IIFE in C++

Immediately Invoked Function Expressions

Jason Turner

- http://chaiscript.com
- http://cppbestpractices.com
- http://github.com/lefticus
- http://cppcast.com
- @lefticus
- Independent contractor

ChaiScript

```
int dosomething(int x, int y,
                const std::function<int (int, int)> &f)
{ return f(x*2, 3); }

int main()
{
  using namespace chaiscript;

  ChaiScript chai(chaiscript::Std_Lib::library());
  chai.add(fun(&dosomething), "dosomething");
  auto i = chai.eval<int>("dosomething(4,3, `+`)"); // i = 11
}
```

IIFE (Pronounced 'iffy')

- Common technique in JavaScript to introduce a new variable scope
- Both define and execute an anonymous function in the same expression
- We don't need a new scope for local variables
- But it has interesting code correctness and performance implications
- In C++, should probably be called IILE

```
// Hard to initialize values have always annoyed me…
// (Simplified real code from ChaiScript http://chaiscript.com)

std::string push_back_name;
if (somecase)
{

  push_back_name = "push_back_ref";
} else {

  push_back_name = "push_back";
}


m->add(fun(&ContainerType::push_back), push_back_name);
```

```cpp
// Hard to initialize values can break many C++ best practices


std::string push_back_name; // Uninitialized variable
if (somecase)
{
  // potentially expensive reassignment
  push_back_name = "push_back_ref";
} else {
  // ditto
  push_back_name = "push_back";
}

// push_back_name is copied in
m->add(fun(&ContainerType::push_back), push_back_name);
// push_back_name is left dangling and useless
// We should use C++11! and std::move! right?


std::string push_back_name;
if (somecase)
{

  push_back_name = "push_back_ref";
} else {

  push_back_name = "push_back";
}


m->add(fun(&ContainerType::push_back), std::move(push_back_name));
```

```
// We should use C++11! and std::move! right? maybe not…


std::string push_back_name; // Uninitialized variable
if (somecase)
{
  // potentially expensive reassignment
  push_back_name = "push_back_ref";
} else {
  // ditto
  push_back_name = "push_back";
}


m->add(fun(&ContainerType::push_back), std::move(push_back_name));
// push_back_name is left in an undefined state
// does a crash lurk if we accidentally use it?
```

```cpp
// Maybe we should take a step back and just focus on C++98:
// Make a function


std::string get_name(const bool t_somecase) {
  if (t_somecase)
  {
    return "push_back_ref";
  } else {
    return "push_back";
  }
}



m->add(fun(&ContainerType::push_back), get_name(somecase));
```

```
// Maybe we should take a step back and just focus on C++98:
// Make a function

// Return Value Optimization solves every performance problem
std::string get_name(const bool t_somecase) {
  if (t_somecase)
  {
    return "push_back_ref";
  } else {
    return "push_back";
  }
}

// No local variables solves problems with unused/bad values
// on the stack
m->add(fun(&ContainerType::push_back), get_name(somecase));
```

```
// Maybe we should take a step back and just focus on C++98:
// Make a function
// But now we have this single use function just lying around
// Return value optimization solves every performance problem
std::string get_name(const bool t_somecase) {
  if (t_somecase)
  {
    return "push_back_ref";
  } else {
    return "push_back";
  }
}

// No local variables solves problems with unused/bad values
// on the stack
m->add(fun(&ContainerType::push_back), get_name(somecase));
```

```
// We could try a lambda…

// But now we have this single use lambda just lying around
// Return value optimization solves every performance problem
auto get_name = [](const bool t_somecase) {
  if (t_somecase)
  {
    return "push_back_ref";
  } else {
    return "push_back";
  }
};

// No local variables solves problems with unused/bad values
// on the stack
m->add(fun(&ContainerType::push_back), get_name(somecase));
```

```cpp
// IIFE.

// Somewhat harder to read if you aren’t used to it
// Outperforms / matches every other option
m->add(fun(&ContainerType::push_back), [&]() {
  if (t_somecase)
  {
    return "push_back_ref";
  } else {
    return "push_back";
  }
}());
```

```
 std::vector<std::vector<std::string>> retval;

 for (int i = 0; i < num_vecs; ++i)
 {
  retval.push_back([&](){
   std::vector<std::string> nextvec;
   nextvec.reserve(vec_size);
   for (int j = 0; j < vec_size; ++j)
   {
    nextvec.emplace_back("Some string that's a little bit longer than a short string");
    // plus whatever else needs to happen
   }
   return nextvec;
  }());
 }

 return retval;
```

http://blog2.emptycrate.com/content/complex-object-initialization-optimization-iife-c11
https://gist.github.com/lefticus/04c644db41e0668ca6c4

http://www.reddit.com/r/cpp/comments/2q1p7x

Const Initialization

```
auto size = sizeof(int) * 8;
if (longlong_)  {

  size = sizeof(int64_t) * 8;
} else if (long_) {

  size = sizeof(long) * 8;
}

// size is used read only after this point
```

Const Initialization

```
auto size = sizeof(int) * 8; // size should be const
if (longlong_)  {
  // reassignment
  size = sizeof(int64_t) * 8;
} else if (long_) {
  // reassignment
  size = sizeof(long) * 8;
}

// size is used read only after this point
```

Const Initialization

```
const auto size = [&](){
  if (longlong_) {
    return sizeof(int64_t) * 8;
  } else if (long_) {
    return sizeof(long) * 8;
  } else {
    return sizeof(int) * 8;
  } }();
```

Const Initialization

```
const auto size = [&](){ // size is const
  if (longlong_) {
    return sizeof(int64_t) * 8;
  } else if (long_) {
    return sizeof(long) * 8;
  } else {
    return sizeof(int) * 8;
  } }(); // intent and value of size is more clear
```

Return Type Deduction

```
auto s = [](){
  if (true) {
    return 1;
  } else {
    return 2;
  }
}();
```

Return Type Deduction

```
jason@jason-VirtualBox:~$ g++-4.6 ./testiife5.cpp -std=c++0x -pedantic
./testiife5.cpp: In lambda function:
./testiife5.cpp:5:14: warning: lambda return type can only be deduced
when the return statement is the only statement in the function body
[-pedantic]
```

Return Type Deduction

```
struct MyType
{
  MyType() { std::cout << "MyType()\n"; }
  MyType(const MyType &) { std::cout << "MyType(const MyType &)\n"; }
  ~MyType() { std::cout << "~MyType()\n"; }
  MyType(MyType &&) { std::cout << "MyType(MyType &&)\n"; }
};

int main()
{
  MyType o;
  const auto &v = [&](){
    return o;
  }();
}
```

Return Type Deduction

```
jason@jason-VirtualBox:~$ ./a.out
MyType()
MyType(const MyType &)
~MyType()
~MyType()
```

Return Type Deduction

```
struct MyType
{
  MyType() { std::cout << "MyType()\n"; }
  MyType(const MyType &) { std::cout << "MyType(const MyType &)\n"; }
  ~MyType() { std::cout << "~MyType()\n"; }
  MyType(MyType &&) { std::cout << "MyType(MyType &&)\n"; }
};

int main()
{
  MyType o;
  const auto &v = [&]()->const MyType & {
    return o;
  }();
}
```

Return Type Deduction

```
jason@jason-VirtualBox:~$ ./a.out
MyType()
~MyType()
```

Why Does this Matter?

Object Selection a more advanced ternary?

```
const auto &idname =
  [&]()->const std::string &{
    if (children[0]->identifier == AST_Node_Type::Reference) {
      return children[0]->children[0]->text;
    } else {
      return children[0]->text;
    }
  }();

try {
  return t_ss.add_global_no_throw(Boxed_Value(), idname);
} catch (const exception::reserved_word_error &) {
  throw exception::eval_error("Reserved word error '" + idname + "'");
}
```

Readability

```
Boxed_Value i;
if (match.length() <= sizeof(int) * 8)
{
  i = const_var(static_cast<int>(temp_int));
} else {
  i = const_var(temp_int);
}


m_match_stack.push_back(make_node<eval::Int_AST_Node>(
  std::move(match), prev_line, prev_col, std::move(i)));
```

Readability

```
Boxed_Value i = [&](){
  if (match.length() <= sizeof(int) * 8)
  {
    return const_var(static_cast<int>(temp_int));
  } else {
    return const_var(temp_int);
  }
}();

m_match_stack.push_back(make_node<eval::Int_AST_Node>(
  std::move(match), prev_line, prev_col, std::move(i)));
```

Readability

```
m_match_stack.push_back(make_node<eval::Int_AST_Node>(
 std::move(match), prev_line, prev_col, [&](){
  if (match.length() <= sizeof(int) * 8)
  {
    return const_var(static_cast<int>(temp_int));
  } else {
    return const_var(temp_int);
  }
}()));
```

Readability what about this...?

```
auto i = [&](){
  if (match.length() <= sizeof(int) * 8)
  {
    return const_var(static_cast<int>(temp_int));
  } else {
    return const_var(temp_int);
  }
};

m_match_stack.push_back(make_node<eval::Int_AST_Node>(
  std::move(match), prev_line, prev_col, i()));
```

Readability or this...?

```
m_match_stack.push_back(make_node<eval::Int_AST_Node>(
  std::move(match), prev_line, prev_col,
    (match.length()<=sizeof(int) * 8)?
      const_var(static_cast<int>(temp_int))
      :const_var(temp_int));
```

Cache Misses And Instruction Counts

```cpp
int main(int argc, char *argv[])
{
  auto s = sizeof(long long int);
  if (argc > 2 && argv) {
    s = sizeof(float);
  } else if (argc == 1) {
    s = sizeof(long double);
  }
  return s;
}
```

```cpp
int main(int argc, char *argv[]) {
  const auto s = [&]() {
    if (argc > 2 && argv) {
      return sizeof(float);
    } else if (argc == 1) {
      return sizeof(long double);
    } else {
      return sizeof(long long int);
    }}();
  return s;
}
```

Cache Misses And Instruction Counts

```
int main(int argc, char *argv[])
{
  auto s = sizeof(long long int);
  if (argc > 2 && argv) {
    s = sizeof(float);
  } else if (argc == 1) {
    s = sizeof(long double);
  }
  return s;
}
```

```
11 instructions
2 branches
```

```
int main(int argc, char *argv[]) {
  const auto s = [&]() {
    if (argc > 2 && argv) {
      return sizeof(float);
    } else if (argc == 1) {
      return sizeof(long double);
    } else {
      return sizeof(long long int);
    }}();
  return s;
}
```

```
12 instructions
1 branch
```

When Not to Use It

- When constructing an object that is passed to const &
- When it impedes readability and maintainability
- When code size increases dramatically
- When performance suffers

A Note About Optimization

http://gcc.godbolt.org/

```
int main() {
  int i = 5;
  ++i;
  i++;
  --i;
  i--;
  i = i + 1;
  i = i - 1;
  i += 1;
  i -= 1;
  i = i * 2;
  return i;
}
```

```
int main() {
  return 10;
}
```

A Note About Optimization

http://gcc.godbolt.org/

```
int main() {
  int i = 5;
  ++i;
  i++;
  --i;
  i--;
  i = i + 1;
  i = i - 1;
  i += 1;
  i -= 1;
  i = i * 2;
  return i;
}
```

```
main:
        movl $10, %eax
        ret
```

```
int main() {
  return 10;
}
```

```
main:
        movl $10, %eax
        ret
```

Best Practices Don’t Apply Here

- Effective Modern C++ says to never use automatic capture by reference [&]
- Because the lambda is being “thrown away” this doesn’t apply
- [&] resulted often in smaller, faster code in experiments
- Remember to apply best practices if you factor the lambda out for re-use

What Other Techniques Are There?

C++ is more expressive than ever, what techniques can we borrow from other languages?

What Other Techniques Are There?

```
// Borrow ‘enumerate’ from python
for (const auto &item : enumerate(vector))
{
  item.first; // index
  item.second; // value
}
```

What Other Techniques Are There?

```
// Borrow ‘times’ from ruby
times(5, [](int i) { std::cout << i
                               << ‘\n’; });


# Ruby equiv
5.times(|i| { print(i) })
```

Conclusions

- Never assume you know what the compiler is doing
- Profile your results
- Look for techniques we can borrow from other languages

Questions?

Jason Turner

- http://chaiscript.com
- http://cppbestpractices.com
- http://github.com/lefticus
- http://cppcast.com
- @lefticus