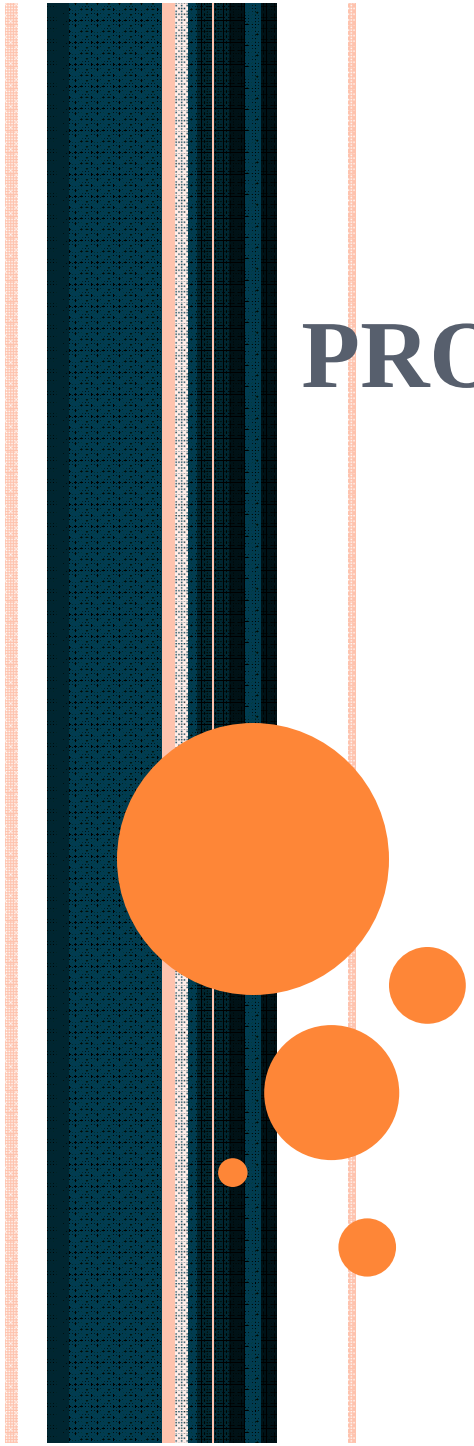

OBJECT ORIENTED PROGRAMMING USING C++

Overview of C++ Overloading

- Overloading occurs when the *same* operator or function *name* is used with *different signatures*

- Both operators and functions can be overloaded

- Different definitions must be distinguished by their signatures (otherwise which to call is ambiguous)
  - Reminder: signature is the operator/function name and the ordered list of its argument types
  - E.g., **`add(int,long)`** and **`add(long,int)`** have different signatures
  - E.g., **`add(const Base &)`** and **`add(const Derived &)`** have different signatures, even if **`Derived`** is-a **`Base`**
  - Most specific match is used to select which one to call

Overloading vs. Overriding

- Overriding a base class method is similar to overloading
  - But definitions are distinguished by their scopes rather than by their signatures

- C++ can distinguish method definitions according to either static or dynamic type
  - Depends on whether a method is virtual or not
  - Depends on whether called via a reference or pointer vs. directly on an object
  - Depends on whether the call states the scope explicitly (e.g., `Foo::baz();`)

Function Overloading

```
class A {
public:
   int add(int i, int j);

   // not allowed, would be
   // ambiguous with above
   // long add(int m, int n);

   // Ok, different signature
   long add(long m, long n);
};

int
main (int argc, char **argv) {
  int a = 7;
  int b = 8;
  int c = add(a, b);
  return 0;
}
```

- Calls to overloaded functions and operators are resolved by
  - Finding all possible matches based on passed arguments
    - May involve type promotion
    - May involve instantiating templates
  - Finding the “best match” among those possible matches
- Signature does not include the return type
  - Which might not help even if it did, i.e., calls may ignore result
  - So, overloading can’t be resolved by return type alone
  - Compiler generates an error if the call can’t be resolved

Operator Overloading

```
class A {
friend ostream &operator<<
  (ostream &, const A &);
private:
  int m_a;
};

ostream &operator<<
  (ostream &out, const A &a) {
  out << "A::m_a = " << a.m_a;
  return out;
}

int main () {
  A a;
  cout << a << endl;
  return 0;
}
```

- Similar to function overloading
  - Resolved by signature
  - Best match is used
- But the list of operators and the "arity" of each is fixed
  - Can't invent operators
  - Must use same number of arguments as for built-in types (except for `operator()`)
  - Some operators are off limits

    `::` (scope) `.` (dot) `?:` (conditional)

    `sizeof` `typeid` (RTTI)

    type casting operators

Operator Symmetry, Precedence

```
class Complex {
public:
 // constructor from real and
 // imaginary parts
 Complex (int r, int i);

 // addition
 Complex operator+ (const Complex &);

 // multiplication
 Complex operator* (const Complex &);

 // exponentiation
 Complex operator^ (const Complex &);

private:
  int real_;
  int imaginary_;
};
```

- In general, make operators symmetric
  - Don't mix base and derived types in their parameter lists
- Operators always obey the same precedence rules (Prata pp. 1058)
  - Can lead to some unexpected mistakes
  - E.g., what's wrong with this Complex number expression?

```
a + b * c ^ 2
```

Member/Non-Member Overloading

```
class A {
  friend bool operator<
    (const A &lhs, const A &rhs);
public:
   bool operator==(const A &a) const;
private:
   int m_a;
};
// member operator
bool A::operator==(const A &a) const {
  // note: object itself is
  // the first argument, can be const
  return m_a == a.m_a;
}
// non-member operator
bool operator<
(const A &lhs, const A &rhs) {
  return lhs.m_a < rhs.m_a;
}
```

- Remember a **`this`** pointer is passed to any non-static member function
  - So, for member functions and operators the object itself does not appear in the argument list
  - For non-member functions and operators all parameters appear
- So, the rule about operator arity is obeyed in code on left
  - Both **`<`** and **`==`** are binary operators
  - Can you see what needs to be added to both of these operators?
- Non-member operators are useful when working with classes you wrote and classes you didn't write
  - E.g., **`ostream <<`** and **`istream >>`**
- Non-member operators are also useful to preserve symmetry
  - May avoid unexpected type conversions, especially up an inheritance hierarchy

Type Cast Operators (and typedef)

```
int main (int,
          const char * argv[]) {

 // cast away constness
 char *p =
  const_cast<char*>(argv[0])

 // convert to smaller type
 int i = 50;
 char c = static_cast<char>(i);

 // downcast a pointer (returns
 // 0 if *bptr isn't a Derived)
 Base * bptr = new Derived;
 Derived * dptr =
  dynamic_cast<Derived*>(bptr);

 // reinterpret a pointer
 typedef unsigned long ulong;
 ulong cookie =
  reinterpret_cast<ulong>(p);
}
```

- Four type cast operators in C++
  - Only use these when you must
  - You cannot overload them
  - Take a type parameter (generic)
- To get a mutable interface from a const one, use const_cast
- To force a static type conversion that's known to be safe at runtime use static_cast
- To force a dynamic type conversion that's known to be safe at runtime use dynamic_cast
- To reinterpret a type as another type (strongest form of casting) use reinterpret_cast
- To alias a type, use typedef

Summary: Tips on Overloading

- Use virtual *overriding* when you want to substitute different subtypes polymorphically
  - E.g., `move()` in derived and base classes

- Use *overloading* when you want to provide related interfaces to *similar* abstractions
  - E.g., `migrate(Bird &)` vs. `migrate(Elephant &)`

- Use different names when the abstractions differ
  - E.g., `fly()` versus `walk()`