SPECIFICATION AND EXCEPTION HANDLING

Today

- Today we will revise a couple of things we need for the next homework:
  - How strings are handled in C++
  - How files are handled in C++
- This material is taken from Pohl, Chapter 9 and Appendix C.
- We will also look at:
  - The use of UML for specification.
  - Exception handling.
- Pohl covers UML very briefly on page 381, and exception handling in Chapter 10.

Strings

- To deal with strings, we need to add:

  `#include<string>`

  at the start of our program.

- With that in place, we can define variables whose type is `string`:

```
string s1 = "Hello";
string s2 = "Simon";
string s3, s4;
```

- This defines `s1` to be a string variable whose value is the word `Hello`, and `s2` to be a string variable who value is the word `Simon`.
- It also defines `s3` and `s4` to be strings, but does not give them a value.

- Since `s1`, `s2`, and `s3` are variables, we can do a lot of the kinds of things we can do to other variables to them.
- We can assign values to them and print their values out.
- For example:

```
s3 = s2;
cout << s3;
```

  will generate:

```
Simon
```

Concatenation

- One operation that is specific to strings is *concatenation*
- For example:

```
s3 = s1 + s2;
cout << s3;
```

- The first line tells C++ to *concatenate* `s1` and `s2` and assign the result to `s3`.
- Thus `s3` now has the value of `s1` followed by the value of `s2`.

- When we print, we get:

```
HelloSimon
```

- There is no space because neither `s1` or `s2` has a space.

```
s3 = s1 + " " + s2;
cout << s3
```

would produce:

```
Hello Simon
```

as would

```
s1 += " " + s2;
cout << s2
```

Member functions

- In C++ a string is an instance of the class `string`.
- Thus:

  ```
  string s1;
  ```

  is just like

  ```
  point p;
  ```

- The class `string` comes with a number of member functions some of whcih we'll explore here.
- For others, see the definition of the class `string`.

- One of the most useful member functions is the function `[ ]`.
- This allows access to the characters that make up the string.

```
string message = "Greetings!"
char ch;

ch = message[4];
cout << ch;
```

will print out

```
t
```

- As with arrays, we start counting from 0.
- This will look familiar to those who were introduced to strings as arrays of characters.
- Other member functions of strings will be less familiar.

- An obvious thing to find out about a string is how long it is.

  ```
  int    len;
  string message;

  len = message.length();
  ```

  will do this for the string `message`.

- So will:

  ```
  len = message.size();
  ```

- So far as I can tell, `length` and `size` give exactly the same thing.

- In fact, `len` shouldn't be an `int`.
- We should really use:

```
string::size_type len;
```

- In other words, what gets returned by `size` and `length` is a value of type `string::size_type`.

Finding things in strings

- Often we want to look for things in a string.
- C++ has a member function to do this:

```
string::size_type pos;
pos = message.find("hello", 0);
```

`pos` gives the location of the start of the first occurence of the string `hello`.

The `0` says to start looking from the first character in `dna`. (Since the string is an array, the first character is numbered 0).

- We can also look for a single character:

```
pos = message.find('h', 0);
```

- If `message.find` doesn't find the thing we are looking for, it returns the value `dna.npos`.
- This gives us a neat way to search for things in `message`.
- We keep looking until we get `message.npos`.
- So, to count how many times we have `g` in `message`, we would do this:

```
int countG = 0;

pos = message.find('g',0);
while (pos != dna.npos)
{
    countG++;
    pos = message.find('g', pos + 1);
}
```

- This code works as follows:

    1. We look for `g` starting at the beginning of the string.
    2. If we don't get `npos` we have found a `g`, so increase the counter.
    3. Look again, starting with the character just after the one you just found.
    4. Go to 2.

- This is a common way of using a `while` loop.
- We'll see later how to use it to read a file.

Replacing part of a string

- If we want to swap one bit of a string for another, we can use `replace`.
- For example:

  ```
  message.replace(7, 4, "gbye");
  ```

  will replace the 4 characters that start in place 7 of the string `mesage` with the string `gbye`.
- This is fine if you want to swap `gbye` for `hola`, but is no good if you want to take out four characters and put in three, or take out three and put in four.

- To swap two bits of a string that aren't the same length, we have to first `erase` one and then `insert` another.
- For example:

```
message.erase(7, 4);
message.insert(7, "adieu");
```

will remove the four characters of `message` that start with the character in place number seven, and then insert the string `adieu` at the same place.

Reading in strings

- One way to read in a string from the user is

```
cin >> s3;
```

- This is fine if you want to read in strings like:

```
Hello
```

and

```
Roustabout
```

but no good if you want to read in:

```
What time is love?
```

- The problem is that `cin` stops reading at the first *whitespace*.
- So, if our program has:

```
cout << "Now type a string";
cin >> s3;
```

and the user types:

```
What time is love?
```

in response to the prompt, then `s3` will have the value `What`.

- The way around this problem is to use the function `getline`.
- There are two ways to use `getline`.
- Like this:

```
cout << "Now type a string";
getline(cin,s3);
```

  it will read everything up to the point the user hits the return key, and assign this to `s3`.
- This is fine for reading in `What time is love?`

- We can also call `getline` with a third parameter.
- This parameter is a character, called a *delimiter*, which tells `getline` when to stop reading.
- If our program has:

```
cout << "Now type a string";
getline(cin,s3,',');
getline(cin,s4,'.');
```

and the user types:

```
First we take Manhattan, then we take Berlin.
```

then …

- `s3` will have the value

  `First we take Manhattan`

  and `s4` will have the value

  `then we take Berlin`

- Note that the delimiters are not read in, and so don't end up in either string.

Files

- File handling involves three steps:
  1. Opening the file (for reading or writing)
  2. Reading from or writing to the file
  3. Closing the file
- Files in C++ are *sequential access*.
- Think of a cursor that sits at a position in the file;
- With each read and write operation, you move that cursor's position in the file

- The last position in the file is called the “end-of-file”, which is typically abbreviated as `eof`
- All the functions described on the next few slides are defined in the either the `<ifstream>` header file (for files you want to read from) or the `<ofstream>` header file (for files you want to write to)

Opening a file for reading

- First you have to define a variable of type `ifstream`
- This “input file” variable will act like the cursor in the file and will point sequentially from one character in the file to the next, as you read characters from the file
- Then you have to open the file:

```
ifstream inFile; // declare input file variable
inFile.open( "myfile.dat", ios::in ); // open the file
```

- You should check to make sure the file was opened successfully

- If it was, then `inFile` will be assigned a number greater than 0.
- If there was an error, then `inFile` will be set to 0, which can also be evaluated as the boolean value `false`; so you can test like this:

```
if ( ! inFile ) {
  cout << "error opening input file!\n";
  exit( 1 );  // exit the program
}
```

- Note that the method `ifstream.open()` takes two arguments:
  - `filename`: a string containing the name of the file you want to open; this file is in the current working directory or else you have to include a full path specification
  - `mode`: which is set to `ios::in` when opening a file for input

Reading from a file.

- Once the file is open, you can read from it
- You read from it in almost the same way that you read from the keyboard
- When you read from the keyboard, you use `cin >> ...`
- When you read from your input file, you use `inFile >> ...`
- Here is an example:

```
int x, y;
inFile >> x;
inFile >> y;
```

- Here is another example:

```
int x, y;
inFile >> x >> y;
```

- When reading from a file, you will need to check to make sure you have not read past the end of the file.
- Do this by calling:

  `inFile.eof()` which will:

  – return `true` when you have gotten to the end of the file (i.e., read everything in the file)
  – return `false` when there is still something to read inside the file.
- For example:

```
while ( ! inFile.eof() ) {
  inFile >> x;
  cout << "x = " << x << endl;
} // end of while loop
```

Opening a file for writing.

- first you have to define a variable of type `ofstream`; this "output file" variable will act like the cursor in the file and will point to the end of the file, advancing as you write characters to the file
- then you have to open the file:

```
ofstream outFile; // declare output file variable
outFile.open( "myfile.dat", ios::out ); // open the file
```

- You should check to make sure the file was opened successfully.
- If it was, then `outFile` will be assigned a number greater than 0.
- If there was an error, then `outFile` will be set to 0, which can also be evaluated as the boolean value `false`;

- You can test like this:

```
if ( ! outFile ) {
  cout << "error opening output file!\n";
  exit( 1 );  // exit the program
}
```

- Note that the method `ofstream.open()` takes two arguments:
  - `filename`: a string containing the name of the file you want to open; this file is in the current working directory or else you have to include a full path specification
  - `mode`: which is set to `ios::out` when opening a file for output
- This is rather like handling an input file, no?

Writing to a file.

- Once the file is open, you can write to it
- You write to it in almost the same way that you write to the screen
- When you write to the screen, you use `cout << ...`
- When you write to your output file, you use `outFile << ...`
- Here is an example:

```
outFile << "hello world!\n";
```

- Here is another example:

```
int x;
outFile << "x = " << x << endl;
```

Closing a file.

- When you are done reading from or writing to a file, you need to close the file
- You do this using the `close()` function, which is part of both `ifstream` and `ofstream`
- So, to close a file that you opened for reading, you have do this:

```
ifstream.close(); // close input file
```

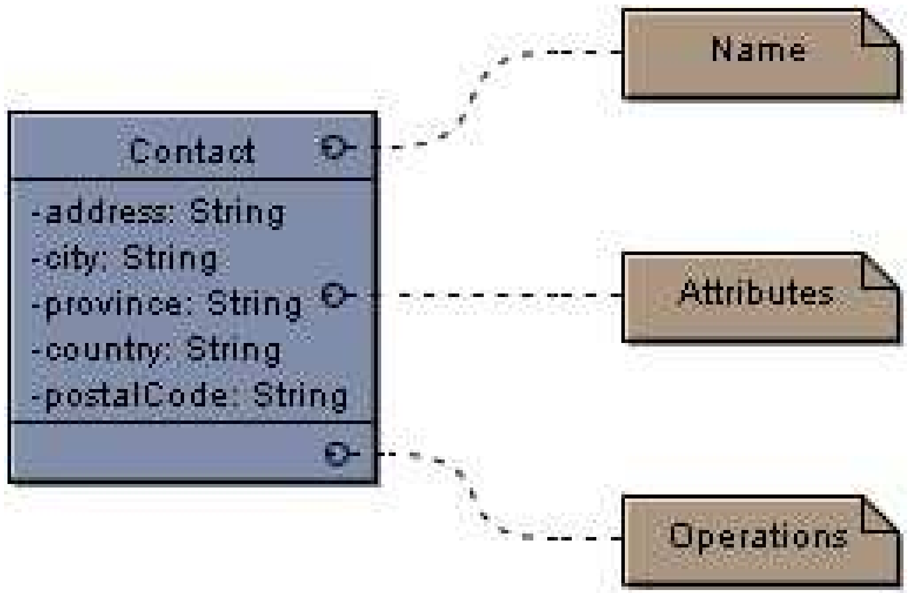
- And, to close a file that you opened for writing, you have do this:

```
ofstream.close(); // close output file
```

- That's all!

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.

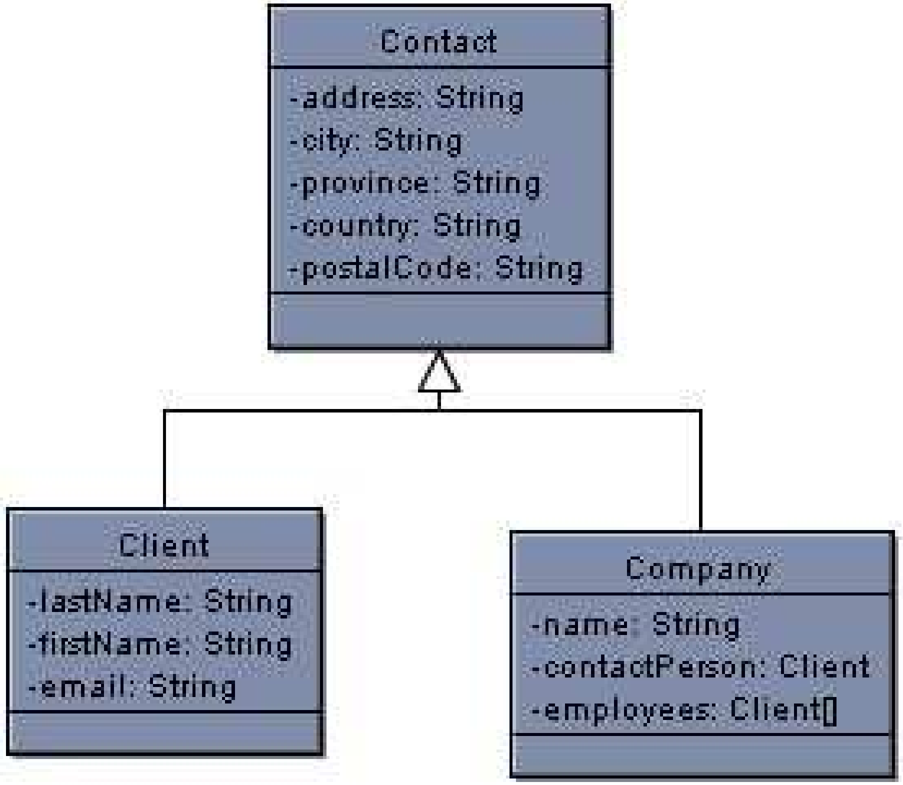
- In this case:

we have:

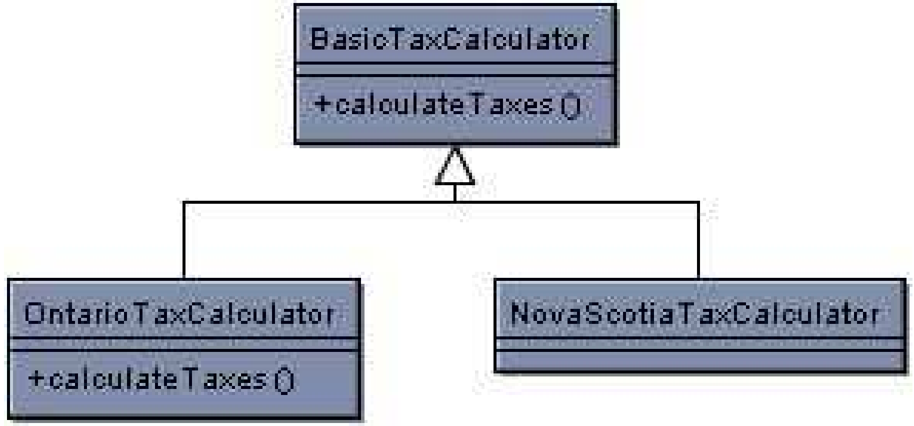
- A private attribute `object`, which is a list of `Object`s
- A public method `getSize` which and returns an integer.
- A public method `set` which is `void` and takes an `int` and an `Object` as arguments.
- A public method `get` which returns an `Object` and takes an `int` as an argument.

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

- In this example:

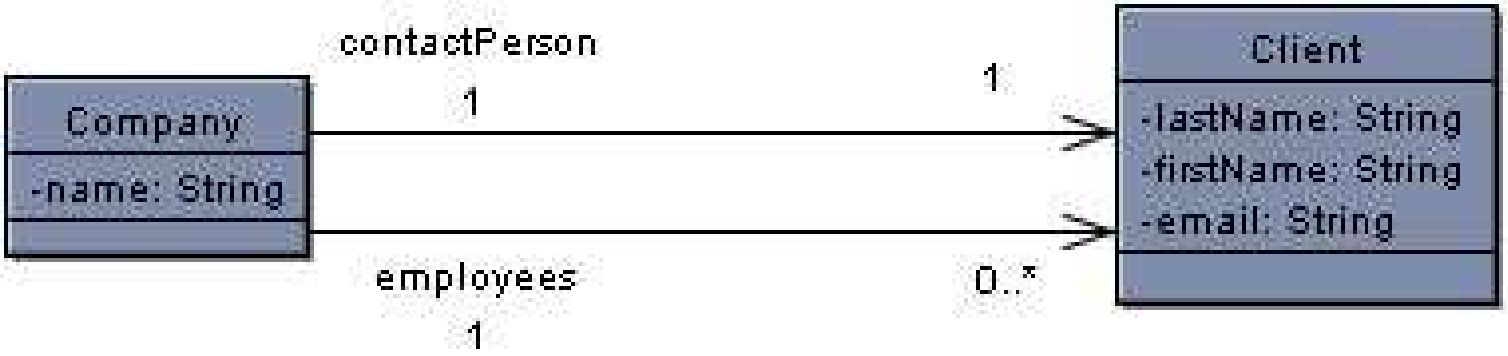


- `Client` and `Company` are both sub-classes of `Contact`.
- They each have some additional attributes on top of those in `Contact`.
- By default sub-classes inherit everything in the parent class.
- They can also provide *redefinitions* of operations.
- Here:



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.

Exceptions

- Exceptions are unexpected error conditions.
- A typical example is a “divide by zero”:

  ```
  x = y / z;
  ```

  where `z` has value 0.
- Hitting such an exception causes your program to crash.
- C++ provides some mechanisms for recovering from such exceptions.

“assert”

- The assert library `cassert` provides a way of checking the correctness of input.
- For example, in our `point` class, as used in the rabbit-world, it doesn’t make much sense to allow values of `x` and `y` that are less than zero.
- `assert` allows us to make sure that this is not the case.
- For example, we can write a new `set` method for `point`.

- Instead of:

```
void point::set(double u) {
  x = u;
  y = 0;
}
```

- we can use:

```
void point::set(double u, double v) {
  assert(u > 0);
  x = u;
  y = 0;
}
```

(see `exception.cpp`)

- If the expression in the `assert` is not true, then the program will abort.
- The idea is that if things go wrong, it is better to detect them at source rather than have to backtrack from some later point in the program where the error shows up.
- You could, of course, do the same with conditionals:

```
void point::set( double u) {
  if(u < 0){
     exit(1);
  }
  x = u;
  y = 0;
}
```

- `assert` is considered to be better style.

“try”, “throw” and “catch”

- `try`, `throw` and `catch` provide a mechanism for detecting and recovering from errors.
- For example we can change the way that we check for errors in out `point` class.
- (see `exception.cpp`)

```
void point::set( double u, double v ) {
  try{
       if(u < 0){
          throw u;
       }
       else {
          x = u;
       }
  }
  catch(double u){
    cout << "That value of x is no good" << endl;
    cout << "I'm setting x to zero" << endl;
    x = 0;
  }
}
```

- Note that we start with a `try`.
- This encloses a `throw`.
- Following the `try` and the `throw`, there is a `catch`.
- The `catch` is called an *exception handler*.
- The signature of the `catch` must match the type of the thing that is `thrown`

Rethrowing exceptions

- If the `catch` can't handle the exception on its own, then it can pass the exception to another handler.
- It does this using a second `throw`.
- The second `throw` does not need an argument.
- The second `throw` can't pass the exception to another `catch` for the same `try`.
- Instead it has to pass the exception to a `catch` associated with a `try` that encompasses the `try/thrown/catch` combination which just did the rethrow.

Multiple handlers for an exception

- A `try` block can be followed by multiple `catch`es.
- In this case, the thing that is `thrown` is tested against the `catch`es in order.
- The first `catch` that has a signature that matches the thing that is `thrown` will be executed.
- A match is when:
  - The `thrown` object is the same type as the `catch` argument.
  - The `thrown` object is of a derived class of the `catch` argument.
  - The `thrown` object can be converted to a pointer type that is the same as the `catch` argument.

- Since a `thrown` object can potentially match several different `catch`es, it is an error to order the `catch`es so that a handler will never be called.
- For example:

```
catch(void *s)
catch(char *s)
```

is not allowed, but:

```
catch(char *s)
catch(void *s)
```

is okay.

- If no matching `catch` is found, the system looks to see if the `try` block that generated the exception is nested in another `try`.
- If so, it will try to match the exception against `catch`es for the outer `try` block.
- This is the same thing that happens when you rethrow an exception.
- If no matching exception handler is found, then a standard handler is called.
- On most systems this is `terminate`.

More "catch"

- A `catch` looks like a function with one argument:

```
catch(double u){
  cout << "I'm setting x to zero" << endl;
  x = 0;
}
```

- The type of the "argument" determines whether the `catch` matches a given `throw`.
- You are allowed to have a `catch` that matches *any* argument:

```
catch(...){
  cout << "You have an error" << endl;
}
```

- That `...` is the syntax for "match anything"

“terminate”

- `terminate()` is called when there is an exception that does not have a handler.
- By default `terminate()` calls `abort()` to stop the program.
- You can redefine `terminate()` using `set_terminate()`
- You call `set_terminate()` with a pointer to the function you want `terminate()` to call when there is an exception that does not have a handler.

Exception specification

- C++ allows you to declare the kinds of exception that a function will throw:
- For example:

```
void translate() throw(unknwn_wd, bad_grammar) {
    .
    .
    <some stuff to do translation>
    .
    .
}
```

will only throw exceptions which are objects of type `unknwn_wd` and `bad_grammar`.

- Convention says that if you don't list the exception types, your function can throw any kind of exception.
- If you have a list of exception types, and your function throws another kind of exception, then this other kind of exception is caught by `unexpected`.
- By default, `unexpected` calls `terminate`.
- You can redfine what `unexpected` calls using `set_unexpected()`
- You use this just like `set_terminate()`.

Summary

- This lecture briefly recapped:
  - Strings
  - Files
- It also covered some UML.
- Finally it looked at exception handling.
- We talked about `assert`.
- Then we looked at the more flexible environment provided by `try`, `throw` and `catch`.