Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

MAT 685: C++ for Mathematicians

Rings and fields of inheritance

John Perry

University of Southern Mississippi

Spring 2017

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Rings

Definition

A **ring** R is a set with two closed operations $+$, $\times$ such that

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Rings

Definition

A **ring** R is a set with two closed operations $+$, $\times$ such that

$+$ is

- commutative and associative
- has an identity 0
- has inverses for all $r \in R$

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Rings

Definition

A **ring** R is a set with two closed operations $+$, $\times$ such that

$+$ is

- commutative and associative
- has an identity 0
- has inverses for all $r \in R$

$\times$ is

- associative
- has an identity 1

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Rings

Definition

A **ring** R is a set with two closed operations $+$, $\times$ such that

$+$ is

- commutative and associative
- has an identity 0
- has inverses for all $r \in R$

$\times$ is

- associative
- has an identity 1
- commtutative? R a **commutative** ring

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Examples

Basic rings

- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- $\mathbb{Q} = \{a/b : a, b \in \mathbb{Z}, b \neq 0\}$

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Examples

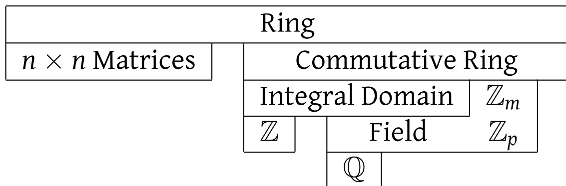
Basic rings

- $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$
- $\mathbb{Q} = \{a/b : a, b \in \mathbb{Z}, b \neq 0\}$

Let R be a ring

- $R^{m \times n}$, set of $m \times n$ matrices over ring R
- $R[x]$, univariate polynomial ring over R
- $R[x, y, z]$, multivariate polynomial ring over R

Types of rings



Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Relationships

We want to use `Ring_Element` class with:

Does it have a Ring_Element, or is it a `Ring_Element`?

Relationships

We want to use `Ring_Element` class with:

- Equation of elements in a ring

Does it have a Ring_Element, or is it a `Ring_Element`?

Relationships

We want to use `Ring_Element` class with:

- Equation of elements in a ring

Does it have a Ring_Element, or is it a `Ring_Element`?

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Relationships

We want to use `Ring_Element` class with:

- Equation of elements in a ring
- Modular ring

Does it have a Ring_Element, or is it a `Ring_Element`?

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Relationships

We want to use `Ring_Element` class with:

- Equation of elements in a ring
- Modular ring

Does it have a Ring_Element, or is it a `Ring_Element`?

MAT 685: C++ for Mathematicians

John Perry

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

“is-a”

B “is an” A

- everything A can do, B can do (better?)
- B’s behavior consistent with A’s
- B *specializes* A in some way

“is-a”

B “is an” *A*

- everything *A* can do, *B* can do (better?)
- *B*’s behavior consistent with *A*’s
- *B specializes A* in some way

In this case,

- *B*’s type can **inherit** *A*’s type, *and*
- *B* can be passed anywhere *A*’s type is placed, *but*
- *A* cannot be placed where *B*’s type is expected

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How to implement?

- many common properties, so
- common interface(s) appealing
  - common code repeatable
  - univariate polynomial rings all kind of behave the same, really

How to implement?

- many common properties, so
- common interface(s) appealing
  - common code repeatable
  - univariate polynomial rings all kind of behave the same, really

but

- most use specialized different data structure
- some methods apply to all, some particular

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Composition v. Inheritance

```
class A {
protected:
  int value;
};
```

B “has an” A

B “is an” A

Composition v. Inheritance

```
class A {
protected:
  int value;
};
```

B "has an" *A*

B "is an" *A*

A should *appear as a field* in *B*

```
class B {
protected: A a;
};

B b;
```

`b.a.value` sensible, not `b.value` (`b` not an `A`)

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Composition v. Inheritance

```
class A {
protected:
  int value;
};
```

B "has an" *A*

A should *appear as a field* in *B*

```
class B {
protected: A a;
};

B b;
```

`b.a.value` sensible, not `b.value` (`b` not an `A`)

B "is an" *A*

B should *inherit* from *A*

```
class B : public A {
}

B b;
```

`b.value` makes sense b/c `b` *is an* `A`

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How to inherit

```
class Parent {
protected:
  int x;
public:
  Parent(int);
  ...
}

class Child : public Parent {
protected:
  int y;
public:
  Child(int, int);
  ...
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Kinds of inheritance

public Access in `Parent` preserved in `Child`

protected Public access in `Parent` is protected in `Child`

private Everything in `Parent` is private in `Child` (default)

Kinds of inheritance

public	Access in `Parent` preserved in `Child`
protected	Public access in `Parent` is protected in `Child`
private	Everything in `Parent` is private in `Child` (default)

This is why I counsel `protected` fields, rather than `private`

- children should have access to their own data!

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Kinds of inheritance

public	Access in `Parent` preserved in `Child`
protected	Public access in `Parent` is protected in `Child`
private	Everything in `Parent` is private in `Child` (default)

This is why I counsel `protected` fields, rather than `private`

- children should have access to their own data!

Aim for `public` inheritance, not `protected` or `private`

- `public` enables code reuse
- `protected`, `private` hide parent's public interface

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Kinds of inheritance

public	Access in `Parent` preserved in `Child`
protected	Public access in `Parent` is protected in `Child`
private	Everything in `Parent` is private in `Child` (default)

This is why I counsel `protected` fields, rather than `private`

- children should have access to their own data!

Aim for `public` inheritance, not `protected` or `private`

- `public` enables code reuse
- `protected`, `private` hide parent's public interface
  - C++ FAQ says private inheritance needed very rarely
  - ***so why is it the default?!?***

Constructing children

Children often need to call parent's constructors

- *definitely* if parent has no default constructor

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Constructing children

Children often need to call parent's constructors

- *definitely* if parent has no default constructor

```
Child::Child(int x0, int y0)
    : Parent(x0), y(y0)
{ }
```

- `Parent`'s constructor called with `x0`, then...
- `y` initialized to `y0`

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Redefining functions

Children can redefine a function, *under the following conditions*

- function declared `virtual` in `Parent`
- `Child` must re-declare in interface and add `override`

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Redefining functions

Children can redefine a function, *under the following conditions*

- function declared `virtual` in `Parent`
- `Child` must re-declare in interface and add `override`

```
class Parent {
...
public:
  void do_nothing() { }
  virtual void print() { cout << x; }
}

class Child : public Parent {
...
public:
  void do_nothing() { // no error, but no override
    cout << "er...";
  }
  virtual void print() override { cout << y; } // OK
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How to use it

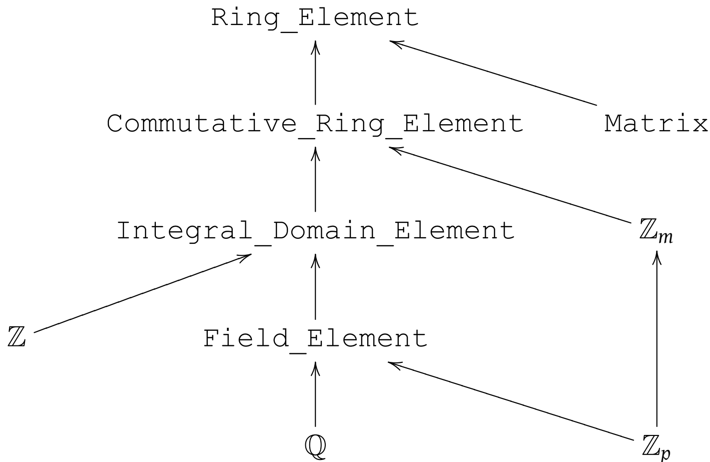
```
Parent P(1);
Child  C(2, 3);
Parent & PC = C;
Parent P = C;

P.do_nothing();  // no output!
PC.do_nothing(); // no output!
C.do_nothing();  // not even an "er..."!

P.print();  // 1
PC.print(); // 2
C.print();  // 3
```

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Class hierarchy

A *simple* class hierarchy.



Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Multiple inheritance

Some matrices may need to inherit from two interfaces

Example

$\mathbb{Z}_p$ inherits from $\mathbb{Z}_m$ and `Field_Element`

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Multiple inheritance

Some matrices may need to inherit from two interfaces

Example

$\mathbb{Z}_p$ inherits from $\mathbb{Z}_m$ and `Field_Element`

Can define by repeating listing in class declaration:

```
class Modp : public Mod, public Field_Element { ... };
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Multiple inheritance

Some matrices may need to inherit from two interfaces

Example

$\mathbb{Z}_p$ inherits from $\mathbb{Z}_m$ and `Field_Element`

Can define by repeating listing in class declaration:

```
class Modp : public Mod, public Field_Element { ... };
```

This can lead to name clashes. Solve with "virtual inheritance" or, ultimately, redefinition:

```
class Modp
    : virtual public Mod, virtual public Field_Element
{
  using Mod::operator +;

  virtual bool has_inverse() { return true; }
};
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

`Ring` lacks information?

`Parent` sometimes doesn't have enough information to do something, but children should

Addition of elements

`Ring_Element` would not know how to add integers, but `Integer` would

Abstract type / "Pure" virtual method

Define such functions "pure virtual"

- cannot instantiate `Parent` b/c it's "abstract"
- cannot instantiate `Child` *unless* it overrides

```
class Ring {
...
  virtual Ring & operator +(const Ring & other) = 0;
...
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Interface examples

`Ring_Element`

- add, subtract, multiply
- is it additive, multiplicative identity?
- is it commutative?
- is it cancellable?
- does it have a multiplciative inverse?
- print to an `ostream`

Interface examples

`Ring_Element`

- add, subtract, multiply
- is it additive, multiplicative identity?
- is it commutative?
- is it cancellable?
- does it have a multiplciative inverse?
- print to an `ostream`

`Mod`

- everything `Ring_Element` can do
- report value

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Interface examples

`Ring_Element`

- add, subtract, multiply
- is it additive, multiplicative identity?
- is it commutative?
- is it cancellable?
- does it have a multiplciative inverse?
- print to an `ostream`

`Mod`

- everything `Ring_Element` can do
- report value

`Rational`

- everything `Field_Element` can do
- report numerator, denominator

Outline

1. Rings
2. `Ring` and its children
3. **Implementing w/templates**
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Important considerations

What size values for `Integer`'s value and `Rational`'s numerator and denominator?

- we might prefer `int64_t`
- client might prefer `short`

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Important considerations

What size values for `Integer`'s value and `Rational`'s numerator and denominator?

- we might prefer `int64_t`
- client might prefer `short`

We don't have to decide this now; we can leave it to the client.

Parametrize the type

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Templates “fill in the blank”

```
template <typename    T    >
   T     gcd(    T     a,     T     b)
{
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  if ( (a == 0) and (b == 0) ) {
    cerr << "WARNING: gcd called with two
zeros.\n";
    return 0;
  }
  if (b == 0) return a;
  if (a == 0) return b;

      T     c = a % b; return gcd(b, c);
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templates “fill in the blank”

```
template <typename int64_t>
int64_t gcd(int64_t a, int64_t b)
{
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  if ( (a == 0) and (b == 0) ) {
    cerr << "WARNING: gcd called with two
zeros.\n";
    return 0;
  }
  if (b == 0) return a;
  if (a == 0) return b;

  int64_t c = a % b; return gcd(b, c);
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templates “fill in the blank”

```
template <typename    Mod    >
  Mod    gcd(  Mod    a,   Mod    b)
{
  if (a < 0) a = -a;
  if (b < 0) b = -b;

  if ( (a == 0) and (b == 0) ) {
    cerr << "WARNING: gcd called with two
zeros.\n";
    return 0;
  }
  if (b == 0) return a;
  if (a == 0) return b;

    Mod    c = a % b; return gcd(b, c);
}
```

Rings

`Ring` and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can we parametrize *any* type?

Only if it can perform everything needed by `T`.

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can we parametrize *any* type?

Only if it can perform everything needed by `T`.

Example

`ostream gcd(ostream, ostream)` won't work, b/c ostream does not have `%`, $<$, negation operators

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can we parametrize *any* type?

Only if it can perform everything needed by `T`.

Example

`ostream gcd(ostream, ostream)` won't work, b/c ostream does not have `%`, <, negation operators

Compiler catches these errors

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templates need to be instantiated

Code using template needs to indicate intent:

```
template gcd<int64_t>(int64_t, int64_t);
```

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Templates need to be instantiated

Code using template needs to indicate intent:

```
template gcd<double >(double , double );
```

This illustrates creation, instantiation of *function* template.
Classes are a little different.

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templates need to be instantiated

Code using template needs to indicate intent:

```
template gcd< Mod >( Mod , Mod );
```

This illustrates creation, instantiation of *function* template.

Classes are a little different.

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templating a class

Simpler definition:

```
template <typename T>
class Integer {
   ...
}
```

(Could also use `length` in place of T)

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templating a class

Simpler definition:

```
template <typename T>
class Integer {
   ...
}
```

(Could also use `length` in place of T)

Simpler instantiation:

```
Integer<short> i;
Integer<uint64_t> j;
```

(No need for **`template <>`** in this case)

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Interface and Implementation

Ah, just click on the link.

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Organization

- Several files
  - `rings.hpp`: Ring_Element, Commutative_Ring_Element, Integral_Domain_Element, Field_Element
  - `integer.hpp`: Integer
  - `rational.hpp`: Rational
  - `mod.hpp`: Integers modulo an integer m
  - `modp.hpp`: Integers modulo a prime p
- One namespace: `Rings`

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

“Namespace”?

More or less what name suggests:

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

"Namespace"?

More or less what name suggests:

- defines a *namespace scope*
- allows distinction of objects with same "short name"
- *element* in `Rings` namespace accessible as `Rings::`*element*

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

“Namespace”?

More or less what name suggests:

- defines a *namespace scope*
- allows distinction of objects with same “short name”
- *element* in `Rings` namespace accessible as `Rings::`*element*
- immediate access: **`using`** `Rings::`*element*`;`
  - already seen as **`using`** `std::cout;`

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

In our case

Creation:

Listing 1: in *.hpp

```
namespace Rings {
  ...
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

In our case

Creation:

Listing 2: in *.hpp

```
namespace Rings {
  ...
}
```

Usage:

```
#include "rings.hpp"

Rings::Ring_Element r;
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

In our case

Creation:

Listing 3: in *.hpp

```
namespace Rings {
  ...
}
```

Usage:

```
#include "rings.hpp"

Rings::Ring_Element r;
```

or:

```
#include "rings.hpp"
using Rings::Ring_Element;

Ring_Element r;
```

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
    - Organization
    - Templating
    - Inheritance
    - Abstract return types
    - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Templating

We templated... a *lot.*

```
// templated only according to type
template <typename T> Integer<T>;
template <typename T> Rational<T>;

// templated according to both type and value
template <typename T, unsigned m> Mod<T,m>;
template <typename T, unsigned p> Modp<T,p>;
```

Templating

We templated... a *lot.*

```
// templated only according to type
template <typename T> Integer<T>;
template <typename T> Rational<T>;

// templated according to both type and value
template <typename T, unsigned m> Mod<T,m>;
template <typename T, unsigned p> Modp<T,p>;
```

Allows:

- size of underlying element to vary
- modulus built-in to type (so it's constant)

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Using a templated class

Suffices merely to “fill in the blanks”

```
Integer<long> i;
Modp<short, 16>;
```

Warning

Use of non-prime modulus will raise an exception.

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C++14 (!!!) feature: templated variables

Before C++14, you could template only functions and classes. Now you can template variables, too!

```
template <typename T, unsigned p> Modp<T,p>
initialize_inverses;
```

MAT 685: C++ for Mathematicians

John Perry

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C++14 (!!!) feature: templated variables

Before C++14, you could template only functions and classes. Now you can template variables, too!

```
template <typename T, unsigned p> Modp<T,p>
initialize_inverses;
```

Why?

- forces construction of this variable
- default constructor tries to create all inverses for this type

MAT 685: C++ for Mathematicians

John Perry

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C++14 (!!!) feature: templated variables

Before C++14, you could template only functions and classes. Now you can template variables, too!

```
template <typename T, unsigned p> Modp<T,p>
initialize_inverses;
```

Why?

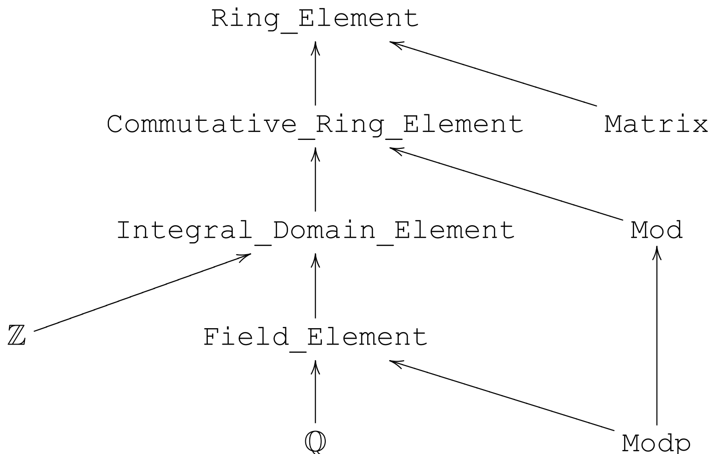
- forces construction of this variable
- default constructor tries to create all inverses for this type
- can't find one?
  - p is not prime
  - throws exception

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Class hierarchy

`Matrix` unimplemented, but otherwise as desired:



Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

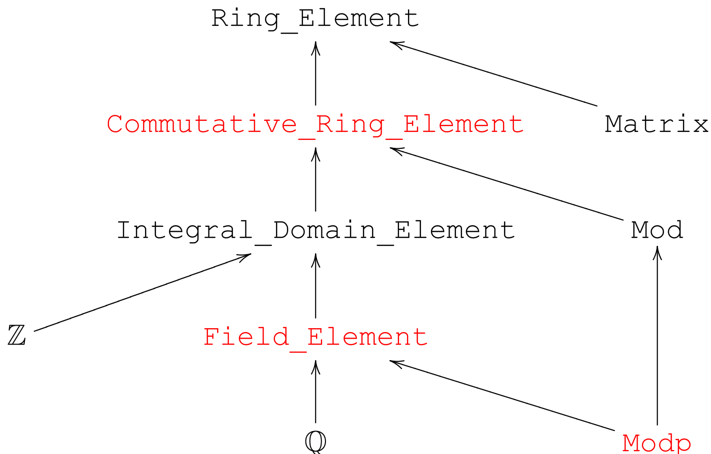
Abstract return types

Casting

Conclusion

Class hierarchy

`Matrix` unimplemented, but otherwise as desired:



Dreaded Diamond of Doom!

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

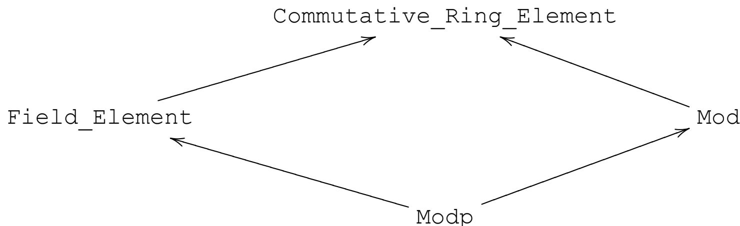
Inheritance

Abstract return types

Casting

Conclusion

Problem 1: Dreaded Diamond of Doom



Challenge w/multiple inheritance:

- `Commutative_Ring_Element` has methods, e.g., `is_one()`
- `Field_Element`, `Mod` both inhert own `is_one()`
- `Modp` inherits whose `is_one()`?

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Solution 1: virtual inheritance

```
class Field_Element
  : virtual public Integral_Domain_Element
{
  ...
};

class Mod
  : virtual public Commutative_Ring_Element
{
  ...
};
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Solution 1: virtual inheritance

```
class Field_Element
  : virtual public Integral_Domain_Element
{
  ...
};

class Mod
  : virtual public Commutative_Ring_Element
{
  ...
};
```

"Virtual" inheritance says that, when merged, inherited virtual methods should *merge as one.*

- works for inherited virtuals, but...
- what about functions redefined by one or the other?

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Problem 2: Merge failure

`Field_Element`, `Mod` both redefine `has_inverse()`

```
virtual bool Field_Element::has_inverse()
const override {
  return not is_zero();
}

virtual bool Mod::has_inverse()
const override {
  return invertible;
}
```

Problem 2: Merge failure

`Field_Element`, `Mod` both redefine `has_inverse()`

```
virtual bool Field_Element::has_inverse()
const override {
  return not is_zero();
}

virtual bool Mod::has_inverse()
const override {
  return invertible;
}
```

Differ, so *cannot* merge

Solution 2: finalization

`Modp` must finalize method in own function, can

- select parent(s) explicitly w/scoping `Field_Element::has_inverse()`
- provide completely new implementation

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Solution 2: finalization

`Modp` must finalize method in own function, can

- select parent(s) explicitly w/scoping `Field_Element::has_inverse()`
- provide completely new implementation

Our choice: “new” implementation

```
virtual bool Modp::has_inverse()
const override {
  return not is_zero();
}
```

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Can’t return abstract type

`Field_Element` has pure virtual functions (inherited & new)

```
class Ring_Element {
  virtual bool is_one() const = 0;
}

class Field_Element
  : virtual public Integral_Domain_Element
{
  ...
};
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can't return abstract type

`Field_Element` has pure virtual functions (inherited & new)

```
class Ring_Element {
  virtual bool is_one() const = 0;
}

class Field_Element
  : virtual public Integral_Domain_Element
{
  ...
};
```

Because of this, C++ *forbids* returning `Field_Element`

```
virtual Field_Element inverse() const = 0;
```

Disallowed even if `inverse()` not "pure virtual"!

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Solution: return *reference* to type...

```
class Field_Element
  : virtual public Integral_Domain_Element
{
  virtual Field_Element & inverse() const = 0;
};
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Solution: return *reference* to type...

```
class Field_Element
  : virtual public Integral_Domain_Element
{
  virtual Field_Element & inverse() const = 0;
};
```

Return elements of *reference* type not resolved until runtime

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

...but don't return stack objects

This *will* cause a runtime error:

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  Mod<T,m> result;
  ...
  return result;
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

...but don't return stack objects

This *will* cause a runtime error:

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  Mod<T,m> result;
  ...
  return result;
}
```

Local variables appear *on "local stack."*

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

...but don't return stack objects

This *will* cause a runtime error:

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  Mod<T,m> result;
  ...
  return result;
}
```

Local variables appear *on "local stack."*

- when function returns, `result` "disappears"

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

...but don't return stack objects

This *will* cause a runtime error:

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  Mod<T,m> result;
  ...
  return result;
}
```

Local variables appear *on "local stack."*

- when function returns, `result` "disappears"
- memory will *probably* be corrupted...

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

...but don't return stack objects

This *will* cause a runtime error:

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  Mod<T,m> result;
  ...
  return result;
}
```

Local variables appear *on "local stack."*

- when function returns, `result` "disappears"
- memory will *probably* be corrupted...
- ...but not always
- ***real pain to debug***

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

“Solution”: use static objects

Static objects appear on “global stack.”

- not “disappeared”/corrupted when function returns

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  static Mod<T,m> result;
  ...
  return result;
}
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

“Solution”: use static objects

Static objects appear on “global stack.”

- not “disappeared”/corrupted when function returns

```
Mod<T,m> & Mod<T,m>::operator +(const Mod<T,m> & other)
{
  static Mod<T,m> result;
  ...
  return result;
}
```

Still not entirely safe: know why?

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can be overwritten

What *will* this print? Is it what you *want* it to print?

```
Modp<long,43> x(2), y(5), u(-6), v(6);
Mod<long,43> & result1 = x + y;
Mod<long,43> & result2 = u + v;
cout << result1 + result2 << endl;
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can be overwritten

What *will* this print? Is it what you *want* it to print?

```
Modp<long,43> x(2), y(5), u(-6), v(6);
Mod<long,43> & result1 = x + y;
Mod<long,43> & result2 = u + v;
cout << result1 + result2 << endl;
```

Actual result is 0!

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Can be overwritten

What *will* this print? Is it what you *want* it to print?

```
Modp<long,43> x(2), y(5), u(-6), v(6);
Mod<long,43> & result1 = x + y;
Mod<long,43> & result2 = u + v;
cout << result1 + result2 << endl;
```

Actual result is 0!

Problem is *not* in storing `Modp` in `Mod`; that is perfectly alright!

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	2
second	5
`result`	-
`result1`	-
`result2`	-

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	2
second	5
`result`	7
`result1`	-
`result2`	-

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	-
second	-
`result`	7
`result1`	7
`result2`	-

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	37
second	6
`result`	7
`result1`	7
`result2`	-

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	37
second	6
`result`	0
`result1`	0
`result2`	-

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	-
second	-
`result`	0
`result1`	0
`result2`	0

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Actual problem? *references*

```
Mod<long,43> &
result1 = x + y;
Mod<long,43> &
result2 = u + v;
```

first	-
second	-
`result`	0
`result1`	0
`result2`	0

...because `result1` is a `reference` to result.

MAT 685: C++ for Mathematicians

John Perry

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Solution: pass *reference*, receive *copy*

```
Modp<long,43> x(2), y(5), u(-6), v(6);
Mod<long,43> result1 = x + y;
Mod<long,43> result2 = u + v;
cout << result1 + result2 << endl;
```

...gives correct result

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Outline

1. Rings
2. `Ring` and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C v. C++ casting

"Casting" changes view of variable's type

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C v. C++ casting

"Casting" changes view of variable's type

Example

- `operator+` is common to all `Ring_Elements`, but...

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C v. C++ casting

"Casting" changes view of variable's type

Example

- `operator+` is common to all `Ring_Elements`, but...
- `Ring_Element` has no idea what its children are, so...

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C v. C++ casting

"Casting" changes view of variable's type

Example

- `operator+` is common to all `Ring_Elements`, but...
- `Ring_Element` has no idea what its children are, so...
- `Ring_Element`'s `operator+` expects `Ring_Element` arguments, so...

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C v. C++ casting

"Casting" changes view of variable's type

Example

- `operator+` is common to all `Ring_Elements`, but...
- `Ring_Element` has no idea what its children are, so...
- `Ring_Element`'s `operator+` expects `Ring_Element` arguments, so...
- `Mod`'s `operator+` receives `Ring_Element` arguments, but...

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

C v. C++ casting

"Casting" changes view of variable's type

Example

- `operator+` is common to all `Ring_Elements`, but...
- `Ring_Element` has no idea what its children are, so...
- `Ring_Element`'s `operator+` expects `Ring_Element` arguments, so...
- `Mod`'s `operator+` receives `Ring_Element` arguments, but...
- `Mod` can only add `Mod`'s, not generic `Ring_Elements`

...so `Mod` must cast `Ring_Element` arguments to `Mod` objects

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How do we do this?

Four types of cast in C++

- C-style: no checking takes place

```
(int )x;
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How do we do this?

Four types of cast in C++

- C-style: no checking takes place

  ```
  (int )x;
  ```

- static cast: more readable & explicit C-style cast

  ```
  int y = static_cast<int>(x);
  ```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How do we do this?

Four types of cast in C++

- C-style: no checking takes place
  ```
  (int )x;
  ```
- static cast: more readable & explicit C-style cast
  ```
  int y = static_cast<int>(x);
  ```
- dynamic cast: checks type & returns `nullptr` if `x` not of type
  ```
  int y = dynamic_cast<int>(x);
  ```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

How do we do this?

Four types of cast in C++

- C-style: no checking takes place
  `(int )x;`
- static cast: more readable & explicit C-style cast
  `int y = static_cast<int>(x);`
- dynamic cast: checks type & returns `nullptr` if `x` not of type
  `int y = dynamic_cast<int>(x);`
- const cast: adds or removes `const` qualifier to variable

```
const int y = const_cast<const int>(x);
// adds const
int z const_cast<int>(u); // removes
const
```

Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Our need: operators

```
virtual bool Ring_Element &
  operator + (const Ring_Element &) const = 0;

virtual bool Mod<T,m> &
  operator + (const Ring_Element & other)
  const override
{
  static Mod<T,m> result;
  // next line makes o look at other as type
  auto o = dynamic_cast<const Mod<T,m> &>(other);
  result.value = value + o.value;
  result.adjust_value();
  return result;
}
```

`Ring_Element` requires `Ring_Element`, so `Mod` must

MAT 685: C++ for Mathematicians
John Perry
Rings
Ring and its children
Implementing w/templates
Code
Observations on the code
Organization
Templating
Inheritance
Abstract return types
Casting
Conclusion

Our need: operators

```
virtual bool Ring_Element &
  operator + (const Ring_Element &) const = 0;

virtual bool Mod<T,m> &
  operator + (const Ring_Element & other)
  const override
{
  static Mod<T,m> result;
  // next line makes o look at other as type
  auto o = dynamic_cast<const Mod<T,m> &>(other);
  result.value = value + o.value;
  result.adjust_value();
  return result;
}
```

`Ring_Element` requires `Ring_Element`, so `Mod` must

casting `Ring_Element` to `Mod` lets us perform computation

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Not to be done lightly

`dynamic_cast` checks type!

- `other` not w/`Mod<T,m>` or descendant? result is `nullptr`
- `o.value` on next line ***will fail***

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Not to be done lightly

`dynamic_cast` checks type!

- `other` not w/`Mod<T,m>` or descendant? result is `nullptr`
- `o.value` on next line ***will fail***

Example

This can go wrong as easily as this:

```
Mod<long, 5> m(3);
Integer<long> n(3);
Mod<long, 5> result = m + n;
```

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Not to be done lightly

`dynamic_cast` checks type!

- `other` not w/`Mod<T,m>` or descendant? result is `nullptr`
- `o.value` on next line ***will fail***

Example

This can go wrong as easily as this:

```
Mod<long, 5> m(3);
Integer<long> n(3);
Mod<long, 5> result = m + n;
```

...but we can fix this by implementing

```
template <typename T, typename U>
Mod<T,m> & Mod<T,m>::operator +(
    const Integer<U> other
) const;
```

Outline

1. Rings
2. Ring and its children
3. Implementing w/templates
4. Code
5. Observations on the code
   - Organization
   - Templating
   - Inheritance
   - Abstract return types
   - Casting
6. Conclusion

Rings

Ring and its children

Implementing w/templates

Code

Observations on the code

Organization

Templating

Inheritance

Abstract return types

Casting

Conclusion

Summary

- Math stuff
  - ring relationships
- Programming stuff
  - namespaces
  - inheritance
    - multiple inheritance
    - Dreaded Diamond of Doom
    - virtual inheritance
    - finalization
  - templates
    - purpose(s)
    - creation
  - casting