Bool - Implicitly Dangerous

Jeff Garland

problem: Implicit Conversions Are Terrible

```
1 bool b = "false";
2 int i = b; //msvc warn
```

problem: Implicit Conversions Are Terrible

```
1 bool b = "false";
2 int i = b; //msvc warn
```

```
1 if ( b == i ) {
2   println("eyeroll"); //prints
3 }
4 if ( b ) {
5   println("true"); //prints
6 }
7 else {
8   println("false");
9 }
```

- line one compiles no warnings with -Wall
- https://godbolt.org/z/n9qhM1cjs

problem: Implicit Conversions are Terrible 2

```
1 bool b{"true"};
2 //   warning: implicit conversion turns string literal into bool:
3 //       'const char[6]' to 'bool' [-Wstring-conversion]
4 //  9 |   bool b = "false"; //clang warn -Wconversion
```

name	default	Wall	Wconversion
gcc	none	none	none
clang	error	error	error
msvc	none	warn	na

problem: Is this Really so Bad?

```
1 void do_something( bool ); //cool api
2
3 // in client app file a far away from do_something
4 do_something( "false" );
```

- do you spot this in code review?
- `do_something` api isn't in the review
- compiler didn't warn you
- your unit test seemed to work

solution: Use `enum class`

```
1 enum class my_bool { False, True };
2 // or even better, something more domain specific
3 enum class state_t { On, Off };
4 void do_something( state_t s ); //no conversions allowed
```

- long recommended for api clarity
- abseil tip94: Callsite Readability and bool Parameters

solution: `safe_bool` client code

```
1  bool f( safe_bool sb) { return bool(sb); }
2
3  safe_bool sb4(true);
4  safe_bool sb5; //false
5  safe_bool sb6 = true; //copy-assign
6  safe_bool sb7 = sb6;
7  safe_bool sb8{};
8
9  if ( sb4 == sb6 ) {};
10 if ( sb4 != sb5 ) {};
11 bool b = f({});
12 bool b1 = f(true);
13
14 safe_bool sb9 = sb7 && sb8;
15 safe_bool sb10 = sb7 || sb8;
```

solution: `safe_bool` client code

```
1  bool f( safe_bool sb) { return bool(sb); }
2
3  safe_bool sb4(true);
4  safe_bool sb5; //false
5  safe_bool sb6 = true; //copy-assign
6  safe_bool sb7 = sb6;
7  safe_bool sb8{};
8
9  if ( sb4 == sb6 ) {};
10 if ( sb4 != sb5 ) {};
11 bool b = f({});
12 bool b1 = f(true);
13
14 safe_bool sb9 = sb7 && sb8;
15 safe_bool sb10 = sb7 || sb8;
```

solution: `safe_bool` client code

```
1  bool f( safe_bool sb) { return bool(sb); }
2
3  safe_bool sb4(true);
4  safe_bool sb5; //false
5  safe_bool sb6 = true; //copy-assign
6  safe_bool sb7 = sb6;
7  safe_bool sb8{};
8
9  if ( sb4 == sb6 ) {};
10 if ( sb4 != sb5 ) {};
11 bool b = f({});
12 bool b1 = f(true);
13
14 safe_bool sb9 = sb7 && sb8;
15 safe_bool sb10 = sb7 || sb8;
```

solution: `safe_bool` client code

```
1  bool f( safe_bool sb) { return bool(sb); }
2
3  safe_bool sb4(true);
4  safe_bool sb5; //false
5  safe_bool sb6 = true; //copy-assign
6  safe_bool sb7 = sb6;
7  safe_bool sb8{};
8
9  if ( sb4 == sb6 ) {};
10 if ( sb4 != sb5 ) {};
11 bool b = f({});
12 bool b1 = f(true);
13
14 safe_bool sb9 = sb7 && sb8;
15 safe_bool sb10 = sb7 || sb8;
```

`safe_bool` definition

```
1  class safe_bool
2  {
3      bool value = false;
4   public:
5      constexpr safe_bool() = default;
6      constexpr ~safe_bool() = default;
7      constexpr safe_bool(bool v) noexcept : value(v) {} //not explicit for operator=
8      constexpr safe_bool(const safe_bool&) = default;
9
10     constexpr safe_bool(non_bool_integral auto) = delete ("integral conversion deni
11     constexpr safe_bool(float) = delete("float conversion denied");
12     constexpr safe_bool(char*) = delete("char* constructor denied");
```

- approach: overload constructors and delete them
- with reason is c++26

`safe_bool` definition

```
1  class safe_bool
2  {
3      bool value = false;
4   public:
5      constexpr safe_bool() = default;
6      constexpr ~safe_bool() = default;
7      constexpr safe_bool(bool v) noexcept : value(v) {} //not explicit for operator=
8      constexpr safe_bool(const safe_bool&) = default;
9
10     constexpr safe_bool(non_bool_integral auto) = delete ("integral conversion deni
11     constexpr safe_bool(float) = delete("float conversion denied");
12     constexpr safe_bool(char*) = delete("char* constructor denied");
```

- approach: overload constructors and delete them
- with reason is c++26

`safe_bool` definition

```
1  class safe_bool
2  {
3      bool value = false;
4   public:
5      constexpr safe_bool() = default;
6      constexpr ~safe_bool() = default;
7      constexpr safe_bool(bool v) noexcept : value(v) {} //not explicit for operator=
8      constexpr safe_bool(const safe_bool&) = default;
9
10     constexpr safe_bool(non_bool_integral auto) = delete ("integral conversion deni
11     constexpr safe_bool(float) = delete("float conversion denied");
12     constexpr safe_bool(char*) = delete("char* constructor denied");
```

- approach: overload constructors and delete them
- with reason is c++26

`safe_bool` integral constructor

```
1 template<typename T>
2 concept non_bool_integral = std::integral<T> && !std::same_as<bool, T>;
3
4 class safe_bool
5 {
6   public:
7 //...
8     constexpr safe_bool(non_bool_integral auto) = delete ("integral conversion denie
```

`safe_bool` other operators

```
1  constexpr explicit operator bool() const noexcept { return value; }
2
3  constexpr bool operator==(const safe_bool&) const = default;
4  template<typename B>
5  constexpr bool operator==(B b) const
6    requires std::is_same_v<bool, B>
7  { return b == this->value; }
8
9  constexpr safe_bool operator&&(const safe_bool& other) const noexcept {
10   return safe_bool(this->value && other.value);
11 }
12 constexpr safe_bool operator||(const safe_bool& other) const noexcept {
13   return safe_bool(this->value || other.value);
14 }
```

`safe_bool` other operators

```
1  constexpr explicit operator bool() const noexcept { return value; }
2
3  constexpr bool operator==(const safe_bool&) const = default;
4  template<typename B>
5  constexpr bool operator==(B b) const
6    requires std::is_same_v<bool, B>
7  { return b == this->value; }
8
9  constexpr safe_bool operator&&(const safe_bool& other) const noexcept {
10   return safe_bool(this->value && other.value);
11 }
12 constexpr safe_bool operator||(const safe_bool& other) const noexcept {
13   return safe_bool(this->value || other.value);
14 }
```

`safe_bool` other operators

```
1  constexpr explicit operator bool() const noexcept { return value; }
2
3  constexpr bool operator==(const safe_bool&) const = default;
4  template<typename B>
5  constexpr bool operator==(B b) const
6    requires std::is_same_v<bool, B>
7  { return b == this->value; }
8
9  constexpr safe_bool operator&&(const safe_bool& other) const noexcept {
10   return safe_bool(this->value && other.value);
11 }
12 constexpr safe_bool operator||(const safe_bool& other) const noexcept {
13   return safe_bool(this->value || other.value);
14 }
```

conclusions

- implicit conversions a madness for correct code
- use strong types in interfaces
  - https://github.com/JeffGarland/safe_bool/
  - see also `std::chrono` on thread api
- better option: **real compiler support**
  - P3081 `type_safety` profile?
  - option to make these not compile