Clean Integral Code

Walther Zwart - Meeting C++ 2018

About me

walther.zwart@gmail.com

> 30 years of programming

20 years of C++

3rd time at MeetingCPP

5 years at Optiver

About Optiver

Market Maker since 1986

Amsterdam, Sydney, Chicago, Shanghai

440 people, 44 nationalities in Amsterdam

Low-latency C++ workshop with David Gross during C++ on Sea 2019

About me

About Optiver

Clean Code

"Have nothing in your houses that you do not know to be useful or believe to be beautiful."

William Morris

"It is not enough for code to work."

Robert C. Martin

About me

About Optiver

Clean Code

Integers

"I thought I was a C++ expert, but I can't even remember the rules about integer promotion."

"What is integer promotion?"

About me

About Optiver

Clean Code

Integers

Quiz

1. How many (real) integer types are there in C++17?

1. 4
2. 8
3. 15
4. 45

1. How many (real) integer types are there in C++17?

1. 4
2. 8
3. 15
4. 45

Answer: C

2. Considering

```
std::uint8_t a{255};
++a;
```

What will happen?

1. overflow
2. just math
3. implementation defined behaviour
4. undefined behaviour

2. Considering

```
std::uint8_t a{255};
++a;
```

What will happen?

1. overflow
2. just math
3. implementation defined behaviour
4. undefined behaviour

Answer: B, math: a == 0

The C++ standards specifically says: no overflow.

Just regular, every day, modulo 2^n math

3. Considering

```
std::int8_t a{127};
++a;
```

What will happen?

1. overflow
2. nothing, just regular math
3. implementation defined behaviour
4. undefined behaviour

3. Considering

```
std::int8_t a{127};
++a;
```

What will happen?

1. overflow
2. nothing, just regular math
3. implementation defined behaviour
4. undefined behaviour

Answer: undefined behaviour

4. Considering

```
    std::int8_t a = std::int8_t(127) + std::int8_t(1);
```

What will happen?

1. overflow, a == -128
2. conversion, a == -128
3. implementation defined behaviour
4. undefined behaviour

4. Considering

```
    std::int8_t a = std::int8_t(127) + std::int8_t(1);
```

What will happen?

1. overflow, a == -128
2. conversion, a == -128
3. implementation defined behaviour
4. undefined behaviour

Answer: implementation defined behaviour

5. Considering

```
auto a = 'A' + static_cast<bool>(2);
```

What is the value and type of a?

1. char 'C'
2. char 'B'
3. int 67
4. int 66

5. Considering

```
    auto a = 'A' + static_cast<bool>(2);
```

What is the value and type of a?

1. char 'C'
2. char 'B'
3. int 67
4. int 66

Answer: D, int 66

6. Considering

```
auto a = -10l / 2u;
```

What is the value of a?

1. compile error
2. -5
3. 2'147'483'644
4. it depends

6. Considering

```
auto a = -10l / 2u;
```

What is the value of a?

1. compile error
2. -5
3. 2'147'483'644
4. it depends

Answer: D. it depends on the sizes

About me

About Optiver

Clean Code

Integers

Quiz

"You need to learn the whole language."

Kate Gregory @Meeting C++ 2017

"And then avoid the obscure parts."

me @Meeting C++ 2018

Clean Code

- Clear, single **purpose**
- Clear, minimal set of **states**
- Clear, minimal set of **operations**

Clear means:

- Raises no questions

Clean Code

- Clear, single **purpose**
- Clear, minimal set of **states**
- Clear, minimal set of **operations**

Clear means:

- Raises no questions

```
template<typename IntT>
void PrintInt(IntT i) {
    std::cout << i;
}

PrintInt<std::int8_t>(65);
```

Clean Code

- Clear, single **purpose**
- Clear, minimal set of **states**
- Clear, minimal set of **operations**

Clear means:

- Raises no questions
- Gives no surprises

Refactoring Client

```
class Client
{
public:
    Client();
    void Configure(const Config&);
    void Connect();
    void Disconnect();

    void GetData(std::function<void(Data)>);
};
```

```
void GetDataFromClient(Client& client)
{
    // what is the state of the client?
    client.GetData([](auto&& data) { ... });
}
```

Refactoring Client

```
class Client
{
public:
    Client();
    void Configure(const Config&);
    void Connect();
    void Disconnect();

    void GetData(std::function<void(Data)>);
};
```

```
class Client
{
public:
    explicit Client(const Config&);

    void GetData(std::function<void(Data)>);
};
```

```
void GetDataFromClient(Client& client)
{
    // what is the state of the client?
    client.GetData([](auto&& data) { ... });
}
```

Clean Code

Rules of clean code:

- Single **purpose**
- Minimal set of **states**
- Minimal set of **operations**
- No **questions**
- No **surprises**

Integers

- Types

	signed	**unsigned**
		bool
char wchar_t	signed char	unsigned char
		char16_t char32_t
	short	unsigned short
	int	unsigned (int)
	long (int)	unsigned long (int)
	long long (int)	unsigned long long (int)

Integers

- Types

- Aliases

signed	unsigned
std::int8_t	std::uint8_t
std::int16_t	std::uint16_t
std::int32_t	std::uint32_t
std::int64_t	std::uint64_t
std::int_least8_t	std::uint_least8_t
std::int_least16_t	std::uint_least16_t
std::int_least32_t	std::uint_least32_t
std::int_least64_t	std::uint_least64_t
std::int_fast8_t	std::uint_fast8_t
std::int_fast16_t	std::uint_fast16_t
std::int_fast32_t	std::uint_fast32_t
std::int_fast64_t	std::uint_fast64_t
std::intmax_t	std::uintmax_t
std::intptr_t	std::uintptr_t
std::ptrdiff_t	
	std::size_t

Clean Code

Integers

Use cases

Clean Code

Integers

Use cases

- Bit manipulation

- **Range of values**: 0 or 1 (N times)
- **Best integer type**: unsigned integers
- **Operations**: | & ^ ~ << >>

- **Stronger type**: std::bitset

Refactoring bits

```
// API
namespace GPIO {
    void SetPin(int pin);
    void SetPins(std::uint8_t pins);
}

// user code
{
    auto pins = 7;
    GPIO::SetPin(pins);
}
```

Refactoring bits

```
// API
namespace GPIO {
    void SetPin(int pin);
    void SetPins(std::uint8_t pins);
}

// user code
{
    auto pins = 7;
    GPIO::SetPin(pins);
}
```

```
// API
namespace GPIO {
    void SetPin(int);
    void SetPins(std::bitset<8>);
}

// user code
{
    std::bitset<8> pins{7};
    GPIO::SetPin(pins);
}
```

Refactoring bits

```
// API
namespace GPIO {
    void SetPin(int pin);
    void SetPins(std::uint8_t pins);
}

// user code
{
    auto pins = 7;
    GPIO::SetPin(pins);
}
```

```
// API
namespace GPIO {
    void SetPin(int);
    void SetPins(std::bitset<8>);
}

// user code
{
    std::bitset<8> pins{7};
    GPIO::SetPin(pins);
}
```

```
namespace GPIO {
    void SetPins(std::uint24_t);
}
```

```
namespace GPIO {
    void SetPins(std::bitset<24>);
}
```

Clean Code

Integers

Use cases

- Bit manipulation

- **Range of values**: 0 or 1 (N times)
- **Best integer type**: unsigned integers
- **Operations**: | & ^ << >>
- **Stronger type**: std::bitset
- **Conclusion**: Don't use integers but use std::bitset if you can

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- **Range of values**: false or true

- **Best integer type**: bool

- **Operations**: || && !

Integers

Use cases

- Bit manipulation

- Truth values

- **Range of values**: false or true

- **Best integer type**: bool

- **Operations**: || && !

```
        if (container.count())
```

```
        if (container.count() != 0)
        // or
        if (!container.empty())
```

```
    void conditional_increment(int& count, bool increment)
    {
        count += increment;
        // or
        count += increment ? 1 : 0;
    }
```

Refactoring with booleans

```
class Socket {
public:
    Socket();
    void SetNonBlocking(bool);
    void EnableNagle();
    void SetDestination(std::string, int);
    void Connect();
    ...
};
```

```
Socket mySocket;
mySocket.SetNonBlocking(true);
mySocket.EnableNagle();
mySocket.SetDestination("localhost", 8080);
mySocket.Connect();
```

Refactoring with booleans

```
class Socket {
public:
 Socket(string, int, bool blocking, bool nagle);
 ...
};
```

```
Socket mySocket("localhost", 8080, true, false);
```

Refactoring with booleans

```
class Socket {
public:
 Socket(string, int, bool blocking, bool nagle);
 ...
};
```

```
class Socket {
public:
 Socket(string, bool blocking, int, bool nagle);
 ...
};
```

```
Socket mySocket("localhost", 8080, true, false);
```

Refactoring with booleans

```
struct SocketConfig {
    std::string host;
    int port = 0;
    bool blocking = true;
    bool nagle = false;
};

class Socket {
public:
 explicit Socket(SocketConfig);
 ...
};
```

Refactoring with booleans

```
struct SocketConfig {
    std::string host;
    int port = 0;
    bool blocking = true;
    bool nagle = false;
};

class Socket {
public:
 explicit Socket(SocketConfig);
 ...
};
```

```
Socket mySocket(
    SocketConfig{
        "localhost",
        8080,
        true,
        false});
```

Refactoring with booleans

```
enum class Mode { Blocking, NonBlocking };
enum class Nagle { Enabled, Disabled };

class Socket {
public:
 Socket(std::string, int, Mode, Nagle);
 ...
};
```

```
Socket mySocket(
        "localhost",
        8080,
        Mode::NonBlocking,
        Nagle::Disabled);
```

Refactoring with booleans

```
class Socket {
public:
 Socket(
    std::string,
    int,
    Bool<Blocking>,
    Bool<struct Nagle>);
```

```
Socket mySocket(
        "localhost",
        8080,
        True<Blocking>,
        False<Nagle>);
```

Refactoring with booleans

```
class Socket {
public:
 Socket(
    std::string,
    int,
    Bool<Blocking>,
    Bool<struct Nagle>);
```

```
Socket mySocket(
        "localhost",
        8080,
        True<Blocking>,
        False<Nagle>);
```

```
template<typename tag>
class Bool
{
public:
    constexpr explicit Bool(bool value): value(value) {}
    constexpr Bool(const Bool<tag>&) = default;

    constexpr explicit operator bool() const { return value; }

private:
    bool value;
};

template<typename tag> constexpr Bool<tag> True{true};
template<typename tag> constexpr Bool<tag> False{false};
```

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- **Range of values**: false or true
- **Best integer type**: bool
- **Operations**: || && !
- **Stronger type**: enum class, or strong bool
- **Conclusion**: Don't use booleans for function arguments unless they are the sole argument.

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- **Range of values** (for chars): [0, 255] or [-128, 127]
- **Best integer type**: char

- **Operations**:

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- **Range of values** (for chars): [0, 255] or [-128, 127]
- **Best integer type**: char

- **Operations**:

```
char a = '}' - 5;

char b = 'a' * 2;

char c = '0' + digit;
```

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- **Range of values** (for chars): [0, 255] or [-128, 127]
- **Best integer type**: char
- **Operations**: == !=

- **Stronger type**: std::string

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- Numeric Identifiers

- **Range of values**: depends
- **Best integer type**: anything big enough
- **Operations**: == < ++
- **Stronger type**:

Clean Code

Integers

Use cases

- Bit manipulation
- Truth values
- Text
- Numeric Identifiers

- **Range of values**: depends
- **Best integer type**: anything big enough
- **Operations**: == < ++
- **Stronger type**:

```
// API
void Hire(std::uint64_t company_id, std::uint64_t employee_id);

// user code
Hire(employee.id, company.id);
```

```
// API
void Hire(ID<Company>, ID<Employee>);

// user code
Hire(employee.id, company.id);
```

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- Numeric Identifiers

- **Range of values**: depends
- **Best integer type**: anything big enough
- **Operations**: == < ++
- **Conclusion**: use a strongly typed ID class

Clean Code

Integers

Use cases

- Bit manipulation
- Truth values
- Text
- Numeric Identifiers
- Amounts, indices

- **Natural range of values**: [0, +inf]
- **Natural integer type**: unsigned
- **Operations**: mathemetical operations (no boolean, no bit manipulation)

Difference types

```
auto subtract(const Date&, const Date&);
auto subtract(time_point, time_point);
auto subtract(void*, void*);


auto subtract(unsigned a, unsigned b)
{
    return a - b;
}
```

```
-> int;
-> duration;
-> std::ptrdiff_t;


-> unsigned
```

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- Numeric Identifiers

- Amounts, indices

- **Range of values**: [-inf, +inf]
- **Natural integer type**: int, std::int64_t
- **Operations**: mathematical operations (no boolean, no bit manipulation)

Common arguments for using unsigned:

- I need the positive range.
- Zero as the lowest possible value expresses the real range better.
- Defined behaviour is good.

```
for (std::size_t i = 0; i < container.size()-1; ++i)
    std::cout << i << ": " << container[i] << '\n';
```

- But the containers in the STL use std::size_t!

Signed vs unsigned

They are wrong. And we're sorry.

Herb Sutter & Chandler Carruth @ CPPCon

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- Numeric Identifiers

- Amounts, indices

- **Range of values**: [-inf, +inf]
- **Natural integer type**: int, std::int64_t
- **Operations**: mathematical operations (no boolean, no bit manipulation)

How to deal with this buggy STL?

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- Numeric Identifiers

- Amounts, indices

- **Range of values**: [-inf, +inf]
- **Natural integer type**: int, std::int64_t
- **Operations**: mathematical operations (no boolean, no bit manipulation)

How to deal with this buggy STL?

- Use iterators if you can
- Convert to signed quickly (don't mix signed and unsigned)

Clean Code

Integers

Use cases

- Bit manipulation

- Truth values

- Text

- Numeric Identifiers

- Amounts, indices

- **Range of values**: [-inf, +inf]
- **Natural integer type**: int, std::int64_t
- **Operations**: mathematical operations (no boolean, no bit manipulation)

How to deal with this buggy STL?

- Use iterators if you can
- Convert to signed quickly (don't mix signed and unsigned)

How to prevent unexpected behaviour?

How to prevent unexpected behaviour?

```
auto CalculateMean(const std::vector<std::int32_t>& values) {
    std::int32_t total = 0;
    for (auto v: values)
    {
        total += v;
    }
    return total / values.size();
}
```

How to prevent unexpected behaviour?

```
std::int32_t CalculateMean(const std::vector<std::int32_t>& values) {
    if (values.empty())
        throw std::invalid_argument("CalculateMean received empty vector");

    std::int64_t total = 0;
    for (auto v: values)
    {
        total += v;
    }
    // no warnings with -Wall, -Wpedantic, -Wextra or -Wconversion
    // only with -Wsign-conversion
    return total / values.size();
}
```

How to prevent unexpected behaviour?

```
std::int32_t CalculateMean(const std::vector<std::int32_t>& values) {
    if (values.empty())
        throw std::invalid_argument("CalculateMean received empty vector");

    std::int64_t total = 0;
    for (auto v: values)
    {
        total += v;
    }
    return total / static_cast<std::int64_t>(values.size());
}
```

How to prevent unexpected behaviour?

```
std::int32_t CalculateMean(const std::vector<std::int32_t>& values) {
    if (values.empty())
        throw std::invalid_argument("CalculateMean received empty vector");

    std::int64_t total = 0;
    for (auto v: values)
    {
        total += v;
    }

    if (values.size() > std::numeric_limits<std::int64_t>::max())
        throw std::runtime_error("CalculateMean received too many values");
    return total / static_cast<std::int64_t>(values.size());
}
```

How to prevent unexpected behaviour?

```
std::int32_t CalculateMean(const std::vector<std::int32_t>& values) {
    if (values.empty())
        throw std::invalid_argument("CalculateMean received empty vector");

    std::int64_t total = 0;
    for (auto v: values)
    {
        constexpr auto max = std::numeric_limits<std::int64_t>::max();
        if (total + v > max)
            throw std::runtime_error("CalculateMean overflowed");
        total += v;
    }
    return total / static_cast<std::int64_t>(values.size());
}
```

How to prevent unexpected behaviour?

```
std::int32_t CalculateMean(const std::vector<std::int32_t>& values) {
    if (values.empty())
        throw std::invalid_argument("CalculateMean received empty vector");

    std::int64_t total = 0;
    for (auto v: values)
    {
        constexpr auto max = std::numeric_limits<std::int64_t>::max();
        constexpr auto min = std::numeric_limits<std::int64_t>::min();
        if (v > 0 && total > max - v)
            throw std::runtime_error("CalculateMean overflowed");
        if (v < 0 && total < min - v)
            throw std::runtime_error("CalculateMean underflowed");
        total += v;
    }
    return total / static_cast<std::int64_t>(values.size());
}
```

- **Range of values**: [-inf, +inf]
- **Natural integer type**: int, std::int64_t
- **Operations**: mathematical operations (no boolean, no bit manipulation)

How to deal with this buggy STL?

- Use iterators if you can
- Convert to signed quickly (don't mix signed and unsigned)

How to prevent unexpected behaviour?

- Manual checking is hard
- Use compiler warnings and static analyzers
- Use a library solution
  - boost multiprecision
  - boost numeric conversion
  - boost safe numerics

The last slide

Raw integers do not limit **states** and **operations** effectively

Use the **strongest type** available

Strive for **no questions**, **no surprises**

Questions?

WE ARE HIRING!

VISIT OPTIVER.COM/JOB-OPPORTUNITIES

FOR MORE INFORMATION