- https://google.github.io/styleguide/cppguide.html c++ style guide
- Book I actually bought: https://moodle.ufsc.br/pluginfile.php/2377667/mod_resource/content/0/Effective_Modern_C__.pdf Effective Modern C++
- Object oriented programming with compile-time typing
  - ```
    #include <cstring>

    template <typename T>
    class Alvec {
    public:
        void push_back(T item);
        size_t size() const { return dataCount; }
        void pop_back();
        T operator[](size_t index) const noexcept(true);
        T at(size_t index) const noexcept(false);
        Alvec() : capacity(2), data(new T[capacity]) {}
        ~Alvec() { delete[] data; }
    protected:
      void resize(size_t newSize);
      size_t capacity;
      T* data;
      size_t dataCount = 0;
    };

    template <typename T> void Alvec<T>::resize(size_t newSize){
      T * newMemory = new T[newSize];
      memcpy(newMemory, data, dataCount);
      delete[] data;
      data = newMemory;
    }

    template <typename T> void Alvec<T>::push_back(T item){
      if (capacity <= dataCount){
        resize(capacity*2);
      }
      dataCount++;
      data[dataCount -1] = item;
    }

    template <typename T> void Alvec<T>::pop_back(){
      dataCount--;
      if (capacity/2 > dataCount){
        resize(capacity/2);
      }
    }

    template<typename T> T Alvec<T>::operator[](size_t index) const noexcept(true){
      return data[index];
    }

    template<typename T> T Alvec<T>::at(size_t index) const noexcept(false){
      if (index >= dataCount){
        throw std::out_of_range("index");
      }
      return operator[](index);
    }
    ```
- Constructors
  - A movable type is one that can be initialized and assigned from temporaries. “A copyable type is one that can be initialized or assigned from any other object of the same type, with the stipulation that the value of the source does not change.”
  - ```
    class Copyable {
     public:
      Copyable(const Copyable& rhs) = default;
      Copyable& operator=(const Copyable& rhs) = default;

      // The implicit move operations are suppressed by the declarations above.
    };

    class MoveOnly {
     public:
      MoveOnly(MoveOnly&& rhs);
      MoveOnly& operator=(MoveOnly&& rhs);

      // The copy operations are implicitly deleted, but you can
      // spell that out explicitly if you want:
      MoveOnly(const MoveOnly&) = delete;
      MoveOnly& operator=(const MoveOnly&) = delete;
    };

    class NotCopyableOrMovable {
     public:
      // Not copyable or movable
      NotCopyableOrMovable(const NotCopyableOrMovable&) = delete;
    ```

```
  NotCopyableOrMovable& operator=(const NotCopyableOrMovable&)
      = delete;

  // The move operations are implicitly disabled, but you can
  // spell that out explicitly if you want:
  NotCopyableOrMovable(NotCopyableOrMovable&&) = delete;
  NotCopyableOrMovable& operator=(NotCopyableOrMovable&&)
      = delete;
};
```

- https://google.github.io/styleguide/cppguide.html#Implicit_Conversions "Every class's public interface should make explicit which copy and move operations the class supports."
- - move construction via rvalue (prvalue) reference `&&`.
  `string(String && other){ data = other.dataptr; other.dataptr = 0; }` we can steal data from rvalues.

▼ Default initialization:

- - value types in file scope are default initialized to zero, while value types in classes or functions are not default initialized.
- - Class types with default constructors are default initialized, except their value types (unless constructor does so)

▼ Function overloading

- ```
  class MyClass {
   public:
    void Analyze(const string &text);
    void Analyze(const char *text, size_t textlen);
  };
  ```

▼ - Pure virtual? `virtual void heapify(int startingIndex) = 0;` set zero

- ```
  class MyBaseClass { //uses default constructor
   public:
    virtual void doYourJob() = 0;
  };
  ```
- - `void doYourJob() override {}` ensures fn is already virtual, overrides virtual
- - virtual on function in base class auto-virtualizes descendants. Also makes the class and all descendants a dynamic class.

▼ Templates— major feature! Class and function templates.. Incomplete classes you populate with types at compile time, and then the compiler handles as if a complete class.

If you're willing to overlook a pinch of pseudocode, we can think of a function template as looking like this:

```
template<typename T>
void f(ParamType param);
```

A call can look like this:

```
f(expr);                    // call f with some expression
```

During compilation, compilers use *expr* to deduce two types: one for `T` and one for *ParamType*. These types are frequently different, because *ParamType* often contains adornments, e.g., `const` or reference qualifiers. For example, if the template is declared like this,

```
template<typename T>
void f(const T& param);     // ParamType is const T&
```

and we have this call,

```
int x = 0;

f(x);                       // call f with an int
```

▼ `T` is deduced to be `int`, but *ParamType* is deduced to be `const int&`.
https://moodle.ufsc.br/pluginfile.php/2377667/mod_resource/content/0/Effective_Modern_C__.pdf

▼ Treated as non-const and non-volatile

▼ Volatile is probably something you *don't* need:

- ```
  volatile int some_int = 100;

  while(some_int == 100)
  {
     //your code
  }
  ```

▼ Standard library
  ```

Strings library

<cctype>	Functions to determine the type contained in character data
<cwctype>	Functions to determine the type contained in wide character data
<cstring>	various narrow character string handling functions
<cwchar>	various wide and multibyte string handling functions
<cuchar> (since C++11)	C-style Unicode character conversion functions
<string>	`std::basic_string` class template
<string_view> (since C++17)	`std::basic_string_view` class template
<charconv> (since C++17)	`std::to_chars` and `std::from_chars`

Containers library

<array> (since C++11)	`std::array` container
<vector>	`std::vector` container
<deque>	`std::deque` container
<list>	`std::list` container
<forward_list> (since C++11)	`std::forward_list` container
<set>	`std::set` and `std::multiset` associative containers
<map>	`std::map` and `std::multimap` associative containers
<unordered_set> (since C++11)	`std::unordered_set` and `std::unordered_multiset` unordered associative containers
<unordered_map> (since C++11)	`std::unordered_map` and `std::unordered_multimap` unordered associative containers
<stack>	`std::stack` container adaptor
<queue>	`std::queue` and `std::priority_queue` container adaptors
**** (since C++20)	`std::span` view

Iterators library

<iterator>	Range iterators

Input/output library

<iosfwd>	forward declarations of all classes in the input/output library
<ios>	`std::ios_base` class, `std::basic_ios` class template and several typedefs
<istream>	`std::basic_istream` class template and several typedefs
<ostream>	`std::basic_ostream`, `std::basic_iostream` class templates and several typedefs
<iostream>	several standard stream objects
<fstream>	`std::basic_fstream`, `std::basic_ifstream`, `std::basic_ofstream` class templates and several typedefs
<sstream>	`std::basic_stringstream`, `std::basic_istringstream`, `std::basic_ostringstream` class templates and several typedefs
<syncstream> (since C++20)	`std::basic_osyncstream`, `std::basic_syncbuf`, and typedefs
<strstream> (deprecated in C++98)	`std::strstream`, `std::istrstream`, `std::ostrstream`
<iomanip>	Helper functions to control the format of input and output
<streambuf>	`std::basic_streambuf` class template
<cstdio>	C-style input-output functions

Localization library

<locale>	Localization utilities
<clocale>	C localization utilities
<codecvt> (since C++11)(deprecated in C++17)	Unicode conversion facilities

Regular Expressions library

<regex> (since C++11)	Classes, algorithms and iterators to support regular expression processing

Atomic Operations library

<atomic> (since C++11)	Atomic operations library

Thread support library

<thread> (since C++11)	`std::thread` class and supporting functions
<mutex> (since C++11)	mutual exclusion primitives
<shared_mutex> (since C++14)	shared mutual exclusion primitives
<future> (since C++11)	primitives for asynchronous computations
<condition_variable> (since C++11)	thread waiting conditions

▼ Iterators, part of standard types. Templates of iterators. Mess to iterate without `**auto**`

Item 5: Prefer `auto` to explicit type declarations.

Ah, the simple joy of

```
int x;
```

Wait. Damn. I forgot to initialize x, so its value is indeterminate. Maybe. It might actually be initialized to zero. Depends on the context. Sigh.

Never mind. Let's move on to the simple joy of declaring a local variable to be initialized by dereferencing an iterator:

```
template<typename It>    // algorithm to dwim ("do what I mean")
void dwim(It b, It e)    // for all elements in range from
{                        // b to e
  while (b != e) {
    typename std::iterator_traits<It>::value_type
      currValue = *b;
    …
```

- What a mess!

```
template<typename It>       // as before
void dwim(It b, It e)
{
  while (b != e) {
    auto currValue = *b;
    …
  }
}
```

- Way better!
- Other benefits of auto

```
int x1;                     // potentially uninitialized

auto x2;                    // error! initializer required

auto x3 = 0;                // fine, x's value is well-defined
```

- Class templates

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

- - variadic `template<typedef ... Types> int sum(int i, Types ... andmore){return i + sum(andMore ... );} int sum(){return 0}`
  - Typedef and class are the same in this context

```
template<class ... Types> void f(Types ... args);
f();       // OK: args contains no arguments
f(1);      // OK: args contains one argument: int
f(2, 1.0); // OK: args contains two arguments: int and double
```

- Pass by value and by reference
  - `void Foo(const string &in, string *out);`
  - Pass by ref makes it clear that null is not an option. https://google.github.io/styleguide/cppguide.html#Ownership_and_Smart_Pointers
- - lValue (located value) last longer than expression. prValue is only for expr, no identity (expression typed), can be moved from, can't take address. xvalue, prValue with identity, like `std::move(x)`.
  - - `void swap(int& x, int& y){}; int a,b; swap(a,b)` — x,y must be movable
- Classes:
  - - static members, initialize exactly once. prevent race: `Fred& global(){ static Fred* kept = new Fred(); return *kept;}`
  - - static declared functions, can only be called on class `X::statFn()`

- Types and conversion
  - - `dynamic_cast<Bird*>(animal)` does upcasting, nullptr if not the type you think
    - typeid
  - - `reinterpret_cast<othertype>(yo)`, uses underlying bit pattern for cast, is "type punning"
    - - convert `char` to `unsigned char` how? `char thechar = -127;`, `*(reinterpret_cast<unsigned char*>(&t))` yields *129* whereas `static_cast<unsigned char>(thechar)` yields *129* gooo twos complement!
  - - `static_cast<new_type>(value)` does downcasting, unsafe upcasting when you're sure, safer
    - https://google.github.io/styleguide/cppguide.html#Casting
    - sometimes you are doing a *conversion* (e.g., `(int)3.5`) and sometimes you are doing a *cast* (e.g., `(int)"hello"`).
    - Integral promotion vs numeric conversion https://en.cppreference.com/w/cpp/language/implicit_conversion
    - Static cast prevents you from from accidentally causing a const_cast or reinterpret_cast which is good.
  - C-style Implicit conversions:
    - https://google.github.io/styleguide/cppguide.html#Implicit_Conversions Use the explicit keyword to prevent them if you want.
      - explicit Foo(int x);
        Foo a = 42; //Compile-time error: can't convert 42 to an object of type Foo
    - When considering the argument to a constructor or to a user-defined conversion function, only one standard conversion sequence is allowed (otherwise user-defined conversions could be effectively chained). When converting from one built-in type to another built-in type, only one standard conversion sequence is allowed.
      https://en.cppreference.com/w/cpp/language/implicit_conversion
      - Implicit conversion sequence consists of the following, in this order:

        1) zero or one *standard conversion sequence*;
        2) zero or one *user-defined conversion*;
        3) zero or one *standard conversion sequence*.
      - A standard conversion sequence consists of the following, in this order:

        1) zero or one *lvalue transformation*;
        2) zero or one *numeric promotion* or *numeric conversion*;
        3) zero or one *function pointer conversion*; (since C++17)
        4) zero or one *qualification adjustment*.
- Visitor pattern http://web.mit.edu/6.031/www/sp17/classes/28-little-languages-2/#visitor
  - One new visitor class with 'on' function, instantiate it and call accept on the types which calls back on visitor class.
  - Allows you to add a function to a bunch of types just in one class
    Each subclass of Visitor implements functionality.
- Smart pointers:
  - • "Smart" pointers are classes that act like pointers, e.g. by overloading the `*` and –> operators.
  - - `include <memory>` `std::unique_ptr `
  - the object is deleted when the `std::unique_ptr` goes out of scope
  - `std::shared_ptr`s can be copied; ownership of the object is shared among all copies, and the object is deleted when the last `std::shared_ptr` is destroyed
  - ```
    std::unique_ptr<Foo> FooFactory();
    void FooConsumer(std::unique_ptr<Foo> ptr);
    ```
- Gotchas:
  - - `Myvector() : capacity(2), data(new T[capacity]) {}` evals in member order => data length is undefined IF `{T* data; size_type capacity;}` data defined first.
  - - 'most vexing parse' `TimeKeeper time_keeper(Timer());` is interpreted as fn, receiving anon **fn returning type timer** as arg. `TimeKeeper time_keeper((Timer()));` disambiguates to variable def
- Neat: compilers usually implement return-by-value using pass-by-reference, when initializing, with no `Foo x; x = retFoo()`, it can prevent any temporary.
    ```