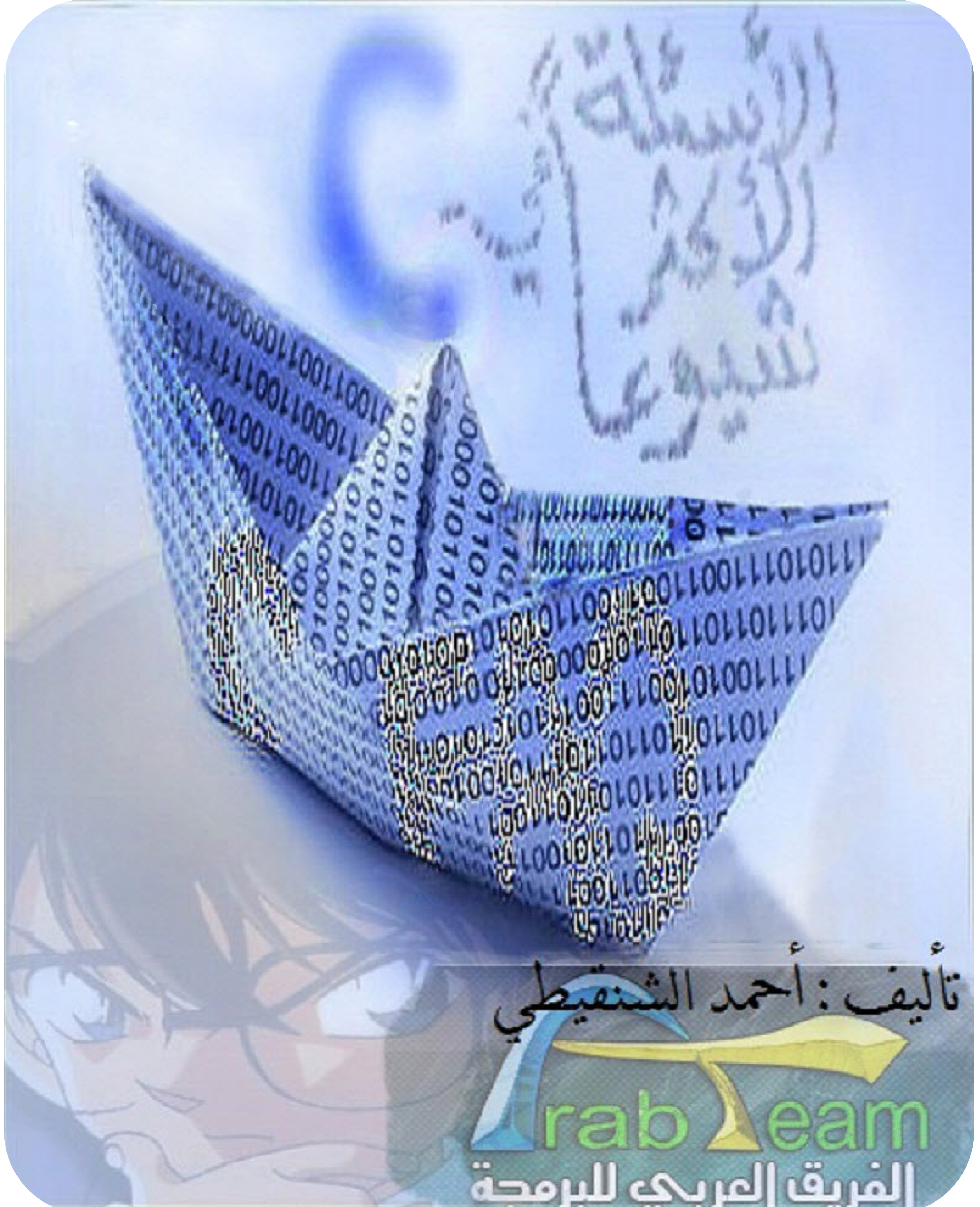
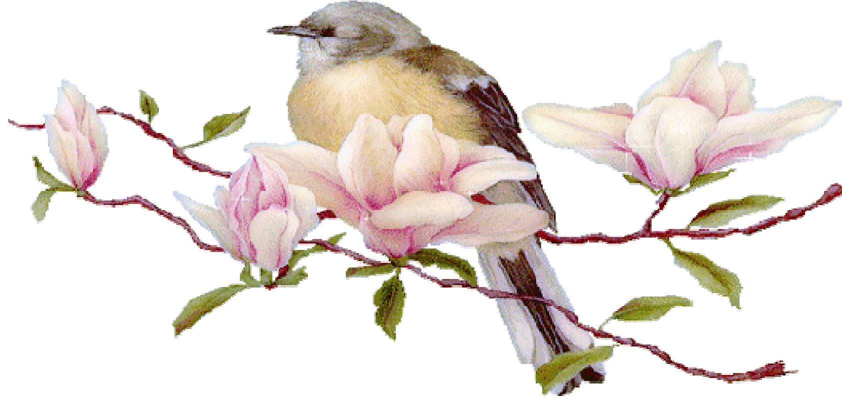

C
الأسئلة الأكثر شيوعا
تأليف : أحمد الشنقيطي
ArabTeam
الفريق العربي للبرمجة



الحمد لله الذي بحمده يُستفتح كل كتاب و بذكره يُصدر كل خطاب وبفضله يتنعم أهل النعيم في دار الجزاء و الثواب والصلاة و السلام على سيد المرسلين و إمام المتقين المبعوث رحمة للعالمين محمد بن عبد الله الصادق الأمين و على صحابته الأخيار و من تبعهم بإحسان إلى يوم الدين أما بعد :

تم بحمد الله الانتهاء من الإصدار الأول من سلسلة الأسئلة الأكثر شيوعا في لغة C. هذا الإصدار يشرح أساسيات C بشكل مُفصل.

قمتُ بتقسيم الأسئلة (حسب نوعها) إلى عدة فصول, الفصول بدورها قد تحتوي على عناوين فرعية, كل عنوان فرعي يجمع الأسئلة التي من نفس النوع, مما يُسهل عملية البحث و يزيد من تنظيم و فهرسة الأسئلة الموجودة.

و هذا فهرس الأسئلة (143 سؤال مُقسمة كالآتي) :

1. الخطوة الأولى لتعلم لغة C
   - نبذة تاريخية عن لغة C (سؤالين)
   - الأدوات اللازمة (6 أسئلة)
2. المتغيرات و المؤثرات
   - المتغيرات (22 سؤال)
   - المؤثرات (13 سؤال)
3. المؤشرات, المصفوفات الحرفية و التراكيب
   - المؤشرات (26 سؤال)
   - المصفوفات الحرفية (12 سؤال)
   - التراكيب (10 أسئلة)
4. الدوال, الملفات و مجالات الرؤية
   - الدوال (19 سؤال)
   - الملفات (6 أسئلة)
   - مجالات الرؤية (3 أسئلة)
5. مُوجهات ما قبل المعالجة (13 سؤال)
6. من هنا و هناك ! [متفرقات في اللغة] (11 سؤال)

كُتب بتاريخ 16/09/2012

الكاتب في سطور:

الاسم: أحمد بن محمد

اللقب: الشنقيطي

سنة الميلاد: 1992

الدولة: بلاد شنقيط و أرض المليون شاعر .. موريتانيا

الهواية: Security & programming

المستوى الأكاديمي: خريج كلية العلوم و التقنيات.

للتواصل: ahmed.ould_mohamed@yahoo.fr

جميع الحقوق محفوظة © All rights reserved

فهرس المحتويات

I - الفصل الأول : الخطوات الأولى لتعلم لغة C.......... 11

1 - I. نبذة تاريخية عن لغة C 11

1.1 – I, من أين جاءت لغة C ؟ 11

1.2 – I, ما هي مميزاتها ؟ 12

2 – I. الأدوات اللازمة 12

2.1 – I, ما هو المترجم !؟ 12

2.2 – I, لماذا نحتاجه !؟ 13

2.3 – I, ولكن .. لماذا يحلل المترجم وبعدها يبدأ بالتوليد ؟ 13

2.4 – I, جميع المترجمات الحديثة متوفرة لها عدة IDE, ماذا يعني هذا المصطلح؟ 14

2.5 – I, ما هي أشهر الـ IDE المُستخدمة حاليا ؟ 14

2.6 – I, ما هي سلبيات و إيجابيات كلّ من Visual C++, Code::Blocks, NetBeans ؟ 14

II - الفصل الثاني : المتغيرات و المؤثرات 15

1 – II. المتغيرات 15

1.1 – II, ما هو النوع char و ما هي المساحة التي يمثلها ؟ 15

1.2 – II, ما هي المساحة التي يمثلها int ؟ 15

1.3 – II, أحيانا, ألاحظ أن حجم int مساوي لـ short أو long أو long long !, لماذا ؟ 15

1.4 – II, ماذا عن اختلاف حجم int بالنسبة للبيئة التي يعمل عليها البرنامج ؟ 16

1.5 – II, ما الحل ؟ 16

1.6 – II, و ما هي الـ Length Modifier ؟ 16

1.7 – II, إذا, توجد أربعة أنواع للأرقام الصحيحة ؟ 16

1.8 – II, هل تحدد اللغة أحجام البيانات ؟ 17

1.9 – II, إذا, اللغة لا تهتم بالمساحة التي يستهلكها int ؟ 17

1.10 – II, كلمة أخيرة عن أحجام البيانات ؟ 17

1.11 – II, ماذا تعني الكلمة signed ؟ 17

1.12 – II, المترجم الخاص بي لا يعرف النوع long long, لماذا ؟ 17

1.13 – II, هل تدعم لغة C المتغيرات المنطقية ؟ 18

II – 1.14, ما فائدة المؤثر sizeof ؟ 18
II – 1.5, هل يمكننا معرفة الحجم الكامل المستعمل للبرنامج عن طريق sizeof ؟ 19
II – 1.16, ما هو النوع size_t ؟ 19
II – 1.17, ما هو النوع wchar_t ؟ 19
II – 1.18, كيف أنشئ اسما مستعارا للأنواع الموجودة في اللغة ؟ 19
II – 1.19, ماذا تعني الكلمة const ؟ 20
II – 1.20, ما هي الرموز المستخدمة مع الأنواع الصحيحة ؟ 20
II – 1.21, متى أستخدم if-else و متى أستخدم switch ؟ 20
II – 1.22, ماذا يحدث للمتغيرات الغير مهيأة (uninitialized variable) ؟ 20
II – 2, المؤثرات 21
II – 2.1, ما الفرق بين المؤثر = و المؤثر == ؟ 21
II – 2.2, ما هو المؤثر الثلاثي (ternary operator) ؟ 21
II – 2.3, هل يمكن استخدام المؤثر الثلاثي في حالة وجود أكثر من statement ؟ 21
II – 2.4, كيف نحسب باقي القسمة ؟ 22
II – 2.5, ما الفرق بين الزيادة القبلية (++i) و الزيادة البعدية (i++) ؟ 22
II – 2.6, كيف يعمل المؤثر AND (&&) ؟ 23
II – 2.7, كيف يعمل المؤثر OR (||) ؟ 23
II – 2.8, كيف يعمل المؤثر AND_BIT (&) ؟ 24
II – 2.9, كيف يعمل المؤثر OR_BIT (|) ؟ 24
II – 2.10, كيف يعمل المؤثر NOT_BIT (~) ؟ 25
II – 2.11, كيف يعمل المؤثر XOR (^) ؟ 25
II – 2.12, كيف تعمل مؤثرات الإزاحة ؟ 26
II – 2.13, ما هو ترتيب أولوية المؤثرات ؟ 27
III - الفصل الثالث : المؤشرات, المصفوفات الحرفية و التراكيب 28
III – 1. المؤشرات 28
III – 1.1, ما هو المؤشر ؟ 28
III – 1.2, متى تُستخدم المؤشرات ؟ 28
III – 1.3, ما معنى NULL ؟ 28
III – 1.4, ما الفرق بين الـ Null Pointer و Uninitialized Pointer ؟ 29

III - 1.5, ما معنى stack, heap, buffer ؟ 29
III - 1.6, ما الفرق بين تحرير و تصفير الذاكرة ؟؟ 30
III - 1.7, لكن, إذا كان المؤشر يشير إلى ذاكرة عنوانها صفر ألا يعني هذا أن قيمة الخانة المؤشر عليها تساوي صفر أيضا !؟ 30
III - 1.8, ما هو المؤشر الثابت ؟ 30
III - 1.9, ما هو المؤشر إلى ثابت ؟ 30
III - 1.10, ما هو المؤشر الثابت إلى ثابت ؟ 30
III - 1.11, ما هو المؤشر العائم (Void Pointer) و فيم يُستخدم ؟ 31
III - 1.12, ما هو الـ Pointer to Pointer ؟ 31
III - 1.13, باعتبار أن t مصفوفة, ما الفرق بين الكتابات التالية t, &t, &(t[0]) ؟ 31
III - 1.14, ما الفرق بين char a[] و char * a ؟ 31
III - 1.15, باعتبار أن t مصفوفة, ما معنى الكتابة *(t+3) ؟ 32
III - 1.16, ما معنى هذه الكتابة *p++ ؟ 32
III - 1.17, ما هو دور المعامل [ ] ؟ 32
III - 1.18, ماذا تعني الكتابة التالية int (*p)[4] ؟ 33
III - 1.19, ما هي فائدة الدالة malloc ؟ 33
III - 1.20, ما هي فائدة الدالة calloc ؟ 34
III - 1.21, ما الفائدة من استخدام calloc إذا كانت malloc تفي بالغرض !؟ 34
III - 1.22, ما هي فائدة الدالة realloc ؟ 34
III - 1.23, كيف أنشئ مصفوفة ديناميكية ثنائية البعد ؟ 35
III - 1.24, ما الفرق بين الدالة malloc و المعامل new ؟ 35
III - 1.25, p و q يُشيران إلى متغيرين يحملان نفس القيمة و مع ذلك فإن p == q تُعيد false دائما !, ما السبب ؟ 36
III - 1.26, كيف نعرف نوع المتغير الذي يُشير إليه مؤشر من النوع void ؟ 36
III - 2. المصفوفات الحرفية 37
III - 2.1, ما هو NUL ؟ 37
III - 2.2, لماذا يحتاج المترجم إلى وضع الرمز NUL في نهاية كل مصفوفة حرفية ؟ 37
III - 2.3, ما معنى الكتابة التالية "char * p = "Bonjour ؟ 38
III - 2.4, كيف أجعل الحروف صغيرة أو كبيرة ؟ 39
III - 2.5, كيف نقوم بتحويل عدد إلى سلسلة محارف ؟ 39
III - 2.6, كيف نقوم بتحويل سلسلة محارف إلى عدد ؟ 39

III – 2.7, كيف ندمج السلاسل الحرفية ؟ 40
III – 2.8, كيف نقارن السلاسل الحرفية ؟ 40
III – 2.9, لماذا لا أستطيع مقارنة السلاسل الحرفية المقروءة بالدالة fgets ؟ 41
III – 2.10, كيف ننشئ مصفوفة من السلاسل الحرفية ؟ 41
III – 2.11, لماذا يجب كتابة الرمز \ هكذا \\ ؟ 41
III – 2.12, كيف نقوم بنسخ مصفوفة ؟ 41
III – 3. التراكيب 42
III – 3.1, كيف أنشئ اسما مستعارا لـ Structure ؟ 42
III – 3.2, ما الفرق بين structure و union ؟ 43
III – 3.3, لماذا حجم البنية لا يُساوي بالضرورة مجموع أحجام العناصر ؟ 44
III – 3.4, كيف نقوم بنسخ structure ؟ 44
III – 3.5, كيف نقارن بين بُنْيَتَيْن ؟ 44
III – 3.6, ماذا تعني الكتابة التالية unsigned int i : <n> ؟ 45
III – 3.7, كيف نستخدم مؤشر يُشير إلى بنية ؟ 45
III – 3.8, ما الفرق بين الكتابتين sizeof(struct data) و sizeof(struct data *) ؟ 45
III – 3.9, كيف يتم الإعلان عن بنية تشير إلى نفسها ؟ 46
III – 3.10, كيف نُحسن تهيئة المتغيرات ؟ 46
IV - الفصل الرابع : الدوال, الملفات و مجالات الرؤية 47
IV – 1. الدوال 47
IV – 1.1, ما السبيل إلى تغيير قيم وسائط الدالة ؟ 47
IV – 1.2, ما هي فائدة الوسائط الافتراضية (Default parameters) ؟ 47
IV – 1.3, ما هي فائدة الدوال الخطية (Inline Function) و أين يتم تخزينها ؟ 48
IV – 1.4, ما هو النموذج المُصغر (prototype) ؟ 48
IV – 1.5, أيهما أفضل, الإعلان عن الدالة أسفل أو أعلى الـ main ؟ 49
IV – 1.6, ما هي التواقيع الصحيحة للدالة الرئيسية ؟ 50
IV – 1.7, كيف تستطيع printf استقبال عدد غير محدود من مُختلف أنواع المتغيرات ؟ 50
IV – 1.8, عند استخدام الرمز %d مع الأعداد الحقيقية تظهر نتائج غريبة !, مالسبب ؟ 50
IV – 1.9, ما هي أهم الرموز المُستخدمة مع الدالة printf ؟ 51
IV – 1.10, ما هي أهم الرموز المُستخدمة مع الدالة scanf ؟ 52

IV – 1.11, ما هي أهم دوال المكتبة math.h ؟ 53
IV – 1.12, ما هي أهم دوال المكتبة ctype.h ؟ 54
IV – 1.13, كيف تُعيد الدالة أكثر من قيمة ؟ 55
IV – 1.14, كيف يتم الإعلان عن دالة تُعيد سلسلة محارف ؟ 55
IV – 1.15, كيف نمرر مصفوفة إلى دالة ؟ 56
IV – 1.16, لماذا لا يُنصح باستخدام المؤشرات المحلية كقيمة مُعادة من طرف الدوال ؟ 57
IV – 1.17, كيف يتم الإعلان عن مؤشر يشير إلى دالة ؟ 58
IV – 1.18, كيف يتم الإعلان عن مصفوفة دوال ؟ 58
IV – 1.19, كيف تتم إعادة مؤشر يشير إلى دالة ؟ 58
IV – 2. الملفات 60
IV – 2.1, كيف أتأكد من وجود ملف ؟ 60
IV – 2.2, كيف أعرف حجم ملف ؟ 60
IV – 2.3, كيف يتم نسخ الملفات ؟ 61
IV – 2.4, كيف أحذف أسطراً من ملف نصي ؟ 62
IV – 2.5, كيف أحذف ملفاً ؟ 62
IV – 2.6, كيف أعرض محتوى مجلد ؟ 62
IV – 3. مدى المتغيرات أو مجالات الرؤية 63
IV – 3.1, ما الفرق بين المتغيرات المحلية (local) و المتغيرات العامة (global) ؟ 63
IV – 3.2, ماذا تعني الكلمة static ؟ 64
IV – 3.3, كيف يتم استخدام متغير عام مُعرف في ملف مصدري آخر (another source file) ؟ 64
V - الفصل الخامس : مُوجهات ما قبل المعالجة 65
V – 1, ما هي فائدة الـ Preprocessor ؟ 65
V – 2, ما هو الـ MACRO ؟ 65
V – 3, كيف أستخدم ماكرو يحتوي على وسائط (Parameterized macro) ؟ 66
V – 4, ماذا تعني الكتابة #define MYMACRO ؟ 66
V – 5, و لماذا تُوفر C هذا النوع من الكتابات الذي لا فائدة من ورائه ؟ 67
V – 6, كيف أعرف ما إذا كان ماكرو مُعرف مُسبقاً أم لا ؟ 67
V – 7, ما هي المشاكل التي قد تحدث نتيجة سوء استخدام الماكرو ؟ 68
V – 8, ما هو دور المؤثر # في تعريف الماكرو ؟ 69

9 – V, ما هو دور المؤثر ## في تعريف الماكرو ؟ 69
10 – V, ما هي فائدة #pragma ؟ 70
11 – V, متى أستخدم #error ؟ 70
12 – V, هل يمكننا استخدام sizeof مع #if ؟ 70
13 – V, ما هي الأسماء المُعرَّفة (Predefined Names) و ما فائدتها ؟ 71
VI - الفصل السادس : من هنا و هناك ! (مُتفرقات في اللغة) 72
1 – VI, كيف أحصل على معيار لغة السي (C standard) ؟ 72
2 – VI, ما هي فائدة Null Statement ؟ (الإجابة للأستاذ خالد الشايع) 72
3 – VI, ماذا يعني الرمز _ أمام اسم دالة أو ماكرو أو متغير ؟ 73
4 – VI, ما هو عمل الدالة fork ؟ 73
5 – VI, ما هي فائدة الدالة system ؟ 74
6 – VI, كيف أحول التاريخ إلى سلسلة حرفية ؟ 74
7 – VI, كيف أقوم بتوليد أرقام عشوائية ؟ 74
8 – VI, ما معنى الخطأ unresolved external symbol _WinMain@16 ؟ 74
9 – VI, ما معنى التحذير no new line at end of file ؟ 75
10 – VI, أين أجد دروس مُفصَّلة في لغة C ؟ 75
11 – VI, أين أجد تمارين جيدة في لغة C ؟ 75
V - ملحق 76
1 – V, ما معنى FAQ ؟ 76
2 – V, أين أجد رابط الموضوع في المنتدى ؟ 76
3 – V, أريد أن أفهم أكثر بعض الإجابات الموجودة في الـ FAQ, كيف ذاك ؟ 76
4 – V, و استفساراتي الأخرى الغير متعلقة بالـ FAQ ؟ 76
5 – V, هل المشاركة في الـ FAQ مفتوحة للجميع ؟ 76
6 – V, إذا, كيف أشارك فيها ؟ 76
7 – V, عفوا, لقد وجدتُ خطأ في إحدى الإجابات ! 76
8 – V, ما هي المصادر التي اعتمدتَ عليها في كتابة هذه المواضيع ؟ 77

I - الفصل الأول : الخطوات الأولى لتعلم لغة C

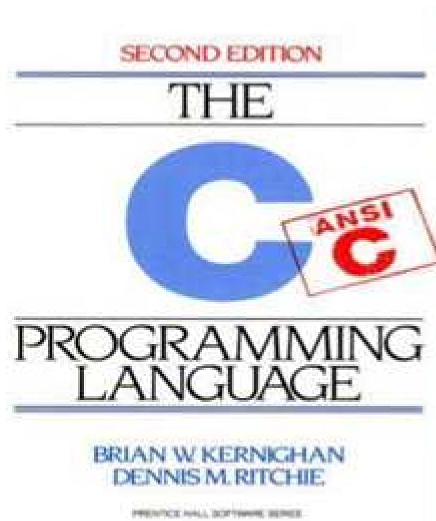
1 - I. نبذة تاريخية عن لغة C

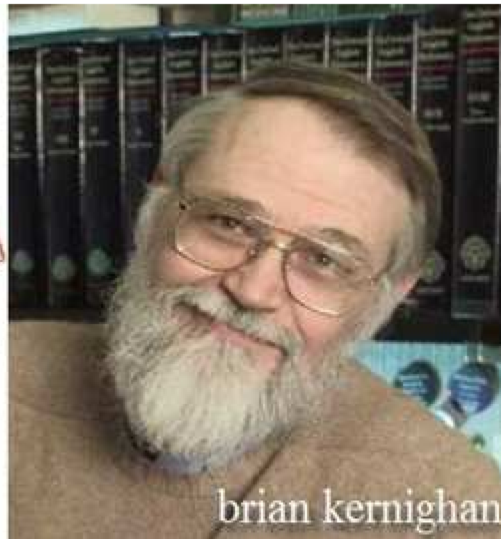
1.1 – I, من أين جاءت لغة C ؟

في الخمسينيات من القرن الماضي كانت توجد مجموعة من لغات البرمجة منها لغة الـ Fortran و الـ Cobol و الـ Basic و Pascal و ..., و كان لكل من هذه اللغات مجالها الخاص !, حيث كانت تُستخدم لغة الفورتران في التطبيقات الهندسية و العلمية .. بينما تُستخدم لغة الكوبول في التطبيقات التجارية .. و هكذا بالنسبة للبقية.

بدأ مطوروا لغات البرمجة يسعون لدمج هذه اللغات في لغة واحدة تتميز برامجها بسرعة التنفيذ و سهولة الاستخدام وكانت المحاولة الأولى من نصيب لغة CPL و هي اختصار لـ :
Combined Programming Language, حيث قامت بتطويرها جامعة كامبريدج (Cambridge) في المملكة المتحدة البريطانية, أما المحاولة الثانية فقد كانت على يد الخبير مارتين ريتشارد (Martin Richards) و هي لغة BCPL, كان ذلك في سنة 1967 في جامعة كامبريدج أيضا, بينما كانت المحاولة الثالثة من طرف السيد كين تومسون (Ken Thompson) وهي لغة B حيث كانت هذه اللغة أحسن من سابقاتها إلا أن ذلك لم يمنع من كونها بطيئة إلى حد ما ! و في سنة 1972 ولدت لغة C على يد خبير يُدعى دنيس ريتشي (Dennis Ritchie) في الولايات المتحدة الأمريكية و بالتحديد في مختبرات Bell.

في عام 1978 قام دنيس ريتشي و براين كارنيغان (Brian Kernighan) بتأليف أول كتاب لهذه اللغة تحت عنوان : The C Programming Language و الذي يعتبر المرجع الأساسي لهذه اللغة.





I – 1.2, ما هي مميزاتها ؟

كانت لغة C تحتوي على كثير من المميزات التي يرغب بها المبرمجون مثل السرعة و كونها متنقلة (حيث يمكن استعمالها تحت أكثر من نظام تشغيل) و هي لغة قياسية أيضا ففي عام 1989 تم تعريف نسخة قياسية من لغة C و سميت بـ ANSI C و هي مختصرة من American National Standards Institute C أي اللجنة الوطنية الأمريكية للمعايير, و بتعاون بين اللجنة الوطنية الأمريكية للمعايير و المنظمة العالمية للمعايير تم إطلاق لغة C القياسية سنة 1990 و سميت بـ ISO C و هي اختصار لـ :

International Organization For Standarizatoin.

I – 2. الأدوات اللازمة

لكي نقوم بتنفيذ أي كود برمجي في لغة C لا بد من وجود برنامج يُقال له الـ compiler أو المترجم, و لكن .. ما هو المترجم ؟ و لماذا نحتاج إليه ؟ و ما هي أشهر المترجمات المتداولة حاليا ؟

I – 2.1, ما هو المترجم !؟

المترجم (بشكل عام) هو برنامج يقوم بتحويل كود مكتوب بلغة برمجة عالية المستوى إلى لغة برمجة أخرى منخفضة المستوى, و بشكل خاص فإن المترجم هو برنامج يقوم بتحويل كود C إلى لغة منخفضة المستوى (قد تشبه لغة الأسمبلي) تسمى Intermediate Representation .

I – 2.2, لماذا نحتاجه !؟

لنفرض أنك لا تعرف اللغة اليابانية و ذهبت إلى اليابان للقاء أحد الأشخاص هناك !
الآن لديك مشكلة .. فأنت تريد التحدث مع شخص لا يعرف سوى اليابانية و أنت لا تعرف لغته !, يعني باختصار: كلاكما لا يعرف لغة الآخر.
إذا أردتُما التحدث إلى بعضكما البعض فلابد من وجود شخص ثالث يعمل كمترجم فيقوم بتحويل كلامك إلى اللغة اليابانية (لكي يفهمه الياباني) و يحول كلام الياباني إلى اللغة العربية (لكي تفهمه أنت), إذن فهو يعمل كوسيط بينكما لكي يفهم كل منكما تعليمات الآخر.
التعريف السابق بنفس الشكل والصورة هو الذي يحدث في عالم البرمجة, فجميعنا نعرف أن الحاسب (المعالج والذاكرة) لا يتعاملان إلا مع الأرقام الثنائية 0 و 1 (سريان تيار أو عدم سريانه), فلكي يفهم الحاسوب برامجنا لا بد أن يتم تحويلها إلى لغة الآلة , وهذا بالضبط ما يفعله المترجم حيث يقوم بالتحويل من لغة إلى أخرى (قد تكون لغة الآلة أو لغة أخرى على حسب نوع المترجم والغرض المصمم من أجله.)
عوداً على بدء .. و لكي يكون الكلام أدق فإن الشخص الثالث (المترجم) يجب أن يقوم بالترجمة بشكل صحيح لذلك عليه أن يفهم ما الذي يقوله الشخص الأول (تحليل الكلام) ومن ثم يقوم بترجمة ما فهمه إلى الشخص الثاني (توليد الكلام).
نفس الأمر يحدث لل compiler حيث يقوم بتحليل النص أولا في مرحلة يمكن أن نطلق عليها إجمالا Analysis Phases, بعدها يقوم بتحويل الكود إلى اللغة الجديدة المراد الوصول إليها وهي مرحلة التوليد أو ما يعرف بـ Synthesis Phases.

I – 2.3, ولكن .. لماذا يحلل المترجم وبعدها يبدأ بالتوليد ؟

سنوضح هذا السؤال بسؤال آخر !, لنفرض أنك تتكلم مع الشخص الثالث (ليترجم ما قلتَه إلى الياباني) و قلت له كلام ليس بالعربية ! (المترجم يحول من العربية إلى اليابانية فقط), الآن سيعترض المترجم .. لأن الكلام الذي قلتَه لا يملك معنى في العربية حتى يتم تحويله إلى اليابانية, نفس الشيء بالنسبة لل compiler , فالمترجم يحلل الكلام للتأكد من أن البرنامج صحيح وموافق تماما لقواعد اللغة, فإذا كان هناك خطأ ما في هذه المرحلة سيتوقف المترجم عن العمل وسيخبرك بمكان هذا الخطأ, إذا كان الخطأ نحوي (يتعلق بقواعد اللغةSyntax Error) سيتوقف المترجم مباشرة عن العمل, أما في الأخطاء الأخرى فلن يتوقف عن العمل وسيكمل مرحلة تحليله واكتشاف باقي الأخطاء, طبعا هذا ما يعرف ب Error Recovery.

2.4 – I, جميع المترجمات الحديثة متوفرة لها عدة IDE, ماذا يعني هذا المصطلح؟

أولا الكلمة IDE مختصرة من Integrated Development Environment أي بيئة تطوير متكاملة, حيث تساعد هذه المترجمات المبرمج في كل من التحرير,الترجمة و الربط ففي السابق كانت الترجمة و الربط يتمان على شكل أوامر ! أما في الـ IDE فأصبحت عملية الربط و الترجمة تتم عن طريق الضغط على زر واحد من لوحة المفاتيح. باختصار, بيئة التطوير هي من يقوم بتنفيذ هذه الأوامر بدلا منك.

2.5 – I, ما هي أشهر الـ IDE المُستخدمة حاليا ؟

يوجد العديد من الـ IDE .. منها, العملاق Visual C++ المقدم من شركة Microsoft و الرائع Dev المقدم من شركة Bloodshed و العجوز Borland, المقدم من شركة Borland, كما يوجد أيضا الأنيق Code::Blocks, بالإضافة إلى الـ IDE الممتاز و الذي أفضله شخصيا وهو NetBeans و إذا أردنا سرد أسماء جميع المترجمات المتميزة فالمقام يطول ...!

2.6 – I, ما هي سلبيات و إيجابيات كلٍّ من Visual C++, Code::Blocks, NetBeans ؟

1. Code::Block
   - ☜ سلبياته : لا يحتوي على العديد من الخيارات المتقدمة, لكنه مُناسب للمبتدئين-المتوسطين.
   - ☜ إيجابياته : سريع, خفيف ومجاني أيضا, كما أنه متعدد المنصات.
2. Visual C++
   - ☜ سلبياته : يعمل على الـ Windows فقط, كما أنه كبير الحجم و بطيء شيئا ما !, غير مجاني أيضا.
   - ☜ إيجابياته : يدعم أكثر من لغة برمجة. يمكنك من خلاله الاتصال بقواعد البيانات. يحتوي على Debugger قوي جدا !
3. Netbeans
   - ☜ سلبياته : حجمه كبير شيئا ما لكنه أصغر حجم من الـ VC.
   - ☜ إيجابياته : يدعم مُعظم لغات البرمجة (C, C++, J2SE, J2EE, J2ME, FORTRAN, PHP, ...), سريع, خفيف ومجاني, كما أنه متعدد المنصات. يمكنك أيضا من خلاله الاتصال بقواعد البيانات.

II - الفصل الثاني : المتغيرات و المؤثرات

II – 1. المتغيرات

II – 1.1, ما هو النوع char و ما هي المساحة التي يمثلها ؟

نوع البيانات char وعلى المستوى المنخفض هو أصغر جزء يمكن حجزه في الذاكرة لتخزين أي عدد صحيح ضمن ترتيب تسلسلي معين وهو ما يُعرف بـ range. حتى وإن كان المتغير من نوع char يُعامل على أنه حرف أو رمز مطبوع أو غير مطبوع على الشاشة فإنه ذو أصول عريقة تنحدر من سلالة نوع البيانات للأعداد الصحيحة وهو int.

حسب التعريف فإن Size Of Char تساوي 1, مما يعني أن النوع char يأخذ من الذاكرة مساحة قدرها 1 بايت, البايت الواحد يُساوي على الأقل 8 بت و الحجم الحقيقي لـ char بالبت مُخزن في الماكرو CHAR_BIT الموجود في المكتبة limits.h, انظر المثال:

```c
#include<stdio.h>
#include<limits.h>

int main() {
    short const SIZE_CHAR = CHAR_BIT;
    printf("Size of char = %d bits", SIZE_CHAR);
    return 0;
}
```

II – 1.2, ما هي المساحة التي يمثلها int ؟

المساحة التي يمثلها int في الذاكرة تتناسب مع الحاسب الذي تعمل عليه, فإذا كنت تعمل على نظام 16 بت (مثل Ms-Dos) فستكون مساحة int تساوي 2 بايت و إذا كنت تعمل على نظام 32 بت (مثل Windows 2000) فستجد أن مساحة int هي 4 بايت أما إذا كنت تعمل على نظام 64 بت (مثل ويندوز 7-x64) فستجد أن مساحة int تساوي 8 بايت.

II – 1.3, أحيانا, ألاحظ أن حجم int مساوي لـ short أو long أو long long !, لماذا ؟

عندما تستخدم int فإن المترجم سيعتبره short أو long أو long long و ذلك حسب إعدادات المترجم الافتراضية.

II – 1.4, ماذا عن اختلاف حجم int بالنسبة للبيئة التي يعمل عليها البرنامج ؟

إذا عمل برنامجك على نظام تشغيل 16 بت و أنت تستخدم int فسيكون مساحته 2 بايت و هذا قد يجعل برنامجك يفشل في تطبيق ما يريد !,فهو يتوقع أن حجم int هو 4 بايت و قد ظهر أن مساحته هي 2 بايت !

II – 1.5, ما الحل ؟

الحل هنا يكمن في استخدام طريقه تسمح لك بتحديد المساحة التي يستهلكها int من الذاكرة أو بمعنى أدق مدى الأرقام الذي يمكن تمثيله بـ int و ذلك سواء كنت تعمل على نظام تشغيل 16 بت أو 32 بت أو 64 بت أو غيرهم حيث ستكون المساحة التي يستخدمها برنامجك واحدة في كل الأحوال. هذه الطريقة تتم باستخدام ما يسمى بـ Length Modifier أو "محددات المساحة" إن صح التعبير !.

II – 1.6, و ما هي الـ Length Modifier ؟

الـ Length Modifier هي كلمات محجوزة مسبقا, باستخدامها مع int تجعل مدى الأرقام الذي يمكن تمثيله به ثابت و لا يتغير من نظام تشغيل لآخر أو بمعنى أدق من Architecture لأخرى و بذلك تكون وصلت إلى ما تريد. الـ Length Modifier هي :

```
signed short
signed long
signed long long
unsigned short
unsigned long
unsigned long long
```

II – 1.7, إذا, توجد أربعة أنواع للأرقام الصحيحة ؟

لم أقل هذا, قلتُ أنه يوجد نوع بيانات واحد للأرقام الصحيحة و هو int و يمكننا تغيير المدى الخاص به عن طريق الـ Length Modifier لنحصل على أنواع جديدة مشتقة منه و هي short و long و long long كأن نكتب:

```
short int//short int = short
long int// long int = long
long long int// long long int = long long
```

II – 1.8, هل تحدد اللغة أحجام البيانات ؟

اللغة لا تحدد حجم البيانات لأنواع البيانات و لكنها تحدد القيمة العظمى و الصغرى للأنواع الرقمية و نظرا لأن القيمة العظمى و الصغرى لهم تستهلك مساحات بقدر 2 و 4 و 8 بايت فقد تم التعارف على أن أحجام البيانات لهم هي بالمساحات السابق ذكرها و يمكنك مراجعة الملف limits.h (خاص بلغةc) و الفئة numeric_limits (خاصة بلغة cpp) حتى تتعرف بشكل أكبر على هذه الأنواع.

II – 1.9, إذا, اللغة لا تهتم بالمساحة التي يستهلكها int ؟

صحيح, اللغة لا تهتم بالمساحة التي يستهلكها int من الذاكرة و إنما تهتم بمدى الأرقام الذي يمكن تمثيله داخل هذا النوع و ذلك لأنه باختلاف العتاد تختلف المعايير المتعلقة بحجز الذاكرة و تحديد المساحات لذا و حتى تصبح اللغة مستقلة عن العتاد لابد من تحديد أسس معينه يتم العمل بها و هي في هذه الحالة مدى الأرقام و ليس المساحة المستهلكة من الذاكرة.

II – 1.10, كلمة أخيرة عن أحجام البيانات ؟

إذا قمت بالنظر داخل الملف limits.h أو الفئه numeric_limits ستجد أنه يعطيك المدى المسموح به من الأرقام لنوع معين و هو ما تشترطه اللغة, أما المساحة التي يحجزها من الذاكرة لتمثيل ذلك المدى فهي تعود للمترجم و يمكن أن تختلف من أحدهم لآخر.

II – 1.11, ماذا تعني الكلمة signed ؟

الكلمة signed هي إحدى الكلمات المحجوزة مسبقا و تنتمي إلى عائلة محددات الأسماء أو ما يعرف بـ Length Modifier , هذه الكلمة تُستخدم مع الأنواع الصحيحة فقط و تعني أن المتغير المستخدم معها سيحتوي على إشارة (ربما تكون موجبة أو سالبة). الكلمة signed تُمثل الحالة الافتراضية للمتغيرات (لأن المتغيرات بالأصل يمكنها تخزين أعداد موجبة أو سالبة).

أما unsigned فتعني أن المتغير سيحوي قيم موجبة فقط.

II – 1.12, المترجم الخاص بي لا يعرف النوع long long, لماذا ؟

لأن هذا النوع لم يتم تضمينه إلا في المعيار C99, ربما تستخدم مترجم لا يدعم هذا المعيار أو ربما قمتَ بإعداد المترجم الخاص بك على المعيار C90.

II - 1.13, هل تدعم لغة C المتغيرات المنطقية ؟

قبل مقياس C99 لم تكن توجد متغيرات من النوع المنطقي bool ولكن من السهل جدا إنشاء هذا النوع من المتغيرات إذا كان المترجم لا يقدم تلك الخدمة :

```
#define BOOL int
#define TRUE 1
#define FALSE 0
```

أو :

```
typedef enum {
    FALSE, TRUE
} BOOL;
```

في لغة C كل قيمة معدومة (0,0.0,...) تعني false بينما تعني كل قيمة غير معدومة (9.06,-4,...) القيمة true. المقياس C99 أدخل النوع _BOOL الموجود في الملف الرأسي stdbool.h بالإضافة إلى الماكرو:

```
__bool_true_false_are_defined
```

الذي يسمح بمعرفة ما إذا كانت المتغيرات المنطقية معتمدة أصلا أم لا.

يمكننا أيضا استخدام الأنواع الأصلية من أجل تعريف نوع جديد في حالة عدم توفر النوع bool :

```
#ifndef __bool_true_false_are_defined
# define bool int
# define true 1
# define false 0
# define __bool_true_false_are_defined
#endif
```

II - 1.14, ما فائدة المؤثر sizeof ؟

هذا المؤثر يسمح بمعرفة أحجام البيانات و تكون النتيجة بالبايت bytes, انظر المثال:

```
#include<stdio.h>

int main() {
    int SizeOfArray[800];

    printf("Char = %d Byte(s)\n", sizeof (char));

    printf("Bool = %d Byte(s)\n", sizeof (bool));

    printf("Short = %d Byte(s)\n", sizeof (short));
    printf("Int = %d Byte(s)\n", sizeof (int));
    printf("Long = %d Byte(s)\n", sizeof (long));
    printf("Long long = %d Byte(s)\n", sizeof (long long));

    printf("Float = %d Byte(s)\n", sizeof (float));
    printf("Double = %d Byte(s)\n", sizeof (double));
    printf("Long double = %d Byte(s)\n", sizeof (long double));

    printf("SizeOfArray = %d Byte(s)\n", sizeof (SizeOfArray));

    return 0;
}
```

II – 1.5, هل يمكننا معرفة الحجم الكامل المستعمل للبرنامج عن طريق sizeof ؟

لا يمكن لأن حجم البرنامج النهائي يمكن أن يدخل فيه مكتبات أخرى بالإضافة الى هيكلة الملف الناتج نفسه ليعمل على Environment محددة وهو شيء خارج عن مهام الCompiler.

II – 1.16, ما هو النوع size_t ؟

هذا النوع يُمثل القيمة المعُادة من طرف المؤثر sizeof وهو مُعرف في الملف الرأسي stddef.h, البعض يعتبره مُكافئا لـ unsigned long.

II – 1.17, ما هو النوع wchar_t ؟

يُستخدم هذا النوع عادة عند التعامل مع الكلمات العربية, يأخذ من الذاكرة مساحة قدرها 2 بايت و الترميز المستخدم يعود للمترجم و في الأغلب يكون إما Universal Character Set الخاص بـ ISO أو UTF-16 الخاص بـ Unicode على عكس char الذي يدعم ترميز الـ ASCII فقط أو بعض الـ encoding المشابهة لها.

يوجد تشابه كبير بين الدوال التي تُستخدم مع char و الأخرى التي تُستخدم مع wchar_t, مثل wprintf بدلا من printf و wcscpy بدلا من strcpy و iswaplha بدلا من isaplha, يمكنك تضمين wchar.h أو wctype.h حسب الدوال التي ستستخدمها, انظر المثال:

```c
#include <stdio.h>
#include <wchar.h>

int main() {
    wchar_t ws[256];
    wcscpy(ws, L"Hello");
    wcscat(ws, L", world !");
    wprintf(L"%s\n", ws);
    return 0;
}
```

II – 1.18, كيف أنشئ اسما مستعارا للأنواع الموجودة في اللغة ؟

يتم ذلك باستخدام الكلمة المحجوزة مُسبقا typedef حيث تُستعمل هذه الكلمة لإعطاء أنواع المتغيرات أسماء مستعارة, و طريقة استخدامها تكون بكتابة الكلمة ثم نوع البيانات متبوعا بالاسم المستعار هكذا:

```c
typedef DataType AliaseName
```

أمثلة :

```
typedef int VECTEUR[3];
typedef void * POINTEUR;
typedef struct { int value; } INTVALUE;
VECTEUR u, v; /* int u[3], int v[3]; */
POINTEUR p1, p2; /* void * p1, * p2; */
INTVALUE n; /* struct { int value; } n; */
```

II – 1.19, ماذا تعني الكلمة const ؟

تُستخدم هذه الكلمة للإعلان عن الثوابت حيث يجب إسناد قيمة ابتدائية للثابت و لا يمكن تغيير تلك القيمة فيما بعد.

II – 1.20, ما هي الرموز المُستخدمة مع الأنواع الصحيحة ؟

- الرموز التالية %d,%i,%o,%x تُمثل القيم التي من النوع int أو unsigned int فقط و إذا أردنا استخدامها مع الأعداد الصحيحة المضاعفة فإننا نستخدم الرموز التالية %ld,%li,%lo,%lx
- إذا أردنا إظهار عدد صحيح قصير موجب unsigned short فإننا نستخدم الرمز %hu و نستخدم الرمز %lu مع unsigned long أما إذا أردنا إظهار عدد صحيح مضاعف ضخم موجب فإننا نستخدم الرمز %llu. (راجع السؤال التاسع من الفصل الرابع لمعرفة المزيد)

II – 1.21, متى أستخدم if-else و متى أستخدم switch ؟

يُنصح باستخدام الجمل الشرطية if,else عند التعامل مع الأعداد الحقيقة أو الجمل. من عيوب switch أنها لا تدعم الأعداد الحقيقة و الجمل ! فهي لا تتعامل إلا مع الأعداد الصحيحة و الحروف, بالنسبة للحروف فهي تتعامل مع الشفرة المقابلة للحرف في جدول ASCII.

II – 1.22, ماذا يحدث للمتغيرات الغير مهيأة (uninitialized variable) ؟

المتغيرات الغير مهيأة هي المتغيرات التي تم التصريح عنها و لكن لم يتم إدخال قيمة لها , يقوم المترجم بإعطائك التحذير unreferenced local variable و في حالة نسخ الـ debug من البرنامج يتم حجز مساحة لها داخل الـ stack و لكن يترك مكانها كما هو بالقيمة التي يحتوى عليها و تسمى القيمة التي يحتوي عليها في هذا الوقت بـ junk أي قيم غير صحيحة.

II – 2, المؤثرات

II – 2.1, ما الفرق بين المؤثر = و المؤثر == ؟

المؤثر = هو معامل الإسناد, حيث يقوم بإسناد القيمة الموجودة على يمينه إلى المتغير الموجود في اليسار, أما المؤثر == فيُستعمل للمقارنة حيث يُعيد true إذا تساوى الطرفان و false في العكس.

II – 2.2, ما هو المؤثر الثلاثي (ternary operator) ؟

المؤثر الثلاثي هو تعبير يستخدم عادة في الشروط البسيطة و شكله العام كالتالي:

```
Condition ? doThis : doThis ;
```

إذا كان الشرط محقق فسيتم تنفيذ الأمر الموجود قبل الرمز : و إلا فسيتم تنفيذ الأمر الموجود بعده و هذا يعني أن تركيبة المؤثر الثلاثي تكافيء:

```
if(Condition == true)
do this;
else
do this;
```

يمكن أيضا استخدام المؤثر الثلاثي في الشروط المتداخلة :

```
if (x > y) {
    if (x > z) return x;
    else return z;
} else {
    if (y > z) return y;
    else return z;
}
```

هكذا:

```
( x > y) ? ((x > z) ? x : z) : (y > z) ? y : z;
```

II – 2.3, هل يمكن استخدام المؤثر الثلاثي في حالة وجود أكثر من statement ؟

```
#include <stdio.h>

int main() {
    int x = 0;
    if (x != 0) {
        x++;
        printf("%d", x);
    } else {
        x--;
        printf("%d", x);
    }
    return 0;
}
```

نعم, يمكن إستخدام الـ ternary operator في حالة وجود أكثر من statement داخل كل block عن طريق استخدام الـ sequence operator (المعامل comma) كالتالي:

```
#include <stdio.h>

int main() {
    int x = 0;
    x != 0 ? x++, printf("%d", x) : x--, printf("%d", x);
    return 0;
}
```

II – 2.4, كيف نحسب باقي القسمة ؟

توجد 5 طرق لحساب باقي القسمة:

```
/******** Methode 1 : *********/
cout << a << " % " << b << " = " << a % b << endl;

/******** Methode 2 : *********/
int c;
c = a / b;
c *= b;
cout << a << " % " << b << " = " << a - c << endl;

/******** Methode 3 : *********/
c = 0;
while (c + b < a)
    c += b;
cout << a << " % " << b << " = " << a - c << endl;

/******** Methode 4 : *********/
c = a;
for (; c > b;)
    c -= b;
cout << a << " % " << b << " = " << c << endl;

/******** Methode 5 : *********/
div_t divresult; //#include<cstdlib>
divresult = div(a, b);
cout << a << " % " << b << " = " << divresult.rem << endl;
```

II – 2.5, ما الفرق بين الزيادة القبلية (++i) و الزيادة البعدية (i++) ؟

يختلفان من ناحية الأسبقية, فالزيادة القبلية تُنفذ قبل تنفيذ جميع الأوامر المرافقة لها أما الزيادة البعدية فيتم تنفيذها بعد تنفيذ جميع الأوامر المرافقة لها.

2.6 - II, كيف يعمل المؤثر AND (&&) ؟

المؤثر && يعتبر حرف العطف "و" في لغة البشر, فلو قلت لك: "ذهبت للبيت أنا و أخي"
ثم أخبرتك أن أخي لم يذهب !, حينئذ ستكون الجملة غير صحيحة و كذلك لو قلت لك أن أخي ذهب لكنني لم أذهب, أولم نذهب نحن الاثنان معا ! ففي الأحوال السابقة تكون الجملة غير صحيحة.تكون الجملة صحيحة في حالة واحدة ووحيدة و هي أن نذهب أنا و أخي معا إلى البيت.
مما سبق نجد أن هذا المؤثر يعمل على ربط علاقتين منطقيتين و تكون نتيجة العبارة الجديدة صحيحة إذا كانت العبارتين صحيحتين في نفس الوقت أما في باقي الحالات فالنتيجة خاطئة والجدول التالي يوضح ما سبق:

P	Q	P&&Q
False	False	False
False	True	False
True	False	Flase
True	True	True

2.7 - II, كيف يعمل المؤثر OR (||) ؟

المؤثر "||" يعتبر "أو" في لغة البشر, لو قلت لك "سأجلس للبرمجة أو تصفح الانترنت"
فلو جلست للبرمجة فقط فكلامي صحيح, أو جلست لتصفح الانترنت فقط فكلامي صحيح أيضا, ولو جلست و فعلت الاثنتين معا لكان كلامي صحيحا أيضا !
تكون هذه الجملة خاطئة في حالة وحيدة, هي عندما لا أجلس للبرمجة و لا أتصفح الانترنت.
مما سبق نجد أن هذا المؤثر يعمل على ربط علاقتين منطقيتين و تكون نتيجة العبارة الجديدة خاطئة إذا كانت العبارتين خاطئتين في نفس الوقت أما في باقي الحالات فالنتيجة صحيحة و الجدول التالي يوضح ما سبق:

P	*Q*	*P*||*Q*
False	*False*	*False*
False	*True*	*True*
True	*False*	*True*
True	*True*	*True*

2.8 - II, كيف يعمل المؤثر AND_BIT (&) ؟

هذا المؤثر يقارن كل بت من العدد الأول مع البت المقابل له من العدد الثاني و هذه طريقة عمله:

```
/*
   Bitwise :And Operator
   True = 1
   False = 0
 */
1 & 1 = 1
1 & 0 = 0
0 & 1 = 0
0 & 0 = 0
```

مثال:

```
/*
   Bitwise :And Operator
   True = 1
   False = 0
*/
23 & 11 = 3
    00010111 // 23 decimal = 00010111 Binary
&
    00001011// 11 decimal = 00001011 Binary
    ------------
=
    00000011// 3 decimal = 00000011 Binary
```

2.9 - II, كيف يعمل المؤثر OR_BIT (|) ؟

هذا المؤثر يقارن أيضا بين البتات المتقابلة لعددين و يعطي عدد جديد بناء على القاعدة التالية:

```
/*
   Bitwise :Or Operator
   True = 1
   False = 0
*/
1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0
```

مثال:

```
/*
   Bitwise :Or Operator
   True = 1
   False = 0
*/
23 | 11 = 31
    00010111 // 23 decimal = 00010111 Binary
&
    00001011// 11 decimal = 00001011 Binary
    ------------
=
    00011111// 31 decimal = 00000011 Binary
```

2.10 – II, كيف يعمل المؤثر NOT_BIT (~) ؟

هذا المؤثر يسمى"المتمم", كل ما يقوم به هو عكس قيمة الـ Bit فلو كان 0 و طبقنا عليه المؤثر فسيصبح 1 و العكس صحيح:

```
/*
   Bitwise :NOT Operator
   True = 1
   False = 0
*/
~0 =1
~1= 0
```

هذا المؤثر يأخذ قيمة واحدة فقط ليعطينا المتمم لها:

```
/*
   Bitwise :NOT Operator
   True = 1
   False = 0
*/
~(255) = 0
~(11111111) = 00000000
```

2.11 – II, كيف يعمل المؤثر XOR (^) ؟

يسمى هذا المؤثر Exclusive OR, حيث يعيد True فقط إذا كان المدخلان مختلفين, و يعيد False إذا كانا متشابهان:

```
/*
   Bitwise :XOR Operator
   True = 1
   False = 0
*/
1 ^ 1 = 0
1 ^ 0 = 1
0 ^ 1 = 1
0 ^ 0 = 0
```

و كمثال على ذلك:

```
/*
   Bitwise :XOR Operator
   True = 1
   False = 0
*/
15 ^ 8 = 7
    00001111 // 15 decimal = 00001111 Binary
^
    00001000 // 8 decimal =  00001000 Binary
    ------------
=
    00000111 // 7 decimal = 00000111 Binary
```

و يُستعمل هذا المؤثر بكثرة في تشفير النصوص.

II – 2.12, كيف تعمل مؤثرات الإزاحة ؟

- **<<** تقوم بإزاحة بتات المتغير إلى اليسار بمقدار معين من الخانات
- **>>** تقوم بإزاحة بتات المتغير إلى اليمين بمقدار معين من الخانات
- **=<<** تقوم بإزاحة بتات المتغير إلى اليسار بمقدار معين من الخانات و يتم إسناد الناتج إلى نفس المتغير
- **=>>** تقوم بإزاحة بتات المتغير إلى اليمين بمقدار معين من الخانات و يتم إسناد الناتج إلى نفس المتغير

```
/*
   Bitwise : Shifting Operator
   Right shift
*/
10010010 >> 5 = 00000100
//10010 this part is taken out because of the boundries of the byte.
//146 decimal = 10010010 Binary
//4 decimal = 00000100 Binary
```

لاحظ أن العدد المكون من خمس خانات على يمين العدد الناتج عن الإزاحة يمثل الجزء الضائع لأنه خرج عن حدود الـ Byte و بالتالي لم يعد موجودا أصلا, أما الخمس خانات الجديدة على اليسار فهي أصفار دائما, أي أن هناك جزء ضائع و أصفار مكتوبة لتعبئة الفراغ المتولد من الإزاحة. هذا ما تفعله الإزاحة إلى اليمين أو إلى اليسار بشكل كامل.

مثال آخر:

```
unsigned char c1, c2, c3, c4;
c1 = 5; /* 00000101 */
c2 = 4; /* 00000100 */
c3 = c1 << 2; /* 00000101(c1) -> 00010100(c3) */
c4 = c2 >> 2; /* 00000100(c2) -> 00000001(c4) */
c1 <<= 2; /* 00000101(c1) -> 00010100(c1) */
c2 >>= 2; /* 00000100(c2) -> 00000001(c2) */
```

2.13 – II, ما هو ترتيب أولوية المؤثرات ؟

الجدول التالي يُرتب جميع المؤثرات الموجودة في C++ (بما في ذلك تلك الموجودة في C), حسب أولويتها :

TABLE 2.4 Operator Precedence

Level	Operator Description	Operator
1	Scope resolution	::
2	Post-increment	++
	Post-decrement	--
	Function call	()
	Array Element	[]
	Pointer to member of	->
	Member of	.
3	Pre-increment	++
	Pre-decrement	--
	Logical NOT	|
	Bitwise NOT	~
	Unary minus	-
	Unary plus	+
	Address of	&
	Indirection of	*
	Size of	sizeof
	New allocation	new
	De-allocation	delete
	Typecast	(type)
4	Pointer to member object	.*
	Pointer to member pointer	->*
5	Multiplication	*
	Division	/
	Remainder	%
6	Addition	+
	Subtraction	-
7	Left shift	<<
	Right shift	>>
8	Less than	<
	Less than or equal to	<=
	Greater than	>
	Greater than or equal to	>=
9	Equal to	=
	Not equal to	!=
10	Bitwise AND	&
11	Bitwise exclusive OR	^
12	Bitwise inclusive OR	|
13	Logical AND	&&
14	Logical OR	||
15	Conditional	? :
16	Assignment	=
17	Compound assignment	+= /= %= += -= <<= >>
18	Comma	,

المصدر

Game Programming All in One, Bruno Miguel Teixeira De Sousa, Premier Press, 1 févr. 2002 - 952

III - الفصل الثالث : المؤشرات, المصفوفات الحرفية و التراكيب

1 – III. المؤشرات

1.1 – III, ما هو المؤشر ؟

هناك 3 أشياء أساسية لفهم تعريف المؤشر :

- كل متغير يملك عنوان في الذاكرة, من خلال هذا العنوان يُمكن للمعالج أن يصل إلى قيمة المتغير و يُجري عليها بعض العمليات (القراءة, الكتابة, ...).
- المؤشر هو متغير يحوي عنوان في الذاكرة (عنوان متغير, عنوان دالة, عنوان بنية, ...).
- هناك خطأ شائع عند البعض و هو استخدام كلمة "مؤشر" من أجل وصف "عنوان".

1.2 – III, متى تُستخدم المؤشرات ؟

من الصعب حصر جميع المجالات التي تُستخدم فيها المؤشرات ولكن إليك بعض الأمثلة:

- عندما نستخدم التمرير بواسطة المرجع (Pass by reference).
- عندما نريد تمرير كائن ذو حجم كبير, التمرير بواسطة المرجع سيكون أقل تكلفة من حيث الوقت و الذاكرة أيضا. تمرير نسخة من الكائن يتطلب نسخ محتويات الكائن من جديد.
- عند الحاجة إلى حجز ديناميكي للذاكرة.

1.3 – III, ما معنى NULL ؟

هذا الثابت عبارة عن MACRO مُعرف في المكتبة stddef.h, يُستخدم عادة في الدوال كقيمة مُعادة للدلالة على حدوث خلل في عمل الدالة.

هناك أيضا المؤشر NULL الذي يملك العنوان 0x00000000 وفي العادة يكون هذا العنوان invalid memory address في أغلب نظم التشغيل لكن ليس دائما وليس في كل الحالات, يُستخدم الـ null pointer للدلالة على أن المؤشر فارغ أي لم يتم حجز ذاكرة له بعد.

III – 1.4, ما الفرق بين الـ Null Pointer و Uninitialized Pointer ؟

المؤشر الغير مُهيأ (Uninitialized Pointer) هو المؤشر الذي لم نحجز له عنوانا محددا في الذاكرة, ربما يتسبب مثل هذا النوع من المؤشرات في تعطيل برنامجك إذا لم تقم بتهيئته لأنه و بكل بساطة يُشير إلى عنوان غير متوقع في الذاكرة. انظر المثال :

```
char *p1 = malloc(sizeof (char)); // (undefined) value of some place on the heap
char *p2; // wild (uninitialized) pointer
*p1 = 'a'; // This is OK, assuming malloc() has not returned NULL
*p2 = 'b'; // This invokes undefined behavior
```

ربما تشير p2 إلى أي مكان في الذاكرة، لذا فإن هذه الخطوة *p2 = 'b' قد تُفسد منطقة غير معروفة من الذاكرة و ربما تحتوي على بيانات حساسة.

أحد الفروق بين الـ Null Pointer و Uninitialized Pointer, يكمن في المقارنة .. حيث يمكننا أن نقارن الـ null pointer مع عنوان أي مؤشر آخر بينما لا يمكننا فعل ذلك مع المؤشرات الغير مهيأة.

III – 1.5, ما معنى stack, heap, buffer ؟

☜ الـ Stack: هو مكان بالذاكرة لابد من تحديد المساحة التي سيتم حجزها فيه وقت ترجمة البرنامج، هذا الجزء من الذاكرة يحتوى على المتغيرات التي تم تعريفها داخل الكود أيا كان نوع هذه المتغيرات (ماعدا المتغيرات العامة) و أيضا المصفوفات التي تم تحديد حجم لها بشكل مباشر أو غير مباشر، أيضا تحتوى هذه الذاكرة على شجرة استدعاءات الدوال و قيم المعاملات. هذه الذاكرة يتم تحريرها بمجرد العودة للدالة التي قامت بالاستدعاء أو بمجرد الانتهاء من تنفيذ الدالة الحالية.

☜ الـ Heap: هو مكان بالذاكرة يمكن تحديد مساحته وقت ترجمة البرنامج أو وقت تشغيل البرنامج و لا يتم الحجز الفعلي للذاكرة إلا باستدعاء الدوال التي تقوم بتنفيذ عملية الحجز و حيث أنك تقوم بعملية الحجز بشكل يدوى فإنك ملزم بتنفيذ عملية تحرير الذاكرة بشكل يدوى أيضا و إن لم تقم بتحرير الذاكرة قد يحدث Memory Leak مما قد يؤدى إلى إنهاء برنامجك بشكل غير طبيعي أو أمور أسوء.

☜ الـ Buffer: هي ذاكرة مؤقتة لتخزين البيانات أثناء انتقالها من طرف إلى آخر بحيث تختلف السرعة بين كلا الطرفين, باعتبار أن وصول البيانات إلى الطرف الثاني أسرع .. فإن دور الـ buffer يكمن في حفظ البيانات ريثما يكتمل وصولها إلى الطرف الأول لتكمل طريقها إلى الطرف الثاني و الذي غالبا ما يكون المعالج. على سبيل المثال : يتم استخدام الـ buffer عند كل عملية إدخال لأن الإنسان (الطرف الأول) بطبيعة حاله أبطأ من المعالج (الطرف الثاني). يمكن أن يكون مكان الـ Buffer في Stack او الـ Heap.

1.6 – III, ما الفرق بين تحرير و تصفير الذاكرة ؟؟

تحرير الذاكرة يعني إلغاء حجز العنوان الذي كان يشير إليه المؤشر. حيث يصبح ذلك العنوان حُراً و بإمكان نظام التشغيل أن يحجزه لبرنامج آخر من جديد. بالنسبة لمصطلح التصفير فيجب التفريق فيه بين شيئين :

- ذاكرة عنوانها صفر = العنوان الذي يشير إليه المؤشر يساوي صفر.
- ذاكرة مصفرة = البيانات التي تحتويها قيمها أصفار.

1.7 – III, لكن, إذا كان المؤشر يشير إلى ذاكرة عنوانها صفر ألا يعني هذا أن قيمة الخانة المُؤشر عليها تساوي صفر أيضا !؟

لا, في العادة الـnull pointer يشير إلى ذاكرة غير صالحة للاستخدام invalid memory address.

1.8 – III, ما هو المؤشر الثابت ؟

المؤشر الثابت هو المؤشر الذي لا يمكن تغيير عنوانه أثناء عمل البرنامج و يُعرف كالآتي :

```
Type * const Pointer = &VarType;
```

حيث Type نوع المؤشر والـ VarType متغير من نوع Type .

1.9 – III, ما هو المؤشر إلى ثابت ؟

المؤشر إلى ثابت هو المؤشر الذي يشير إلى قيمة لا نستطيع تغييرها من خلال المؤشر ولكن نستطيع تغيير العنوان الذي يشير إليه المؤشر, يمكن كذلك استخدام هذا النوع من المؤشرات في الإشارة للمتغيرات المعرفة على أنها ثابتة وفي هذه الحالة لن تستطيع تغيير القيمة الموجودة في المؤشر من خلال المؤشر أو من خلال المتغير, ويعرف على الشكل التالي :

```
const Type *Pointer = &ValType;
or
const Type *Pointer = &ConstValType;
```

1.10 – III, ما هو المؤشر الثابت إلى ثابت ؟

المؤشر الثابت إلى ثابت هو المؤشر الثابت العنوان, الثابت القيمة أي لا نستطيع تغيير عنوانه أو قيمته ولكن إن كانت القيمة المسندة للمؤشر من متغير غير ثابت ففي هذه الحالة يمكن تغيير قيمة المؤشر ولكن من المتغير وليس المؤشر , ويعرف على الشكل التالي :

```
const Type * const Pointer = &ValType;
or
const Type * const Pointer = &ConstValType;
```

III – 1.11, ما هو المؤشر العائم (Void Pointer) و فيم يُستخدم ؟

يمكنك اعتبار أن مؤشر الـ void يشير إلى كائن غير معروف الطول أو الهوية (مجرد كتلة من الذاكرة), أنت من يحدد كيف يستخدمه, فمثلا يمكنك كتابة دالة تقوم بتصفير ذاكرة أي كائن, في هذه الحالة ستحتاج أن تمرر لها طول الكائن ومؤشر void لأنك لا تعرف إن كنت ما ستقوم بتصفيره هو int أم char أم float أم structure, هنا تظهر أهمية مؤشر الـ void.

تُستخدم أيضا المؤشرات العائمة في تكوين مصفوفة لأنواع مختلطة أو تمرير متغيرات لدوال تتوقع عدة أنواع من البيانات.

III – 1.12, ما هو الـ Pointer to Pointer ؟

هذا النوع من المؤشرات يحوي عنوان, و العنوان بدوره يحوي قيمة. فهو إذا لا يشير إلى قيمة بل إلى مؤشر و يُستخدم عادة للتعامل مع مصفوفة ديناميكية ثنائية البعد, يتم الإعلان عنه هكذا :

```
Type **Pointer;
```

III – 1.13, باعتبار أن t مصفوفة, ما الفرق بين الكتابات التالية (t, &t, &(t[0] ؟

الكتابة الأولى تمثل الكائن (المصفوفة), الكتابة الثانية عبارة عن عنوان المصفوفة, أما الثالثة فتمثل عنوان أول عناصر المصفوفة.

إذا, الكتابات الثلاثة مختلفة عن بعضها البعض و لا توجد بينها نقطة مشتركة.

الكتابة الثانية و الثالثة يملكان نفس العنوان و لكن يختلفان في النوع.

III – 1.14, ما الفرق بين []char a و char * a ؟

تذكر جيدا أنه في لغة C, المصفوفة ليست مؤشر .. حتى ولو كان استخدامهما متقارب إلى حد كبير.

الكتابة []char a تعني مصفوفة حرفية غير معروفة الحجم في البداية, و لكن عند استخدامها هكذا:

```
char a[] = "Hello";
```

تتحول إلى [6]char و تكون مصفوفة حرفية مُكونة من 6 حروف.

أما الكتابة char * a فتعني مؤشر يشير إلى متغير حرفي.

1.15 – III, باعتبار أن t مصفوفة, ما معنى الكتابة (t+3)* ؟

هذه الكتابة مُكافئة لـ t[3], لأن t يمثل عنوان أول عناصر المصفوفة و في حالتنا هذه ستتم إزاحته بمقدار ثلاث خانات إلى الأمام ثم يتم أخذ قيمة الذاكرة التي يشير إليها ذاك المؤشر. انظر المثال:

```c
#include <stdio.h>

int main() {
    int t[] = {10, 20, 30, 40};
    int * p = t;
    /* p pointe sur t[0] donc *p equivaut a t[0]. */
    printf("t[0] = %d\n", *p);
    /* p pointe sur t[0] donc p + 1 pointe sur t[1].
     *(p + 1) equivaut donc a t[1]. */
    printf("t[1] = %d\n", *(p + 1));
    /* p pointe sur t[0] donc p + 2 pointe sur t[2].
     *(p + 2) equivaut donc a t[2]. */
    printf("t[2] = %d\n", *(p + 2));
    /* p pointe sur t[0] donc p + 3 pointe sur t[3].
     *(p + 3) equivaut donc a t[3]. */
    printf("t[3] = %d\n", *(p + 3));
    return 0;
}
```

أو بدون استخدام المؤشر p :

```c
#include <stdio.h>

int main() {
    int t[] = {10, 20, 30, 40};
    printf("t[0] = %d\n", *t);
    printf("t[1] = %d\n", *(t + 1));
    printf("t[2] = %d\n", *(t + 2));
    printf("t[3] = %d\n", *(t + 3));
    return 0;
}
```

1.16 – III, ما معنى هذه الكتابة p++* ؟

المؤثرات الأحادية (*,++,--) لهم أولوية قوية ويتم تنفيذهم من اليمين إلى اليسار, الكتابة p++* تعني قيمة الخانة الموالية للخانة التي يشير إليها المؤشر p أما الكتابة (p*)++ فتعني زيادة قيمة المؤشر p بـ 1.

1.17 – III, ما هو دور المعامل [ ] ؟

هذا المعامل يسمح بالوصول إلى عناصر المصفوفة, يتطلب وسيطين, الأول مؤشر و الثاني عدد صحيح, العبارة X[Y] مُكافئة لـ ((X) + (Y))*. بعبارة أخرى, إذا كان p مؤشر فالكتابات التالية متكافئة :

p[1], 1[p] and *(p + 1).

III – 1.18, ماذا تعني الكتابة التالية int (*p)[4] ؟

هذه الكتابة تقوم بإنشاء متغير p بحيث يكون p* عبارة عن مصفوفة مكونة من 4 خانات, إذا p يشير إلى مصفوفة من 4 خانات, انظر المثال:

```c
#include <stdio.h>

int main() {
    int t[10][4];
    int (*p)[4] = t;
    /* p pointe sur t[0]. *p <=> t[0]. */;
    (*p)[0] = 0; /* t[0][0] = 0 */
    (*p)[1] = 0; /* t[0][1] = 1 */
    (*(p + 1))[0] = 1; /* t[1][0] = 1 */
    (*(p + 1))[1] = 1; /* t[1][1] = 1 */
    printf("t[0][0] = %d\n", t[0][0]);
    printf("t[0][1] = %d\n", t[0][1]);
    printf("t[1][0] = %d\n", t[1][0]);
    printf("t[1][1] = %d\n", t[1][1]);
    return 0;
}
```

III – 1.19, ما هي فائدة الدالة malloc ؟

الدالة malloc تستقبل عدد البايتات (نوعية حجم البيانات) و تعيد مؤشر void يشير إلى مكان الحجز مع العلم أنها لا تقوم بتصفير الذاكرة لذلك سنحتاج إلى الدالة memset (إن أردنا التصفير) التي تستقبل ثلاث معاملات, المعامل الأول هو المؤشر نفسه و الثاني هو القيمة المراد وضعها بداخل الخانة التي يشير إليها المؤشر و الثالث هو عدد البايتات أو كمية البيانات المُؤشر عليها انظر المثال:

```c
char* ptr = malloc(sizeof (char));
if (ptr != NULL) {
    memset(ptr, 0x00, sizeof (char));
    //...
    free(ptr);
}
```

III – 1.20, ما هي فائدة الدالة calloc ؟

الدالة calloc تستقبل معاملين, الأول هو عدد العناصر المراد حجزها، و المعامل الثاني هو حجم العنصر الواحد بالبايت, هذه الدالة تُعيد مؤشر void يشير إلى مكان أول خانة من هذه الخانات مع العلم أنها متتالية في الترتيب لذلك يمكننا الوصول إلى بقية الخانات عن طريقة تحريك المؤشر. مثلا إذا أردنا حجز 5 عناصر من نوع int نكتب :

```
malloc(5 * sizeof(int));
```

أو

```
calloc(5, sizeof(int));
```

مع اختلاف بسيط وهو أن الدالة calloc تقوم بتصفير البيانات التي تحجزها، وفي العادة calloc تقوم باستدعاء malloc داخلها.

III – 1.21, ما الفائدة من استخدام calloc إذا كانت malloc تفي بالغرض !؟

عمليا ليس هناك فرق بينهما, لكن لجعل الكود أكثر وضوحا يفضل استخدام malloc عند حجز ذاكرة لكائن واحد و calloc عند حجز مصفوفة.

لتقريب الفكرة, يمكنك اعتبار أن calloc = malloc + memset.

III – 1.22, ما هي فائدة الدالة realloc ؟

realloc تقوم بإعادة حجز ذاكرة تم حجزها مسبقا عن طريق calloc/malloc بحجم مختلف وتقوم بنسخ البيانات من الذاكرة القديمة إلى الأحدث إذا كان هذا ممكنا ثم تقوم بتحرير الذاكرة القديمة وإرجاع عنوان للذاكرة الأحدث. (يمكن استخدامها عندما نريد حذف أو إضافة عناصر إلى المصفوفة)

III - 1.23, كيف أنشئ مصفوفة ديناميكية ثنائية البعد ؟

توجد ثلاث طرق للإعلان عن المصفوفات الديناميكية :

✎ الطريقة الأولى : نحجز مؤشر P هكذا

```
P = malloc(NBLIG* NBCOL* sizeof(élément));
```

حيث NBLIG هو عدد الأسطر و NBCOL هو عدد الأعمدة و élément هو نوع عناصر المصفوفة. بهذه الطريقة يمكننا الوصول إلى العنصر الذي يوجد عند تقاطع السطر i و العمود j بالكتابة التالية :

```
*( (P+j)+(i*NBCOL) );
```

✎ الطريقة الثانية : إنشاء مصفوفة مؤشرات أحادية البعد, عدد عناصرها NBLIG, بهذه الطريقة يمكنك التعامل مع المصفوفات الغير ثابتة البعد (كل سطر له بعد مختلف – معالجة النصوص مثلا) كما تسمح لك هذه الطريقة بالتعامل السهل و السريع مع الأسطر (تبادل سطرين مع بعضها البعض دون نسخ محتوياتهما), هذه الطريقة أحسن و أسرع من السابقة, لأنه يمكننا الوصول إلى عنوان كل سطر دون أي حسابات.

✎ الطريقة الثالثة : استخدام مؤشر لمؤشر, نفس الطريقة السابقة و لكن في هذه الطريقة سنستبدل جدول المؤشرات (حجمه مُعرف مُسبقا) بالحجز الديناميكي.

III - 1.24, ما الفرق بين الدالة malloc و المعامل new ؟

يكمن الفرق الأساسي في النقاط التالية:

- ☜ malloc لا تستدعي المشيد كما يفعل new.
- ☜ malloc لا تحتوي على آلية لمعالجة الأخطاء داخليا.
- ☜ تشترط malloc تحديد حجم البايتات المراد حجزه حسب نوع المتغير.

III – 1.25 , p و q يُشيران إلى متغيرين يحملان نفس القيمة و مع ذلك فإن q == p تُعيد false دائما !, ما السبب ؟

هذا طبيعي جدا !, الشرط q == p يُقارن بين المؤشرين و ليس ما يُشيران إليه, في هذه الحالة تكون العناوين مختلفة و لكن المحتوى متساوي, لمقارنة محتوى الذاكرة التي يشير إليها كل مؤشر, قم باستدعاء الدالة memcmp أو استخدم p* و q*.

```
int * p, * q, n = 10;
/* On suppose ici que malloc ne peut pas échouer */
p = malloc(sizeof(int));
q = malloc(sizeof(int));
memcpy(p, amp;&n, sizeof(int)); /* Ou simplement *p = n; */
memcpy(q, amp;&n, sizeof(int)); /* Ou simplement *q = n; */
/* (*p == *q) mais (p != q) */
free(p);
p = q;
/* Maintenant (p == q) et donc naturellement (*p == *q) */
free(p); /* Ou free(q) */
```

III – 1.26, كيف نعرف نوع المتغير الذي يُشير إليه مؤشر من النوع void ؟

لا يمكن معرفة نوع الكائن الذي يشير إليه مؤشر void, إذا كان المؤشر مستخدم لإنشاء generic function فإنه من الضروري أن نضيف إلى وسائط الدالة وسيط إضافي يُحدد حجم أو نوع المتغير المستعمل:

```
typedef enum {
    TYPES_char,
    TYPES_short,
    TYPES_int,
    TYPES_long,
    TYPES_float,
    TYPES_double
} TypePtr;
void MyFunction(void * Ptr, TypePtr Type);
```

2 – III. المصفوفات الحرفية

2.1 – III, ما هو NUL ؟

المحرف NUL (أو 0\) هو الرمز الذي يُستخدم للدلالة على نهاية المصفوفة, جميع بتات هذا المحرف تساوي 0.

2.2 – III, لماذا يحتاج المترجم إلى وضع الرمز NUL في نهاية كل مصفوفة حرفية ؟

تحتوي لغة C على أنواع بسيطة مثل short و long و float و char. المصفوفات التي يتم إنشائها من الأنواع السابقة إن كانت داخل الـ stack فيمكنك معرفة حجمها – بدون تحويلها لمؤشر – و ذلك عن طريق إحضار حجم المصفوفة باستخدام sizeof و قسمته على حجم نوع المصفوفة.

بالنسبة للمصفوفة الحرفية فهي ليست نص و إنما مجموعة من الحروف يمكن الوصول لأحد عناصرها باستخدام موقعه داخل المصفوفة تماما كما تصل لموقع أحد عناصر المصفوفات ذات الأنواع الأخرى.

و حيث أننا نحتاج لاستخدام النصوص داخل لغة لا تدعم النصوص فلابد من وضع مفاهيم و تعاريف حتى نستطيع تمييز النص, هذه القواعد تتلخص في نقطتين:

- النص هو مصفوفة من النوع char أو wchar_t
- لابد و أن ينتهي النص بعلامة مميزة نستطيع من خلالها معرفة الموقع الذي يمثل نهاية النص و يتم هذا الأمر باستخدام الـ Null Character او Null Terminator.

إذا كنت ستستخدم المصفوفة الحرفية كنص فلابد و أن تنتهي بـ 0 لكي تستطيع دوال التعامل مع النصوص معرفة نهاية النص أما إن كنت ستتعامل مع المصفوفة الحرفية كمصفوفة حروف فلابد أن تعلم عدد الحروف المكونة لها.

كما سبق و أوضحت أنه يمكنك معرفة حجم المصفوفة الموجودة داخل الـ Stack فقط إذا كنت ستتعامل معها داخل الـ scope الذي تم تعريفها فيه و لم تقم لتحويلها لمؤشر، أيضا المصفوفة التي تم تعريفها داخل الـ Heap يمكن معرفة حجمها داخل ويندوز كالتالي (لاحظ انه لابد من معرفة حجم نوع البيانات الذي تمثله المصفوفة)

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int getSize(int* arr) {
    return HeapSize(GetProcessHeap(), 0, arr) / sizeof (int);
}

int main() {
    int* w = malloc(100 * sizeof (int));
    int size = getSize(w);
    printf("%d", size);
    return 0;
}
```

الدالة getSize تعود لك بمساحة الدالة التي تم انشائها داخل الـ heap.

لا تقم بتجربة المثال السابق داخل Visual Studio لأنه يستخدم Debug Heap خاص به داخل وضع الـ Debug و أيضا الـ Release، إذاكنت تريد تجربة المثال فقم بتجربته داخل مساحة الذاكرة الخاصة به.

2.3 – III, ما معنى الكتابة التالية "char * p = "Bonjour ؟

هذه الكتابة تقوم بإنشاء مؤشر يشير إلى السلسلة الحرفية Bonjour, مكان تخزين السلسلة يختلف باختلاف البيئة التي تعمل عليها, لغة C لا تضمن لك أن مساحة الذاكرة ستكون غير قابلة للقراءة

```
char * p = "Bon*our";
p[3] = 'j';
/*Incorrect(mais accepte par le compilateur !).
 * Il n'est pas garanti que p[3] soit modifiable. */
strcpy(p, "Salut"); /* Incorrect (mais accepte par le compilateur !).
                     * Pour la meme raison. */
p = "Salut"; /* Correct.
              * p pointe maintenant sur une chaine de caracteres valant "Salut". */
```

لذا يجب اعتبار أن مساحة الذاكرة للقراءة فقط عن طريق تعريف المؤشر على أنه مؤشر إلى ثابت:

```
char const * p = "Bon*our";
p[3] = 'j'; /* Incorrect. Immediatement rejete par le compilateur.
             * On ne peut pas modifier une 'constante' ! */
strcpy(p, "Salut"); /* Incorrect. Immediatement rejete par le compilateur.
                     * Pour la meme raison. */
p = "Salut"; /* Correct.
              * p pointe maintenant sur une chaine de caracteres valant "Salut". */
```

III - 2.4, كيف أجعل الحروف صغيرة أو كبيرة ؟

يمكنك استخدام الدالة tolower لتصغير الأحرف أو toupper لتكبير الأحرف, الدالتين السابقتين يقومان بتحويل حرف واحد, لذا سيلزمك استخدام حلقة. في الويندوز يمكن استخدام الدالتين CharLower و CharLowerBuff لتصغير الحروف أو CharUpper و CharUpperBuff لتكبيرها.

III - 2.5, كيف نقوم بتحويل عدد إلى سلسلة محارف ؟

يمكنك استخدام الدالة sprintf الموجودة في الملف الرأسي stdio.h و التابعة للمعيار ANSI-C و بالتالي يمكن استخدامها تحت أي منصة, طريقة استخدامها تكون كالآتي:

```
#include <stdio.h>
char buf[32];
int n = 10;
sprintf(buf, "%d", n);
```

المعيار C99 أدخل الدالة snprintf و هي مشابهة لـ sptintf جدا, الفرق الوحيد هو أن snprintf تستقبل الحجم الأقصى للسلسلة الحرفية كوسيط إضافي.

III - 2.6, كيف نقوم بتحويل سلسلة محارف إلى عدد ؟

إذا كانت السلسلة الحرفية تحوي عدد, يمكننا إسناده لمتغير صحيح باستخدام الدالة strtol :

```
char buf[32] = "15";
long n;
n = strtol(buf, NULL, 10);
```

الوسيط الثالث للدالة عبارة عن قاعدة النظام المستعمل (في حالتنا هذه فإن النظام المستعمل هو النظام العشري), بعد تنفيذ الدالة سيحتوي الوسيط الثاني على عنوان السلسلة الحرفية بعد توقفها, إذا تم التحويل بنجاح فإن الوسيط الثاني سيشير إلى \0 :

```
char buf[32] = "15";
long n;
char * end
n = strtol(buf, &end, 10);
if (*end == '\0') {
    /* La chaine a enteirement pu etre convertie */
} else {
    /* La chaine contenait au moins un caractere ne pouvant pas etre converti en nombre */
}
```

2.7 - III, كيف ندمج السلاسل الحرفية ؟

توجد عدة طرق, إذا أردنا دمج سلسلتين و تخزين الناتج في سلسلة ثالثة فمكننا استخدام الدالة sprintf هكذا:

```
#include <stdio.h>
char Chaine1[32], Chaine2[32], ChaineFinale[64];
sprintf(ChaineFinale, "%s%s", Chaine1, Chaine2);
```

يمكننا أيضا استدعاء الدالة strcat التي تقوم بدمج السلسلتين في السلسلة الأولى:

```
#include <string.h>
char Chaine1[64], Chaine2[32];
strcat(Chaine1, Chaine2); /* Chaine1 contient maintenant Chaine1 + Chaine2 */
```

طول السلسلة الثانية يجب أن يكون كافياً لإضافة السلسلة الأولى.

2.8 - III, كيف نقارن السلاسل الحرفية ؟

لا تستخدم المعامل == في المقارنة. استخدم الدالة strcmp لمثل هذا الأمر, إذا كنت تريد مقارنة جزء معين من السلسلة الحرفية فيمكنك استخدام الدالة strncmp, توجد ثلاث حالات:

- القيمة 0 تعني أن السلسلتين متساويتين.
- القيمة السالبة تعني أن السلسلة الحرفية الأولى تحتوي على حروف ترميزها بال ASCII أصغر من ترميز حروف السلسلة الثانية.
- القيمة الموجبة تعني أن السلسلة الحرفية الأولى تحتوي على حروف ترميزها بال ASCII أكبر من ترميز حروف السلسلة الثانية.

```
const char * chaine1 = "blabla";
const char * chaine2 = "blabla";
const char * chaine3 = "blablu";
if (strcmp(chaine1, chaine2) == 0) {
    /* les chaines sont identiques */
}
if (strcmp(chaine2, chaine3) != 0) {
    /* les chaines sont différentes */
}
if (strncmp(chaine2, chaine3, 3) == 0) {
    /* les chaines sont identiques en se limitant a comparer les 3 premiers caracteres */
}
```

III – 2.9, لماذا لا أستطيع مقارنة السلاسل الحرفية المقروءة بالدالة fgets ؟

عند قراءة سلسلة محارف من ملف نصي باستخدام الدالة fgets يبقى رمز الرجوع إلى بداية السطر \n مُخزن في السلسلة و بالتالي يجب حذف الرمز قبل المقارنة

```
char s[256];
FILE * fp;
...
if (fgets(s, sizeof s, fp) != NULL) {
    char * p = strchr(s, '\n');
    if (p != NULL)
        *p = 0;
}
```

III – 2.10, كيف ننشئ مصفوفة من السلاسل الحرفية ؟

يكفي إنشاء مصفوفة ثنائية البعد, إذا أردنا إنشاء مصفوفة تحتوي على 10 كلمات, كل كلمة تحتوي على 256 حرف كحد أقصى فيمكننا كتابة:

```
#include <string.h>
...
char Tableau[10][256];
...
strcpy(Tableau[0], "azerty");
```

في المثال السابق, قمنا بإسناد الكلمة azerty إلى المصفوفة رقم 0.

III – 2.11, لماذا يجب كتابة الرمز \ هكذا \\ ؟

لأن \ يُمثل بداية الرموز الخاصة بلغة C, على سبيل المثال, الرمز \n يُمثل الرجوع إلى بداية السطر.

لكي يتم التعامل مع السلسلة السابقة باعتبار أنها الرمز \ متبوعا بالحرف n يكفي إضافة الرمز \ إلى العبارة لتصبح \\n.

نسيان الخطوة السابقة هو خطأ شائع, عادة ما يقع فيه المبتدئون في لغة C, انظر المثال:

```
remove("c:\arabteam\faqc\test.txt"); /* Ne fonctionne pas */
remove("c:\\arabteam\\faqc\\test.txt"); /* Fonctionne */
```

III – 2.12, كيف نقوم بنسخ مصفوفة ؟

توجد طريقتان لفعل هذا الأمر, الطريقة الأولى تكمن في نسخ المصفوفة عنصرا بعنصر, هذه الطريقة معقدة جدا و بطيئة أيضا.

الطريقة الثانية تتم باستخدام الدالة memcpy, تستقبل هذه الدالة 3 وسائط, المصفوفة التي نريد نسخها و المصفوفة الجديدة و حجم البيانات المراد نسخها بالبايت, انظر المثال :

```
#include <string.h>
int Tab1[10], Tab2[10];
memcpy(Tab2, Tab1, sizeof Tab2);
```

3 – III. التراكيب

3.1 – III, كيف أنشئ اسما مستعارا لـ Structure ؟

يمكنك فعل ذلك بأكثر من طريقة:

```
typedef struct {
    int x;
    int y;
} POINT2D;
```

أو

```
struct point2d_s {
    int x;
    int y;
};
typedef struct point2d_s POINT2D;
```

أو

```
typedef struct point2d_s {
    int x;
    int y;
} POINT2D;
```

3.2 – III, ما الفرق بين structure و union ؟

البنية عبارة عن مجموعة من البيانات مختلفة النوع مُسجلة تحت اسم واحد:

```c
#include <stdio.h>

struct point_s {
    int x;
    int y;
};

int main() {
    struct point_s A;
    A.x = 1;
    A.y = 2;
    printf("A = (%d, %d)\n", A.x, A.y);
    return 0;
}
```

أما ال union فهو عبارة عن قائمة من العناصر التي تستخدم نفس العنوان في الذاكرة

```c
#include <stdio.h>

union duo_u {
    int n;
    double x;
};

int main() {
    union duo_u d;
    d.n = 1;
    printf("d.n = %d\n", d.n);
    d.n = 2;
    printf("d.n = %d\n", d.n);
    d.x = 3.0;
    printf("d.x = %f\n", d.x);
    d.x = 4.0;
    printf("d.x = %f\n", d.x);
    return 0;
}
```

أهم الفوارق بين structure و union :

- في أغلب الأحيان يكون حجم البنية مُساوي لمجموع أحجام العناصر المُكونة له أما حجم ال union فيُساوي حجم أكبر العناصر.
- قيمة أي متغير في ال union تساوي قيمة آخر متغير تم استعماله لأن جميع العناصر تملك نفس العنوان بينما يملك كل عنصر في البنية عنوان مُختلف.
- يمكن استخدام أكثر من عنصر في آن واحد في structure أما في ال union فلا يمكن العمل على عنصرين في نفس الوقت.
- ال union تأخذ مساحة أقل من الذاكرة مُقارنة مع structure.

III – 3.3, لماذا حجم البنية لا يُساوي بالضرورة مجموع أحجام العناصر ؟

لأن عناصر البنية ليست بالضرورة مُخزنة في أماكن متتالية في الذاكرة, يتعلق الأمر بالطريقة التي يعتمد عليها المعالج في الـ Alignment Constraints.

III – 3.4, كيف نقوم بنسخ structure ؟

يتم ذلك عن طريق المعامل = أو الدالة memcpy الموجودة في الملف الرأسي string.h :

```
#include <stdio.h>
#include <string.h>

struct personne_s {
    char nom[21];
    int age;
};

int main() {
    struct personne_s A, B, C;
    strcpy(A.nom, "A");
    A.age = 26;
    B = A;
    memcpy(&C, &A, sizeof (C));
    printf("A.nom = %s\n", A.nom);
    printf("A.age = %d\n", A.age);
    printf("B.nom = %s\n", B.nom);
    printf("B.age = %d\n", B.age);
    printf("C.nom = %s\n", C.nom);
    printf("C.age = %d\n", C.age);
    return 0;
}
```

III – 3.5, كيف نقارن بين بُنْيَتَيْن ؟

المعامل == غير مُعرف في التراكيب, لا يمكننا أيضا استخدام الدالة memcmp لأن عناصر البنية ليست بالضرورة متجاورة كما هو الحال مع المصفوفات مثلا, لذا يلزم أن نقارن بين البنيتان عنصرا بعنصر.

III – 3.6, ماذا تعني الكتابة التالية unsigned int i : <n> ؟

هذه الكتابة عبارة عن تصريح خاص بالتراكيب, هذا التصريح يسمح بإنشاء عنصر جديد اسمه i و حجمه n بت (bits), يُسمى هذا النوع من العناصر بـ Bit field :

```
struct test_s {
    unsigned int i : 1;
    unsigned int j : 2;
};

int main() {
    struct test_s s;
    s.i = 0; /* A la fin de l'instruction, on aura s.i = 0 en binaire soit 0 en decimal. */
    s.i++; /* A la fin de l'instruction, on aura s.i = 1 en binaire soit 1 en decimal. */
    s.i++; /* A la fin de l'instruction, on aura s.i = 0 en binaire soit 0 en decimal. */
    s.j = 0; /* A la fin de l'instruction, on aura s.j = 00 en binaire soit 0 en decimal. */
    s.j++; /* A la fin de l'instruction, on aura s.j = 01 en binaire soit 1 en decimal. */
    s.j++; /* A la fin de l'instruction, on aura s.j = 10 en binaire soit 2 en decimal. */
    s.j++; /* A la fin de l'instruction, on aura s.j = 11 en binaire soit 3 en decimal. */
    s.j++; /* A la fin de l'instruction, on aura s.j = 00 en binaire soit 0 en decimal. */
    return 0;
}
```

III – 3.7, كيف نستخدم مؤشر يُشير إلى بنية ؟

يتم الإعلان عنه هكذا:

```
struct base {
    int a;
    double b;
};
struct base Elem;
struct base * p = &Elem;
```

لكي نستطيع الوصول إلى أحد أعضاء البنية نقوم بعمل dereference للمؤشر, هكذا (*p).a = 5, يُمكن تبسيط الكتابة السابقة إلى p->a = 5.

الكتابة الأولى ليست مُكافئة للكتابة الثانية, لأن الأولى تقوم بعمل dereference للمؤشر قبل الدخول إلى محتواه أما الكتابة الثانية فتقوم بالذهاب مُباشرة إلى ذاكرة المتغير.

III – 3.8, ما الفرق بين الكتابتين sizeof(struct data) و sizeof(struct data *) ؟

الكتابة الأولى تُعيد حجم البنية أما الكتابة الثانية فتعيد حجم مؤشر البنية.

III - 3.9, كيف يتم الإعلان عن بنية تشير إلى نفسها ؟

توجد عدة طرق لمثل هذا الأمر, المشكلة أنه عند الإعلان عن البنية باستخدام typedef سيكون النوع غير معرف بعد !.

هذه أحد أبسط الحلول, لحل المشكلة نقوم بإضافة tag إلى البنية :

```
typedef struct node {
    char * item;
    struct node * next;
} node_t;
```

هناك حل آخر يكمن في الإعلان عن نوع البنية قبل تعريفها عن طريق مؤشر:

```
typedef struct node * node_p;

typedef struct node {
    char * item;
    node_p next;
} node_t;
```

الإعلان السابق يُستخدم عادة من أجل الحصول على قوائم مترابطة أو بُنى شجرية.

III - 3.10, كيف نُحسن تهيئة المتغيرات ؟

المتغيرات العامة أو المتغيرات الساكنة (static) تكون مهيأة تلقائيا عند الإعلان عنها, إذا لم يتم إسناد أي قيمة لها فالقيمة الافتراضية لها هي صفر (حسب نوع المتغير فقد تكون القيمة الافتراضية : 0, 0.00, NULL)

أما بقية المتغيرات فيلزم أن نقوم بتهيئتها في الدالة الرئيسية main. المتغيرات المحجوزة ديناميكيا تدخل هي أيضا في الصنف الثاني, يمكن أن نستخدم الدالة calloc لتهيئة تلك المتغيرات, هذه الدالة تقوم بتصفير الذاكرة (تماما كما تفعل memset), هذه التهيئة تكون صالحة للأنواع الصحيحة (char, short, int, long) ولكنها لا تصلح مع المؤشرات و الأنواع الحقيقية (float).

الطريقة الأفضل لتهيئة المتغيرات حسب نوعها تكون كالتالي:

```
[static] const struct s x0 = {0};
{
    struct s *px = malloc(sizeof *px);
    *px = x0;
}
```

و بالنسبة للمتغيرات الديناميكية:

```
{
    type a[10] = {0};
    struct s x = {0};
}
```

IV - الفصل الرابع : الدوال, الملفات و مجالات الرؤية

1 – IV. الدوال

1.1 – IV, ما السبيل إلى تغيير قيم وسائط الدالة ؟

في C, يتم تمرير نُسخ من الوسائط, تُخزن داخل المكدس (Stack), أو على الأقل هذا ما يحدث في أغلب الـ implementations !, عند الوصول إلى return و انتهاء عمل الدالة يتم حذف جميع القيم المُخزنة في المكدس و بالتالي تضيع التغييرات التي حدثت على قيم الوسائط.

للتغلب على هذه المشكلة يكفي إرسال الوسائط عن طريق العنوان بدلا من القيمة, انظر المثال:

```
void echange(int * a, int * b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

1.2 – IV, ما هي فائدة الوسائط الافتراضية (Default parameters) ؟

تظهر فائدة الوسيط الافتراضي إذا لم تُمرر له قيمة, حيث يحتفظ بقيمته الافتراضية.

يتم الإعلان عن الوسائط الافتراضية عن طريق إسناد قيم افتراضية لها في النموذج المصغر للدالة, مثلاً:

```
int myFunc(int x= 10) ;
```

فهذه الدالة يمكن استدعاؤها هكذا ; c= myFunc(25) لتكون قيمة x مساوية لـ 25. أو هكذا c= myFunc() لتكون قيمة x مساوية لـ 10 لأنه تم تجاهل تمرير قيمة لهذا الوسيط, و بالتالي احتفظ الوسيط بقيمته الافتراضية.

لاحظ أن القيم المفترضة توضع في النموذج المصغر فقط, أما ترويسة الدالة فتبقى : int myFunc(int x)

أخيرا, هناك 3 ملاحظات مهمة :

- ✌ يتم تعيين القيم المفترضة بالموقع وليس بالاسم, لذلك ليس من الضروري أن يطابق اسم الوسيط في النموذج المصغر اسمه في التعريف.
- ✌ عند تعيين قيمة مفترضة لوسيط ما, يجب تعيين قيم مفترضة لكل الوسائط بعده.
- ✌ في حال كان هناك أكثر من وسيط مفترض في دالة واحدة, فلا يمكن عند الاستدعاء تمرير قيمة لأحد هذه الوسائط المفترضة دون تمرير قيمة للوسيط المفترض الذي يسبقه.

IV - 1.3, ما هي فائدة الدوال الخطية (Inline Function) و أين يتم تخزينها ؟

الدوال الخطية تُنسخ شفرتها كما هي وتوضع في المكان الذي تم استدعاؤها منه, أي أنها تصبح جزءاً من الدالة المُستدعية.

تظهر فائدتها عندما تكون الدالة صغيرة جداً, وذلك لتوفير الوقت المُستهلك لاستدعاء الدالة من مكانها في الذاكرة لو كانت عادية.

عند تعريف دالة, يقوم المترجم بإنشاء نسخة منها في الذاكرة, ويقفز إليها كلما تم استدعاء هذه الدالة.
بعض الدوال تكون صغيرة, وسيكون القفز إليها في كل استدعاء مضيعة للوقت, لذلك يمكن جعلها خطية inline بحيث يقوم المترجم عند التنفيذ بنسخ تعليماتها كما هي, ووضعها في كل مكان يتم استدعاؤها فيه, وذلك بوضع كلمة inline قبل نوع إرجاع الدالة في النموذج المصغر كالتالي على سبيل المثال:

```
inline int myFunc(int) ;
```

قد يتجاهل المترجم كون دالة ما خطية ويتعامل معها على أنها دالة عادية في حال وجد حجمها كبيراً جداً.

IV - 1.4, ما هو النموذج المُصغر (prototype) ؟

الـ prototype هو النموذج المُصغر للدالة أو توقيع الدالة كما يُسميه البعض, هذا التوقيع يشمل تعريف الدالة (نوع "الكائن" المُعاد + اسم الدالة + الوسائط), فمثلا لو أردنا كتابة دالة تقوم بإجراء القسمة بين عددين و تُعيد بنية تحتوي على قيمة الناتج و باقي القسمة أيضا, سيكون توقيع الدالة هكذا :

```
struct Mod(int x, int y) ;
```

مثال :

```c
#include<stdio.h>

struct Div {
	int quot;
	int rem;
};

Div Mod(int x, int y) {
	Div Result;
	Result.quot = x / y;
	Result.rem = x % y;
	return Result;
}

int main() {
	Div test = Mod(38, 5);
	printf("38 div 5 => %d, remainder %d.\n", test.quot, test.rem);
	return 0;
}
```

1.5 – IV, أيهما أفضل, الإعلان عن الدالة أسفل أو أعلى الـ main ؟

يُفضل دائما وضع prototype الدوال فوق الـ main ثم كتابة تفاصيل الدوال أسفل الدالة الرئيسية.

يُمكننا تلخيص فوائد وضع الـ prototype في أربع نقاط :

- تنظيم الشفرة وتسهيل تصحيحها.
- لو كانت لديك دالتان تستدعيان بعضهما البعض, فحينها يجب أن يكون هناك نموذج مصغر لإحداهما على الأقل, لكي تستطيع القيام بالعملية, حيث أن الدالة حين تستدعي دالة أخرى, فإنها تبحث عنها فوقها في الشفرة, فإما أن تجد تعريف الدالة, أو أن تجد تصريحها (نموذجها المصغر Prototype), أو لا تجد ما تريده, وحينها لن تعمل.
- لا يمكن تعريف القيم الافتراضية لبعض الوسائط إلا في النموذج المصغر.
- لا تستطيع تحديد إن كانت الدالة خطية inline أم لا إلا في النموذج المصغر.

IV - 1.6, ما هي التواقيع الصحيحة للدالة الرئيسية ؟

الـ main تُعيد int دائما, الكتابات السليمة تكون هكذا :

```
int main(void);
int main(int argc, char * argv[]);
```

كل توقيع آخر ليس بالضرورة أن يكون قياسيا, و لا يُنصح باستخدامه حتى لو قبله المترجم الذي تعمل عليه !

أيضا, القيمة المُعادة من طرف الدالة الرئيسية يجب أن تكون موجبة أو معدومة, القيم القياسية المُستخدمة في الإعادة هي : 0, EXIT_SUCCESS, EXIT_FAILURE.

يُمكن أيضا أن تجد في Unix التوقيع التالي:

```
int main (int argc, char* argv[], char** arge);
```

عند الحاجة إلى استخدام Shell Environment Variables, هذا التوقيع ليس محمولا (portable) و لا قياسيا أيضا (standard).

IV - 1.7, كيف تستطيع printf استقبال عدد غير محدود من مُختلف أنواع المتغيرات ؟

printf هي دالة غير محددة الوسائط, توقيعها كالتالي:

```
int printf(const char * format, ...); /* C 90 */
int printf(const char * restrict format, ...); /* C 99
```

لاحظ أن نوع و عدد الوسائط غير مُحدد في التوقيع.

IV - 1.8, عند استخدام الرمز %d مع الأعداد الحقيقية تظهر نتائج غريبة !, مالسبب ؟

لا تحاول استخدام الرمز %d مع النوع float أو ما شابه لأن هذا سينتج عنه الـ Undefined Behavior أو السلوك الغير معروف و هو تصرف غير مسئولة عنه الدالة التي تناديها أو الشيء (Construct) الذي تقوم باستعماله في اللغة.مثلاً, من المعروف أن القسمة على صفر خطأ منطقي. في لغات أخرى, يكون هناك ما يسمى بالـ Exception في اللغة اسمه Divide by Zero أو ما شابه ليخبرك أنك قسمت على صفر, في C هناك شروط يجب أن تتوفر قبل عمل أي دالة بشكل صحيح تسمى أحياناً بـ Preconditions , ببساطة أكثر, صاحب الدالة التي سوف تستعملها يقول لك يجب أن تنادي الدالة بالطريقة التالية, و إن لم تناديها بالطريقة المعطاة فالدالة ليست مسئولة عن التحقق من صحة ما يمرر لها, بالتالي صاحب الدالة لا يقوم بالتحقق من شيء و إنما يفترض أنك أنت المسؤول عن المناداة بشكل صحيح. و أنت قمت بمناداة الدالة printf بشكل خاطئ. هذا هو السلوك الغير المعروف بكل بساطة, و لا يمكنك بأي حال من الأحوال أن تقول ظهرت لي نتيجة ما عند تجربة الكود, إذاً فهو صحيح.

لأن الدالة نفسها لا تتحقق أصلاً من شيء, بالتالي الناتج خاطئ حتى لو كان صحيحاً على مترجم معين بنسخة معينة على نظام تشغيل معين باستخدام مكتبات معينة و جميع الظروف التي تحيط لحظة تشغيل الكود !

IV – 1.9, ما هي أهم الرموز المُستخدمة مع الدالة printf ؟

الجدول التالي يبين أهم الرموز المُستخدمة مع الدالة printf :

الرمز	النوع	يتم الإظهار باعتبار أنه
%c	char	حرف من حروف ASCII
%d,%i	int	عدد صحيح
%o	int	عدد صحيح مكتوب في النظام الثماني (Octal)
%x	int	عدد صحيح مكتوب في النظام السداسي عشر (Hex)
%u	unsigned int	عدد صحيح (موجب)
%ld	long	عدد صحيح مضاعف
%lld	long long	عدد صحيح مضاعف ضخم
%f	float	عدد حقيقي
%e	float	عدد حقيقي مكتوب بصيغة الأس مثل $3,68^{e+7}$
%g	float	عدد حقيقي مكتوب بصيغة الأس أو الصيغة العادية (أيهما أنسب)
%lf	double	عدد حقيقي مضاعف
%le	double	عدد حقيقي مضاعف مكتوب بصيغة الأس
%lg	double	عدد حقيقي مضاعف مكتوب بصيغة الأس أو الصيغة العادية (أيهما أنسب)
%Lf	Long double	عدد حقيقي مضاعف ضخم
%Le	Long double	عدد حقيقي مضاعف ضخم مكتوب بصيغة الأس
%Lg	Long double	عدد حقيقي مضاعف ضخم مكتوب بصيغة الأس أو الصيغة العادية (أيهما أنسب)
%s	char*	سلسلة حرفية
%p	مؤشر	عنوان المتغير الذي يشير إليه المؤشر

IV – 1.10, ما هي أهم الرموز المُستخدمة مع الدالة scanf ؟

الجدول التالي يبين أهم الرموز المُستخدمة مع الدالة scanf :

الرمز	النوع	تتم القراءة باعتبار أنه
%c	char	حرف من حروف ASCII
%d,%i	int	عدد صحيح
%o	int	عدد صحيح مكتوب في النظام الثماني (Octal)
%x	int	عدد صحيح مكتوب في النظام السداسي عشر (Hex)
%u	unsigned int	عدد صحيح (موجب)
%ld	long	عدد صحيح مضاعف
%lld	long long	عدد صحيح مضاعف ضخم
%f	float	عدد حقيقي
%e	float	عدد حقيقي مكتوب بصيغة الأس مثل $3,68^{e+7}$
%g	float	عدد حقيقي مكتوب بصيغة الأس أو الصيغة العادية (أيهما أنسب)
%lf	double	عدد حقيقي مضاعف
%le	double	عدد حقيقي مضاعف مكتوب بصيغة الأس
%lg	double	عدد حقيقي مضاعف مكتوب بصيغة الأس أو الصيغة العادية (أيهما أنسب)
%Lf	Long double	عدد حقيقي مضاعف ضخم
%Le	Long double	عدد حقيقي مضاعف ضخم مكتوب بصيغة الأس
%Lg	Long double	عدد حقيقي مضاعف ضخم مكتوب بصيغة الأس أو الصيغة العادية (أيهما أنسب)
%s	char*	سلسلة حرفية

IV, 1.11 – ما هي أهم دوال المكتبة math.h ؟

الجدول التالي يبين دوال المكتبة math.h, الأكثر استخداما :

الدالة	الوصف
sin(x)	الجيب
cos(x)	جيب تمام
tan(x)	الظل
asin(x)	معكوس الجيب
acos(x)	معكوس الجيب تمام
atan(x)	معكوس الظل
ceil(x)	تقريب x لأصغر قيمة صحيحة أكبر من x
floor(x)	تقريب x لأكبر قيمة صحيحة أصغر من x
fabs(x)	القيمة المطلقة
log10(x)	اللوغاريتم ذو الأساس 10
log(x)	اللوغاريتم الطبيعي
exp(x)	رفع x للأساس e
sqrt(x)	الجذر التربيعي
pow(x,y)	x^y
ldexp(x,y)	$x*2^y$

IV – 1.12, ما هي أهم دوال المكتبة ctype.h ؟

الجدول التالي يبين دوال المكتبة ctype.h, الأكثر استخداما :

الدالة	الوصف
isalnum(x)	إذا كانت القيمة المُعطاة للدالة تُقابل حرفا أو رمزا أو رقما في جدول آسكي فستعيد الدالة 1 و إلا فستعيد 0
isalpha(x)	إذا كانت القيمة المُعطاة للدالة تُقابل حرفا أبجديا في جدول آسكي فستعيد الدالة 1 و إلا فستعيد الدالة 0
isdigit(x)	إذا كانت القيمة المُعطاة لهذه الدالة لا تُقابل رقما في جدول آسكي فستعيد الدالة 0 و إلا فستعيد عدد يختلف عن الصفر
isgraph(x)	إذا كانت القيمة المعطاة لهذه الدالة تقابل رمزا مرئيا في جدول آسكي فستعيد الدالة 0 و إلا فستعيد قيمة تختلف عن الصفر
islower(x)	إذا كانت القيمة المُعطاة لهذه الدالة تقابل حرفا صغيرا في جدوا آسكي فستعيد نتيجة تختلف عن الصفر و إلا فستعيد 0
isupper(x)	إذا كانت القيمة المعطاة لهذه الدالة تقابل حرفا كبيرا في جدوا آسكي فستعيد نتيجة

IV, 1.13 - كيف تُعيد الدالة أكثر من قيمة ؟

لغة C لا تسمح للدوال بإعادة أكثر من كائن, لكن توجد عدة حلول لهذه المشكلة, أحد أبرز هذه الحلول هو التمرير بالمرجع لأن التغييرات ستحدث على مستوى العنوان و ليس القيمة, يوجد أيضا حل آخر و هو تمرير بنية أو مؤشر بنية تحوي مجموعة العناصر التي نريد تغيير قيمها.

IV, 1.14 - كيف يتم الإعلان عن دالة تُعيد سلسلة محارف ؟

توجد 3 طرق لمثل هذا الأمر, الطريقة الأولى تتعلق بتمرير مؤشر يشير إلى المصفوفة الحرفية :

```
#include <stdio.h>
#include <string.h>

void get_string(char * Buffer, size_t BufferLen) {
    /* On ne peut pas utiliser strcpy car cette fonction
    ne permet pas d'indiquer le nombre max de caracteres qu'on veut copier */
    strncpy(Buffer, "MaChaine", BufferLen);
    /* Il est possible que Buffer n'ait pas pu contenir toute la chaine ... */
    Buffer[BufferLen - 1] = '\0';
}

int main(void) {
    char Buffer[100];
    get_string(Buffer, sizeof (Buffer));
    printf("%s\n", Buffer);
    return 0;
}
```

الطريقة الثانية تعتمد على إعادة مؤشر يشير إلى سلسلة محارف موجودة في global memory و يمكننا فعلها بإحدى طريقتين :

1. باستخدام Static string :

```
#include <stdio.h>

const char * get_string(void) {
    return "MaChaine";
}

int main(void) {
    printf("%s\n", get_string());
    return 0;
}
```

2. باستخدام Dynamic string :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char * alloc_string(void) {
    char * Buffer, s[] = "MaChaine";
    Buffer = malloc(strlen(s) + 1);
    if (Buffer != NULL)
        strcpy(Buffer, s);
    return Buffer;
}

int main(void) {
    char * Buffer = alloc_string();
    if (Buffer != NULL) {
        printf("%s\n", Buffer);
        free(Buffer);
    }
    return 0;
}
```

IV - 1.15, كيف نمرر مصفوفة إلى دالة ؟

التعامل مع المصفوفات في الدوال يتم على أساس العنوان (عنوان المصفوفة). عادة, نقوم بتمرير مؤشر (يشير إلى أول عناصر المصفوفة) و متغير صحيح عبارة عن حجم المصفوفة (إذا كانت هذه الخطوة ضرورية), انظر المثال:

```c
#include <stdio.h>

void initialiser_tab(int * ptr, size_t n_elements) {
    size_t i;
    for (i = 0; i < n_elements; i++)
        ptr[i] = (int) i;
}

int main() {
    int t[10];
    size_t i, n_elements = sizeof (t) / sizeof (t[0]);
    /* t peut egalement s'ecrire &(t[0]) */
    initialiser_tab(t, n_elements);
    for (i = 0; i < n_elements; i++)
        printf("%d\n", t[i]);
    return 0;
}
```

عنوان المصفوفة هو عنوان أول عناصرها, وبالتالي فالكتابات التالية متكافئة:

```c
void initialiser_tab(int * ptr, size_t n_elements)
void initialiser_tab(int ptr[], size_t n_elements)
/* Le 10 ne sert absolument a rien */
void initialiser_tab(int ptr[10], size_t n_elements)
```

هذا مثال مع مصفوفة ثنائية البعد:

```
#include <stdio.h>

void initialiser_tab_2(int * ptr, size_t N, size_t M) {
    /* ptr = &(t[0][0]) d'ou : */
    /* - ptr[i] <=> t[0][i] */
    /* - ptr[M * j] <=> t[j][0] */
    /* - ptr[M * j + i] <=> t[j][i] */
    size_t j, i;
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++) {
            ptr[M * j + i] = (int) j;
        }
    }
}

int main() {
    int t[10][4];
    size_t j, i;
    size_t N = sizeof (t) / sizeof (t[0]), M = sizeof (t[0]) / sizeof (t[0][0]);
    /* &(t[0][0]) peut egalement s'ecrire t[0] ou encore (int *)t */
    initialiser_tab_2(&(t[0][0]), N, M);
    for (j = 0; j < N; j++) {
        for (i = 0; i < M; i++)
            printf("%d\t", t[j][i]);
        printf("\n");
    }
    return 0;
}
```

1.16 – IV, لماذا لا يُنصح باستخدام المؤشرات المحلية كقيمة مُعادة من طرف الدوال ؟

لأن المتغيرات المحلية لا يتم "تحريرها" عند الخروج من الـ Scope الخاص بالدالة, بل يتم استخدام Frame مختلف عن طريق زيادة الـ Stack pointer مما يؤدي الى جعل الـ Frame السابق قابل للكتابة عليه عن طريق تعريف متغيرات محلية أخرى في الـ Parent Routine.

في هذه الحالة سيظل المؤشر يشير إلى ذاكرة صالحة للاستخدام لكننا لا نضمن أنه لن يتم الكتابة عليها Overwrite عن طريق حجز متغير محلي آخر.

.. لم أفهم شيئا مما قلت ..!

بعبارة أبسط, الدالة التي تحتوي على المتغير من نوع مؤشر قد انتهى عملها و تمت العودة منها ليتم تنفيذ باقي البرنامج, في هذه الحالة المكان الموجود داخل ذاكرة الـ stack و الذي كان مستخدما من قبل المتغير من نوع المؤشر سيظل محتفظ بالقيمة التي به طالما لم يُستخدم ذلك المكان من قبل دالة أخرى.

IV – 1.17, كيف يتم الإعلان عن مؤشر يشير إلى دالة ؟

في البداية, نقوم بتعريف الدالة التي سيشير إليها المؤشر:

```
typedef int F(void); /* F : type de la fonction f. */

/* Voici la fonction f en question : */
int f(void) {
    return 1;
}
```

ثم نعلن عن المؤشر P الذي يشير إلى f هكذا: F * p = &f, يقوم المترجم بتحويل اسم الدالة (f في هذه الحالة) إلى &f, إذا يمكننا استبدال الكتابة السابقة بالكتابة التالية F * p = f ولاستدعاء الدالة f يمكننا استخدام إحدى الكتابتين ; y = p() أو ; y = (*p)().

أما بدون استخدام typedef سيكون إعلان المؤشر p هكذا int (*p)(void) = f.

المؤشر p يشير إلى الدالة f التي لا تأخذ أي وسائط و تعيد int و بالتالي فإن p* يمثل عنوان الدالة, إذا يُمكننا كتابة تعريف الدالة هكذا int (*p)(void).

IV – 1.18, كيف يتم الإعلان عن مصفوفة دوال ؟

توجد حالتان لمثل هذا الأمر, إذا كانت الدوال تملك نفس الـ prototype, يكفي الإعلان هكذا:

```
extern char * f(int, int); /* une fonction */
char * (*fp[N])(int, int); /* Un tableau de N fonctions */
char * sz;
fp[4] = f; /* affectation de f dans le tableau */
sz = fp[4](42, 12); /* utilisation *
```

أما إذا كانت الدوال تختلف في الـ prototype فيجب الإعلان عن مصفوفة من الـ Generic Functions.

هذه الأخيرة لا توجد بشكل حقيقي في اللغة, لذا يجب استخدام دالة بدون وسائط و تعيد int :

```
int (*fp[N])(); /* Un tableau de N fonctions quelconques*/
```

الكتابة السابقة تصلح لجميع الدوال باستثناء تلك الغير محدودة الوسائط مثل printf.

IV – 1.19, كيف تتم إعادة مؤشر يشير إلى دالة ؟

إليك هذا المثال:

```
int (*func(void))(const char *) {
    return puts;
}
```

الدالة func لا تأخذ وسيطا, و تُعيد مؤشرا يشير إلى دالة من نوع int f(const char * s).

لفهم المثال السابق, يمكنك قراءته كالتالي:

func تُعيد مؤشرا, الكتابة func(void) تُمثل المؤشر, الكتابة *func(void) تُمثل الشيء الذي يُشير إليه المؤشر, هذا الشيء عبارة عن دالة تأخذ * const char كوسيط و تُعيد int.

إذا النموذج المُصغر لـ f هو : int (*func(void))(const char *).

بالطبع, يُمكننا تخفيف الكتابة السابقة باستخدام typedef :

```
#include <stdio.h>
typedef int F(const char *);
F * func(void);

int main(void) {
    func()("Bonjour."); /* <=> puts("Bonjour."); */
    return 0;
}

F * func(void) {
    return puts;
}
```

IV – 2. الملفات

IV – 2.1, كيف أتأكد من وجود ملف ؟

بشكل عام, لا توجد طريقة قياسية لمعرفة سبب عدم فتح الملف. في معيار POSIX, توجد الدالة access ولكن بعض الأنظمة لا يطبق هذه الواجهة.

في معيار C ISO, الحل الوحيد لاختبار وجود ملف هو محاولة فتحه:

```
FILE *fp = fopen("fichier.txt", "r");
if (fp == NULL) {
    fputs("Le fichier n'existe pas,\n"
          "ou vous n'avez pas les droits necessaires\n"
          "ou il est inaccessible en ce moment\n"
          , stderr);
} else {
    /* ... operation sur le fichier */
    fclose(fp);
}
```

مع أن هذه الطريقة ليست سليمة 100% لأنه إذا لم يتم فتح الملف فهذا لا يعني بالضرورة أن الملف غير موجود !, توجد عدة أسباب أخرى :

- قد لا تملك الصلاحيات الضرورية لفتح الملف
- قد لا يمكن الوصول إلى محتوى الملف في تلك اللحظة (ملف مُشفر مثلا)
- قد لا يملك النظام مساحة الذاكرة الكافية لفتح الملف

في نظام DOS/Windows و أنظمة Nix* يمكننا فحص قيمة errno, هل هي مُساوية لـ : ENOENT (no such file or directory).

IV – 2.2, كيف أعرف حجم ملف ؟

لسوء الحظ, الدالتان stat و fstat التابعتان للمعيار POSIX غير مُدرجتان في معيار ISO.

الحل القياسي الوحيد يكمن في استخدام الدالتين fseek و ftell :

```c
int fsize(const char * fname, long * ptr) {
    /* Cette fonction retourne 0 en cas de succes,
     * une valeur differente dans le cas contraire. */
    /* La taille du fichier, si elle a pu etre calculee,
     * est retournee dans *ptr */
    FILE * f;
    int ret = 0;
    f = fopen(fname, "rb");
    if (f != NULL) {
        fseek(f, 0, SEEK_END); /* aller a la fin du fichier */
        *ptr = ftell(f); /* lire l'offset de la position courante
                          * par rapport au debut du fichier */
        fclose(f);
    } else
        ret = 1;
    return ret;
}
```

الحل السابق يصلح فقط عندما يكون حجم الملف أقل من القيمة العظمى لـ int.

2.3 – IV, كيف يتم نسخ الملفات ؟

لا توجد في C دالة تنسخ الملفات, لكن يمكننا كتابة دالة تقرأ الملف ثم تقوم بنسخه في الملف المطلوب :

```c
int copier_fichier(char const * const source, char const * const destination) {
    FILE* fSrc;
    FILE* fDest;
    char buffer[512];
    int NbLus;
    if ((fSrc = fopen(source, "rb")) == NULL) {
        return 1;
    }
    if ((fDest = fopen(destination, "wb")) == NULL) {
        fclose(fSrc);
        return 2;
    }
    while ((NbLus = fread(buffer, 1, 512, fSrc)) != 0)
        fwrite(buffer, 1, NbLus, fDest);
    fclose(fDest);
    fclose(fSrc);
    return 0;
}
```

من السهل جدا تغيير عمل الدالة السابقة من أجل دمج محتوى الملف المراد نسخه في محتوى الملف المطلوب عن طريق فتح الملف على النمط ("ab").

في Windows, توجد الدالة CopyFile التي تغنينا عن كتابة دالة جديدة للنسخ في كل مرة:

```c
BOOL CopyFile(LPCTSTR lpExistingFileName, /* Nom du fichier source */
        LPCTSTR lpNewFileName, /* Nom du fichier destination */
        /* Si != 0, la copie sera annulée si le fichier existe déjà */
        BOOL bFailIfExists
        );
```

IV - 2.4, كيف أحذف أسطراً من ملف نصي ؟

لغة C لا تُوفر دالة لفعل هذا الأمر, و بالتالي يجب قراءة الملف ونسخ جميع الأسطر باستثناء تلك التي نريد إزالتها, المثال التالي يحذف الأسطر التي يبدأ بـ @, لتبسيط الكود, اعتبرتُ أن طول كل سطر لا يتجاوز 255 حرف:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char ligne[256];
    FILE * fIn;
    FILE * fOut;
    if ((fIn = fopen("texte.txt", "r")) == NULL)
        return EXIT_FAILURE;
    if ((fOut = fopen("texte.tmp", "w")) == NULL) {
        fclose(fIn);
        return EXIT_FAILURE;
    }
    while (fgets(ligne, sizeof ligne, fIn)) {
        if (ligne[0] != '@')
            fputs(ligne, fOut);
    }
    fclose(fIn);
    fclose(fOut);
    rename("texte.tmp", "texte.txt");
    return 0;
}
```

نفس الشيء بالنسبة لحذف سجل من ملف ثنائي (binary file).

IV - 2.5, كيف أحذف ملفاً ؟

في Windows, استخدم الدالة DeleteFile الموجودة في المكتبة windows.h و في UNIX (بشكل عام, جميع الأنظمة التي تتوافق مع المعيار POSIX), استخدم الدالة unlink الموجودة في المكتبة unistd.h.

المكتبة القياسية لـ C توفر أيضا الدالة remove التي تستقبل مسار الملف المراد حذفه و تعيد قيمة صحيحة تدل على نجاح أو فشل العملية:

```c
#include <stdio.h>
int remove(const char * pathname);
```

IV - 2.6, كيف أعرض محتوى مجلد ؟

في Windows, استخدم الدالتين FindFirstFile و FindNextFile للبحث عن الملفات باستخدام التعابير المنطقية (*.* تعني جميع الملفات و جميع الامتدادات). المقبض (HANDLE) الذي تُعيده الدالة FindFirstFile ينبغي إغلاقه حالما لم يعد ضروريا باستخدام الدالة FindClose.

هذه الدوال لا تبحث في المجلدات الفرعية, المثال التالي يعرض محتوى Current Directory :

```
#include <stdio.h>
#include <windows.h>

int main(void) {
    WIN32_FIND_DATA File;
    HANDLE hSearch;
    hSearch = FindFirstFile("*.*", &File);
    if (hSearch != INVALID_HANDLE_VALUE) {
        do {
            printf("%s\n", File.cFileName);
        } while (FindNextFile(hSearch, &File));
        FindClose(hSearch);
    }
    return 0;
}
```

إن كنت تعمل في بيئة متوافقة مع المعيار POSIX فاستخدم الدوال opendir, readdir and closedir :

```
#include <stdio.h>
#include <dirent.h>

int main(void) {
    DIR * rep = opendir(".");
    if (rep != NULL) {
        struct dirent * ent;
        while ((ent = readdir(rep)) != NULL) {
            printf("%s\n", ent->d_name);
        }
        closedir(rep);
    }
    return 0;
}
```

3 – IV. مدى المتغيرات أو مجالات الرؤية

3.1 – IV, ما الفرق بين المتغيرات المحلية (local) و المتغيرات العامة (global) ؟

يتم الإعلان عن المتغيرات المحلية داخل دالة معينة, و لا يمكن استخدامها إلا داخل الدالة أو الإطار التي أُعلنت بداخله أما المتغيرات العامة فيتم الإعلان عنها خارج جميع الدوال, ويمكن لأي دالة في البرنامج استعمال هذا النوع المتغيرات.

بالنسبة للمتغيرات العامة فتنتهي أعمارها عند انتهاء تنفيذ البرنامج بينما ينتهي عمر المتغيرات الخاصة عند الانتهاء من تنفيذ الدالة الموجودة بها.

IV, 3.2 – ماذا تعني الكلمة static ؟

تُستخدم هذه الكلمة مع المتغيرات الساكنة, و عند إضافتها إلى أي متغير فهذا يعني أنك قد أضفت له صفتين رئيسيتين هما:

- أصبح عمر المتغير مثل عمر المتغيرات العامة التي لا تُمسح من الذاكرة إلا إذا طلبت ذلك برمجيا, أو بانتهاء تنفيذ البرنامج .. لأن المتغيرات العامة تكون مُعرضة للقراءة في أي وقت ومن أي دالة.
- لا يُمكن رؤيتها إلا داخل الإطار التي أُعلنت بداخله.

المتغيرات الساكنة تأخذ مزايا النوعين السابقين .. فهي مثل المتغيرات الخاصة لأن هناك دالة وحيدة تستطيع رؤيتها وهي الدالة التي تم الإعلان عن المتغير بداخلها, و من ناحية أُخرى فالمتغيرات الساكنة مثل المتغيرات العامة لأنها لا تنتهي أو تُمسح من الذاكرة عندما ينتهي تنفيذ الدالة التابعة لها .. بل تظل مخزنة في الذاكرة (جاهزة للاستدعاء) حتى ينتهي تنفيذ البرنامج.

أيضا, إضافة الكلمة static إلى الدوال تعني أنه لا يُمكن رؤية تلك الدوال خارج الملف الذي أعلن عنهم فيه.

IV, 3.3 – كيف يتم استخدام متغير عام مُعرف في ملف مصدري آخر (another source file) ؟

تسمح الكلمة المُعرفة مُسبقا extern بالإعلان عن متغير عام سبق تعريفه في نفس الملف المصدري أو في ملف آخر. في المثال التالي, يتكون المشروع من ملفين هما globales.c و main.c, الملف الأول يحتوي على الإعلان عن متغيرين عامَّيْن و الملف الثاني يستخدم تلك المتغيرات :

globales.c :

```
int globale_1 = 1;
int globale_2 = 2;
```

main.c :

```
#include <stdio.h>
extern int globale_1;
int get_globale_2(void);

int main(void) {
    printf("globale_1 = %d\n", globale_1);
    printf("globale_2 = %d\n", get_globale_2());
    return 0;
}

int get_globale_2(void) {
    extern int globale_2;
    return globale_2;
}
```

V - الفصل الخامس : مُوجهات ما قبل المعالجة

V – 1, ما هي فائدة الـ Preprocessor ؟

قبل الوصول إلى مرحلة الترجمة (Compilation), تتم معالجة الملفات المصدرية (Source Files) باستخدام الـ Preprocessor, الذي يتولى مهمة تنفيذ كافة الأوامر التي تبدأ بـ # مثل #include, #define, ... , بعد ذلك تبدأ مرحلة الترجمة : ترجمة الملف المصدر (Source file) إلى ملف كائن (File object).

V – 2, ما هو الـ MACRO ؟

الماكرو عبارة عن تصريح DEFINE, قد يستقبل (أو لا يستقبل) بارامترات أو وسائط.
يُستخدم الماكرو عادة لتعيين قيمة ثابتة في كل الكود, يمكن اللجوء إليها فيما بعد و شكله العام يكون كالتالي :

```
#define identifier replacement-text
```

عندما يظهر السطر السابق ضمن ملف معين, فإنه يتم استبدال كافة الكلمات المطابقة لـ identifier بالسلسلة replacement-text بشكل تلقائي قبل أن تتم الترجمة, فمثلا يتم تحويل الكتابة التالية بعد المعالجة :

```
#define PI 3.14159

duoble CircleArea(float Radius) {
    return PI * Radius * Radius;
}
```

إلى :

```
duoble CircleArea(float Radius) {
    return 3.14159 * Radius * Radius;
}
```

3 – V, كيف أستخدم ماكرو يحتوي على وسائط (Parameterized macro) ؟

كما قلنا سابقا, يستطيع الماكرو استقبال وسائط بشكل مشابه للدوال, القوس المفتوح يجب أن يتبع اسم الماكرو مباشرة (بدون Space) ثم بقية الوسائط, يليها قوس الإغلاق, ثم يأتي بعد ذلك محُتوى الماكرو الذي يمُكن كتابته على أكثر من سطر, بشرط أن ينتهي كل سطر بـ backslash ماعدا الأخير, انظر المثال :

```
#include<stdio.h>
#define PRINT(x) print\
f("MESSAGE : \
%s\n", x)

int main() {
    PRINT("Hello, world !");
    return 0;
}
```

بعد انتهاء معالجة الماكرو ستتحول الكتابة السابقة إلى:

```
#include<stdio.h>

int main() {
    printf("MESSAGE : %s\n", "Hello, world !");
    return 0;
}
```

4 – V, ماذا تعني الكتابة #define MYMACRO ؟

الكتابة السابقة تُعرِّف ماكرو باسم MYMACRO, يتم تحويله بعد عملية المعالجة إلى .. لاشيء !

انظر المثال :

```
#define IN
#define OUT

int f(IN int n, OUT int * p1, OUT int * p2) {
    *p1 = n - 1;
    *p2 = n + 1;
    return 2 * n;
}
```

بعد المعالجة :

```
int Func(int n, int * p1, int * p2) {
    *p1 = n - 1;
    *p2 = n + 1;
    return 2 * n;
}
```

V - 5, و لماذا تُوفر C هذا النوع من الكتابات الذي لا فائدة من ورائه ؟

لا تستعجل !, الكتابة السابقة مهمة جدا عندما نريد محاكاة الحلقات التكرارية باستخدام الماكرو, لأنها تُعتبر آلية جيدة لمنع الحلقات اللانهائية !

الإستدعاء المتكرر للماكرو يُولد حلقة تكرارية .. و عند استدعاء ماكرو عادي في نهاية الحلقة قد يُدخلنا هذا في حلقة لا نهائية, لذا من الأفضل استخدام الماكرو الصامت عند نهاية الإستدعاء كإشارة على نهاية الحلقة.

V - 6, كيف أعرف ما إذا كان ماكرو مُعرف مُسبقاً أم لا ؟

يُمَكنك الشرط if defined(MYMACRO) أو ifdef MYMACRO من معرفة ما إذا كان الماكرو MYMACRO مُعرف أو لا, انظر المثال :

```
#include <stdio.h>
#if defined(UPPERCASE)
#define MESSAGE "HELLO, WORLD !"
#else
#define MESSAGE "Hello, world !"
#endif

int main() {
    printf("%s\n", MESSAGE);
    return 0;
}
```

في هذه الحالة, سيتم إبدال الكلمة MESSAGE بـ Hello, world (على التوالي HELLO, WORLD) إذا كان الماكرو UPPERCASE غير مُعَرَّف (على التوالي مُعرف).

بما أنه لم يتم تعريف الماكرو UPPERCASE في المثال السابق, سيقوم الـ Preprocessor بتجاهل (أو حذف الجزء الأول من الإختبار, الموجود بين if defined و else.

V - 7, ما هي المشاكل التي قد تحدث نتيجة سوء استخدام الماكرو ؟

النقاط الثلاثة التالية تُمثل أشهر المشاكل :

1. عدم الاستخدام الصحيح للأقواس : ليكن الماكرو SQUARE ذو التعريف :

```
#define SQUARE(x) x * x
```

هو المسؤول عن حساب مساحة المربع من خلال القيمة التي تُرسل له. إذا تم إرسال البارامتر 9+1 مثلا على الماكرو السابق فإن النتائج ستكون خاطئة و غير متوقعة لدى البعض !, لأن SQUARE(9 + 1) سيتم إبدالها بـ 9+1 * 9+1 الذي يكافئ في لغة C _حسب أولوية المؤثرات_ الكتابة 9+(1*9)+1 ! و بالتالي الخطأ نتج بسبب تقديم أولوية الضرب على الجمع في هذه الحالة, لذا يُنصح دائما باستخدام الأقواس عند إجراء عمليات كهذه : #define SQUARE(x) ((x) * (x))

2. التأثيرات الجانبية (Side effects) : ليكن الماكرو MAX ذو التعريف :

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
```

هو المسؤول عن إيجاد أكبر العددين x و y. عندما نستدعي الماكرو MAX هكذا k = MAX(3, 2) فإن قيمة k ستكون 3 و هذا شيء بديهي ! لكن عندما نستدعي MAX هكذا k = MAX(++i, j), حيث i و j متغيرين من النوع int, يحملان نفس القيمة (و لتكن 2 مثلا), فإن قيمة k ستكون 4 !, بالرغم من أنها يجب أن تكون 3 !! الخطأ في هذه الحالة نتج عن طريقة ترجمة الكتابة k = MAX(++i, j), لأن الـ Preprocessor حَوَّلها إلى : k = ((++i) > (j) ? (++i) : (j)) و بالتالي, من الطبيعي جدا أن يُصبح k=4 لأن المتغير i تمت زيادته مرتين.

3. أسماء الوسائط (Parameter name) :

هذا النوع من الأخطاء عادة ما يحدث عند الإعلان عن الماكرو هكذا :

```
#define ERR_PRINT_INT(n) fprintf(stderr, "%d\n", n)
```

عند تمرير القيمة 10 إلى الماكرو, سيُصبح هكذا :

```
fprintf(stderr, "%d\10", 10)  !
```

8 – V, ما هو دور المؤثر # في تعريف الماكرو ؟

المؤثر # يسمح بتحويل الوسيط (مهما كان) إلى سلسلة محارف, انظر المثال:

```c
#include <stdio.h>
#define TOSTR(x) #x
#define NNNNN 99999

int main() {
    printf("%s\n", TOSTR(10000)); /* --> "10000" */
    printf("%s\n", TOSTR(1 + 1)); /* --> "1 + 1" */
    printf("%s\n", TOSTR(n + 1)); /* --> "n + 1" */
    printf("%s\n", TOSTR(float)); /* --> "float" */
    printf("%s\n", TOSTR(NNNNN)); /* --> "NNNNN" */
    return 0;
}
```

كما نُلاحظ, يتم تحويل الكتابة TOSTR(NNNNN) إلى NNNNN# مع الأخذ في الاعتبار أن المعالج سيقوم بتحويل الوسيط السابق إلى "NNNNN" و ليس "99999", لكي نحصل على الكتابة الأخيرة بدل الأولى, يُمكننا استخدام الحيلة التالية:

```c
#define TOSTR(x) __STR(x)
#define __STR(x) #x
```

في هذه الحالة, سيتم تحويل الكتابة TOSTR(NNNNN) إلى ___STR(99999) التي سيتم تحويلها ثانية إلى "99999".

9 – V, ما هو دور المؤثر ## في تعريف الماكرو ؟

يستقبل المؤثر ## نصين و يقوم بدمجهما, الماكرو الآتي:

```c
#define CAT(x, y) x##y
```

يُحوِل CAT(C, 90) إلى C90, انظر المثال

```c
#include <stdio.h>
#include <wchar.h>
#define WIDESTR(x) L##x
#define AU_REVOIR "Au revoir"

int main(void) {
    wprintf(WIDESTR("%s\n"), WIDESTR("Bonjour")); /* --> wprintf(L"%s\n", L"Bonjour"); */
    return 0;
}
```

على عكس ما سبق, سيتم تحويل الكتابة WIDESTR(AU_REVOIRE) إلى L##AU_REVOIR و من ثم إلى LAU_REVOIR !, للحصول على الكتابة "L"Au revoir بدلا من الكتابة السابقة, يمكننا استخدام الحيلة التالية:

```c
#define WIDESTR(x) ___WSTR(x)
#define ___WSTR(x) L##x
```

في هذه الحالة, سيتم تحويل الكتابة WIDESTR(AU_REVOIR) إلى ___WSTR("Au revoir") التي سيتم تحويلها ثانية إلى "L"Au revoir.

V - 10, ما هي فائدة #pragma ؟

تكمن فائدة التوجيه #pragma في إرسال أمر معين إلى المترجم (أحد خيارات الترجمة أو الربط, ...), هذه الأوامر تكون دائما خاصة بالمترجم. ينص المعيار القياسي لـ C على أن المترجم يُمكنه تجاهل الأمر المرسل عندما لا يستطيع التعرف عليه, كما يُمكنه أيضا إرسال تحذير (Warning) ليكون المستخدم على علم بما يحدث.
عادة ما يُستخدم هذا التوجيه للتحكم في المترجم حسب رغبة المبرمج و بالتالي قد تجد أن بعض الأوامر تختلف باختلاف المترجم, لذا يُستحسن مراجعة المساعد الخاص بالمترجم الذي تستعمله.

V - 11, متى أستخدم #error ؟

يُساعد هذا التوجيه في تنبيه المبرمج (في مرحلة الترجمة) إلى أخطاء يقوم هذا الأخير بتجهيزها خوفا من الوقوع فيها.
يعتبر البعض أن هذا التوجيه ما هو إلا محاكاة كلاسيكية للـ Exception الموجودة في C++, فمثلا إذا أردنا ترجمة كود معين بشرط أن يكون حجم char هو 8 bits فيمكننا كتابة :

```
#include <limits.h>
...
#if (CHAR_BIT != 8)
#error Ce programme requiert que la taille d'un char soit de 8 bits.
#endif
...
```

V - 12, هل يمكننا استخدام sizeof مع #if ؟

لا !, لأن sizeof تتم معالجتها أثناء الترجمة و ليس قبلها.
الطريقة (أو الحيلة) التالية تسمح بتوليد خطأ مباشر أثناء الترجمة عندما لا يتحقق الشرط, في هذا المثال سيتم توليد خطأ عندما تكون sizeof(int) تختلف عن sizeof(long) :

```
#include <limits.h>
...
#if (CHAR_BIT != 8)
#error Ce programme requiert que la taille d'un char soit de 8 bits.
#endif
...
```

V ,13 - ما هي الأسماء المُعـرَّفة (Predefined Names) و ما فائدتها ؟

هي مجموعة من المختصرات Macros جاهزة, و يُمكن تمييزها بالرمزين Underscore في بداية و نهاية اسم المختصر. الجدول التالي يُبين جميع الأسماء المُعرفة :

المختصر	الشرح
__LINE__	ثابت عشري يحمل رقم سطر المصدر الحالي
__FILE__	سلسلة حرفية تحمل اسم الملف الجاري ترجمته
__TIME__	سلسلة حرفية تحمل الوقت الذي تمت فيه الترجمة
__DATE__	سلسلة حرفية تحمل التاريخ الذي تمت فيه الترجمة
__STDC__	تكون قيمة هذا الثابت 1 إذا كان المترجم المستعمل مترجم قياسي للغة C

و هذا مثال حول طريقة الاستخدام :

```
#include<stdio.h>

int main() {
    printf("Line: %d\n", __LINE__);
    printf("File: %s\n", __FILE__);
    printf("Time of compilation: %s\n", __TIME__);
    printf("Date of compilation: %s\n", __DATE__);
    printf("AINSI C(False = 0, True = 1): %d\n", __STDC__);
    return 0;
}
```

VI - الفصل السادس : من هنا و هناك ! (مُتفرقات في اللغة)

VI - 1, كيف أحصل على معيار لغة السي (C standard) ؟

تم توحيد لغة C من طرف ISO. معيار لغة C مُتاح للبيع على موقع الـ ISO :

C90 : http://www.iso.org/iso/com/catalogue_detail?csnumber=17782
C99 : http://www.iso.org/iso/com/catalogue_detail?csnumber=29237

و كذلك على موقع الـ ANSI :

C89 : http://webstore.ansi.org/RecordDetail.aspx?sku=AS%203955-1991

في كل الحالات تجد أن الثمن مكلف بعض الشيء.

النسخة التجريبية الأخيرة من C99 التي تحتوي على TC1, TC2, TC3 يمكن تنزيلها مجانا من موقع الـ OpenStd

http://www.open-std.org/

و بالتحديد من هنا :

http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

VI - 2, ما هي فائدة Null Statement ؟ (الإجابة للأستاذ خالد الشايع)

الـ statement التي تتكون من فاصلة منقوطة فقط, تسمى null statement في C و C++ على حد سواء.

قد تجد فكرة مشابهة لها في لغات أخرى كـ Assembly و تسمى في بعض المنصات NOP أو no operation.

بمعنى أن ما تقوم به هو أن لا تقوم بأي شيء. فائدتها أننا أحياناً, نحتاج لها. مثلاً, نريد جمع مكعب الأعداد في مدى معين – من سالب ثلاثة إلى ثلاثة مثلاً!

```
int sum = 0;
for(int n = -3; n <= 3; sum += n++);
```

أو كما هو الحال في الـ implementation الشهير لـ strcpy من كتاب K&R:

```
// src & dest are pointers to char arrays.
// forget the details for moment!
// src is the source string.
// dest is the destination string.
while(*dest++ = *src++);
```

ببساطة إن كان هناك موضع في اللغة يحتاج إلى statement فإنه يمكن استخدام null statement إن كنت لا تريد تنفيذ شيء على الإطلاق. مثلاً, assert تستخدم في حالة الـ debugging و في حالة تفعيل الـ optimizations في المترجمات فإنه يتم إزالتها عن طريق استخدام null statement. هناك طرق أخرى لعمل implementation لـ assert و لكن هذه إحداها.

VI – 3, ماذا يعني الرمز _ أمام اسم دالة أو ماكرو أو متغير ؟

في أغلب الأحيان, يدل هذا الرمز على أن الدالة أو الماكرو المُرافق له غير قياسي, يُمكننا القول أنه متوافق مع المكتبة القياسية التي تُوفرها البيئة التي تعمل عليها لكنه ليس متوافق مع معيار اللغة. على سبيل المثال, في نظام DOS/Windows, تَتْبع ثوابت الماكرو أو الدوال التالية :

_getch,_kbhit, _spawn*, _P_WAIT, …

للمكتبة CRT و هي اختصار لـ C Run-Time Library و ليس للمعيار القياسي.

نفس الشيء بالنسبة للدوال _read, _write, _exec* التابعة لمعيار POSIX.

أما بالنسبة للكلمات المعرفة مسبقا و الغير قياسية فتجد أمامها رمزين من Underscore بدلا من رمز واحد, هكذا :

__declspec, __stdcall, __asm, __int32, __int64, …

VI – 4, ما هو عمل الدالة fork ؟

تقوم الدالة fork بإنشاء Process ابن من الـ Process الأب للبرنامج بحيث يكون الـ Process الابن مُطابق تماما لأبيه من ناحية الكود . يعمل الأب و الابن في نفس الوقت إلا أن الأخير يبدأ بالتنفيذ انطلاقا من الأمر الذي يلي frok مباشرة.

frok لا تأخذ أي معاملات و لكنها تعيد متغير من نوع pid_t, .. لتقريب الفكرة, يمكنك اعتبار أن الدالة frok تُعيد متغير int.

القيمة المعادة من طرف الدالة تختلف باختلاف وضع أو طبيعة الـ Process. توجد 3 حالات :

- إذا كانت القيمة المُعادة تساوي 1- : حدث خطأ أثناء إنشاء الـ Process الإبن.
- إذا كانت القيمة المُعادة تساوي 0 : التحكم الآن بيد الـ process الابن.
- إذا كانت القيمة المُعادة أكبر من 0 : هذه القيمة تعبر عن الـ PID الخاص بالابن.

VI - 5, ما هي فائدة الدالة system ؟

راجع هذا الموضوع:

http://www.arabteam2000-forum.com/index.php?showtopic=211463

VI - 6, كيف أحول التاريخ إلى سلسلة حرفية ؟

تُمَكِّن الدالة ctime من تحويل timestamp إلى سلسلة حرفية من خلال ضبط التوقيت المحلي :

```
#include <stdio.h>
#include <time.h>

int main() {
    time_t t = time(NULL);
    printf("%s\n", ctime(&t));
    return 0;
}
```

أيضا, يمكننا تحويل بنية من النوع struct tm إلى سلسلة محارف باستخدام الدالة asctime و بالتالي الكود السابق مكافئ لـ :

```
#include <stdio.h>
#include <time.h>

int main() {
    time_t t = time(NULL);
    printf("%s\n", asctime(localtime(&t)));
    return 0;
}
```

VI - 7, كيف أقوم بتوليد أرقام عشوائية ؟

راجع هذا الموضوع:

http://www.arabteam2000-forum.com/index.php?showtopic=217773

VI - 8, ما معنى الخطأ unresolved external symbol _WinMain@16 ؟

هذا الخطأ ينتج عن عدم وجود الدالة WinMain في الكود المُستعمل. WinMain تقوم مقام الدالة main إذا كان الكود مُخصص لـ Windows. لإصلاح الخطأ غيِّرْ نوع المشروع إلى Console Application project.

VI - 9, ما معنى التحذير no new line at end of file ؟

المعيار القياسي ينص على أن كل سطر يجب أن ينتهي بعلامة سطر جديد (Line break or New line),
هذا التحذير يدل على أن السطر الأخير من الملف غير مكتمل, قم بتكملة السطر الأخير من الملف لتخطي هذا التحذير.

VI - 10, أين أجد دروس مُفصَّلة في لغة C ؟

راجع فهرس المواضيع المميزة:
http://www.arabteam2000-forum.com/index.php?showtopic=204712

VI - 11, أين أجد تمارين جيدة في لغة C ؟

راجع هذا الموضوع:
http://www.arabteam2000-forum.com/index.php?showtopic=268761
أيضا, يحتوي موقع Developpez.com على مجموعة من الكتب الرائعة جدا في لغة C :
http://c.developpez.com/livres/
و التي تضم العديد من التمارين و النصائح القيمة.
من أهم هذه الكتب, الكتاب الأب The C Programming Language (الذي يُعتبر المرجع الأساسي للغة C) :
http://c.developpez.com/livres/#L2100487345
و الذي يحتوي بدوره على مجموعة من التمارين الجيدة في جميع المستويات, و هنا تجد حل التمارين:
http://users.powernet.co.uk/eton/kandr2/

V - ملحق

V - 1, ما معنى FAQ ؟

FAQ اختصار لـ "Frequently Asked Questions" أي "الأسئلة الأكثر شيوعا", تحتوي عادة على إجابات مُفصلة للأسئلة الأكثر تكراراً في موضوع مُعين, حيث تكمن فائدتها في تقديم الإجابات و منع التكرار الدائم في طرح الأسئلة.

V - 2, أين أجد رابط الموضوع في المنتدى ؟

تفضل :

http://www.arabteam2000-forum.com/index.php?showtopic=266986

V - 3, أريد أن أفهم أكثر بعض الإجابات الموجودة في الـ FAQ, كيف ذاك ؟

إذا كان لديك استفسار في أحد الفصول, يمكنك طرح سؤالك في هذا الموضوع:
http://www.arabteam2000-forum.com/index.php?showtopic=266991

V - 4, و استفساراتي الأخرى الغير متعلقة بالـ FAQ ؟

تلك الأسئلة لا علاقة لها بهذا الموضوع بشكل مباشر, اكتب موضوع جديد في القسم الرئيسي :
http://www.arabteam2000-forum.com/index.php?showforum=14
و ستجد المساعدة إن شاء الله.

V - 5, هل المشاركة في الـ FAQ مفتوحة للجميع ؟

نعم و لكن بشروط.

V - 6, إذا, كيف أشارك فيها ؟

تأكد جيدا من صحة الإجابات التي ستقدمها, بعد التأكد يمكنك مراسلتي على الخاص لإضافة مشاركتك إلى الموضوع المناسب.

V - 7, عفوا, لقد وجدتُ خطأ في إحدى الإجابات !

جلَّ من لا يُخطئ !, أرسل لي رقم الفصل و اسم السؤال و دليل الخطأ, و سأقوم بتصحيحه إن شاء الله.

8 - V, ما هي المصادر التي اعتمدتَ عليها في كتابة هذه المواضيع ؟

أغلب المواضيع سبق و أن تعرضتُ لها في كتاب "الكافي".

إضافة إلى ذلك, استعنت بـ :

- Programmer en langage C - Claude Delannoy, 5ème édition, Éditions Eyrolles
- FAQ C - Club des professionnels en informatique

تم بحمد الله