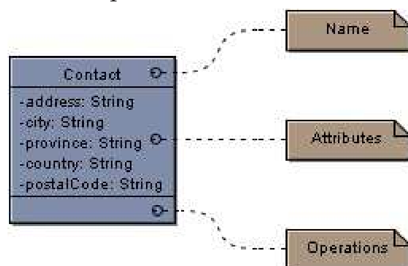
SPECIFICATION AND MULTIFILE COMPILATION

Today

- Today we will look at:
  - The use of UML for specification.
  - Multifile compilation
- Pohl covers UML very briefly on page 381, and does not explicitly cover multifile compilation.

UML

- The Universal Modeling Language (UML) is a technique for specifying software.
- The bit of UML that we will consider allows us to specify classes.
- For example:



- This sketches the class `Contact`.
- It lists a number of attributes:
  - `address`
  - `city`
  - `province`
  - `country`
  - `postalCode`
- It is also possible to specify the methods of the class, the “operations”.
- UML also specifies how private the attributes and methods are to the class.
  - The - before the attribute names specifies that they are private.

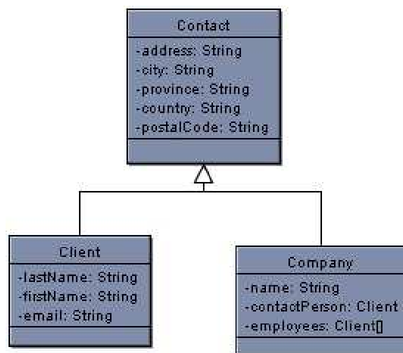
- We have:
  - – for private attributes;
  - + for public attributes; and
  - # for protected attributes.
- In this case:

we have:

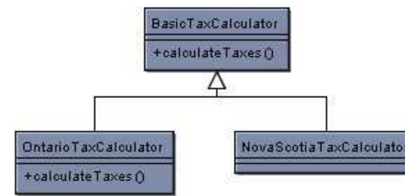
- A private attribute `object`, which is a list of `Objects`
- A public method `getSize` which and returns an integer.
- A public method `set` which is `void` and takes an `int` and an `Object` as arguments.
- A public method `get` which returns an `Object` and takes an `int` as an argument.

- The next piece of graphical notation relates sub-classes to super-classes (specializations to generalizations).

- In this example:

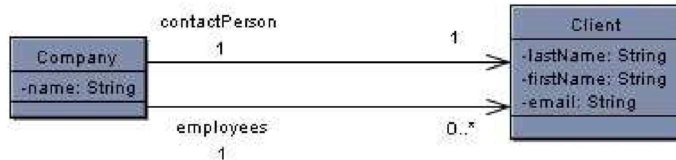


- `Client` and `Company` are both sub-classes of `Contact`.
- They each have some additional attributes on top of those in `Contact`.
- By default sub-classes inherit everything in the parent class.
- They can also provide *redefinitions* of operations.
- Here:



the `OntarioTaxCalculator` *overrrides* the definition of `calculateTaxes`.

- Finally, we can specify the relationship between different classes that refer to one another.



says there is one `Client` that is the `contactPerson` for a `Company` and that a `Company` has zero or more `employees` who are `Clients`.

- The notation ..* is the bit that means "or more".
- Thus $1, 3..^*$ means one, three or more.

Multifile compilation

- So far we have written all of our code in one file.
- Often it is annoying to do this.
  - Files get very big
  - We end up doing a lot of cut and paste (as with `point`).
- In these cases, it is often better to split the code into multiple files.
- We can do this easily — it just takes a bit of care.

- Let's start by describing a common way to set this up.
- We can divide a class definition, like that for class point, into two bits.
- One is the bit that contains the `class` statement:

```
class point{
  :
  :
};
```

we put this in a file called `point.h`

- The other bit is part that contains the method/function descriptions:

```
void point::print() const{
  :
}
```

and so on.

- We put that in a file called `point.cpp`
- We link the two files by adding:

```
#include"point.h"
```

at the start of `point.cpp`

- Note the speech marks rather than the usual < and >.

- This (roughly speaking) tells the compiler to add the contents of the `.h` file to the `.cpp` file.
- (Actually it tells the *preprocessor*, but the call to the prepreocessor is done from within `g++` so it is invisible to the programmer.)
- We will also need to `include` any libraries that are used in `point.cpp`.

- Now, we will typically want to use `point` in some program, and so we will have another file, say `testpoint.cpp` which contains something like:

```
int main (){

  point p;
  p.set(2, 3);
  p.print();

return 0;
}
```

- This clearly needs to

```
#include"point.h"
```

as well.

- How do we compile these files?
- If we try to compile `point.cpp`:

```
g++ point.cpp -o point
```

we get an error message like:

```
/usr/lib/gcc/x86_64-linux-gnu/4.2.4/../
 ../../../lib/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: ld returned 1 exit status
```

and we remember that every C++ program needs to have a `main`.

- If we try to compile `testpoint.cpp`:

```
g++ testpoint.cpp -o testpoint
```

we get an error message like:

```
/tmp/ccT2sdKH.o: In function `main':
main.cpp:(.text+0x1c0):
  undefined reference to `point::print()'
main.cpp:(.text+0x1e1):
  undefined reference to `point::set(double, double)'
main.cpp:(.text+0x1f9):
  undefined reference to `point::print()'
collect2: ld returned 1 exit status
```

because `testpoint.cpp` doesn't include the definitions of the function members of `point.cpp`.

- However, if we compile everything together:

```
g++ testpoint.cpp point.cpp -o testpoint
```

  we get the result we want.

- There's another way to do this also, one that is better when creating programs with multiple classes (which is, of course, typical).
- Here we do the compilation in two steps:

```
g++ -c point.cpp -o point
g++ testpoint.cpp point -o testpoint
```

- The first step uses the `-c` switch to stop compilation before running the *linker*.
- If we do this, the compiler does everything except check that it knows how to execute all the functions in the program.
- So it is happy with compiling `point.cpp` because it doesn't bother to check for a definition of `main`.

- The check only takes place in the second step, when there *is* a `main`, and when the compiler also knows about the member functions of class `point` (from the file `point`).

- Now, let's assume we have a new class, `line`, say, which has some `point` members.
- There will be a file `line.h` which gives the class definition and a file `line.cpp` which defines the members.
- We will also write a file `testline`, which uses the `line` class.
- You can find these files on the web page for Unit IV.
- Since `line` uses `point`, we include:

```
#include"point.h"
```

  in `line.h`.

- We can compile `line` successfully using the same approach as with `point`:

```
g++ -c line.cpp -o line
```

- But you can't compile `testline` in the same way as before. This:

```
g++ testline.cpp line -o testline
```

  will cause an error.

- The reason is that `testline` (indirectly) needs `point`.
- So now we have a three step compilation process:

```
g++ -c line.cpp -o line
g++ -c testline.cpp -o testline
g++ testline line point -o testline.exe
```

- We have to use a different name than `testline` for the final output.

- Now let's imagine we have another file with a `main` program, let's call it `prog.cpp`.
- `prog` uses both `line` and `point`.
- You might naturally think that you should have:

```
#include"point.h"
#include"line.h"
```

  in `prog.cpp`
- If you do that, you'llk get an error when you compile:

```
g++ prog.cpp line point -o prog
```

  because `point.h` will be included twice. Once through `line.h` and once directly.

- We can just figure out we only need `line.h`, but in complex code this is hard to do.
- Instead we use one of these methods:
  - `pragma once`
  - include guards
- To use the first, we put:

```
#pragma once
```

  at the start of `point.h`.
- This tells the compiler (well, the preprocessor) to only ever include that file once in a compilation.

- Include guards a an older idea, and a bit clunkier.
- Here you wrap the definition of the `point` class:

```
class point {

 private:
   double x, y;
 public:

   :

}
```

in some commands that mean the definition is only ever read once.


```
#ifndef POINT_H
#define POINT_H

class point {

private:
  double x, y;

public:

   :

};

#endif
```


Make

- With lots of source files, doing all the compilation can be a pain.
  - Typically it is hard to remember which files need to be recompiled when you have made changes.
- As you can see, when we start splitting programs into separate files, we tend to end up with lots of source files.
- `make` is a Unix tool that makes managing this a bit easier.
- To use `make`, you need a file called `Makefile`.
- A sample `Makefile` is on the next slide.
- Note that a “project” in Code::Blocks and other IDEs is really no more than a `Makefile` with a fancy front-end.


```
# A simple makefile
#
# Simon Parsons
# May 31st 2009

# Default target
.PHONY: all
all: craps

# A rule to build craps
craps: craps.cpp
g++ -o craps craps.cpp

# This rule tells make to delete all the output files
.PHONY: clean
clean:
rm craps
```

- Lines beginning with # are comments.
- This is a rule:

```
craps: craps.cpp
       g++ -o craps craps.cpp
```

it says:

> When I tell you to make the *target* `craps`, you need the file `craps.cpp`, and you do the making by executing `g++ -o craps craps.cpp`

- Note that the command part is indented one tab stop.
- To execute this rule you would type:

```
make craps
```

which tells `make` to look in the `Makefile` for a rule which starts with `craps`.

- The first line of the rule:

```
craps: craps.cpp
```

tells `make` to watch the file `craps.cpp`.

- It will only do the compilation in;

```
       g++ -o craps craps.cpp
```

if `craps.cpp` has changed since the last compilation.

- Otherwise it will report:

```
make: ‘craps’ is up to date.
```


- This is another kind of rule:

```
.PHONY: all
all: craps
```

- This tells `make` that for the target `all`, that is if you type `make all`, it should make the target `craps`.
- The `all` target is typically the main thing you are writing.
- The `all` target is what gets made when you just type `make`.
- The:

```
.PHONY: all
```

says that there is no input file that is associated with the `all` target, so `make` has no need to check if the file has changed.

- The final rule:

```
.PHONY: clean
clean:
rm -f craps
```

is another typical rule to see.

- This tells `make` how to clean up all the results of compilation.
- We use `rm -f` so that we don’t get an error if the file doesn’t exist.
- Often `make` files will have an `install` target which moves the final object file to another part of the file system, and so a typical use of `make` is:

```
make all
make install
make clean
```

- The advantages of `make` are more obvious when you have lots of source files to manage.
- Look at the `Makefile` that is zipped up with the multifile examples.
- This has a rule for building:
  - `point`
  - `line`
  - `testpoint`
  - `testline`
  - `prog`
- `make` keeps track of which files need to be compiled. and when you use it to compile any target, it will automatically recompile any files that need to be recompiled.

Summary

- This lecture described some aspects of UML.
- There is a lot (looooooot) more to UML than what we covered, but what we looked at should give you an idea of how to use UML to specify classes and the relationship between classes.
- We also looked at how to split source code among several files.
- Since it is a handy tool when we have lots of source files, we also briefly discussed `make`.
- Again, there is much to know about `make` that we did not have time to go into.