Classes Part 2: Operator Overloading, Breaking up Files, makefiles, UML and Umbrello

Operator overloading

- Definition: Providing a user-defined meaning to a pre-defined operator (e.g., +, ==, <<) for a user-defined type (class).

- New keyword: const
  - constant, does not change
- New keyword: this
  - This object’s instance
- New keyword: friend
  - Grants access to private variables

Operator Overloading

- You can define only existing operators
  - You can't create your own custom operator such as $$ or @
- You can define operators only with the usual number of operands
  - *E.g.*, no unary <= (less than or equal) and no binary **!** (not)
- Overloaded operators must have *at least* one user-defined type as an operand
  - **int operator+(int,int);** // error: you can't overload built-in +
  - **Vector operator+(const Vector&, const Vector &);** // ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - **+** should be addition, ***** be multiplication, **[]** be access, **()** be call, etc.
  - You must determine what is "conventional" with the classes you define
- Advice (not language rule):
  - Don't overload unless you really have to

Permissible Operator Overloading

- The following operators **can** be overloaded:
  - + - * / % ^
  - & | ~ ! , =
  - < > <= >= ++ --
  - << >> == != && ||
  - += -= /= %= ^= &=
  - |= *= <<= >>= [ ] ( )
  - → ->* new new [ ] delete delete [ ]
- The following operators **cannot** be overloaded:
- :: .* . ?:

Breaking up files

- Normally, everything isn’t included in one file
  - If you write good code at least
- We use separate our classes from our main.
- Two cpp files
- Difference between <> and “” includes
  - <> for system files
  - “” for your own files

Breaking up files

- Normally, not all of a class is in one file
- Broken up between .h and .cpp files

Now how to compile all these functions

- Our compile statement has to include all the files, or be broken up into multiple commands
- Hard to do for big projects
- Hard to compile for new machines
- Makefiles are the solution

Makefiles

- Compile our code for us
- Look at a simple makefile first
- Then look at a more complicated make file
- Make one for our Date program.

Simple Makefile

- Comment
- Macros

- All = default command
- Additional commands

- Building coordinate.o if dependencies have changed
  - Bash Command – CXX = c++, CXXFLAGS = C++ flags

- Clean - Clear out what's there.

- Tabs not spaces

```
# Makefile for Roving Robots
CXXFLAGS = -std=c++11

all: coordinate.o

debug: CXXFLAGS += -g
debug: coordinate.o

coordinate.o: coordinate.cpp coordinate.h
	$(CXX) $(CXXFLAGS) -c coordinate.cpp

clean:
	rm -f *.o a.out
```

More complicated make file

```
# Makefile for Roving Robots
CXXFLAGS = -std=c++11

all: executable

debug: CXXFLAGS += -g
debug: executable

rebuild: clean executable
executable: test_coordinate.o coordinate.o
	$(CXX) $(CXXFLAGS) test_coordinate.o coordinate.o
test_coordinate.o: test_coordinate.cpp coordinate.h
	$(CXX) $(CXXFLAGS) -c test_coordinate.cpp
coordinate.o: coordinate.cpp coordinate.h globals.h
	$(CXX) $(CXXFLAGS) -c coordinate.cpp
clean:
	rm -f *.o a.out
```

Let make our own makefile

Look back at our UML