Code Refactoring and Defects

Reusing this material

Defects Overview

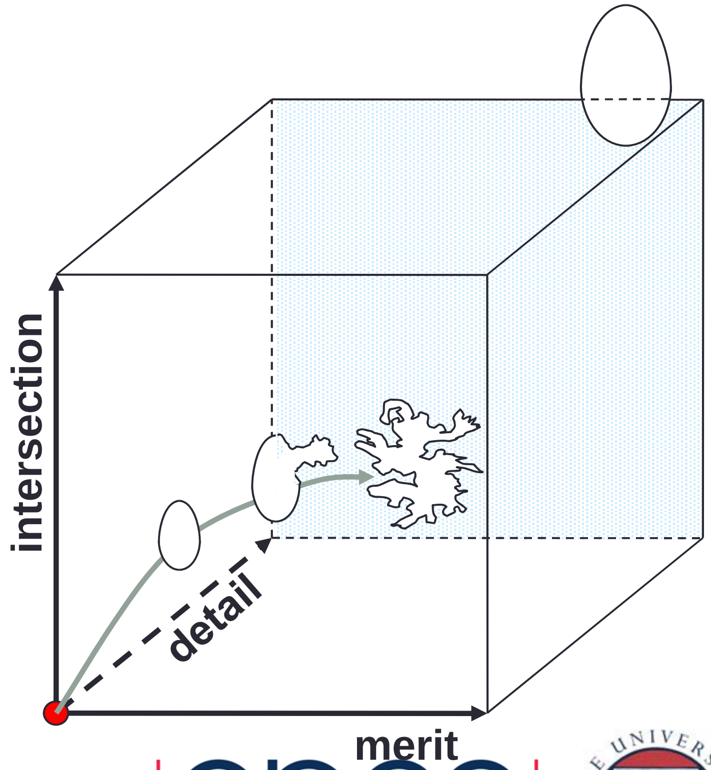
- Low complexity in software
- Bad code: ugly, inefficient, interdependent
- Interdependency
  - Complexity Explosions
- Encapsulation
  - Different levels
- Conclusions

Before Coding

- System Design should have ensured the following:
- Reasonable level of detail
- High intersection
  - what features there are should be common to a good final item
- High merit
  - low complexity (of what detail we have)
  - low difficulty of use
    - main features of the user interface should have been planned
  - low runtime and space
    - main data structures and algorithms should have been planned

The Code Design Challenge

- Complexity can explode as detail increases
- "The more code you add, the worse things get"
  - bad code starts to appear
  - in the short term it's the easy option
  - get even worse code when you build on top of that
- What's the solution?
  - identify and tackle bad code before it cripples the project
- What is bad code?



What is Bad Code?

- Regardless of whether the code behaves correctly, it can still be 'bad' in a number of ways and lead to 'scrambled egg'
- 1. Ugliness or sloppiness
- 2. Inefficiency
- 3. Interdependency

1. Ugly or Sloppy Code

- No comments or poor comments
- Cryptic naming
- Messy layout
  - inconsistent indentation or spacing
- Huge functions
  - more than half a page is probably too long
- Makes extension, debugging and maintenance difficult

The Cure

- Decide on coding style guidelines up front
  - Such as Code Conventions for the Java Programming Language
    - Sun’s own code style guide
- Tidy it up according to the guidelines
  - Makes maintenance easier
  - Can identify problems just by looking at the code

2. Inefficient Code

- Passing large arguments by value
  - More efficient memory use, little copy overheads
- Data duplication
  - same data stored in multiple parts of the program
  - synchronisation overheads
- Poorly designed algorithms
  - Inefficient or unneeded calculations or searches

The Cure

- Identify problems by observing performance and profiling
  - Profile the memory usage to identify problems
  - Profile the code to identify bottlenecks
- Design better algorithms
  - Do use existing efficient algorithms
  - Do not take as ‘gospel’ that they are perfect
- Measure improvement
  - Record and replicate all tests
  - Show how changes should and do affect performance

3. Interdependent Code

- Not all dependencies are bad
  - if a function calls other functions, that's good!
  - building functions out of lower-level functions keeps bugs down
    - fix the low-level function and all its callers become fixed too
- Nasty dependencies in programs:
  - code dependent on data being in a certain state
  - code dependent *implicitly* on other code

```
char** buffer = new char*[45];    // Creates an array of 45 string pointers.
…
void printBuffer() {
  for (int i = 0; i < 45; i++) …    // Assumes that the size of the array is 45.
}
```

Dependencies

- Nasty dependencies can be very hard to identify
  - by their implicit nature
  - makes them hard to identify and pervasive
- Nasty dependencies are very bad for a project
  - a change can cause major ripple effects in the software
  - leads to unreliability, inconsistency
  - makes extension very difficult
  - can destroy your schedule and your project!
- Examples then solutions

Dependency Examples

- Convention dependency
  - Positive account numbers belong to individuals, negative ones belong to companies
  - Get lots of these sprinkled through the code

    if (order.accountNumber < 0) then … else ...

    – If you change the convention somewhere, have to change it everywhere
    – Conventions may not be obvious (implicit)
    – Conventions may not be documented at all

Dependency Examples

- Implementation Dependency
  - dealing with raw data structures introduces dependencies on the current implementation

```
int* rainfallSeq;   // Array of ints.          int rainInMarch = rainfallSeq[2];
```

  – client code depends on the rainfall sequence being an array
  – a real pain if you decide to use a linked list to store the rainfall sequence

Dependency Examples

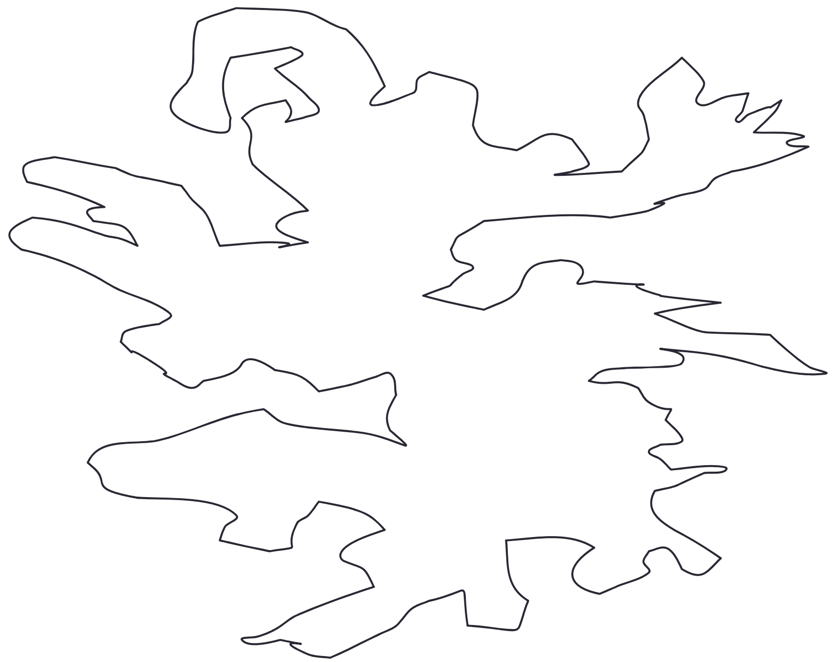
- Behaviour dependency
  - if two functions are expected to have some aspect of their behaviour in common, then that aspect had better stay the same

```
void printNames() {
  // Print out all the names.
  for (int i = 0; i < numPeople; i++) {
    printf("Name is %s \n", names(i));
  }
}
```

```
void printNameAndAge(int i) {
  // Print out the name and age of
  // the person at the argument index.
  printf("Name is %s \n", names(i));
  printf("Age is %d \n", ages(i));
}
```

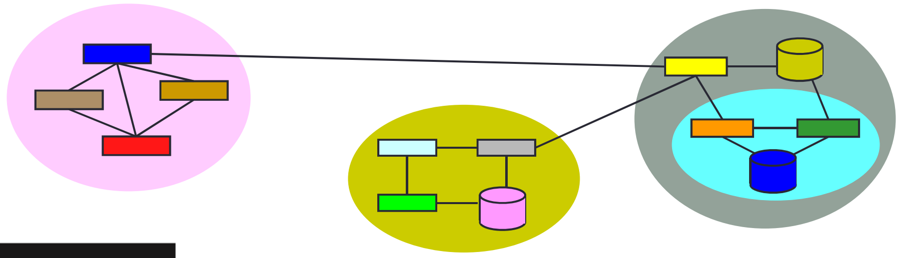
Dependencies

- A single dependency on its own may not look very threatening
  - but if you let them proliferate, things go downhill rapidly
- What's the cure?



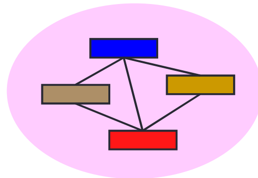
The Cure

- Encapsulation
  - “the grouping of related ideas into one unit, which can thereafter be referred to by a *single name*”
- Group related software elements (code and/or data) that have some common purpose or destiny
- Group so that:
  - dependencies are collected together inside encapsulation boundaries
  - dependencies across encapsulation boundaries are minimised



Some Benefits of Encapsulation

- Small parts are combined into a larger whole which has *meaning*
  - reduces complexity
  - refer to the whole instead of the parts

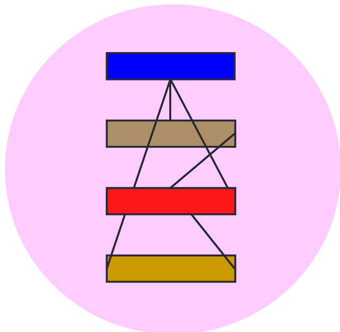


- By grouping dependencies inside encapsulation boundaries they become *explicit*
  - makes them easier to spot and less pervasive
- Reduced dependencies across encapsulation boundaries
  - fewer knock-on effects of a code change

Low Level Encapsulation

- There are various levels of encapsulation
- Dependency examples seen so far can be cured with low level encapsulation
  - encapsulating a constant into a name
  - encapsulating an idea into a function
  - encapsulating related functions into another function

lines of code
grouped into
a function



Curing Dependency

- Array example
  - encapsulate the idea of a buffer size with a meaningful name then refer to it explicitly

  - Using a meaningful name helps explain the purpose of a value

```
char** buffer = new char*[45];
…
void printBuffer() {
  for (int i = 0; i < 45; i++) …
}
```

```
bufSize = 45
char** buffer = new char*[bufSize];
…
void printBuffer() {
  for (int i = 0; i < bufSize; i++) …
}
```

Curing Dependency

- Convention dependency
  - positive account numbers belong to individuals, negative ones belong to companies

if (order.accountNumber < 0) then … else ...

– encapsulate the underlying meaning into a function and refer to it explicitly

if (isCompany(order)) then … else ...

– Allows the convention to be changed to reflect new requirements without affecting existing code or function

Curing Dependency

- Implementation Dependency
  - dealing with raw data structures introduces dependencies on the current implementation

```
int* rainfallSeq;   // Array of ints.
```

```
int rainInMarch = rainfallSeq[2];
```

– encapsulate the monthly rainfall concept into a function

```
int* rainfallSeq;   // Array of ints.
…
int rainInMonth(int month) {
  if (month < 1 or month > 12) then error;
  return rainfallSeq[month - 1];
}
```

```
int rainInMarch = rainInMonth(3);
```

Curing Dependency

- Behaviour dependency

```
void printNames() {
  // Print out all the names.
  for (int i = 0; i < numPeople; i++) {
    printf("Name is %s \n", names(i));
  }
}
```

```
void printNameAndAge(int i) {
  // Print out the name and age of
  // the person at the argument index.
  printf("Name is %s \n", names(i));
  printf("Age is %d \n", ages(i));
}
```

  – as seen earlier, encapsulate the common code into a printName(…) function then call it from both
  – removes code duplication and scope for inconsistency
  – Makes changes to functions easier

Good Low Level Encapsulation

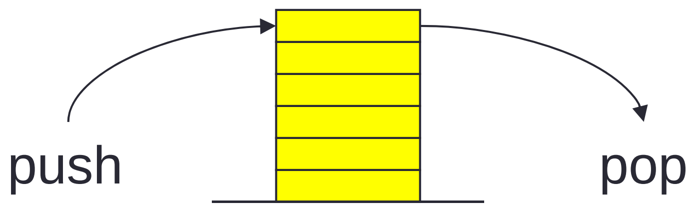
- Naming is important
  - the name of a function should capture the idea being encapsulated
  - the function's comment should capture the programmer's intent
- Avoid dependencies across encapsulation boundaries
  - minimise side-effects of functions or modification of global variables within functions
  - write functions which take arguments rather than reading global variables
  - Keep global variables to a minimum

Code-Data Dependency

- Will low level encapsulation solve all our dependency problems?
  - No
- Critically, it doesn't address data protection issues
  - huge source of problems in many programs
- Need higher level encapsulation

Motivation

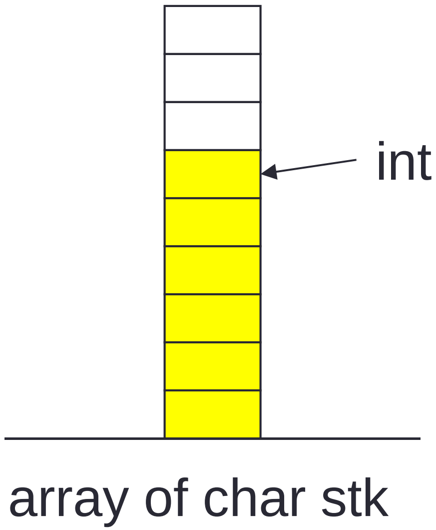
- Consider a classic data structure, the stack



- Very useful for bracket matching for example
  - [[[a + b] * 4] + [c * d]]
  - push opening brackets onto the stack
  - pop a bracket each time you discover a closing bracket
  - correct matching if you end with an empty stack

Traditional Implementation

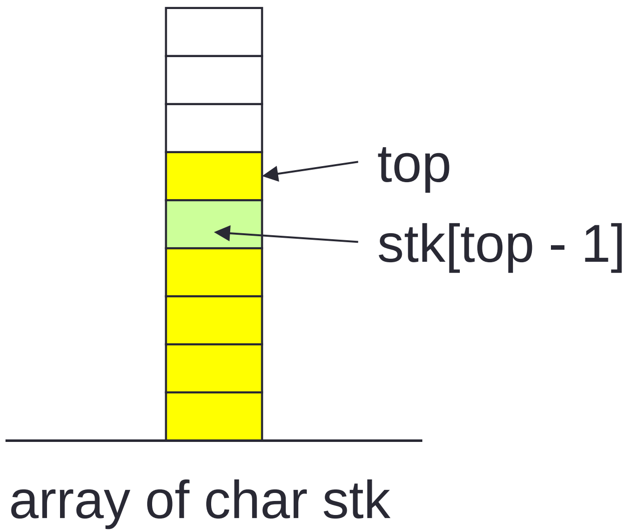
- Implemented using an array and a top index




```
void push(char c) {
  # Add argument character to the stack.
  top = top + 1
  stk[top] = c
}

char pop() {
  # Remove top character and return it.
  if (top < 0) then error
  char res = stk[top]
  top = top - 1
  return res
}
```

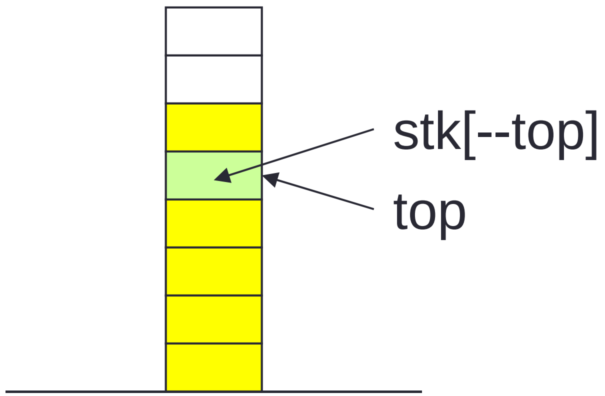
Problems with Public Variables

- What if people are interested in the second-top character?
  - easy, they can just use stk[top - 1]



Problems with Public Variables

- What if new boy Johnny uses stk[--top] by mistake?



- He has used the decrement operator (--) so his function will give the right answer (once), but after that the stack is corrupt!

Problems with Public Variables

- The symptom could appear in one of *your* functions!
- You may have to go and debug it
- Johnny has messed up *your* afternoon!
- He hasn't touched a single line of your code, but has still managed to make it misbehave
  - by tampering with public data
  - By not understanding the ideas in use

Too Much Rope

- Why has this waste of time been allowed to happen?
- Programming languages are almost *too* flexible
- They invite quick dirty hacks to solve small problems
  - quicker to shove in stk[top - 1] than write a function everyone can use
- This approach collapses in larger software
  - complexity becomes unmanageable
- Need to impose additional structure and restrictions

Private Data

- There is no reason why stk or top should be public to the whole program
- They should be made *private* to the stack functions
  - make variables accessible only by those functions that have a fundamental right to know about them
- The compiler will complain if an attempt is made to access them from elsewhere
  - Johnny's erroneous modification wouldn't even have compiled

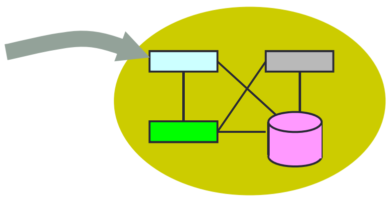
Public Interface

- If someone has a legitimate need to know the second-top character, add a function to the public interface

```
char secondTop() {
  # Return the character which is one below the top one in
  # the stack, or NULL if there are <=1 characters in the stack.
  if (top < 1) then return NULL
  return stk[top - 1]
}
```

High Level Encapsulation

- Means code and data is grouped together into an ‘abstract data type’ (ADT)
- Data is made private to the functions of the ADT
- Access to the data from outside the ADT is provided via the ADT’s public interface
- An ADT’s interface can be updated and its function extended



functions and data grouped into an ADT

Bad Code Summary

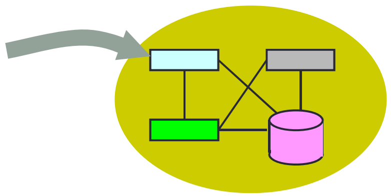
- Try to keep complexity low as coding progresses
  - bad code leads to complexity explosion
- Several types of bad code
  - Lots of different forms of bad code
- Interdependency can be the most sinister
  - because of its subtlety and pervasive effects
- Encapsulation
  - protects against dependencies by making them explicit
  - other benefits: conceptual clarity, reduced scope for bugs etc.
  - low level - grouping lines of code into functions
  - high level - grouping functions and data into ADTs

ADT Overview

- Abstract Data Types
- Advantages and disadvantage
- Implementation
  - general strategy
  - examples in C and Fortran 90
- A word on Object Orientation
- Conclusions

Abstract Data Types

- An Abstract Data Type ties code and data together into a coherent logical unit
- An ADT is a user-defined type such that
  - private data is hidden from external functions
  - external functions can only access the data via a public interface



- ADTs can be implemented in most programming languages

Abstract Data Types

- Why 'Abstract'?
  - they allow you to 'abstract away' from the lower level detail
- ADT concept used successfully for years
  - but still not as prevalent as it could/should be
- It's a genuine shift in programming emphasis away from logic and towards data
  - as a program gets bigger it's the complexity of the data which tends to cause the worst problems
  - ADTs are a good (only?) way to manage that data complexity

Advantages of ADTs

- They reduce the *scope* for bugs, as seen already
  - it is impossible to corrupt a stack via its public interface
  - you can't access the private data directly to mess it up
- Ease of re-use

```
# Stack for bracket matching.
Stack s
push(s, bracket1)
push(s, bracket2)

# Stack for brace matching.
Stack t
push(t, brace1)
push(t, brace2)
```

Advantages of ADTs

- Interfaces
  - there is now a clean interface between the data and the rest of the program
  - if the implementation needs to change, you change it in one place only (the ADT), while the interface stays the same
- Opens the possibility for correctness proofs
  - if you can define axioms for your functions

Disadvantages of ADTs?

- Accessing data via functions is slower than doing it directly
- But…
  - the overhead is small
  - the saving in development and debugging time for an ADT-based program will usually dwarf any runtime penalty
  - lots of programs spend 90% of the time in 10% of the code

Implementing ADTs

- A notation for ADTs (UML)
  - minus signs mean ‘private’
  - from here on assume all ADT data is private

Stack

private data / attributes —
-stk: Array of char
-top: int

public interface —
new(): Stack
push(s: Stack, c: char)
pop(s: Stack): char
top(s: Stack): char
secondTop(s: Stack)
isEmpty(s: Stack): Boolean
size(s: Stack): int

private interface —
-grow(s: Stack)

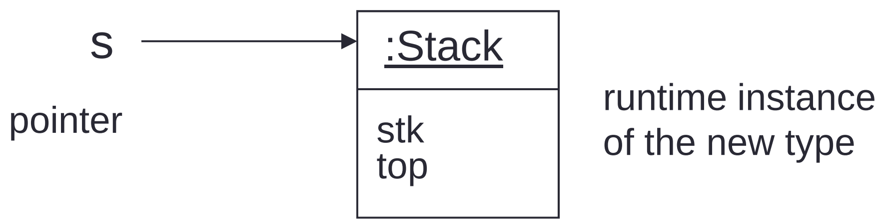
:Stack

-stk
-top

instance of the ADT at runtime

Implementing ADTs

- The Data
  - define a new type (or struct in C) to represent the data of the ADT
  - create instance(s) of that type at runtime
  - refer to instances of the type using pointers (or equivalent)
    - cheaper to pass the pointer around than the thing it points to



Implementing ADTs

- The Functions
  - arrange for the contents of the new type to be accessible only by the ADT's functions
  - other functions cannot access the contents directly
  - they must go via the ADT's public interface

Implementing ADTs

- Some programming languages are better suited to ADTs than others
  - Stack example implemented in C and Fortran 90
  - object-oriented languages are very well-suited to ADTs

Implementing Stack in C

Code File: Stack.c

- define the stack functions
- add a function that creates a *struct Stack* (using malloc) and returns a pointer to it
- the other functions take a pointer to a *struct Stack*

Header File: Stack.h

- declare the stack data as a *struct* called Stack
- declare the Stack functions

Implementing Stack in C

Code File: Stack.c

```
#include "Stack.h"

struct Stack* stackNew() {
  /* Create a new stack and return */
  /* a pointer to it. */
  ...
}

void stackPush(struct Stack* s, char c) {
  /* Push the argument character */
  /* onto the argument stack. */
  ...
}

...
```

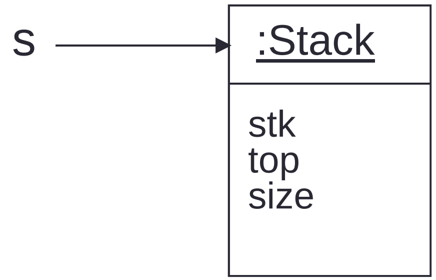
Header File: Stack.h

```
/* This module implements a stack */
/* of characters using an array. */

/* Private data. */
struct Stack {
  char* stk;   /* Array for the stack. */
  int     top;  /* Index of top element. */
  int     size; /* Size of the array. */
}

/* Public interface. */
struct Stack* stackNew(void);
void stackPush(struct Stack* s, char c);
…

/* Private interface. */
static void stackGrow(struct Stack* s);
```

Implementing Stack in C

Code File: Stack.c

```
struct Stack* stackNew() {
  /* Create a new stack and return a pointer to it. */
  struct Stack* s = (struct Stack*)malloc(sizeof(struct Stack));
  s->size = 1;
  s->stk = (char*)malloc(s->size * sizeof(char));
  s->top = -1;
  return s;
}
…

void stackPush(struct Stack* s, char c) {
  /* Push the argument character onto the argument stack. */
  s->top++;
  if (s->top >= s->size) stackGrow(s);
  s->stk[s->top] = c;
}
...
```



Using an ADT in C

main.c

```
#include "Stack.h"

main(int argc, char *argv[])
{
  struct Stack* s = stackNew();
  stackPush(s, 'Y');
  stackPush(s, 'o');
  stackPrint(s);
  stackPop(s);
  stackPrint(s);
  stackDelete(s);
}
```

output

```
Yo
Y
```

Problems of Using C for ADTs

- In C there is no mechanism for making the contents of the struct Stack strictly private
  - so have to rely on programmer discipline
  - i.e. the one thing we didn't want to rely on!
- Also, what if you want a stack of integers, say?
- For the int case, duplicate the code but...
  - append 'I' (say) to the Stack type and all its functions
  - replace char with int
- This is nasty
  - code duplication always increases the scope for bugs
  - behaviour dependency

Benefits of Using C for ADTs

- So why bother?
- Even with C, adopting an ADT style still gives important benefits
  - conceptual benefits of encapsulation
  - coherent structure
  - ease of re-use
  - interfaces insulate you from changing implementation details
  - fewer bugs, easier to fix
  - easier to extend the program

Stack in FORTRAN 90

- F90 is very well-suited to ADTs
  - define a MODULE for the Stack
  - define a new TYPE for the Stack's data
  - use PRIVATE to restrict access
  - implement the Stack's functions as SUBROUTINEs or FUNCTIONs
  - add a function to create a Stack using ALLOCATE
- Implementing a stack of characters as an array in F90 is perhaps not very realistic
  - but it's worth seeing an ADT in practice

Stack in FORTRAN 90

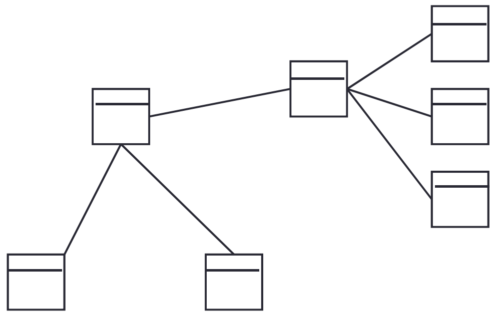
Stack.f90

```
MODULE StackMod
IMPLICIT NONE

PRIVATE :: stackGrow  ! This fn will be private
                       ! to the ADT.

! Private data.
TYPE Stack
  PRIVATE
  CHARACTER, POINTER :: stk(:)  ! Pointer to
                                ! an array of characters.
  INTEGER  top                  ! Index of the topmost char.
  INTEGER  size                 ! Size of the array.
END TYPE Stack

CONTAINS
```

```
! Public Interface.

SUBROUTINE stackNew(s)
  ! Allocate and initialise a new stack.
  TYPE(Stack), INTENT(INOUT) :: s

  s%size = 1
  ALLOCATE(s%stk(s%size))
  s%top = 0
END SUBROUTINE stackNew
...
SUBROUTINE stackPush(s, c)
  ! Push argument character onto the stack.
  TYPE(Stack), INTENT(INOUT) :: s
  CHARACTER, INTENT(IN) :: c

  s%top = s%top + 1
  IF( s%top .gt. s%size ) CALL stackGrow(s)
  s%stk(s%top) = c
END SUBROUTINE stackPush
...
```

Building Entire Programs with ADTs

- ADTs aren't just for basic data structures like Stack
- Ideally, every part of the program should be an ADT of some kind
- At runtime the program is a graph of ADT instances pointing to other ADT instances



A Word on Object-Orientation

- OO languages (Smalltalk, Java, C++) build on the ADT concept
  - they provide a slightly more advanced form of high level encapsulation
- Fundamentally, OO is ADTs with better syntax
  - a ‘class’ is an abstract data type
- But inheritance allows easy customisation of ADTs
  - e.g. define a Car ADT as a specialisation of a Vehicle ADT
- (Most) OO languages also solve the ‘stack of ints’ problem in C mentioned earlier
- Well worth learning about

ADT Summary

- Abstract Data Types really can help you
  - conceptual clarity, ease of re-use, reduced scope for bugs, use of interfaces etc.
- Relatively straightforward to implement in most languages
- It's only a small step from ADTs to object orientation
- See the handout for C and F90 implementations of Stack

References

- For encapsulation and interdependency:
  - Page-Jones, M., *Fundamentals of Object-Oriented Design in UML*, 2000, Addison-Wesley
- Code Style Guidelines
  - Code Conventions for the Java Programming Language
    - Sun Microsystems, Inc., 1999
  - Ada 95 Quality and Style Guide: Guidelines for Professional Programmers
    - Software Productivity Consortium, 1995
- For object orientation:
  - Page-Jones, M., Fundamentals of Object-Oriented Design in UML, 2000, Addison-Wesley