EXCEPTION HANDLING

Today

- Today we will look at exception handling.
- This is a gentle introduction to the idea of engineering your program so that it is robust against errors.
- It also leads naturally to testing and specification.
- Pohl covers exception handling in Chapter 10.

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.
- As the library name suggests, this is a hold-over from C.
- For example, in our `point` class, as we have been using it, 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.

- `assert` is clearly limited.
- It allows us to trap an error and quit the program, but it doesn't allow us to try and *fix* the error.
- C++ includes some features which allow us to try to correct errors.

`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 our `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` needs to come *immediately* after the `try`.
- 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 since you can only rethrow the same thing you threw before.
- The second `throw` can't pass the exception to another `catch` for the same `try`.
- Instead it has to pass the exception *out*.

- That is you can rethrown from a `catch` if and only if that `catch` is *inside* another `try`.
- The handler that `catch`s the second `throw` has to have the correct signature for the thing that was originall `thrown`.
- Look at `exception2.cpp` for an example of rethrowing.
- Rethrowing gives you a way to check a single value for two exceptions.

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 looked at exception handling.
- We talked about `assert`.
- Then we looked at the more flexible environment provided by `try`, `throw` and `catch`.
- While `assert` is simnple, `try`, `throw` and `catch` are more complex.
- Once again there is a trade-off between complexity and power — the more powerful and flexible mechanism is more complex.