

OCR Computer Science A Level

2.2.1 Programming Techniques Intermediate Notes



Specification:

2.2.1 a)

- **Programming constructs**
 - Sequence
 - Iteration
 - Branching

2.2.1 b)

- **Recursion**
 - How it can be used
 - How it compares to an iterative approach

2.2.1 c)

- **Global and local variables**

2.2.1 d)

- **Modularity, functions and procedures**
 - Parameter passing by value
 - Parameter passing by reference

2.2.1 e)

- **Use of an IDE to develop / debug a program**

2.2.1 f)

- **Use of object-oriented techniques**



Programming Constructs

A crucial part of solving a problem is simplifying it to represent it in a way that makes it easier to understand and thus program. The following constructs are used to represent a program's **control flow**:

- **Sequence**

Code is executed **line-by-line**, from top to bottom.

- **Branching**

A certain block of code is run **if a specific condition is met**, using IF statements. This is also known as 'selection'.

- **Iteration**

A block of code is executed a **certain number of times** or **while a condition is met**. Iteration uses FOR, WHILE or REPEAT UNTIL loops. Iteration can be either:

- Count-controlled

Iteration is repeated a given number of times.

```
for i in range (0,10):  
    print i  
next i
```

- Condition-controlled

Iteration continues until a given condition is met.

```
while i <= 20:  
    print "Not true";  
    i=i+1  
endwhile
```



Recursion

Recursion is a programming construct in which a **subroutine calls itself** during its execution. This continues until a certain condition - called the **stopping condition** - is met, at which point the recursion stops.

The advantage of using recursion for certain problems is that they can be represented in **fewer lines of code**.

Some problems can also be **more easily expressed** using recursion over iteration.

A common example of a naturally recursive function is factorial, shown below:

```
function factorial(number)
  if number == 0 or 1:
    return 1
  else:
    return number * factorial(number - 1);
  endif
end function
```

Each time the function calls itself, a new **stack frame** is created within the **call stack**, where **parameters**, **local variables** and **return addresses** are stored. This continues until the stopping condition, or **base case**, is reached at which point the subroutine **unwinds**. This refers to the process of information from the call stack being popped off the stack.

The biggest downside to recursion is its inefficient use of memory. If the subroutine calls itself too many times, there is a danger of a **stack overflow**, which is when the **call stack runs out of memory**. This would cause the program to crash. Another problem is that recursion is **difficult to trace**.

Synoptic Link

Recursion is an example of the **divide and conquer** technique which is discussed further in **2.2.2**.

Call Stack

A **'Last In First Out'** data structure which stores details about the order in which subroutines have been called



Global and Local Variables

Variables can be defined with either global or local scope. **Scope** refers to the **section of code in which the variable is available**.

Local variables have limited scope which means that they can only be **accessed within the subroutine in which they were defined**. Therefore, multiple local variables with the same name can exist in different subroutines. Using local variables is considered to be good programming practice because it ensures subroutines are **self-contained**, with no danger of variables being affected by code outside of the subroutine.

Synoptic Link

The idea of **self-contained subroutines** follows the principle of **encapsulation in object-oriented programming**, encountered in 1.2.4.

Global variables, on the other hand, can be **accessed across the whole program**. These are useful for values that need to be used by multiple parts of the program. However, using global variables is not recommended because they can be **unintentionally overwritten**. As global variables are not deleted until the program terminates, they **require more memory** than local variables which are deleted once the subroutine has been completed.

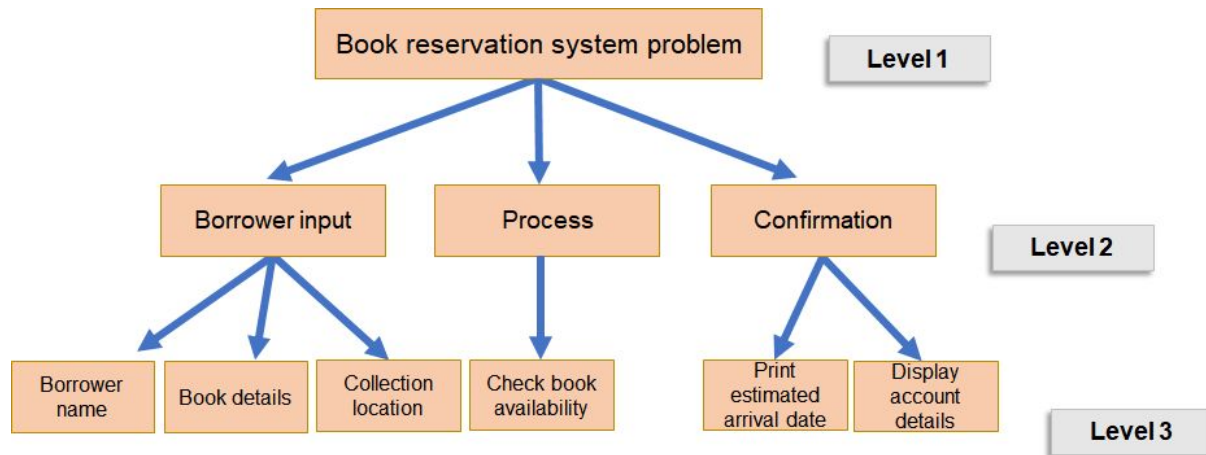
In the event that a local variable exists within a subroutine with the same name as a global variable, the local variable will take precedence.

Modularity, Functions and Procedures

Modular programming is a technique used to **split large, complex programs into smaller, self-contained modules**. A **modular design** also makes it easier to **divide tasks between a team** and manage projects, whilst simplifying the process of testing and maintenance, as each component can be **dealt with individually**. This improves the **reusability** of components, as once a module has been tested, it can be reused with confidence.

A popular technique used to modularise programs is **top-down design**, in which the problem is **continually broken down into sub-problems**, until each can be represented as an **individual, self-contained module which performs a certain task**. This process is also called **stepwise refinement**. These modules form blocks of code called **subroutines**, which can be categorised as either functions or procedures. Below is an example of a top-down design for a problem involving a book reservation system:





Procedures and functions are both **named blocks of code that perform a specific task**. While **procedures do not have to return a value**, **functions must always return a single value**.

The subroutine below is an example of a function as it always returns a value of either True or False regardless of the input.

```

function isEven(number):
    if number MOD 2 = 0:
        return True
    else:
        return False
end function
  
```

When parameters are passed into a subroutine, they can be passed either **by value** or **by reference**. When a parameter is passed by value, a **copy of the value** is passed to the subroutine and discarded at the end. Therefore its value outside of the subroutine remains unaffected. Passing by reference means that the **address of the parameter** is given to the subroutine, so the value of the parameter will be **updated at the given address**.

Parameters

Values passed into a function.

In exam questions, you should assume parameters are passed by value unless you are told otherwise. The following format will be used:

```

function multiply(x:byVal, y:byRef)
  
```



Use of an IDE

An **Integrated Development Environment**, or IDE, is a **program** which provides a **set of tools** to make it easier for programmers to **write, develop and debug code**.

Common features of IDEs include:

- **Stepping**
This allows you to **monitor the effect of each individual line of code** by executing a single line at a time.
- **Variable watch**
This allows users to observe how the **contents of a variable change** in real-time.
- **Breakpoint**
IDEs allow users to **set a point in the program at which the program will stop**.
- **Source code editor**
The editor provides features such as **autocompletion** of words, **indentation**, syntax **highlighting** and automatic bracket completion.
- **Debugging tools**
Some IDEs also provide **run-time detection of errors** with a guide as to where in the code they are likely to have occurred.

Use of object-oriented techniques

Object-oriented languages are built around the idea of classes. A **class** is a **template for an object** and defines the **state and behaviour of an object**. State is given by **attributes** while behaviour is defined by **methods**.

Classes can be used to create objects by a process called **instantiation**. An **object** is a **particular instance of a class**, and a class can be used to create multiple objects with the same set of attributes and methods.

In object-oriented programming, **attributes are declared as private so can only be altered by public methods**. This is called **encapsulation**. Encapsulation is a technique used throughout programming to implement the principle of **information hiding**. This is when programs are made **less complex** by **protecting data from being accidentally edited** by other parts of the program. Top-down design implements the same principle of encapsulation, as each module is designed to be self-contained.

