

## EXCEPTIONS

---

- In our class `list`:

```
int list::car (void) const {  
    return content -> car;  
}
```

- What we do if the list is empty?
  - Ideally, our code should make sure that `car` is never called on an empty list.
  - Still, other programmers using our module may forget about this.
  - So we decide that `car` on an empty list is an error condition and we throw an **exception**.
- **Exceptions** are useful when we want a function to signal a special condition (including error conditions) but we have no room in the domain of returned values to do so by simply returning.

## USING EXCEPTIONS

---

- **Throwing:**

- When a special condition happens, a function can **throw an object**:

```
throw Object
```

- *Object* is called an **exception**.
- Just like `return`; the function stops executing.
- Not quite like `return`; the whole block that called the function stops executing and throws the object forward to its callee.

- **Catching:**

- The object is thrown on and on until
  - \* either function `main` throws it (the program prints out “Abort” and terminates),  
or
  - \* somebody **catches** the exception.

## CATCHING EXCEPTIONS

---

- If you want to catch an exception, you include the code that may throw it in a `try` statement:

```
try{
    code that might throw exceptions of type ex_t1 or ex_t2
}
catch (ex_t1 v) {
    do something useful with v, of type ex_t1
} catch (ex_t2 v) {
    do something useful with v, of type ex_t2
}
```

- Special cases:
  - `catch (...)` catches everything
  - `throw;` throws the current exception forward

- As in Java, except that
  - There are no special exception types, and no predefined exceptions. An exception is just an object, of some (i.e., no matter what) type. So you need to roll your own exceptions.
  - There is no need to say explicitly that your function throws an exception.
    - \* but saying it is a good idea (part of program self-documentation)
  - An uncaught exception does not generate a terribly useful error message.

## EXAMPLE

---

**list.h**

```
class list {
    ...
public:
    class null_exception {
        char* func;
    public:
        null_exception()
            { func = 0; }
        null_exception(char* f)
            { func = f; }
        friend ostream& operator<<
            (ostream& out,
             const null_exception e)
            { out << e.func; return out; }
    };

    list(void);
    ...
    int car(void) const;
    ...
};
```

**list.cc**

```
...
int list::car (void) const {
    if (content == 0)
        throw null_exception("list::car");
    return content -> car;
}
...
```

**function main**

```
int main () {
    try {
        list l;
        cout<<"Created an (empty) list\n";
        int i = l.car();
        cout<<"We will not reach this\n";
    }
    catch (list::null_exception e) {
        cout<<"### "<<e<<": null list\n";
    }
}
```

## VIRTUAL FUNCTIONS

---

Virtual, adj.; Being in essence or effect, not in fact; as, the virtual presence of a man in his agent or substitute.

- Say you have a class `mail` that handles the process of sending a letter (be it through Canada Post, or XpressPost, or Fedex, ...).

```
class mail {
public:
    address sender; address recipient;
    void send (void);
};
```

- How do we write `send`? We can enumerate all the possible services...

```
void mail::send (void) {
    switch(service) {
        case CANADA_POST: put_in_mailbox(); break;
        case XPRESS_POST: affix_ep_label(); put_in_mailbox(); break;
        case FEDEX: fill_in_waybill(); call_for_pickup(); break;
        // ... and so on for every imaginable service
    }
}
```

- ...but this will get quite tiresome eventually.

## VIRTUAL FUNCTIONS (CONT'D)

---

- Solution: we redefine `send` in each class that is derived from `mail`, e.g.,

```
class cp_mail: public mail {
public:
    void send (void) { put_in_mailbox(); }
};
```

- Suppose now that we want to write a function that gets the address and sends **any** kind of letter:

```
void get_address_and_send(mail& letter) {
    letter.from = my_address;
    letter.send();
}
```

- **But** then if we pass a `cp_mail` to `get_address_and_send` we get into problems.
- We need to tell to C++ “Please call the member function `send` of the derived rather than the base class.”
- We do this by using the `virtual` keyword for function `send` in the base class.

## VIRTUAL FUNCTIONS (CONT'D)

---

- Declaring a member function virtual makes C++ to use the member function of the derived class whenever possible.

```
class mail {  
    public:  
        address sender; address recipient;  
        virtual void send (void) { cout << "Don't know how to send this."; }  
};
```

- What do we put in the body of `send` in class `mail`?
  - We don't know how to send a letter if we don't know what kind of letter it is. Hence the above body.
  - The impossibility of sending a particular letter reveals itself at runtime.
  - It would be nicer if we could obtain the information of such kind of problem at compile time.
  - In other words, we would like to specify that a particular member function **must** be overridden in the derived classes.

## PURE VIRTUAL FUNCTIONS

---

- We tell the compiler that the member function `send` must be overridden in any class derived from `mail` by declaring this function **pure virtual**:

```
class mail {
public:
    address sender; address recipient;
    virtual void send (void) = 0;
};
```

- Fun with virtual functions (pure or not):

```
cp_mail m1, m2, m3;
fedex_mail m4, m5;
//... write the letters m1 to m5;
mail* mailings = { &m1, &m2, &m3, &m4, &m5 };
for (int i = 0; i < 5; i++)
    get_address_and_send(*(mailings[i]));
```