

http://en.cppreference.com/w/cpp/language/lambda

```
struct MyFunctor {
  MyFunctor(double a) : a(a) {}

  int operator()(double v) { return v * a; }
  double a;
};
```

Calling:

```
  double a = 2.;
  MyFunctor f(a);
```

Writing **readable** code: Lambda functors

http://en.cppreference.com/w/cpp/language/lambda

```
struct MyFunctor {
  MyFunctor(double a) : a(a) {}

  int operator()(double v) { return v * a; }
  double a;
};
```

Calling:

```
  double a = 2.;
  MyFunctor f(a);
```

Replaced with:

```
  auto f_lambda = [a](double d) { return a * d; };
```

Writing **readable** code: Lambda functors

```
template <typename VecType, typename T> // _
void for_each(VecType &vec, T f) {
  for (auto d : vec) {
    auto res = f(d);
  }
}

int main() {
  std::vector<double> vec(10);
  for_each(vec, [](double d) { return 2 * d; });
}
```

Writing **readable** code: Lambda functors

```
template <typename VecType, typename T> // _
void for_each(VecType &vec, T f) {
  for (auto d : vec) {
    auto res = f(d);
  }
}

int main() {
  int a = 2;
  std::vector<double> vec(10);
  for_each(vec, [a](double d) { return d * a; });
}
```

Writing **readable** code: For each

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

int main() {
  std::vector<double> v(10);
  std::iota(v.begin(), v.end(), 0);
  std::for_each(v.begin(), v.end(), [](double &x) { x = x * x; });
  std::for_each(v.begin(), v.end(),
                [](double x) { std::cout << x << std::endl; });
}
```

What is this code doing ? (homework)
help @ http://en.cppreference.com/

Writing **readable** code: Concepts

```
#include <iostream>

void foo(int n) { std::cout << "Number: " << n << std::endl; }

void foo(double n) { std::cout << "Number: " << n << std::endl; }

void foo(std::string n) { std::cout << "String: " << n << std::endl; }

void foo(const char *n) { std::cout << "String: " << n << std::endl; }

int main() {
  double a = 3.14;
  int b = 2;
  foo(a);
  foo(b);
  foo("toto");
}
```

Conceptually we would like to factor the code
(specialization ?)

Writing **readable** code: Concepts

Ideally we would like to write:

Writing **readable** code: Concepts

Ideally we would like to write:

```
template <Number T> void foo(T n) { //
  std::cout << "Number: " << n << std::endl;
}

template <String T> void foo(T n) { //
  std::cout << "String: " << n << std::endl;
}
```

Writing **readable** code: Concepts

Ideally we would like to write:

```
template <Number T> void foo(T n) { //
  std::cout << "Number: " << n << std::endl;
}

template <String T> void foo(T n) { //
  std::cout << "String: " << n << std::endl;
}
```

This is a C++ (20) concept

https://en.cppreference.com/w/cpp/language/constraints

Writing **readable** code: Concepts

```
template <typename T>
concept Number = std::is_arithmetic_v<T>;

template <typename T>
concept String =
    std::is_same_v<T, std::string> or std::is_same_v<T, const char *>;
```

- This is a C++ (20) concept

Writing **readable** code: Concepts

A concept is defined like:

```
template <typename T> concept Number =
std::is_arithmetic_v<T>;

template <typename T> concept String = std::is_same_v<T,
std::string> or std::is_same_v<T, const char *>;
```

- This is a C++ (20) concept

Writing **readable** code: Concepts

A concept is defined like:

```
template <typename T> concept Number =
std::is_arithmetic_v<T>;

template <typename T> concept String = std::is_same_v<T, std::string> or
std::is_same_v<T, const char *>;
```

- This is a C++ (20) concept
- Constraints on template types
- Use of STL *type metaprogramming*

Writing **readable** code: Concepts

A concept is defined like:

```
template <typename T> concept Number = std::is_arithmetic_v<T>;

template <typename T> concept String = std::is_same_v<T,
std::string> or std::is_same_v<T, const char *>;
```

- This is a C++ (20) concept
- Constraints on template types
- Use of STL *type metaprogramming*

Writing **readable** code: Concepts

Ideally we would like to write:

Writing **readable** code: Concepts

Ideally we would like to write:

```
void foo(Number auto n) { //
  std::cout << "Number: " << n << std::endl;
}

void foo(String auto n) { //
  std::cout << "String: " << n << std::endl;
}
```

Writing **readable** code: Concepts

Ideally we would like to write:

```
void foo(Number auto n) { //
  std::cout << "Number: " << n << std::endl;
}

void foo(String auto n) { //
  std::cout << "String: " << n << std::endl;
}
```

This is a C++ (20) concept
https://en.cppreference.com/w/cpp/language/constraints

Modern C++

Take away message

- **auto**: automatic declaration of type on a function return
- **range loop**: Efficient syntax to loop over generic containers (vector, list, set)
- **smart pointers**: objects managing raw pointers in a safe way
- **functors**: object with () operator, to store functions
- **lambda**: compact declaration of functors
- **std::for_each**: apply a functor to every item of a container
- **concept**: define constraints on templated types
- **STL meta-programming**: allows to manipulate types (doc)