**The 15 C++11 features you must really use in your C++ projects.**

1. **auto**
2. **2- nullptr**
3. **3- shared\_ptr**
4. **4- Strongly-typed enums**
5. **5- static assert**
6. **Variadic template**
7. **Range-based for loops**
8. **Initializer lists**
9. **Noexcept**
10. **Move**
11. **Lambda**
12. **Explicitly defaulted and deleted special member functions**
13. **override identifier**
14. **std::thread**
15. **Unordered containers**
    1. unordered\_map
    2. unordered\_set
    3. unordered\_multimap
    4. unordered\_multiset

# Each piece of code has a story, give it a good end from the beginning.

# Easy steps to modernize your C++ code.

int partition(int\* input,int p,int r){

int pivot = input[r];

while( p < r ){

while( input[p]< pivot )

p++;

while( input[r]> pivot )

r--;

if( input[p]== input[r])

p++;

elseif( p < r ){

int tmp = input[p];

input[p]= input[r];

input[r]= tmp;

}

}

return r;

}

// The quicksort recursive function

void quicksort(int\* input,int p,int r){

if( p < r ){

int j = partition(input, p, r);

quicksort(input, p, j-1);

quicksort(input, j+1, r);

}

}

**Step1: Replace containers by iterators**

**Step2: Make the comparator generic if possible**

**Step3: Replace treatments by standard ones**

template< typename BidirectionalIterator, typename Compare >

void quick\_sort( BidirectionalIterator first, BidirectionalIterator last, Compare cmp ) {

if( first != last ) {

BidirectionalIterator left = first;

BidirectionalIterator right = last;

BidirectionalIterator pivot = left++;

while( left != right ) {

if( cmp( \*left, \*pivot ) ) {

++left;

} else {

while( (left != right) && cmp( \*pivot, \*right ) )

--right;

std::iter\_swap( left, right );

}

}

--left;

std::iter\_swap( pivot, left );

quick\_sort( first, left, cmp );

quick\_sort( right, last, cmp );

}

}

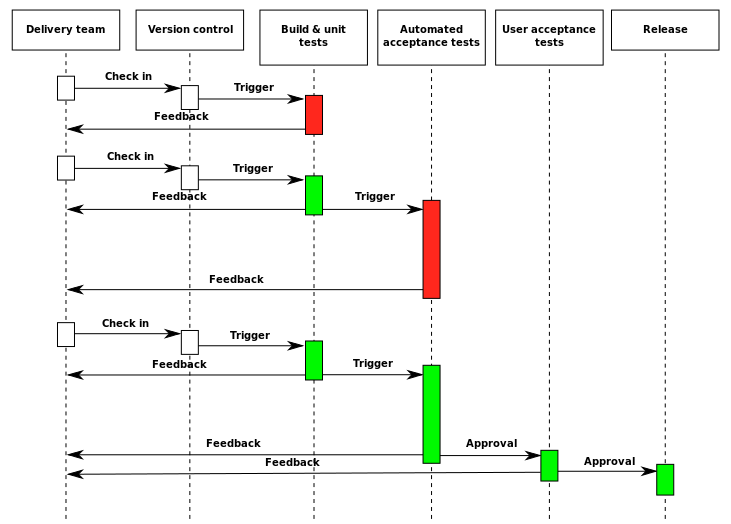
template< typename BidirectionalIterator >

inline void quick\_sort( BidirectionalIterator first, BidirectionalIterator last ) {

quick\_sort( first, last,

std::less\_equal< typename std::iterator\_traits< BidirectionalIterator >::value\_type >());

}



* The only valid measurement of code quality: WTF/minute
* High level language like C++ not written to communicate with compilers. They’re usually written to communicate with people.
* - Every comment is a failure to express yourself in code.

What is your code trying to tell you???

Code that is hard to write is hard to read also maintain, test and fix( range based for , algorithms , auto)

Compile at high level of warning

Stop writing C code and write C++ code

Avoid #ifdef when ever possible; when necessary keep them simple

User RAII everywhere, even in the absence of exception

Keep functions linear and don’t write arrow code

Const qualify everything

Don’t use C casts

Macros are ugly

Walls of code are ugly

Lambdas are beautiful

Invisible code is beautiful

Removing effort is beautiful

Other people’s code is beautiful

Comments are ugly

Macros that should be functions

Replace function-like macros with functions

|  |  |  |
| --- | --- | --- |
| No documentation | Coding standards | Other people code |
| Magic numbers | Tightly coupled | Large classes |
| Large classes | Global variables | Bad names |
| Broken code | Bad formatting | Improper scoping |

Remove 3 C’s

* Clutter is any things in your code that does not add value
  + Format your code
  + Delete Comments
  + Delete dead code
  + Delete unnecessary codes
* Complexity
  + Bad names
  + Long methods
  + Deep conditionals (if / for / while / switch)
  + Magic numbers
  + Improper variable scoping
  + Missing encapsulation
  + Obscure code blocks
* Cleverness

Remove Duplicationsssssssssss

Must be check:

* Const keyword
* Auto keyword
* Typedef keyword
* Unsigned data type
* Preferring sizeof(varname) to sizeof(type)
* Destroy objects in destructor
* Avoiding using namespace directive
* Define a guard or pragma at the beginning of every headers
* Make\_shared instead of reset
* Nullptr instead of null
* Static const type X instead of #define X val

Most ‘if’s can be replaced by polymorphism

Because function without if are easier to : ‘read’ to ‘test’ to ‘maintain’

Never return null in function

Don’t return error code instead of throw an exception

Replace conditions with polymorphism

Replace the switch with polymorphism

Switch almost always means you should use polymorphism

If is more subtle … some times an if just an if

**Clean Code**

**Code must be clean, safe, fast**

**Type and memory safety**

**Type safe, ex: avoid union use variant**

**Bounds safe, ex: avoid pointer arithmetic use array\_view**

**Life time safe, ex: (don’t leak: forget to delete) (don’t corrupt - double delete) (don’t dangle return &local)**

**Bounds safety**

**GSL types**

**Array\_view , string\_view**

**Rules:**

**Don’t use pointer arithmetic . use array view instead**

**Only index into array using constant expression**

**Don’t use array-to-pointer decay**

**Don’t use::function and types that are not bounds-checked**

# What make C++ C++??

* Objects and class
* Operator overloading
* Using templates
* Const
* Exceptions
* The standard library
* Lambdas
* User defined types

# The worst question in C++

## Is shared\_ptr thread safe??? NO

## Use a lock on shared\_ptr

Atomic<shared\_ptr>

Auto& ref = \*returns\_a\_shared\_ptr();

Ref.boooom;

RAII types + default constructor -> DANGER

How do you write HARD TO TEST Code??

Doing work in constructor is dangerous

New operators are dangerous

Components:

* Well define boundaries
* Dependency injection
* Global state free
* Factories for easy instatntiation

<http://misko.hevery.com/>