EASING INTO MODERN C++

A LIGHTNING TALK FOR THE UNSURE

(OR THOSE WHO TALK TO THE SKEPTICAL)

BEN DEANE / bdeane@blizzard.com / @ben_deane

C++NOW / MONDAY MAY 6TH, 2018

A COMMON NOTION

"Modern C++ doesn't help me with the kind of programming problems I have."

– A C++ 98/03 Programmer

EASY FEATURES TO ADOPT

Here are 7 (± 2) features of C++11 that you can adopt today.

No downside. All upside. Can be applied piecemeal.

#1: `override`

Any time you declare a `virtual` function override in a derived class, put `override` after it.

```
struct Foo { virtual void Frob(); };

struct Bar : Foo {
  virtual void Frob();
};
```

```
struct Foo { virtual void Frob(); };

struct Bar : Foo {
  virtual void Frob() override;
};
```

If you do nothing else, *do this*.

#2: underlying_type FOR enum

```
enum ChipShopOrder
{
  COD_AND_CHIPS,
  PLAICE_AND_CHIPS,
  MUSHY_PEAS,
  CURRY_SAUCE,
  ...
  INVALID_ORDER = 0xffffffff;
}
```

```
enum ChipShopOrder : uint32_t
{
  COD_AND_CHIPS,
  PLAICE_AND_CHIPS,
  MUSHY_PEAS,
  CURRY_SAUCE,
  ...
  INVALID_ORDER = 0xbadf00d;
}
```

Works for old-style (unscoped) enum as well new scoped enum.

#3: using OVER typedef

typedef: never straightforward, never readable.

```
typedef int AnimId;

typedef int (*FP)(int, int);
```

```
using AnimId = int;

using FP = int (*)(int, int);
// or:
using FP = auto(*) (int, int) -> int;
```

auto (*) means "pointer-to-function" now.

#4: DEFAULT MEMBER INITIALIZATION

```
struct S
{
  S() : value(5) {}
  int value;
};

int foo() {
  S s;
  return s.value;
}
```

https://godbolt.org/g/Wv9wge

```
struct S
{
  // constructor is not needed...
  int value = 5;
};

int foo() {
  S s;
  return s.value;
}
```

https://godbolt.org/g/AGZcwL

#5: `delete` UNIMPLEMENTED SMFS

Can turn a link error into a (more understandable) compile error.

```
class Foo {
private:
  Foo(const Foo&);            // unimplemented
  Foo& operator=(const Foo&); // unimplemented
};
```

```
class Foo {
public:
  Foo(const Foo&) = delete;
  Foo& operator=(const Foo&) = delete;
};
```

#6: constexpr ARRAY SIZE

```
// x had better actually be an array!
#define lengthof(x) \
  (sizeof(x) / sizeof(x[0]));
```

```
template <typename A, std::size_t N>
constexpr std::size_t lengthof(T (&)[N])
{
  return N;
}
```

#7: `explicit` CONVERSION TO `bool`

Ditch the safe bool idiom, use `explicit` conversion to bool.

```
struct Foo
{
  // to prevent unwanted conversion
  // to bool, do a trick
  // e.g. with a "magic" PMF type
};
```

```
struct Foo
{
  explicit operator bool() const {
    // whichever member we want to test
  };
};
```

± 2: static_assert

```
// something homegrown using sizeof trickery
#define STATIC_ASSERT(cond, msg) ...

STATIC_ASSERT(x, "x should hold");
```

```
// nothing: it's in the language now
// #define STATIC_ASSERT ...

static_assert(x, "x should hold");
```

± 2: <chrono>

Use <chrono> for typed time.

- no runtime cost
- expressive
- easy to apply piecemeal
- any questions are probably already answered (by Howard) on SO

Never accidentally pass milliseconds to a function expecting seconds again!

TAKE WHAT YOU WANT

Start using these "no-brainer" recipes.

You don't have to change your whole style or codebase.

They'll just make your life better.

- `override`
- `enum` type
- `using`
- default member init
- `= delete`
- `constexpr` array size
- `explicit bool`
- `static_assert`
- `#include <chrono>`