Level 3 Technical Level
IT: PROGRAMMING
F/507/6465

Unit 2 Computer programming

Mark scheme

January 2018

Version: 1.0 Final

1 8 1 A F 5 0 7 6 4 6 5 / M S

Mark schemes are prepared by the Lead Assessment Writer and considered, together with the relevant questions, by a panel of subject teachers. This mark scheme includes any amendments made at the standardisation events which all associates participate in and is the scheme which was used by them in this examination. The standardisation process ensures that the mark scheme covers the students' responses to questions and that every associate understands and applies it in the same correct way. As preparation for standardisation each associate analyses a number of students' scripts. Alternative answers not already covered by the mark scheme are discussed and legislated for. If, after the standardisation process, associates encounter unusual answers which have not been raised they are required to refer these to the Lead Assessment Writer.

It must be stressed that a mark scheme is a working document, in many cases further developed and expanded on the basis of students' reactions to a particular paper. Assumptions about future mark schemes on the basis of one year's document should be avoided; whilst the guiding principles of assessment remain constant, details will change, depending on the content of a particular examination paper.

Further copies of this mark scheme are available from aqa.org.uk

Level of response marking instructions

Level of response mark schemes are broken down into levels, each of which has a descriptor. The descriptor for the level shows the average performance for the level. There are marks in each level.

Before you apply the mark scheme to a student's answer read through the answer and annotate it (as instructed) to show the qualities that are being looked for. You can then apply the mark scheme.

Step 1 Determine a level

Start at the lowest level of the mark scheme and use it as a ladder to see whether the answer meets the descriptor for that level. The descriptor for the level indicates the different qualities that might be seen in the student's answer for that level. If it meets the lowest level then go to the next one and decide if it meets this level, and so on, until you have a match between the level descriptor and the answer. With practice and familiarity you will find that for better answers you will be able to quickly skip through the lower levels of the mark scheme.

When assigning a level you should look at the overall quality of the answer and not look to pick holes in small and specific parts of the answer where the student has not performed quite as well as the rest. If the answer covers different aspects of different levels of the mark scheme you should use a best fit approach for defining the level and then use the variability of the response to help decide the mark within the level, ie if the response is predominantly level 3 with a small amount of level 4 material it would be placed in level 3 but be awarded a mark near the top of the level because of the level 4 content.

Step 2 Determine a mark

Once you have assigned a level you need to decide on the mark. The descriptors on how to allocate marks can help with this. The exemplar materials used during standardisation will help. There will be an answer in the standardising materials which will correspond with each level of the mark scheme. This answer will have been awarded a mark by the Lead Examiner. You can compare the student's answer with the example to determine if it is the same standard, better or worse than the example. You can then use this to allocate a mark for the answer based on the Lead Examiner's mark on the example.

You may well need to read back through the answer as you apply the mark scheme to clarify points and assure yourself that the level and the mark are appropriate.

Indicative content in the mark scheme is provided as a guide for examiners. It is not intended to be exhaustive and you must credit other valid points. Students do not have to cover all of the points mentioned in the Indicative content to reach the highest level of the mark scheme.

An answer which contains nothing of relevance to the question must be awarded no marks.

Question	Guidance	Mark
01	C	1
02	B	1
03	C	1
04	C	1
05	A	1

06	**Explain the difference between assembly language and machine code.** **2 marks** for clear explanation, **1 mark** for partial explanation, eg: • mnemonic/numeric or hex • symbolic/bits or bytes • labels/fixed memory addresses • hierarchy of languages • hardware specific • compiled (translated)/machine understandable • instead of representing the machine language as binary, the instructions and registers are given names/meaning.	**2**

07.1	**Explain why the totals are different** **2 marks** for clear explanation, **1 mark** for partial explanation, eg: • There is a different scope between global and local variables (**1 mark**) so the total inside the function does not change the value outside **(1 mark**) • A variable can have a different value inside and outside a function (**1 mark**) • The variable 'total' in the function is not assigned on return (**1 mark**) • The function is called but the return value not stored (**1 mark**)	**2**

07.2	**Without changing the function definition, fix the code so the totals are the same.** **1 mark** for `11` `total = distance(0,0,8,6)` **Allow:** equalise the totals, eg `total=100` on line 2; `distance(8,6,8,6)` or `distance(0,0,0,0)` on line 11; replace total on line 13 with line 11; make the total variable global (allow: static).	**1**

08	**In programming, what is functional decomposition?** **Example answers:** • Functional decomposition is the process of breaking down a complex problem into smaller parts **(1 mark)**, these parts can then be broken down further **(1 mark)**. Therefore, developing and testing is easier/more efficient **(1 mark)**. • Decomposition of software into sub-problems **(1 mark)** that can be developed independently **(1 mark)**, this is also known as modularisation **(1 mark)**. **Allow:** other valid approaches.	**3**

09.1	**Use examples to explain the difference between a pre-condition loop and a post-condition loop.** **Pre-condition (2 marks)** • Check condition first, only run loop if it passes • eg While, For, For-each **Post-condition (2 marks)** • Check condition at end *after executing code once* / single pass • eg do… while **Allow:** sequence of code that includes a pre-condition loop **(1 mark)** and a post-condition loop **(1 mark)** with comments or other annotation/listing of outputs which illustrates the difference stated in bullets above **(up to 2 marks)**.	**4**

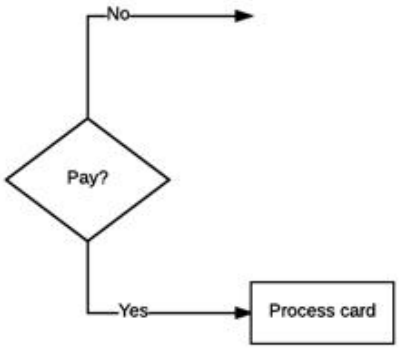
09.2	**The following code should add the numbers 1, 2, 3, and 4 together and output 10. What needs to be corrected to make this happen?** **1 mark** for either: 02 `For (int x = 1; x <= 4; x++)` 02 `For (int x = 1; x < 5; x++)`	**1**

10.1	**Explain the concept of a programming paradigm.** **Example answer**: A paradigm is a style of programming / way to classify **(1 mark)** based on features / system of ideas **(1 mark)** eg in the functional paradigm computations are expressed as the evaluation of mathematical functions **(1 mark)**. Some languages support one paradigm, others multiple paradigms **(1 mark)**. Supporting examples (**1 mark**)	**4**

10.2	**State one language associated with the functional programming paradigm.** **1 mark** for example, eg: • LISP • Logo • Scheme • C# • C++ • JavaScript • Python.	**1**

10.3	**Name one other common programming paradigm.** **1 mark** for example, eg: • Procedural • Object-oriented • Event-driven • Scripting • Logic.	**1**

11.1	**Give three reasons why you would use a flowchart to demonstrate a program to a client.** **1 mark** for each reason, eg: • ease of understanding/present complex ideas in simple form • represent process rather than code • graphically rather than words • something developers can refer back to and check requirements.	**3**

11.2	**Using appropriate symbols, draw a flowchart extract that shows an example of a decision and a process.** **1 mark** (max **1 mark**) for appropriate symbols **1 mark** (max **1 mark**) for an example of each **1 mark** (max **1 mark**) for appropriate labels eg decision Y/N, direction of flow  	**3**

12.1	**Explain the difference between a real and an integer variable.**
	
	1 mark for one definition, **2 marks** for contrast, eg:
	• Integer is a whole number
	• Real variable/data type:
	  ○ has a fractional part
	  ○ stores numbers like single and double precision
	  ○ is an approximation of a real number
	  ○ **do not allow**: 'is any number'.
		2

12.2	**What does the keyword const stand for and why is this different from a variable?** **1 mark** for • constant • cannot change the assigned value	**2**

12.3	**Explain what is meant by the terms variable, assignment and expression.** **Example answers** (max **3 marks**): • Assignment copies a value of an expression into the variable **(1 mark)** eg testscore = 30/50 **(1 mark)** would enumerate 30/50 **(1 mark)** and assign 0.6 to the variable **(1 mark)**; the variable could then be referenced/have its value changed elsewhere in the program **(1 mark)** • *var x = score/marks* **(1 mark)** is variable = expression **(1 mark)** and the expression is assigned to variable 'x' **(1 mark)**. A variable is storage location paired with a symbolic identifier **(1 mark)**. An expression is a combination of one or more explicit values, constants, variables, operators, and functions **(1 mark)** that the programming language interprets • a variable stores a value that can be changed **(1 mark)** • an assignment is the process that gives a value to a variable/of giving a variable data to store **(1 mark)** • an expression is an equation which resolves to a value **(1 mark)**.	**3**

12.4	**Give a line of code which shows the relationship between them.** **1 mark** for example, eg: • testscore = 30/50 • var x = score/marks • var x = expression	**1**

13.1	**Explain why a developer could learn different things from client testing and user testing.** **1 mark** for each point, eg: • client and user may have different roles/expectations/understanding, eg checking against requirements/brief and testing bugs • client might not ever use the system • a user might point out design flaws the client is not aware of • users might be external to the company, eg using a website • practical examples of different roles.	**3**

13.2	**Student test scores are input into a database as a raw score out of 60 and as a percentage. Complete Table 1 with appropriate test data.** **1 mark** for each correct row **Allow:** other examples of appropriate data, eg where candidate provides a context which justifies the data chosen.	**3**

TEST DATA	**Raw score**	**Test score (percentage)**
Normal	0 – 60	0 to 100
Extreme	towards 0 OR 60	towards 0 OR 100
Invalid	<0 OR >60 **Allow:** fractional, eg 13.2	<0 OR >100 **Allow:** Null; error.

14	**Explain how the following code could be improved so that it demonstrates the principles of good programming practice.** **1 mark** for each point, eg: • The repeated code is inefficient • Insert comments • Line 7/8 could be on one line **(1 mark)**, eg: age1 = input(name1+", how old are you?") (**1 mark)** • Create loops **(1 mark)** with explanation of this **(1 mark)** • Create player class • Use arrays rather than variables **(1 mark)**, eg: name(1), name(2)	**6**

15.1 **Compare the Incremental and V-model software development approaches, including the advantages and disadvantages of each model.**

Indicative content:

Model	Advantages	Disadvantages
Incremental • Requirements divided into various builds; easier to manage • Major requirements of complete system clearly defined • Some details can evolve over time • Used to get a product to the market early; or project with high-risk objectives • Series of mini-waterfalls (requirements, design, implementation, testing, deployment) • Each module adds functionality • Work incrementally until each stage fully finished	• Balance of simplicity and adaptability • Generates working software / business / customer value early in software development life cycle (SDLC) • Flexible – less costly to change scope and requirements • Easier to test and debug during a smaller iteration; problems detected early • Better use of scarce resources by defining increments • Customer can respond to each build ∴ easier to manage risk / gain feedback due to frequent design cycles • Lowers initial delivery cost • Good for paired programming	• Needs good planning and design • Needs heavy documentation • Needs clear/complete definition of whole system/processes before it can be broken down and built incrementally • Total cost is higher than waterfall • Level of customer involvement (can be adv.) • Separation of functions and features may be difficult • Much longer development time

12

Model	Advantages	Disadvantages
V-model • 'Verification and Validation' model • Modified waterfall model with sequential execution of processes (V: requirements, high-level design, low-level design – implementation – unit, integration, system testing, deployment) • Each phase must be completed before next phase begins • Testing planned in parallel (developers v testers life cycle) • High-level (architecture/ integration testing)/low-level (class diagrams/ component testing)/ implementation phases • Used for small to medium sized projects where requirements clearly defined/fixed/ understood; technical resources/expertise available • Development and testing form the 'two sides of the V-shape • Coding at bottom of V-model	• Ease of use • Testing activities like planning, test designing, happen before coding: saves time, develops very good understanding of project at initial stage • Time management • Defects found at early stage, so cheaper to fix • Higher chance of success over waterfall model • System test plan created before development; addresses functionality in requirements gathering • Unit testing followed by integration testing • Product release and on-going support (ie top right of V)	• Very rigid/least flexible • Each stage must be completed before next stage begins • Software developed during implementation phase ∴ no early prototypes; risk of not meeting client expectations • If changes happen midway, test/requirements documents need to be updated • Design changes are expensive as require reengineering of tests • Applicable mostly to big companies (resources intensive) • Can favour managers and users, over developers and designers

Award a mark from each levels of response table:

Q15 Descriptor (first component) Compare • Max. **4 marks** if candidate has not compared two models. • Max. **3 marks** if only one approach considered.	**Marks**
Candidate has compared each approach, with clear understanding of both.	5-6
Candidate has described each approach, with some understanding of both; or clear understanding of one approach (max. 3 marks).	3-4
Candidate has described each approach with some general understanding (or compared advantages/disadvantages); or shown some understanding of one approach (max. 2 marks).	1-2
No creditworthy response.	0

Q15 Descriptor (second component) Advantages/disadvantages	**Marks**
Candidate has explained some advantages and disadvantages, or made a list (5+) which shows clear understanding of both.	5-6
Candidate has described some advantages and disadvantages, or made a list (3+) which shows some understanding of both.	3-4
Candidate has listed some advantages or disadvantages, with some general understanding.	1-2
No creditworthy response.	0

15.2 **Explain two other software development approaches.**

Examples in specification are (**3 marks** for each explanation):

- Waterfall
- Spiral
- Agile
- Iterative
- Modular

Allow other valid approaches.

Indicative content:

Model	Advantages	Disadvantages
Waterfall • A linear model / step-by-step design (requirements, design, implementation, testing, deployment) • Each phase must be completed before the next can begin • Used for small/medium projects with clearly defined requirements from outset	• Simple to understand and use • Easy to manage – each phase has specific deliverables and review process • Phases processed/ completed one at a time/do not overlap • Works well for smaller projects where requirements are very well understood	• Scope needs to be clear and detailed from the outset • Working software not given to client until late in the SDLC • Difficulty of managing team resources effectively • Difficult/expensive to change client requirements later/or in testing phase

6

	Modular • Separates functionality into interchangeable modules • Rules, tools and methods that together prescribe how modules are deployed over time • Closely related to structured programming and object-oriented programming • Allows many programmers to collaborate/expertise to be split • More flexible/scalability, easier to maintain code / large programs easier to design and manage • Cost-effective; simplified planning and engineering, standardisation	• Reuse of code • Scoping of variables easier • Thorough testing • Organisation of code; allows library programs to be inserted / programming codes shortened • Errors localised/can be easily identified • Develop and test functionality independently	• Requires thorough documentation of modules • Approach needs to be agreed across teams, eg naming conventions	
	Iterative • Iteration: steady refinement of design based on testing/evaluation/repeated circle of events • Delivered first in rough form, then subject to user feedback and testing before final version • Part of each iteration involves studying how intuitive and efficient (eg interface) is. Cycle would then repeat to refine the previous delivery until a final accepted and tested design • Refine based on user testing/feedback to improve usability; present sketches and blueprints of product to users for feedback	• Testing/fixing at each stage instead of at the end • Acquire quantitative and qualitative feedback, eg to increase productivity while using interface • Build and improve product step-by-step • Track defects at early stages; avoids downward flow of the defects • Less time spent on documenting, more time for design	• Each phase rigid with no overlaps • Potential/costly system architecture issues because not all requirements are gathered up front for entire SDLC	

Agile • Type of incremental model • People and interactions, rather than processes and tools • Allows interaction with customers, developers, testers • Small incremental releases build on previous functionality • New changes can be implemented at very little cost because of frequency • Easy to roll back and implement new features • More freedom of options / time to make/manage/postpone decisions • Regularly adapted to suit environment/clients' needs; easy to effect new features based on feedback	• Software delivered frequently • Very little planning required • Realistic client expectations/on time/budget • Alerted early to problems • Rapid, continuous delivery of useable software • Continuous attention to technical excellence/quality/ good design • Adapt regularly to changing circumstances	• Some software deliverables difficult to assess for effort required at the beginning of the SDLC • Lack of emphasis on necessary design and documentation • Can get taken off track if customer representative not clear on outcomes • Only senior programmers capable of taking decisions required during development process • Novice programmers need paired with experienced
Spiral • Similar to incremental • For medium to high-risk projects • More emphasis on risk analysis/minimisation of risk • Planning/risk analysis (prototypes, design), engineering (implementation, testing)/evaluation (deployment) • Repeats/spirals/multiple-development phases • Requirements are complex • Significant changes are expected (research and exploration), eg new product line	• High amount of risk analysis • Good for large and mission-critical projects • Strong approval and documentation control • Additional functionality can be added later • Software is produced early in the SDLC	• Completed phases cannot be revisited easily • Users can be unsure of needs • Scope for revising/ backtracking • Needs close risk assessment • Client may have to spend a lot of time with development team • Often no documentation • Difficult to fix the start/end of phase • Difficult to commit long-term because of potential changes to economic priorities

16 **Design a test strategy for installing and testing the system. Justify your choices. For example, you could consider:**

- **what is critical to success**
- **different users and audience**
- **tools and testing techniques**
- **structure and data**
- **remedial action.**

Q16 Descriptor (first component) Design a test strategy	**Marks**
Candidate has carefully designed a clear and viable strategy using the bullets provided or sound alternatives	5-6
Candidate has designed a strategy which is mostly viable using the bullets provided or sound alternatives	3-4
Candidate has attempted a strategy, some of which is viable	1-2
No creditworthy response	0

Q16 Descriptor (second component) Justify your choices	**Marks**
Candidate has fully justified their choices in relation to the scenario	5-6
Candidate has made some justification of their choices, some of which is focused on the scenario	3-4
Candidate has attempted to justify their choices, though in a fairly generic way	1-2
No creditworthy response	0

12

Assessment Outcomes					
Question	**AO1**	**AO2**	**AO3**	**AO4**	**Question Total**
Section A					
01			3a (1)		1
02			3e (1)		1
03			3a (1)		1
04	1a (1)				1
05		2c (1)			1
06	1a (2)				2
07.1			3e (2)		2
07.2			3e (1)		1
08		2d (3)			3
09.1			3a (4)		4
09.2			3a (1)		1
10.1	1c (1)				1
10.2	1b (1)				1
10.3	1b (4)				4
11.1		2d (3)			3
11.2		2d (3)			3
12.1			3a (2)		2
12.2			3a (2)		2
12.3			3a (3)		3
12.4			3a (1)		1
13.1		2a (3)			3
13.2			3d (3)		3
14				4a (6)	6

Section B					
15.1		2b (12)			12
15.2		2b (6)			6
16			3f (12)		12
Totals	**9**	**31**	**34**	**6**	**80**