C++ Foundation

Functions, Operators, and Data Structures

Functions, Operators, and Data Structures

- member functions
- private access
- overloading
- default arguments
- function templates
- references
- passing arguments
- operators

Member Data and Member Functions

- C++ structs and classes can contain functions

C

```
struct date
{
        int year;
        int month;
        int day;
};
int day_number(const date *);
```

```
void eg(struct date when)
{
    when.day = 30;
    day_number(&when);
}
```

C++

```
struct date
{
        int year;
        int month;
        int day;

        int day_number() const;
};
```

```
void eg(date when)
{
    when.day = 30;
    when.day_number();
}
```

Access Control

- C++ structs and classes can control access using the public and private keywords

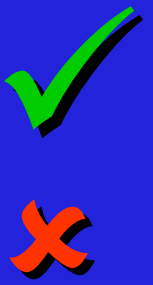
```
struct date
{
public:
      ...
      int year;

private:
      ...
      int month;
};
```

```
date when;

when.year;

when.month;
```



No Abstraction

- C has limited scope for abstraction based on use
  - C structs typically expose their representation
  - An open invitation to access their “private parts”! not safe! not polite!

```
struct date
{
     int year, month, day;
};
```

```
void oops(struct date * when)
{
    when->day = 42;
}
```

A Model of Politeness

- C++ structs and classes offer a model *not* based on representation
  - Lights, camera, abstraction!

```
struct date
{
        ...
        int year() const;
        int month() const;
        int day() const;
        ...
};
```

```
void eg(date when)
{
    today.day()
}
```

don't forget the parentheses!

What the heck does that const mean?

```
struct date
{
      ...
      int day() const;
      ...
};
```

a const at the end of a member function declaration promises...

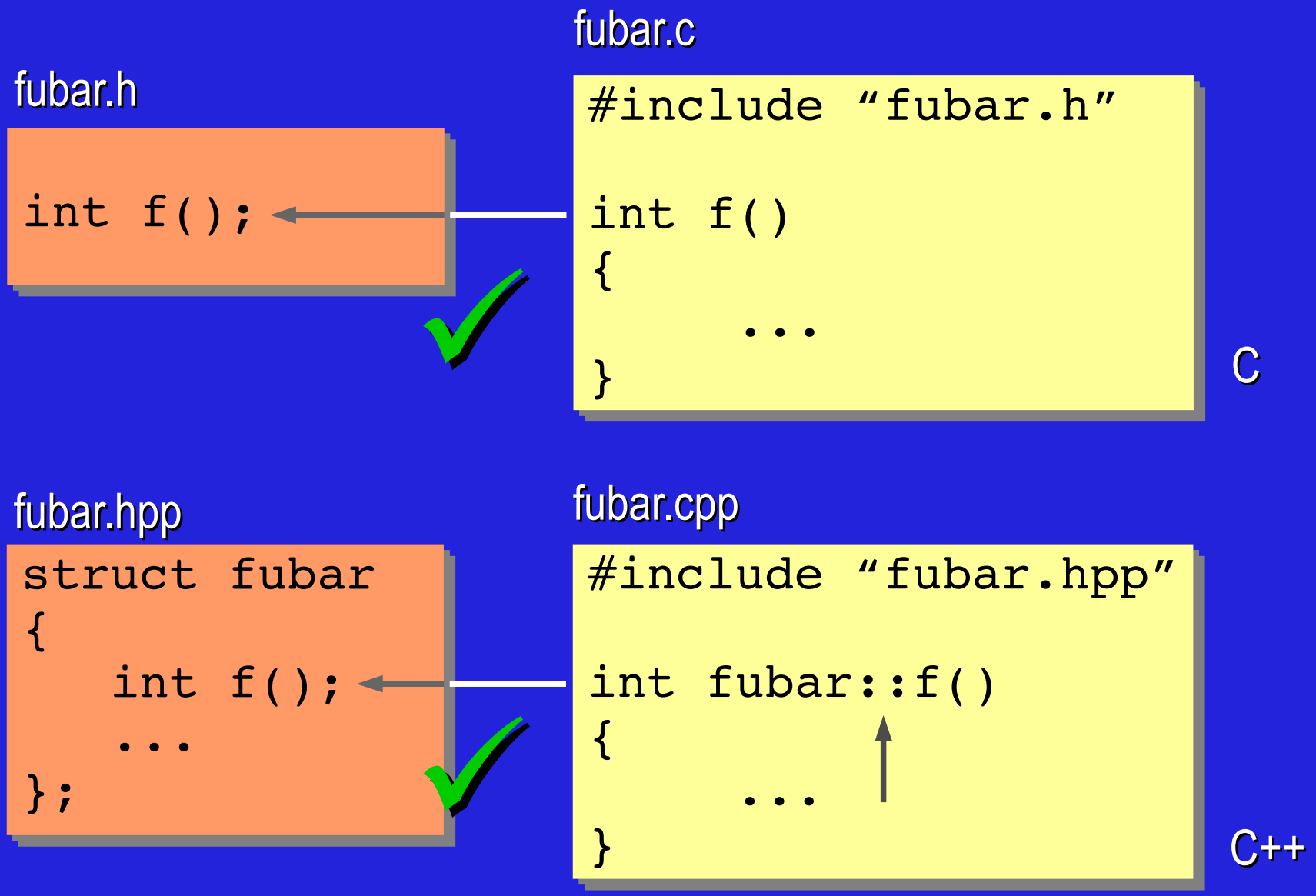
```
today.day()
```

...that if you call that member function on an object...

```
date today;
```

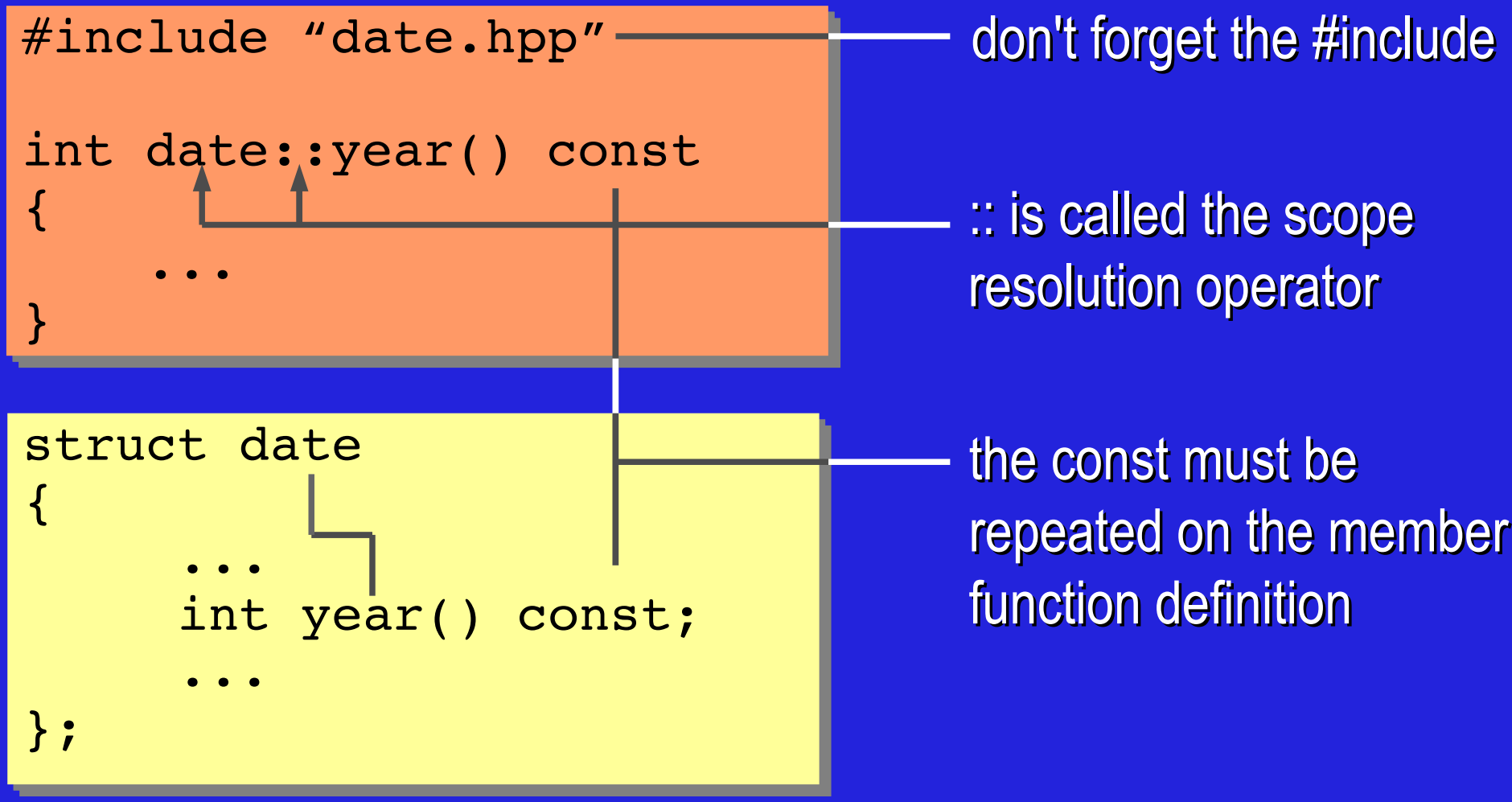
...the member function's definition won't change the visible state of that object

How to Define Member Functions?



The Scope Resolution Operator

- Member function declarations require a matching member function definition



Private Access

- Only member functions have access to private members
- All members are accessible to all other members through the implicit this pointer

```
struct date
{
      ...
      int year() const;
      ...
private:
      long time_stamp;
};
```

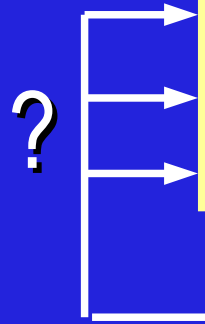
```
#include “date.hpp”

int date::year() const
{
      ... time_stamp ...
}
```

Overloading

- Functions with the same name and in the same scope are said to *overload* each other
- Function parameters must differ somehow
- A difference in return type alone is not sufficient

```
int    min(int    lhs, int    rhs);
long   min(long   lhs, long   rhs);
double min(double lhs, double rhs);
```

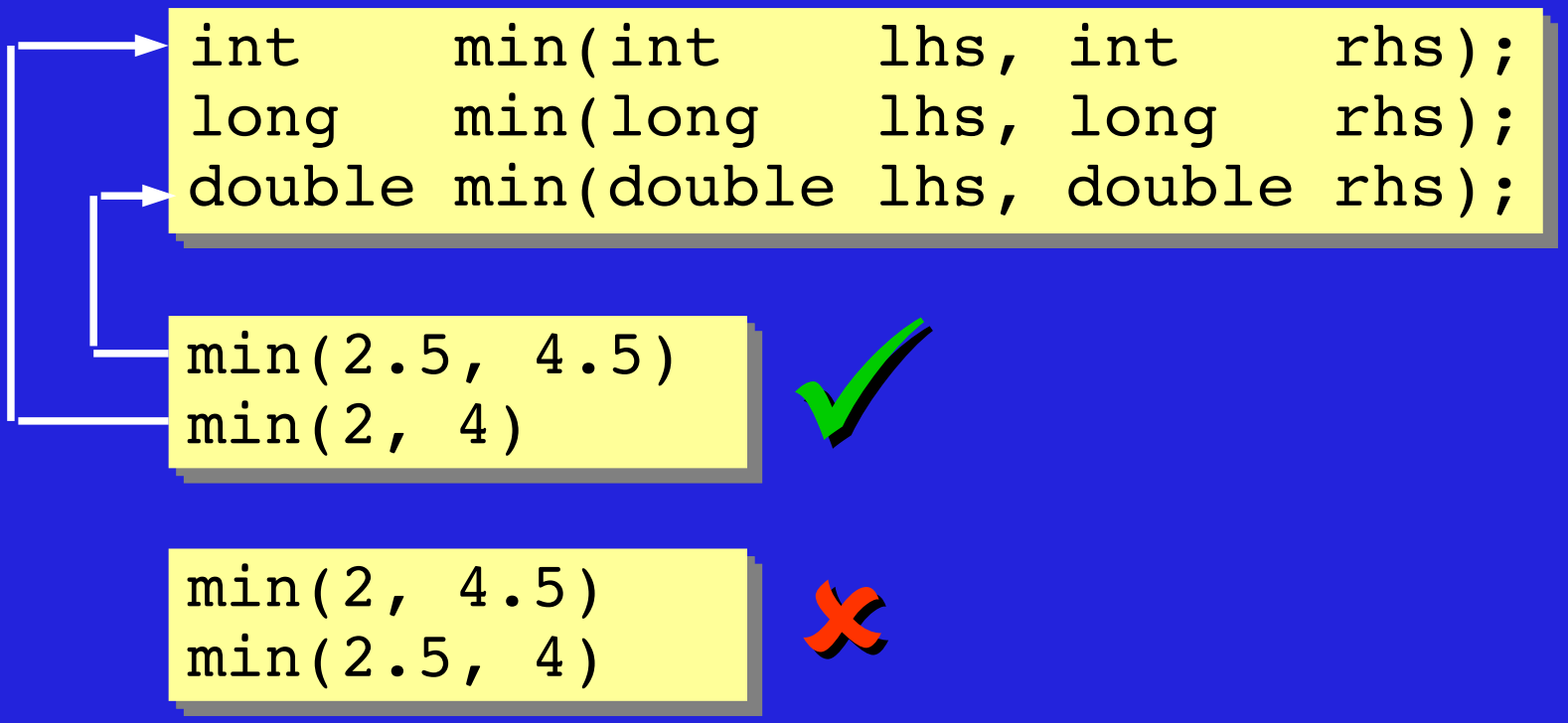

```
int    random();
long   random();
double random();
```

```
random();
```

overloading != overriding

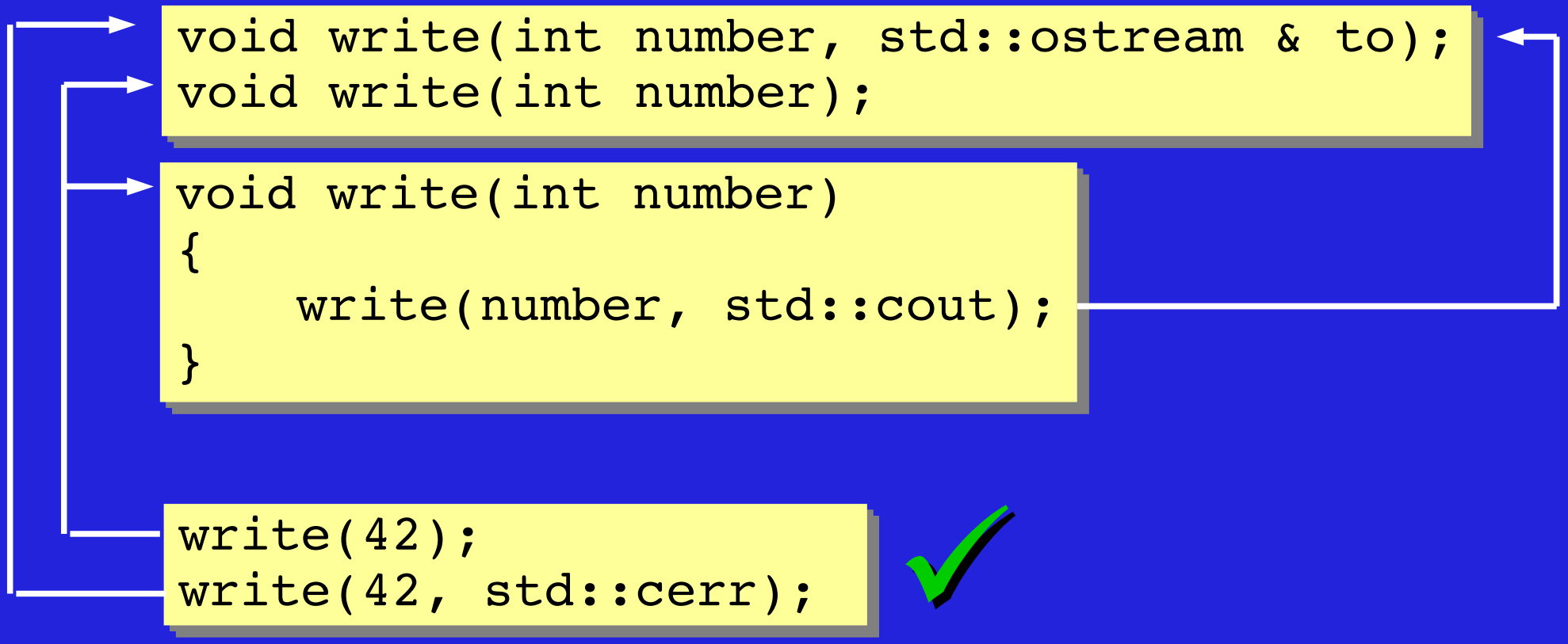
Overloading

- The compiler resolves a call to the overload with the best argument-parameter match
- Resolution must be unambiguous



Internal Default Arguments

- Overloading and forwarding can provide “internal” defaults



External Default Arguments

- An explicit “external” =default syntax also exists

```
void write(int number,
           std::ostream & to = std::cout);
```

```
write(42);
```

compiler rewrites to...

```
write(42, std::cout);
```

the default is written on the declaration, not on the definition

```
write(42, std::cerr);
```

defaults are written on the rightmost parameters

External Defaults...?

- Can increase header dependencies
- Can be surprising - best avoided

```
coord make_coord(int x = 0, int y = 0);
```

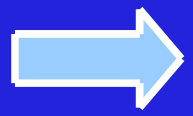
```
make_coord();
```

```
make_coord(0,0);
```

```
make_coord(2,5);
```

```
make_coord(2,5);
```

```
make_coord(42);
```



```
make_coord(42,0);
```

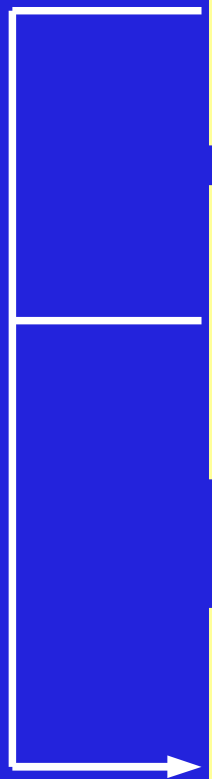
?!

Function Templates

- What about when functions differ in their *types* but not their definition?

```
int min(int lhs, int rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

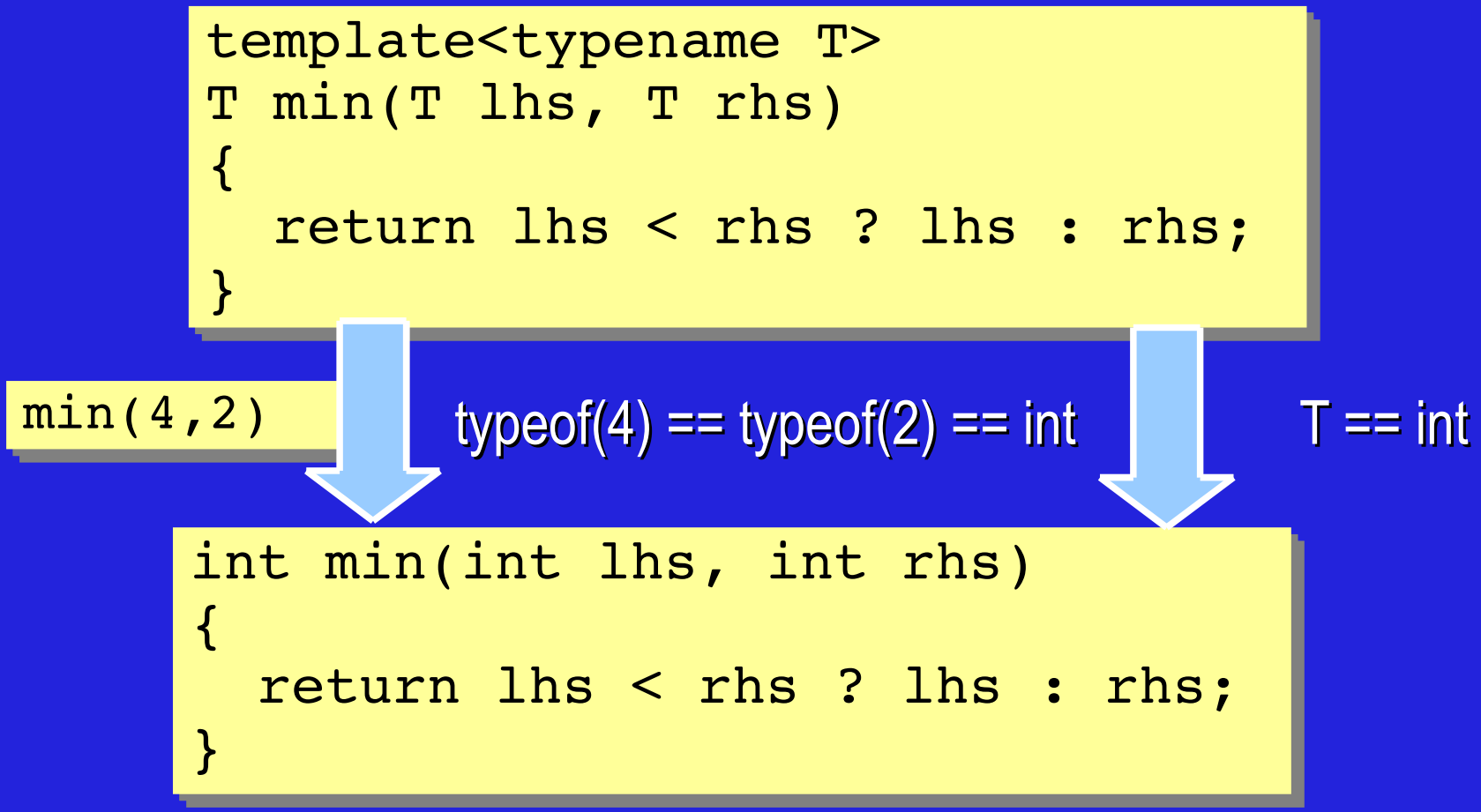
```
double min(double lhs, double rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```



```
?? min(?? lhs, ?? rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

Function Templates

- The compiler can use a function template to generate functions to match function calls!



Template Type Names

- It is common to express the template type's requirements in its name

```
template<typename T>
T min(T lhs, T rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

s/T/Comparable/

```
template<typename Comparable>
Comparable min(Comparable lhs, Comparable rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

Function Templates

- Can have more than one templated type

```
template<typename Iterator,
         typename Function>
Function for_each(Iterator at,
                  Iterator end,
                  Function f)
{
    for (; at != end; ++at)
        f(*at);
    return f;
}
```

```
const char * str = “Hello”;
for_each(str, str + 5, std::putchar);
```

Function Template Diagnostics

- Can be *hard* to understand!

```
void wtf()
{   ...
    std::for_each(begin, end, s());
}
```

```
In file included from /usr/include/c++/4.4/algorithm:62
...
/usr/include/c++/4.4/bits/stl_algo.h: In function
  '_Funct std::for_each(_IIter, IIter, _Funct)
  [with __Iter = int*, _Funct = s]'
...perhaps 450 lines of babble...
instantiated from here
/usr/include/c++/4.4/bits/stl_algo.h:4200:
  error: no match for call to '(s) (int&)'
```

References

- Look like pass by copy, but isn't
- An alias to an object

C

```
void f(int value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 42);
}
```

C

```
void f(int * value)
{
    (*value)++;
}
void eg()
{
    int x = 42;
    f(&x);
    assert(x == 43);
}
```

C++

```
void f(int & value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 43);
}
```

This is not
f(&x);

const references

- Look like pass by copy, but isn't
- An alias to a readonly object

C

```
void f(int value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 42);
}
```

C++

```
void f(int & value)
{
    value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 43);
}
```

C++

```
void f(const int & value)
{
    //value++;
}
void eg()
{
    int x = 42;
    f(x);
    assert(x == 42);
}
```

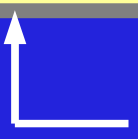
Operators

- Highly stylized functions
- You can overload most operators

```
int deadline, today;
...
if (deadline == today)...
```

```
struct date { ... };
bool operator==(const date &, const date &);
```

```
date deadline, today;
if (deadline == today)...
```



infix notation

Template Type Requirements

- Template types require a *uniform* syntax

```
template<typename T>
T min(T lhs, T rhs)
{
  return lhs < rhs ? lhs : rhs;
}
```

this definition requires T supports the < operator

```
struct date { ... };
bool operator<(const date &, const date &);
```

```
date earliest,estimate;
...
earliest = min(earliest, estimate);
```

Parameter Passing

- By const & to mimic pass by copy for non-primitive types is a common idiom
- By copy when the argument is an <algorithm> parameter is also common
- By non-const & when the function modifies the argument
- By copy when the argument is a primitive type (eg int, bool, enum)
- By plain pointer to indicate object lifetime considerations