

static members

- static data members – “class variables”
 - Just one copy (in global space) shared by all instances
 - Declare inside class; initialize in global space (w/out static)
 - Use for instance counters, common constants, shared data, ... – see `String::count()` example in `.../demo07/`
- static methods – also instance independent
 - In fact, don’t need any instances to invoke them
`int result = Foo::aStaticFunction();`
 - Caution: no `this` pointer, because no instance involved
 - So there is no way to access non-static data or methods

friends

- A class can grant friendship to a function

```
class Foo
{ friend void showValue(Foo const &me); ... }
void showValue(Foo const &obj)
{ cout << obj.value; }
```

 - Now okay for global function to access private data
 - See `String::join()` in `.../demo07`
- A class can grant friendship to a class too

```
friend class FooFriend;
```

 - Now all of `FooFriend`’s functions can access value

Exceptions

- No Exception class like Java
- Instead `throw` any valid C++ expression
 - Then exception propagates to calling function
 - Process continues until the expression is caught
 - Program terminates if exception is never handled
 - Any objects on the stack are popped first
- Use `try/catch` block to handle (like Java)

```
try { /* call function that might throw an exception */ }
catch (TypeX e) { /* handler if TypeX is thrown */ }
catch (...) { /* default handler – catches any type */ }
```
- See `quotient.cpp` example in `.../demo08/`

`new` may throw exception

- If insufficient memory, `new` throws type `std::bad_alloc` (defined in `<new>`)

```
try { int *ptr = new int(7); ... delete ptr; }
catch(std::bad_alloc const &)
{ cerr << "out of memory\n"; }
```
- Can prevent by `nothrow` expression
 - Then `new` returns 0 pointer instead (like `malloc` does)

```
int *ptr = new (std::nothrow) int(7);
if (ptr) { /* process */ }
else cerr << "out of memory\n";
```
- Note: dtors are not allowed to throw exceptions

Exception specifications

- A little bit like Java’s `throws` clauses
- Limits what a function may throw

```
void func() throw(double, TypeX) {...}
// may only throw a double or a TypeX
void func2() throw() // may not throw anything
void func2() // may throw anything
```
- Beware: compiler might not catch violations
 - Might not be able to detect indirect violations
 - Might choose to ignore, or just issue warning

Function overloading

- Same basic idea as Java – but more versatile
 - A function’s signature is `name(argument list)`
 - Overloading means reusing a name in the same scope
 - Requires a different argument list – number/order/type
 - The return type does not matter
 - `const`-qualified arguments or member functions do matter
 - Compiler “mangles” the names for the linker (see p. 250)
- Can overload class member, or global functions
 - But not overloading to have one of each type – use `::`
- C++ bonus: can overload operators (`+`, `-`, ...) too!

Operator function overloading

- Can overload almost any C++ operator
 - At least one argument must be user-defined type
 - Precedence, “narity”, and associativity rules apply as always for each operator
 - e.g., + has usual precedence, is binary or unary, l-r
 - e.g., = has lower precedence, is binary only, r-l
 - “Just because you can does not mean you should”
 - e.g., don’t overload , or && or || – see charts, pp. 255-7, 284
- Defining – `ADT operator+(ADT &, ADT &);`
 - Overloads + to return an ADT object (hopefully the sum of the two ADT arguments – best to not change operator’s meaning)

3 free member operators

- Assume *any* class – even `class C {};`
- It has an assignment operator
 - `C &operator=(C const &);`
 - Makes a shallow copy, like free copy ctor
- And it has 2 different address-of operators
 - One for mutable objects:
`C *operator&();`
 - And one for constant objects:
`C const *operator&() const;`
- If you’re skeptical, see `.../demo08/freeops.cpp`

Member vs. non-member ops

- Some functions are better as class members
 - Like any one that needs a `this` pointer: `++`, `+=`, ...
 - And there are four operators that can only be overloaded as class members: `=`, `()`, `[]`, and `->`
 - Usually a better choice – avoids need to be `friend`
 - And it always has a required user-defined argument
- Sometimes non-member functions better though
 - e.g., binary functions, where arg order doesn’t matter:
 - `==`, `<`, ..., and binary forms of `+`, `-`, `*`, `/`, `%`
 - Also when other types must access – like `<<` and `>>` that must be accessed by `cout` and `cin`

Non-member operator functions

- e.g., `operator+()` – (binary version)
 - Usually a friend – to access data with less overhead
- ```
friend ADT const operator+
 (ADT const &, ADT const &);
```
- Can invoke by using either functional or infix notation
- ```
ADT a, b; ...
ADT c = operator+(a, b); // functional notation
ADT d = a + b; // infix notation
```
- See
String:
+ and << in
.../demo08
- Note: either `a` or `b` must be an ADT instance for both notations
- Another example: `operator<<`

```
friend ostream &operator<< // returns ostream &
    (ostream &, ADT const &);
```
 - Note: `ostream&` is left argument; `ADT&` is right (must be explicit)

Member operator functions

- First argument is `this` – but it’s hidden
 - Is the left argument of binary operations
 - So there can be no implicit conversion of left argument
 - Is the only argument of unary operations
- e.g., `operator+=` – a compound assignment op

```
ADT &ADT::operator+=(ADT const &right)
- {return *this = *this + right;} // assumes = and + exist
```
- Sometimes need 2 versions – `operator[]` – subscript op
 - e.g., `String` might allow access to chars

```
char &String::operator[](int index) {...} // mutable for lvalue
char String::operator[](int index) const {...} // not mutable
```
 - Ops `++` and `--` even more interesting – see `.../demo08/String`

Inheritance in C++

- Usual C++ terms: Base and Derived classes
- Usual definition – `class D : public B {...}`
 - Makes `D` a public derivation of class `B`
 - See `.../demo08/employees/example`
- Rarely: `class E : private B {...}` // or just: `class E : B`
 - `E` inherits `B` members, but not `B`’s interface
 - Can make specific member, `m`, available by using `B::m`;
 - Also protected derivations/members – subclass access only
- Note: if `struct` instead of `class` – default is `public`
- “Is-a” rule only applies for public derivations