OCR A-Level Computer Science Spec Notes

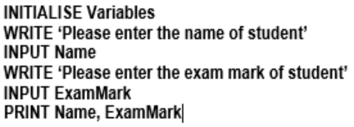
2.2 Problem solving and programming

2.2.1 Programming techniques

(a) Programming constructs: Sequence, iteration, branching

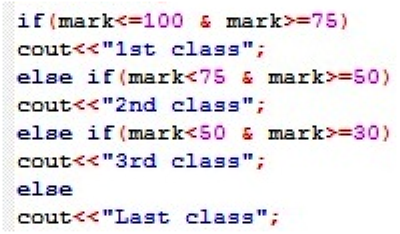
Programming Constructs (Methods of writing code**):**

- **Sequence**
  - **Series of statements** which are **executed one after another**
  - Most common programming construct



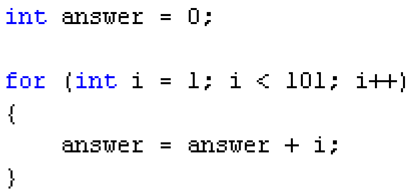
```
INITIALISE Variables
WRITE 'Please enter the name of student'
INPUT Name
WRITE 'Please enter the exam mark of student'
INPUT ExamMark
PRINT Name, ExamMark
```

- **Branching/Selection**
  - Decisions made on the state of a **Boolean expression**
  - Program **diverges** to another part on program based on whether a **condition** is **true or false**
  - **IF statement** is a common example of Selection

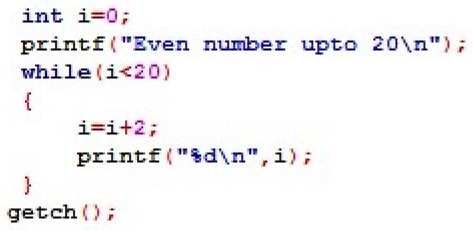


```
if(mark<=100 & mark>=75)
cout<<"1st class";
else if(mark<75 & mark>=50)
cout<<"2nd class";
else if(mark<50 & mark>=30)
cout<<"3rd class";
else
cout<<"Last class";
```

- **Iteration**
  - = **repetition**. A section of code **repeated** for a **set amount of time / until condition is met**
  - Loop: When a section of code is repeated
  - Example of **For Loop ->**
  - Example of **While Loop** ↓



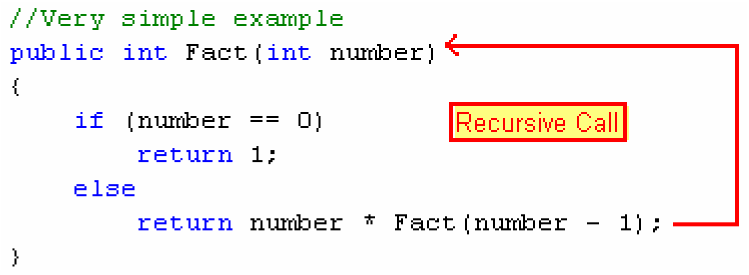
```
int answer = 0;

for (int i = 1; i < 101; i++)
{
    answer = answer + i;
}
```



```
 int i=0;
 printf("Even number upto 20\n");
 while(i<20)
 {
      i=i+2;
      printf("%d\n",i);
 }
getch();
```

(b) Recursion, how it can be used and compares to an iterative approach

- **Subroutine/Subprogram/Procedure/Function** that **calls itself**
- Another way to produce **iteration**



```
//Very simple example
public int Fact(int number)
{
    if (number == 0)
        return 1;
    else
        return number * Fact(number - 1);
}
```

(c) Global and local variables

Variables: Named locations that store data in which contents can be changed during program execution

- Assigned to a **data type**
- **Declared/Explicit statement**

Global Variables

- **Defined/declared** outside **subprograms** (Functions/Procedures etc)
- Can be **'seen' throughout** a program
- Hard to **integrate** between **modules**
- **Complexity** of program **increases**
- Causes **conflicts** between names of other **variables**
- **Good programming practice** to not use global variables (**Can be altered**)

Local Variables

- Declared in a **subroutine** and **only accessible within** that subroutine
- Makes **functions/procedures reusable**
- Can be used as a **parameter**
- **destroyed/deleted** when **subroutine exits**
- **same variable names** within two different modules will not **interfere** with **one another**
- Local variables **override global variables** if they have the **same name**

(d) Modularity, functions, procedures, parameters

Modularity: Named locations that store data in which contents can be changed during program execution

- Program **divided** into **separate tasks**
- Modules **divided** into **smaller modules**
- Easy to **maintain, update and replace** a part of the system
- Modules can be **attributed** to different **programmers strength**
- Less code produced

Functions

- Subroutine/subprogram/module/named sub-section of program/block which most of the time **returns a value**
- Performs **specific calculations & returns a value of a single data type**
- Uses **local variables** & is used commonly
- Value returned **replaces function call** so it can be used as a **variable** in the **main body of a program**

Procedures

- Performs **specific operations** but **don't return a value**
- Uses **local variables**
- **Receives** & usually accepts **parameter values**
- Can be called my **main program**/another **procedure**
- Is used as any **other program instruction** or **statement** in the main program

Parameters

- **Description/Information** about **data supplied** into a **subroutine** when called
- May be given **identifier/name** when called
- Substituted by **actual value/address** when called
- May **pass values** between **functions & parameters** via **reference/ by value**
- Uses local variables

Passed by Value:

- A copy is made of the actual value of the **variable** and is passed into the procedure.

- Does not change the **original variable value**.
- If changes are made, then only the **local copy** of the **data** is **amended** then **discarded**.
- No **unforeseen effects** will occur in other modules.
- Creates new **memory space**

Passed by Reference:

- The **address/pointer/location** of the value is passed into the **procedure**.
- The **actual value** is not **sent/received**
- If changed, the **original value** of the **data** is also **changed** when the **subroutine ends**
- This means an **existing memory space** is used.

(e) Use of an IDE to develop/debug a program

IDE (Integrated Development Environment) contains the tools needed to **write/develop/debug a program**. Typical IDE has the following tools:

- **Debugging tools**
    - **Inspection** of variable names
    - **Run-time detection** of errors
    - Shows **state of variables** at where **error occurs**
- **Translator diagnostics:**
    - Reports **syntax errors**
    - Suggests **solutions** & informs programmer to **correct error**
    - Error message can be **incorrect/misinterpreted**
- **Breakpoint:**
    - Tests program at **specified points/lines of code**
    - Check **values** of **variables** at that **point**
    - Set **predetermined point** for program to **stop & inspect code/variables**
- **Variable watch:**
    - Monitors **variables/objects**
    - Halt **program** if condition is **not met**
- **Stepping:**
    - Set program to **step through one line at a time**
    - Execution **slows down** to observe path of **execution** + **changes to variable names**
    - Programmer can **observe the effect** of each line of code
    - Can be used with **breakpoints** + **variable watch**

(f) Use of object orientated techniques

- Many programs written using objects (**Building blocks**)
- **Self contained**
- Made from **methods & attributes**
- Based on **classes**
- Many objects can be based in the **same class**
- **Most programs** made using object-oriented techniques

2.2.2 Computational methods

(a) Features that make a problem solvable by computational methods

Computability: Something which is not affected by the speed/power of a machine

Computational methods can help to break down problems into sections for example:

- **Models of situations/hypothetical solutions** can be **modelled**
- **Simulations** can be run by **computers**
- **Variables** used to **represent data items**
- Algorithms used to **test possible situations** under **different circumstances**

Features that make a problem solvable by computational methods:

- Involves **calculations** as some issues can be **quantified** - these are easier to process **computationally**
- Has **inputs, processes and outputs**
- Involves **logical reasoning**.

(b) Problem recognition

- A problem should be **recognised/identified** after looking at a **situation** and **possible solutions** should be divided on how to **tackle the problems** using **computational methods**

(c) Problem decomposition

Problem Decomposition

- **Splits problem** into **subproblems** until each problem can be **solved**.
- Allows the use of **divide and conquer**
- Increase **speed of production**.
- Assign areas to specialities.
- Allows use of **pre-existing modules & re-use of new modules**.
- Need to ensure **subprograms** can **interact correctly**.
- Can **introduce errors**.
- Reduces **processing/memory requirements**.
- Increases **response speeds of programs**.

(d) Use of divide and conquer

Divide and Conquer: When a task is split into **smaller tasks** which can be tackled more easily

(e) Use of abstraction

Abstraction: Process of separating ideas from particular instances/reality

- **Representation of reality** using various methods to display real life features
- **Removes unnecessary details** from the main purpose of the program
- E.g Remove parks/roads on an Underground Tube Map

Examples of Abstraction: Variables/data structure/network/layers/symbols (maps)/Tube Map

(f) Applying computational methods

Other computational Methods:

- **Backtracking**
    - **Strategy** to **moving systematically** towards a **solution**
    - **Trial & Error** (Trying out series of actions)
    - If the pathway **fails** at some point = **go to last successful stage**
    - Can be used **extensively**
- **Heuristics**
    - **Not always worth trying to find the 'perfect solution'**
    - Use '**rule of thumb**' /educated guess **approach** to arrive at a solution when it is unfeasible to analyse all possible solutions
    - Used to **speed up finding solutions** for **A* algorithm**
    - Useful for too many **ill-defined variables**
- **Data mining**
    - Examines **large data sets** and looks for **patterns/relationships**
    - **Brute force** with **powerful computers**
    - Incorporates: **Cluster analysis, Pattern matching, Anomaly detection, Regression Analysis**
    - Attempts to show relationships between **facts/components/events** that may not be obvious which can be used to **predict future solutions**
- **Visualisation**
    - A computer process presents data in an **easy-to-grasp way** for humans to **understand** (visual model)
    - Trends and patterns can often be better comprehended in a **visual display**.
    - Graphs are a **traditional form** of visualisation.
    - **Computing techniques** allow **mental models** of what a **program** will do to be produced.
- **Pipelining**
    - Output of **one process fed into another**
    - **Complex jobs** placed in **different pipelines** so **parallel processing** can occur
    - Allow **simultaneous processing of instructions** where the processor has **multi-cores**
    - Similar to factory production in real life
- **Performance modelling**
    - Example of **abstraction**
    - **Real life objects/systems** (computers/software) can be **modelled** to see how they perform & behave when in use
    - **Big-O notation** used to measure **algorithm behaviour** with increasing input
    - **Simulations predict performance** before real systems created