Elevate your Code to Modern C++ with Automated Tooling

Peter Sommerlad

Download IDE at:
www.cevelop.com

Simple C++

- Less Code == More Software
- Know your language and its (modern) idioms
- Don’t be afraid of STL or templates
- Start small. Tools can help.

C++ hello world
(as generated by Eclipse CDT)

- What is wrong here?

```
//============================================================================
// Name        : helloworld.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
using namespace std;

int main() {
    cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
    return 0;
}
```

belongs into version management system

bad practice, very bad in global scope

ridiculous comment

redundant

inefficient, redundant

using global variable! really bad if not in main() :-(

Namespaces

```
using namespace std;
```

- common, but discouraged. Verboten in headers.
- But very convenient when writing code using std:: namespace components (or any other, like boost::)
- Cevelop's "Namespactor" helps refactoring code:
  - type it with using namespace and then inline the namespace where it was implicitly used

Namespace Refactorings

- Inline using namespace/declaration
  - write code with using namespace and then get rid of it
- Extract using declaration
  - avoid repeating std:: in front of the same identifier, e.g., using std::string;
- Extract using namespace
  - into the most local scope
- Qualify unqualified name
  - make clear, which namespace a name belongs to (prohibit ADL)

C++11: auto

- deduction like function template typename argument
- type deduced from initializer, use =

```
auto var= 42;
auto a  = func(3,5.2);
auto res= add(3,5.2);
```

  - use for local variables where value defines type

```
auto func(int x, int y) -> int {
   return x+y;
}
```

```
template <typename T, typename U>
auto add(T x, U y)->decltype(x+y){
   return x+y;
}
```

  - use for late function return types (not really useful, except for templates, better in C++14)

C++14: auto

```
auto func(int x, int y) {
   return x+y;
}
```

- type deduction even for function return types (not only lambdas)

```
template <typename T, typename U>
decltype(auto) add(T &&x, U &&y){
   return x+y;// may be overloaded
}
```

- can even use decltype(auto) to retain references

useful auto

- Use auto for variables with a lengthy to spell or unknown type, e.g., container iterators

```
std::vector<int> v{1,2,3,4,5};

auto it=v.cbegin();
std::cout << *it++<<'\n';

auto const fin=v.cend();
while(it!=fin)
    std::cout << *it++ << '\n';
```

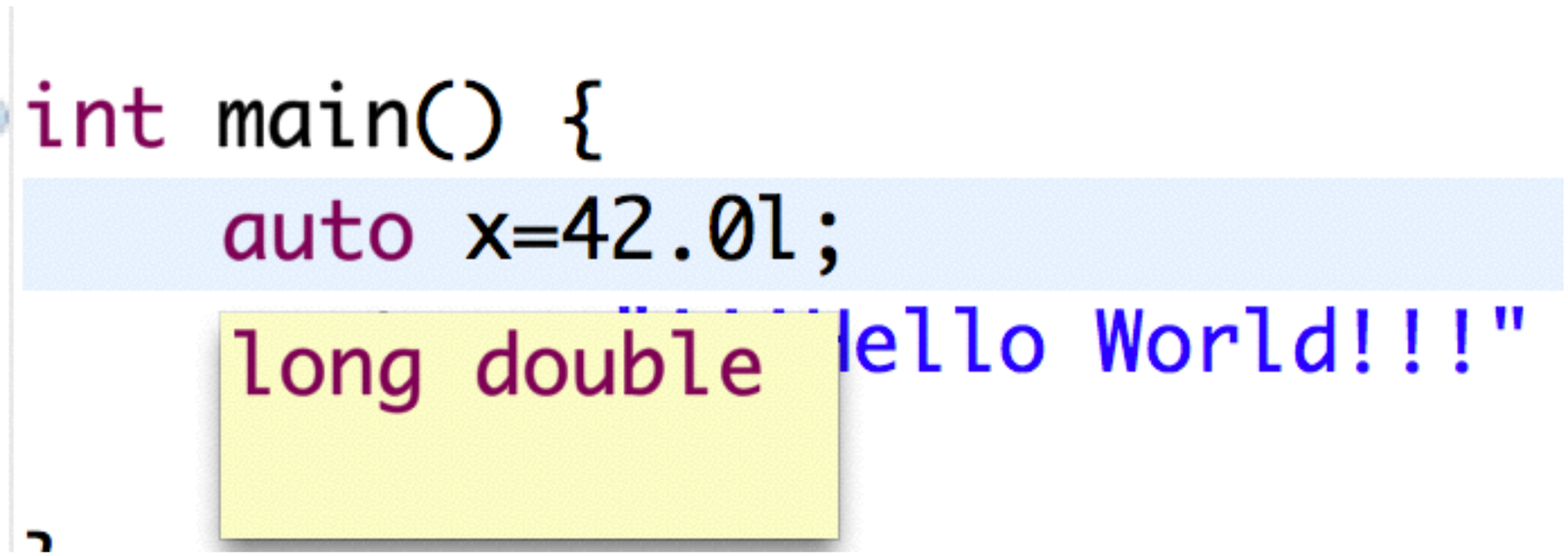
- Also for for() loop variables
  - especially in range-based for()
  - could use &, or const if applicable

```
for (auto i=v.begin(); i!=v.end();++i)
    *i *=2;
```

```
for (auto &x:v)
    x += 2;
for (auto const x:v)
    std::cout << x << ", ";
```

Cevelop/CDT auto

- hover shows deduced type:



Uniform Initialization {}

```
int x;
std::string y;
```

What's the difference?

```
std::string a("");
std::string b();
```

- unfortunately uniform is not (yet) universal
- some gotchas, but at least it allows to mark variables to be default initialized
  - which unfortunately might mean they aren't
- C++11 allows us clearly mark initialization with {}
  - The "Elevator" suggests this and changes it

uniform initialization {}

- C-struct and arrays allow initializers for elements for ages, C++ allows constructor call

```
struct point{
   int x;
   int y;
   int z;
};
point origin={0,0,0};
point line[2]={{1,1,1},{3,4,5}};
```

```
int j(42);
std::string s("hello");

int f();
std::string t();
```

What's wrong here?

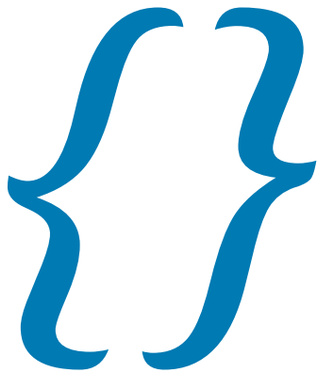
- C++11 uses {} for "universal" initialization:

```
int i{3};
int g{};
std::vector<double> v{3,1,4,1,5,9,2,6};
std::vector<int> v2{10,0};
std::vector<int> v10(10,0);
```

Caveat: use () if ambiguous!

caveat: auto and initializer

```
auto i={3};
```



- might be fixed in C++17
- auto i{5}; // int

“most vexing” parse {}

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

int main() {
   using in=istream_iterator<int>;
   vector<int> v(in(cin),in());
   for (auto x:v){
      cout << x <<", ";
   }
}
```

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

int main() {
   using in=istream_iterator<int>;
   vector<int> v{in{cin},in{}};
   for (auto x:v){
      cout << x <<", ";
   }
}
```

- using {} initializers avoids getting trapped

Macros

- Macros in C++ are obsolete (almost) or damned
  - Work on Reflection will help rid those needed for Unit Testing and Logging
  - Most other uses of Macros in C++ are no longer needed due to C++11/14 mechanisms for guaranteed compile-time computation
- “cleanly”-written macros can be eliminated

Macro Replacement

```
#define PI 3.1415926
#define RUN_FUNC(X) do { X; } while (0)
#define SQUARE(X) ((X) * (X))
#define UNUSED_OBJECTSTYLE_MACRO X
#define UNUSED_FUNCTIONSTYLE_MACRO(X) ((X) * (X))
int main() {
   RUN_FUNC(foo());
   cout << "The square of 42 is " << SQUARE(42) << '\n';
   cout << "The square of PI is " << SQUARE(PI) << '\n';
   return 0;
}
```

- Can use constexpr constants or functions
- Or, eliminate macro by inlining it everywhere

Macro Replacement

```
constexpr auto PI = 3.1415926;
template<typename T1>
inline void RUN_FUNC(T1&& X)
{
    do{
        X;
    } while (0);
}
template<typename T1>
inline constexpr auto SQUARE(T1&& X) -> decltype(((X) * (X)))
{
    return (((X) * (X)));
}
```

- Can use constexpr constants or functions
- Or, eliminate macro by inlining it everywhere

Arrays

- C-Arrays are bad! use std::array instead
  - no loss of dimension/ degeneration to pointer when passing as argument
- STL API, can do bounds check with .at(idx)

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

int main() {
   int a[]={1,2,3,4,5};
   double b[4] = { 3.14, 15 };

   using out=ostream_iterator<int>;
   copy(begin(a),end(a),out{cout});
   std::cout << *b;
}
```

Arrays elevated

- Just press CTRL-1/CMD-1

```
#include <iostream>
#include <array>


int main() {
   std::array<int, 5> a = { { 1, 2, 3, 4, 5 } };
   std::array<double, 4> b = { { 3.14, 15 } };
   using out=ostream_iterator<int>;
   copy(begin(a),end(a),out{cout,"-"});
   std::cout << *b.data();
   std::cout << b.at(42); // show bounds check
}
```

Pointer Parameters

- Pointers are evil…
  - as Parameter types often a C-legacy, especially when they can not be nullptr
- Better use C++'s references then
- Or std::optional or smart pointers in the future

```
void myswap(int *x, int *y);

int main() {
   int x { 1 };
   int y { 2 };
   myswap(&x, &y);
}
void myswap(int *x, int *y) {
   int t { *x };
   *x = *y;
   *y = t;
}
```

Reference Parameters

- Refactor pointer parameters to references
- adjust call sites and declarations accordingly

```
void myswap(int& x, int& y);

int main() {
   int x { 1 };
   int y { 2 };
   myswap(x, y);
}
void myswap(int& x, int& y) {
   int t { x };
   x = y;
   y = t;
}
```

Strings

- char * and char[] are evil… std::string OK
  - stay tuned for std::string_view if you really have to care about avoiding copies and allocation
- However, changing code by hand is hard
- CharWars attempts to solve that

PERFORMANCE

```
void strcat(char *dest, char *src) {
  while(*dest) dest++;
  while(*dest++ = *src++);
}
```

SAFETY

```
int main() {
   char pw[7] = "secret";
   char name[5];
   strcpy(name, "John_123456");
   std::cout << pw;    //123456
}
```

MAPPINGS

`strlen(a)`	▷	`a.size()`
`strcmp(a, b)`	▷	`a.compare(b)`
`strcat(a, b)`	▷	`a += b`
`strcpy(a, b)`	▷	`a = b`
`strstr(a, b)`	▷	`a.find(b)`
`strchr(a, ch)`	▷	`a.find_first_of(ch)`
`strspn(a, b)`	▷	`a.find_first_not_of(b)`
`a = strdup(b)`	▷	`a = b`

OPTIMIZATIONS

```
if(strcmp(pwd, "secret") == 0) {
   //access granted
}
```

```
if(pwd.compare("secret") == 0) {
   //access granted
}
```

EXAMPLE REFACTORING

```
#include <iostream>



int main() {
    char str[100] = "myfile";

    if(strchr(str, '.') == nullptr) {
       strcat(str, ".txt");
    }
}
```

EXAMPLE REFACTORING

```
#include <iostream>
#include <string>

int main() {
    char str[100] = "myfile";

    if(strchr(str, '.') == nullptr) {
        strcat(str, ".txt");
    }
}
```

EXAMPLE REFACTORING

```
#include <iostream>
#include <string>

int main() {
    std::string str = "myfile";
    str.reserve(100);
    if(strchr(str, '.') == nullptr) {
       strcat(str, ".txt");
    }
}
```

EXAMPLE REFACTORING

```
#include <iostream>
#include <string>

int main() {
    std::string str = "myfile";
    str.reserve(100);
    if(str.find('.') == std::string::npos) {
        strcat(str, ".txt");
    }
}
```

EXAMPLE REFACTORING

```
#include <iostream>
#include <string>

int main() {
    std::string str = "myfile";
    str.reserve(100);
    if(str.find_first_of('.') == std::string::npos) {
        str.append(".txt");
    }
}
```

Generalize Function into Template

- Functions might be generalized to be used in more context and avoid duplication
- Caveat:
  definition must be visible
- Extract Template…

```
int mymax(int i, int j) {
   return i > j ? i : j;
}
int enter(std::istream& in) {
   int i { };
   in >> i;
   return i;
}
int main() {
   cout << "enter two numbers “
        << endl;
   cout <<  mymax(enter(cin),
                  enter(cin))
      << " was larger\n";
}
```

Generalize Function into Template

```
template<typename T1>
T1 mymax(T1 i, T1 j) {
   return i > j ? i : j;
}
template<typename T1>
T1 enter(std::istream& in) {
   T1 i { };
   in >> i;
   return i;
}
int main() {
   cout << "enter two numbers "<< endl;
   cout << mymax(enter<int>(cin), enter<int>(cin))
        << " was larger\n";
}
```

Compile-time Dependency Injection

- Code might be hard to test, because it uses dependencies to concrete classes

- Extract such a hard-coded type dependency as a template parameter with the original type as its default argument for a new type alias

- Now you can replace the dependency with a different one in a test, for example.

Eliminate concrete type dependency

```
#ifndef GAMEFOURWINS_H_
#include "Die.h"
#include <iostream>
class GameFourWins
{
    Die die;
public:
    void play(std::ostream & out = std::cout);
};
#endif /*GAMEFOURWINS_H_*/
```

Hard-coded type dependency

Easy to replace already polymorphic

First Step:
Toggle Function Definition

```
#ifndef GAMEFOURWINS_H_
#include "Die.h"
#include <iostream>
class GameFourWins
{
    Die die;
public:
   void play(std::ostream& out = std::cout) {
      if (die.roll() == 4) {
         out << "You won!\n";
      } else {
         out << "You lost!\n";
      }
   }
};
#endif /*GAMEFOURWINS_H_*/
```

Select here to extract type Die

Second Step: Extract Template…

```
#ifndef GAMEFOURWINS_H_
#include "Die.h"
#include <iostream>
template<typename DIE> class GameFourWinsT {
   DIE die;

public:
   void play(std::ostream& out = std::cout) {
      if (die.roll() == 4) {
         out << "You won!\n";
      } else {
         out << "You lost!\n";
      }
   }
};
typedef GameFourWinsT<Die> GameFourWins;
#endif /*GAMEFOURWINS_H_*/
```

Adjusted Name of Template Class

Compile-time parameter

Original type name available as alias

What brings the future?

- We have plans and look for students to work on them for much more refactoring and transformation support. ***Sponsors are welcome!***

- Stay tuned for:

    - Template instantiation visualization (like macro expansion)
    - constexpr evaluation in IDE (like with auto's type deduction)
    - Concept support
    - const suggestions (already available in Linticator)
    - loop parallelization, loop to STL algorithm transformation
    - more ideas…

Questions?

- You could apply all these modernization suggestions by hand, or you can download and use Cevelop:

Cevelop
C++
Your C++ code deserves it

Download IDE at:
www.cevelop.com