PH502: Scientific Programming Concepts

Irish Centre for High End Computing (ICHEC)

September 23, 2020

Overview

ICHEC
Irish Centre for High-End Computing

- Object-Orientated Programming is a more modern approach to programming.
- We are going to illustrate the benefits of using OOP over traditional programming methods with an example.
- We don't expect you to know how to construct an OOP but it would be remiss if it was not part of the course.

Object-Oriented Programming

ICHEC
Irish Centre for High-End Computing

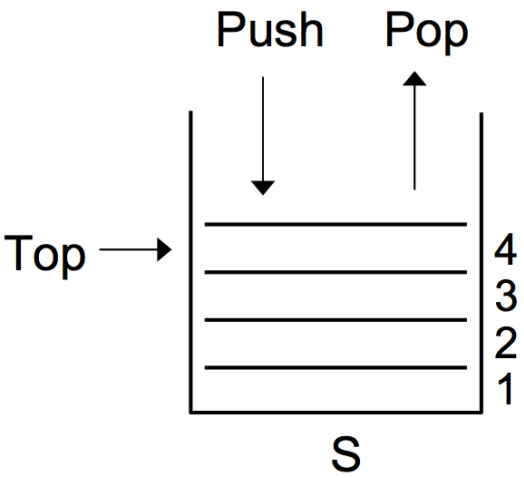
- OOP is a common programming style, offered by many modern languages.
- While C isn't object-oriented (C++ is a C-based OOP language), Fortran 2003 offers OO functionalities.
- Through examples, we will introduce problems, and use them to introduce the fundamentals of OOP.

Managing people

- Let's now pretend we're writing software to manage the people that matter in an ISP company.
- We mainly have two kinds of people to deal with: employees and customers.
  - Employees may be hired and fired. When one needs to be fired, it's the last one in gets the boot.
  - Customer helpdesk requests will appear randomly, and require said person to be called, on a first come first serve basis.
- Employees and Customers both have a name and contact details to be contacted (for help or P45). Employees also have a salary.

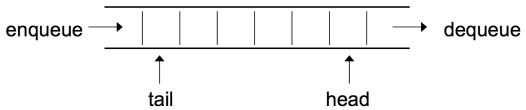
A list of employees

- Let's imagine we have a list of employees, in employment order.
- Hiring a new employee consists in 'pushing' him at the end of the list.
- When a redundancy is required, we'll 'pop' the last employee out of the list and use his contact details to notify him.
- The employee list works as a *stack*: it works on a Last In, First Out basis.



A list of customers

- Let's imagine we have a list of customers, in order of help requests reception.
- Any requests by a new customer will be 'pushed' at the end of the list.
- When a support staff is available, he'll 'pop' the first customer from the list and use his contact details to call him.
- The customer list works as a *queue*: it works on a First In, First Out basis.



Implementing people: data structure

- Let's define Employees and Customers as composite types to help our code readability.

```
type Employee:
  name
  contactdetails
  salary

type Customer:
  name
  contactdetails
```

Implementing people: employee functions

- We want to be able to create a new Employee in the system, as well as read/modify his details. We also want to be able to fire him and delete his details from the system.
- If we want to ensure some quality control, like capital letters in names, we shouldn't allow the program to directly access the data fields of our composite type. So we'll provide functions to do it.
- We'll need at least the following functions:

```
createEmployee(name, contactdetails, salary)
getEmployeeName(employee)
getEmployeeContactdetails(employee)
getEmployeeSalary(employee)
modifyEmployeeName(employee, newname)
modifyEmployeeContactdetails(employee, newdetails)
modifyEmployeeSalary(employee, newsalary)
sendP45(employee)
deleteEmployee(employee)
```

Implementing people: customer functions

- We want to be able to create a new Customer in the system, as well as read/modify his details. We also want to be able to send him answers to his queries and delete his details from the system.
- If we want to ensure some quality control, like capital letters in names, we shouldn't allow the program to directly access the data fields of our composite type. So we'll provide functions to do it.
- Sounds familiar doesn't it? However, unless you're using a language with very loose typing, Customer and Employee are two distinct types, so you can't use the same functions as for an Employee.
- We'll need at least the following functions:

```
createCustomer(name, contactdetails)
getCustomerName(customer)
getCustomerContactdetails(customer)
modifyCustomerName(customer, newname)
modifyCustomerContactdetails(customer, newdetails)
helpCustomer(customer)
deleteCustomer(customer)
```