MAT 685: C++ for Mathematicians

A modular class

John Perry

University of Southern Mississippi

Spring 2017

Modular arithmetic

Observations from the code...

...on the language

...on the mathematics

Summary

Outline

1. Modular arithmetic
2. Observations from the code...
   - ...on the language
   - ...on the mathematics
3. Summary

Modular arithmetic

Observations from the code...

...on the language

...on the mathematics

Summary

Outline

1 Modular arithmetic

2 Observations from the code...
...on the language
...on the mathematics

3 Summary

From Modern Algebra or Number Theory

$\mathbb{Z}_n = \{0, 1, \ldots, m-1\}$

- add, subtract, multiply: take remainder after dividing by m

From Modern Algebra or Number Theory

$\mathbb{Z}_n = \{0, 1, \ldots, m-1\}$

- add, subtract, multiply: take remainder after dividing by m
- has properties of a **ring**
  - addition, subtraction work "as desired" (closed, commutative, associative, identity, inverses)
  - multiplication works "almost as desired" (closed, commutative, associative, identity, *not always inverses*)

From Modern Algebra or Number Theory

$\mathbb{Z}_n = \{0, 1, \ldots, m-1\}$

- add, subtract, multiply: take remainder after dividing by m
- has properties of a **ring**
  - addition, subtraction work "as desired" (closed, commutative, associative, identity, inverses)
  - multiplication works "almost as desired" (closed, commutative, associative, identity, *not always inverses*)
- if m prime, has properties of a **field**
  - addition, subtraction same
  - multiplication "as desired": inverses, too!

Very useful

- anything cyclical
- "clockwork" arithmetic: 3 hours after 10 is 1 ($10 + 3 = 1$)
- coding theory
- cryptography
- computer science

Let's implement as a class

- Constructors
  - include default constructor
    - default value obviously 0
    - default modulus? *not* 0

Let's implement as a class

- Constructors
  - include default constructor
    - default value obviously 0
    - default modulus? *not* 0
- Getters
- Setters
- I/O
- Arithmetic operators
- Comparison operators

Default modulus?

Whose choice?

Default modulus?

Whose choice?

- ours?
  - need *something* to setup

Default modulus?

Whose choice?

- ours?
  - need *something* to setup
- user's?
  - better idea of needs
  - annoying & wasteful to change from *our* default modulus for every new object

Default modulus?

Whose choice?

- ours?
  - need *something* to setup
- user's?
  - better idea of needs
  - annoying & wasteful to change from *our* default modulus for every new object
- why not both?

`constexpr` and `static`

`constexpr` expression can be evaluated to a constant at compile time (C++11 only)

- all involved expressions must be `constexpr` too
- appropriate choice for *our* default modulus
- observe different meaning from `const`, as book uses

`constexpr` and `static`

`constexpr` expression can be evaluated to a constant at compile time (C++11 only)

- all involved expressions must be `constexpr` too
- appropriate choice for *our* default modulus
- observe different meaning from `const`, as book uses

`static` field w/same value in *all* objects of class

- changing it in one object changes it in all
- can change w/out object by scoping: `Class::static_field = new_value`
- appropriate choice for *user's* default modulus

Possible problem

What if user selects invalid modulus?

Possible problem

What if user selects invalid modulus?

Throw exception

exception special class for "bad data"

- `#include <stdexcept>`
- see available exceptions
- can create your own, too!

- terminates program immediately
- can provide useful information
- usage: **`throw`** `exception_type(``string description'');`

Notes on throwing exceptions

1. Do it rarely...
   - can slow down code
   - better to write good code from the get-go
2. ...but do it when necessary
   - undefined behavior hard to debug
   - client can `try` and `catch()` exceptions, allowing recovery

Interface

Ah, just look at the source code.

Outline

1 Modular arithmetic

2 Observations from the code...
...on the language
...on the mathematics

3 Summary

Modular arithmetic

Observations from the code...

...on the language

...on the mathematics

Summary

Outline

1. Modular arithmetic
2. Observations from the code...
   - ...on the language
   - ...on the mathematics
3. Summary

Defined methods in class

Not typically done (or advisable)

- necessary for templates, inline code (see below)
- sometimes inadvisable: linker complaints?
- no need for class scope (`Mod::`) if in class

Can also define outside *class* but in *interface file*

- need full signature (class scope)

```
inline Mod Mod::pow (int64_t k) const {
  // a miracle occurs here
}
```

Inlining code

Function calls expensive

- involve small, but nonzero time penalty
- pushing, popping stack

Inlining code

Function calls expensive

- involve small, but nonzero time penalty
- pushing, popping stack

"Inline" functions inserted directly

- no *real* function call takes place
- time penalty eliminated, but...
- ...space penalty possible
- okay with very short functions

Inlining code

Function calls expensive

- involve small, but nonzero time penalty
- pushing, popping stack

"Inline" functions inserted directly

- no *real* function call takes place
- time penalty eliminated, but...
- ...space penalty possible
- okay with very short functions

`inline` keyword *requests*, but does not *guarantee* inlining

- compiler free to ignore
- compiler can inline even when not requested
- many counsel against using this keyword, but...
- ...I've seen it make a difference

Static fields

```
static int64_t default_modulus;
```

- Universal to class: accessing `default_modulus` anywhere changes for all instances

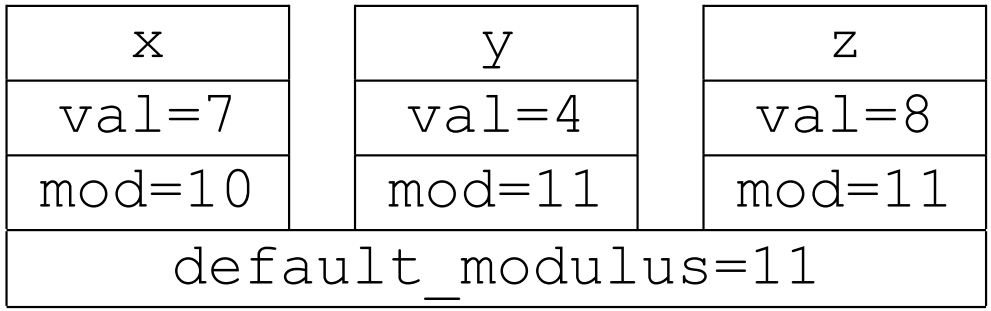
x	y	z
val=7	val=4	val=8
mod=10	mod=11	mod=11
default_modulus=11		

- reading/modifying should *not* be bound to variable, but...
- ...field is protected member of class

Static fields

```
static int64_t default_modulus;
```

- Universal to class: accessing `default_modulus` anywhere changes for all instances



- reading/modifying should *not* be bound to variable, but...
- ...field is protected member of class

solution: `static` functions

```
static int64_t Mod::get_default_modulus();
static void Mod::set_default_modulus(int64_t);
```

Initializing static fields

May not initialize non-constant static variable *in the class:*

```
static const int64_t default_modulus
    = INITIAL_DEFAULT_MODULUS; // cool!
static int64_t default_modulus
    = INITIAL_DEFAULT_MODULUS; // error!
```

`default_modulus` not constant, so this is out.

Initializing static fields

May not initialize non-constant static variable *in the class:*

```
static const int64_t default_modulus
    = INITIAL_DEFAULT_MODULUS; // cool!
static int64_t default_modulus
    = INITIAL_DEFAULT_MODULUS; // error!
```

`default_modulus` not constant, so this is out.

May initialize *anywhere outside the class:*

```
class Mod {
public:
  static void set_default_modulus(int64_t m) {
    // a miracle occurs here
  }
};

Mod::set_default_modulus(INITIAL_DEFAULT_MODULUS);
```

Constructors and fields

```
Mod() : mod(default_modulus), val(0) { }

Mod(int64_t x)
    : mod(default_modulus), val(x) { adjust_val(); }

Mod(int64_t x, int64_t m) : val(x), mod(m) {
  check_mod(); adjust_val();
}

Mod(const Mod & other) : val(other.val), mod(other.mod)
{ }
```

Constructors and fields

```
Mod() : mod(default_modulus), val(0) { }

Mod(int64_t x)
    : mod(default_modulus), val(x) { adjust_val(); }

Mod(int64_t x, int64_t m) : val(x), mod(m) {
  check_mod(); adjust_val();
}

Mod(const Mod & other) : val(other.val), mod(other.mod)
{ }
```

- In each case, `mod` initialized before code is run.
- In last three cases, `val` also initialized before code is run.
- In each case, *we construct before we compute.*

Constructors and fields

```
Mod() : mod(default_modulus), val(0) { }

Mod(int64_t x)
    : mod(default_modulus), val(x) { adjust_val(); }

Mod(int64_t x, int64_t m) : val(x), mod(m) {
  check_mod(); adjust_val();
}

Mod(const Mod & other) : val(other.val), mod(other.mod)
{ }
```

- In each case, `mod` initialized before code is run.
- In last three cases, `val` also initialized before code is run.
- In each case, *we construct before we compute.*
- Sometimes this is *required* (`const` fields).
- It is typically *more efficient*.

Constructors and fields

```
Mod() : mod(default_modulus), val(0) { }

Mod(int64_t x)
    : mod(default_modulus), val(x) { adjust_val(); }

Mod(int64_t x, int64_t m) : val(x), mod(m) {
  check_mod(); adjust_val();
}

Mod(const Mod & other) : val(other.val), mod(other.mod)
{ }
```

- In each case, `mod` initialized before code is run.
- In last three cases, `val` also initialized before code is run.
- In each case, *we construct before we compute.*
- Sometimes this is *required* (`const` fields).
- It is typically *more efficient*.
- So: *construct before computing*

Assignment is sometimes construction

Test code has

```
z = -3;
```

Assignment is sometimes construction

Test code has

```
z = -3;
```

Interface does not define

```
Mod::operator = (int64_t x);
```

Assignment is sometimes construction

Test code has

```
z = -3;
```

Interface does not define

```
Mod::operator = (int64_t x);
```

It *could* define it, but if not, corresponding constructor is called:

```
Mod::Mod(int64_t x);
```

Arithmetic operators

C++ assumes nothing: defining

```
Mod operator + (
    const Mod & m, const int64_t other
);
```

does not likewise define

```
Mod operator + (
    const int64_t other, const Mod & m
);
```

If you want both, you must define both.

Arithmetic operators

C++ assumes nothing: defining

```
Mod operator + (
    const Mod & m, const int64_t other
);
```

does not likewise define

```
Mod operator + (
    const int64_t other, const Mod & m
);
```

If you want both, you must define both.
In our case,

```
Mod Mod::operator + (const int64_t other);
```

does the same as the first.

Arithmetic operator's return value

Standard practice is to return result of *any* operation, ***including* increment, decrement, scaling**

```
Mod & Mod::operator ++ () {
  ++val; adjust_val();
  return Mod(val, mod);
}
```

Arithmetic operator's return value

Standard practice is to return result of *any* operation, ***including* increment, decrement, scaling**

```
Mod & Mod::operator ++ () {
  ++val; adjust_val();
  return Mod(val, mod);
}
```

Allows us to chain operations, an old C++ trick

```
cout << "++x = " << ++x << endl;
```

(Many languages do not allow this.)

Should we really check correct modulus?

In arithmetic operations we check `mod` & `throw()` if different, *but...*

- checking `mod` slows arithmetic
- arguably client's job, not ours

Pre- and post-inc/decrement

Pre-increment

- "increment before returning value"
- `symbol` before variable
- return new value
- signature: `Type Type::operator ++ () ;`

Pre- and post-inc/decrement

Pre-increment

- "increment before returning value"
- `symbol` before variable
- return new value
- signature: `Type Type::operator ++ () ;`

Post-increment

- "increment after returning value"
- `symbol` after variable
- return old value
- signature: `Type Type::operator ++ (int) ;`
  - argument is "dummy" variable
  - used *only* to distinguish from pre-increment

Pre- and post-inc/decrement

Pre-decrement

- "decrement before returning value"
- `symbol` before variable
- return new value
- signature: `Type Type::operator -- () ;`

Post-decrement

- "decrement after returning value"
- `symbol` after variable
- return old value
- signature: `Type Type::operator -- (int) ;`
  - argument is "dummy" variable
  - used *only* to distinguish from pre-increment

Outline

1. Modular arithmetic
2. Observations from the code...
   - ...on the language
   - ...on the mathematics
3. Summary

Finding inverses

Recall Bézout's identity:

$$ax + by = \gcd(a, b)$$

Finding inverses

Recall Bézout's identity:

$$ax + by = \gcd(a, b)$$

Modulo b, this reduces:

$$ax + 0 \equiv \gcd(a, b)$$

Finding inverses

Recall Bézout's identity:

$$ax + by = \gcd(a, b)$$

Modulo b, this reduces:

$$ax + 0 \equiv \gcd(a, b)$$

If $\gcd(a, b) = 1$, then

$$ax \equiv 1 \pmod{b}$$

Finding inverses

Recall Bézout's identity:

$$ax + by = \gcd(a, b)$$

Modulo b, this reduces:

$$ax + 0 \equiv \gcd(a, b)$$

If $\gcd(a, b) = 1$, then

$$ax \equiv 1 \pmod{b}$$

This explains `inverse()`

Finding powers

Recursive algorithm for a^k

Finding powers

Recursive algorithm for a^k

- if $k = 2\ell$,

$$a^k = a^{2\ell} = \left(a^\ell\right)^2$$

so just find a^ℓ, multiply to self

Finding powers

Recursive algorithm for a^k

- if $k = 2\ell$,

$$a^k = a^{2\ell} = \left(a^\ell\right)^2$$

  so just find a^ℓ, multiply to self
- if $k = 2\ell + 1$,

$$a^k = a^{2\ell+1} = a \times \left(a^\ell\right)^2$$

  so find a^ℓ, multiply to self, then to a

Modular arithmetic

Observations from the code...

...on the language

...on the mathematics

Summary

Outline

1 Modular arithmetic

2 Observations from the code...
...on the language
...on the mathematics

3 Summary

Homework

pp. 174–175 #9.1, 9.5

Homework

pp. 174–175 #9.1, 9.5

Hint on 9.1: From Bézout, find $c, d \in \mathbb{Z}$ such that

$$cm + dn = 1 \, .$$

Homework

pp. 174–175 #9.1, 9.5

Hint on 9.1: From Bézout, find $c, d \in \mathbb{Z}$ such that

$$cm + dn = 1 \, .$$

Notice this simplifies to

$$0 + dn \equiv 1 \pmod{m} \quad \text{and} \quad cm + 0 \equiv 1 \pmod{n} \, .$$

Homework

pp. 174–175 #9.1, 9.5

Hint on 9.1: From Bézout, find $c, d \in \mathbb{Z}$ such that

$$cm + dn = 1 \, .$$

Notice this simplifies to

$$0 + dn \equiv 1 \pmod{m} \quad \text{and} \quad cm + 0 \equiv 1 \pmod{n} \, .$$

Let $x = adn + bcm$; then

$$\begin{aligned} x &\equiv adn + bcm \equiv a\,(dn) + 0 \equiv a \times 1 \equiv a \pmod{m} \\ y &\equiv adn + bcm \equiv 0 + b\,(cm) \equiv b \times 1 \equiv b \pmod{n} \, . \end{aligned}$$

So x is our solution, and the solution is unique modulo mn.

Summary

- Math stuff
  - additional use for Bézout
  - fast exponentiation
- Programming stuff
  - assignment can be construction
  - `constexpr`
  - construct before computing
  - inlining code
  - returning from arithmetic operations
  - `static` (variables *and* methods)
  - throwing exceptions