Développer pour l'Amiga 500 en C

un guide pas à pas pour créer un jeu de quiz avec Visual Studio Code
et le tester sur émulateur et matériel réel

version corrigée — mars 2026

Table des matières

(Cliquez avec le bouton droit → mettre à jour le champ pour générer la table)

1. Introduction au Développement sur Amiga

Bienvenue dans ce guide complet dédié à la programmation en langage C sur l'une des machines les plus emblématiques des années 80 et 90 : le Commodore Amiga. Plus précisément, nous nous concentrerons sur le modèle phare, l'Amiga 500. Ce cours est conçu pour vous guider, étape par étape, depuis la configuration d'un environnement de développement moderne jusqu'à la création d'un jeu de quiz fonctionnel, en explorant les concepts fondamentaux de la programmation sur cette plateforme unique.

1.1. Pourquoi développer sur Amiga aujourd'hui ?

En 2025, se lancer dans le développement sur une machine vieille de près de 40 ans peut sembler anachronique. Pourtant, la scène "rétro-informatique" est plus vivante que jamais. Développer sur Amiga, c'est avant tout :

- **Un défi technique stimulant :** Vous apprendrez à travailler avec des ressources limitées (CPU à 7 MHz, 512 Ko de RAM), ce qui force à écrire du code optimisé et efficace. C'est un excellent exercice pour tout programmeur.
- **Comprendre l'architecture matérielle :** L'Amiga est célèbre pour ses "custom chips" (Agnus, Paula, Denise) qui déchargent le processeur principal de nombreuses tâches graphiques et

sonores. Apprendre à les manipuler directement ("hardware banging") est une expérience unique et profondément formatrice.
- **Rejoindre une communauté passionnée :** La communauté Amiga est active, avec des forums, des événements (demoscenes) et une multitude de projets open source. Vous trouverez toujours de l'aide et de l'inspiration.
- **La nostalgie et le plaisir :** Pour beaucoup, c'est une façon de revivre la magie de leur premier ordinateur, mais cette fois-ci en passant de l'autre côté de l'écran, en devenant créateur.

Ce guide s'inspire de méthodes modernes pour rendre cette expérience aussi agréable que possible, en utilisant des outils contemporains comme Visual Studio Code pour pallier les limitations des éditeurs de l'époque, qui manquaient souvent de coloration syntaxique, d'IntelliSense ou de formatage automatique .

1.2. L'écosystème Amiga : Kickstart, Workbench et AmigaOS

Pour programmer sur Amiga, il est crucial de comprendre son architecture logicielle. Contrairement aux systèmes modernes où tout est intégré, l'AmigaOS est composé de deux parties distinctes mais interdépendantes .

Le Kickstart

Le Kickstart est le firmware de l'Amiga, stocké sur une puce ROM sur la carte mère. On peut le comparer au BIOS d'un PC. Il contient les éléments essentiels du système d'exploitation :

- **Exec :** Le noyau multitâche préemptif, qui gère les tâches, la mémoire et les interruptions. C'est la bibliothèque fondamentale sur laquelle tout le reste repose .
- **Intuition :** La bibliothèque qui gère l'interface utilisateur graphique (fenêtres, écrans, gadgets, menus).
- **Graphics.library :** Fournit les fonctions de dessin de bas niveau (lignes, polygones, texte).
- **Dos.library :** Gère le système de fichiers, l'accès aux disques et l'exécution des programmes.
- Et bien d'autres bibliothèques essentielles.

La version du Kickstart détermine les fonctionnalités de base de la machine. Pour un Amiga 500, la version la plus courante est la **1.3**.

Le Workbench

Le Workbench est l'interface graphique utilisateur qui se charge depuis une disquette ou un disque dur. Il fournit l'environnement de bureau avec ses icônes, ses fenêtres et ses outils. Il dépend entièrement des bibliothèques présentes dans le Kickstart pour fonctionner.

Une particularité de l'Amiga est que de nombreux jeux et démos démarraient directement depuis leur propre disquette "bootable", sans charger le Workbench. Ils prenaient le contrôle total de la machine en utilisant uniquement les routines du Kickstart, libérant ainsi un maximum de mémoire et de ressources . Pour notre projet de quiz, nous développerons une application qui se lance depuis le Workbench, ce qui est la méthode standard pour les logiciels "applicatifs".

1.3. Développement Natif vs. Cross-Compilation

Il existe deux approches principales pour développer sur Amiga :

1. **Le développement natif :** C'est la méthode "à l'ancienne". On utilise un Amiga (réel ou émulé) avec un compilateur C installé directement sur AmigaOS, comme SAS/C ou StormC. On écrit le code dans un éditeur de texte Amiga comme CygnusED ou même l'éditeur de SAS/C. C'est une expérience authentique, mais qui peut être lente et fastidieuse. Une simple erreur de pointeur peut faire planter tout le système (le fameux "Guru Meditation"), obligeant à redémarrer .

2. **La cross-compilation :** C'est l'approche moderne et celle que nous allons privilégier. On utilise un PC puissant avec un environnement de développement moderne (comme Visual Studio Code) pour écrire le code. Un compilateur spécial, appelé "cross-compiler" (comme VBCC ou une version modifiée de GCC), s'exécute sur le PC mais génère un exécutable compatible avec le processeur Motorola 68000 de l'Amiga. On teste ensuite cet exécutable sur un émulateur ou on le transfère sur un vrai Amiga.

Ce guide va présenter une méthode hybride, popularisée par des développeurs comme l'auteur du blog "Joe's Blog" . Nous allons :

- Utiliser **Visual Studio Code sur Windows** pour l'édition de code, profitant de tout son confort.
- Utiliser un **émulateur (WinUAE)** pour faire tourner un environnement AmigaOS complet.
- Installer le compilateur **SAS/C** à l'intérieur de l'émulateur.
- Partager le dossier de notre projet entre Windows et l'Amiga émulé.
- Écrire le code dans VS Code, puis basculer sur la fenêtre de l'émulateur pour lancer la compilation et tester.

Cette méthode combine le meilleur des deux mondes : le confort d'un IDE moderne et la compatibilité d'un compilateur natif historique. Plus loin dans le guide, nous explorerons également une approche de cross-compilation pure avec des outils plus récents.

2. Mise en Place de l'Environnement de Développement

Avant d'écrire la moindre ligne de code, nous devons préparer notre atelier numérique. Cette section vous guidera dans l'installation et la configuration des trois piliers de notre environnement : l'émulateur WinUAE, le compilateur SAS/C et l'éditeur Visual Studio Code.

2.1. L'Émulateur : WinUAE, votre Amiga virtuel

WinUAE est l'émulateur Amiga le plus avancé et le plus complet pour Windows. Il nous permettra de créer un Amiga 500 virtuel sur notre PC.

Ce dont vous avez besoin :

- **WinUAE :** Téléchargez la dernière version depuis le site officiel.
- **Le Kickstart ROM :** C'est le firmware de l'Amiga. Pour un A500, vous aurez besoin du fichier **Kickstart 1.3** (souvent nommé `kick34005.A500`). Les ROMs sont sous copyright. La manière la plus simple et légale de les obtenir est d'acheter le pack Amiga Forever de Cloanto, qui inclut toutes les versions des ROMs et des Workbench pré-configurés .
- **Les disquettes du Workbench :** Vous aurez besoin des disquettes du **Workbench 1.3** pour installer le système d'exploitation sur un disque dur virtuel. Elles sont également incluses dans Amiga Forever.

Configuration de base de WinUAE

L'installation de WinUAE est simple. Une fois installé, lancez-le. L'interface peut paraître intimidante, mais nous n'aurons besoin que de quelques réglages.

3. **Chemins (Paths) :** Allez dans l'onglet `Paths`. WinUAE tentera de détecter automatiquement vos ROMs si vous avez installé Amiga Forever. Sinon, indiquez manuellement le dossier où vous avez placé votre fichier Kickstart 1.3.
4. **Configuration rapide (Quickstart) :** Allez dans l'onglet `Quickstart`. Sélectionnez `Amiga 500` comme modèle. WinUAE devrait automatiquement sélectionner la ROM Kickstart 1.3 correspondante.
5. **CPU et FPU :** Dans l'onglet `CPU and FPU`, assurez-vous que le CPU est bien `68000`. Pour la compatibilité, laissez la vitesse sur `Approximate A500/A600/A1200 speed`. Pour le développement, l'auteur de Joe's Blog recommande le réglage "fastest possible" pour accélérer la compilation, mais nous commencerons par une configuration standard .
6. **RAM :** Dans l'onglet `RAM`, une configuration standard pour un A500 étendu est **512 KB de Chip RAM** et **512 KB de Slow RAM**. La Chip RAM est la mémoire accessible par les puces custom, essentielle pour les graphismes et le son. La Slow RAM (ou Fast RAM) est réservée au CPU .

Création et Installation d'un Disque Dur Virtuel

Pour installer SAS/C et stocker nos projets, nous avons besoin d'un disque dur.

7. **Créer le disque dur :** Allez dans l'onglet `Hard drives`. Cliquez sur `Add Hardfile...`. Dans la nouvelle fenêtre, spécifiez une taille (par exemple, 200 Mo, ce qui est énorme pour un Amiga 1.3) et un emplacement sur votre PC pour enregistrer le fichier `.hdf`. Laissez les autres options par défaut et cliquez sur `Create`. Le fichier est maintenant listé.
8. **Installer le Workbench :** Allez dans l'onglet `Floppy drives`. Insérez la disquette `Workbench 1.3` dans le lecteur `DF0:`. Insérez la disquette `Install 1.3` dans `DF1:`.
9. **Démarrer l'émulation :** Cliquez sur `Start`. L'Amiga virtuel va démarrer. Vous devriez voir les icônes des disquettes et du disque dur (nommé `DH0:`, probablement non formaté).
10. **Formater et Installer :** Ouvrez la disquette `Install 1.3`, lancez l'installeur et suivez les instructions pour formater votre disque dur `DH0:` et y copier les fichiers du Workbench. Cela implique généralement de changer de disquette plusieurs fois.

Une fois l'installation terminée, vous pouvez éjecter les disquettes virtuelles et redémarrer. L'Amiga devrait maintenant booter directement sur votre disque dur virtuel. C'est une bonne idée de sauvegarder cette configuration dans WinUAE. Allez dans l'onglet `Configurations`, donnez un nom (ex: "A500 Dev SAS/C") et cliquez sur `Save`.

2.2. Le Compilateur C : SAS/C, le choix classique

SAS/C (originellement Lattice C) était l'un des compilateurs C les plus populaires et les plus robustes pour l'Amiga classique. Même si son développement est arrêté depuis longtemps, il reste une référence pour le développement sur AmigaOS 1.x à 3.x .

Obtention et Installation de SAS/C

SAS/C est un logiciel commercial abandonné ("abandonware"). Vous pouvez le trouver sur des sites d'archives comme le Internet Archive ou via des forums spécialisés comme l'English Amiga Board (EAB) . Vous aurez besoin des fichiers ADF (images de disquettes) de la version 6.50 et des patchs successifs jusqu'à la 6.58.

L'installation se fait entièrement dans l'environnement émulé de WinUAE :

11. Démarrez votre configuration WinUAE avec le disque dur Workbench 1.3.
12. Dans l'interface de WinUAE (appuyez sur F12 pour y revenir), insérez la première disquette d'installation de SAS/C dans le lecteur `DF0:`.
13. Retournez à l'émulation. Ouvrez l'icône de la disquette et lancez le programme d'installation.

14. Suivez les instructions à l'écran. L'installeur vous demandera de choisir un disque de destination (votre `DH0:`) et vous invitera à insérer les autres disquettes au fur et à mesure .
15. Une fois l'installation de base terminée, appliquez les patchs de la même manière pour mettre à jour votre installation vers la version 6.58.

L'installeur ajoutera des lignes à votre fichier de démarrage `S:User-Startup` pour définir les chemins nécessaires au compilateur (comme `sc:`, `lib:`, `include:`) . Après un redémarrage, SAS/C devrait être prêt à l'emploi.

2.3. L'Éditeur de Code : Visual Studio Code

Nous allons utiliser Visual Studio Code (VS Code) sur Windows pour écrire notre code. C'est un éditeur gratuit, puissant et extensible.

Installation et Extensions

16. Téléchargez et installez VS Code depuis le site officiel.
17. Lancez VS Code et allez dans l'onglet des extensions (icône des carrés sur la gauche).
18. Installez l'extension indispensable : **C/C++** de Microsoft. C'est elle qui fournira la coloration syntaxique, l'IntelliSense (auto-complétion, informations sur les fonctions) et les capacités de navigation dans le code .

L'image ci-dessus, tirée du tutoriel original, montre une liste d'extensions. Pour notre approche, seule l'extension **C/C++** est strictement nécessaire au début. Nous aborderons d'autres extensions plus avancées plus tard.

À ce stade, notre environnement est prêt. Nous avons un Amiga virtuel avec un compilateur, et un éditeur de code moderne sur notre machine hôte. La prochaine étape est de les faire communiquer.

3. Le Flux de Travail : Un Pont entre Windows et Amiga

L'astuce de cette configuration réside dans le partage d'un dossier entre le système d'exploitation hôte (Windows) et le système invité (AmigaOS dans WinUAE). Cela nous permet d'éditer les fichiers avec VS Code et de les compiler instantanément dans l'émulateur sans aucune étape de transfert manuel. Cette section suit de près la méthode décrite dans l'article de Joe's Blog.

3.1. Création et partage du dossier de projet

Étape 1 : Créer le dossier sur Windows

Créez un dossier sur votre PC qui contiendra tous vos projets Amiga. Pour une organisation claire, vous pouvez recréer une structure qui imite celle d'un disque dur Amiga. Par exemple, créez un dossier `C:\AmigaDev`. À l'intérieur, créez un dossier `Work` (qui correspondra au volume `Work:` sur l'Amiga), puis `Projects` à l'intérieur de `Work`, et enfin un dossier pour notre premier projet, par exemple `QuizGame`.

Le chemin complet serait : `C:\AmigaDev\Work\Projects\QuizGame`.

Étape 2 : Partager le dossier dans WinUAE

Nous allons maintenant "monter" ce dossier Windows comme un disque dur dans WinUAE.

19. Arrêtez votre émulation Amiga si elle est en cours.
20. Dans l'interface de WinUAE, allez dans l'onglet `Hard drives`.
21. Cliquez sur `Add directory...`.
22. Dans la fenêtre qui s'ouvre :

- **Device Name :** Entrez un nom pour le volume, par exemple `Work`.
- **Volume Label :** Vous pouvez laisser le même nom, `Work`.
- **Path :** Cliquez sur `...` et sélectionnez le dossier que vous avez créé sur Windows, c'est-à-dire `C:\AmigaDev\Work`.

23. Cliquez sur `OK`. Le dossier partagé apparaît maintenant dans la liste des disques durs.

Dans l'exemple de l'image, l'auteur a monté deux dossiers : un pour le Workbench (`DH0`) et un pour ses projets (`DH1`). Dans notre cas, nous avons un disque dur virtuel `DH0:` et nous ajoutons un volume `Work:` qui pointe vers un dossier de notre PC.

Étape 3 : Vérifier le partage dans AmigaOS

Sauvegardez votre configuration et démarrez l'émulation. Sur le bureau du Workbench, une nouvelle icône de disque dur nommée `Work` devrait apparaître. Si vous double-cliquez dessus, vous verrez son contenu, qui est le reflet exact de votre dossier `C:\AmigaDev\Work` sur Windows. Vous pouvez maintenant naviguer jusqu'à `Work:Projects/QuizGame` depuis le Shell Amiga (CLI).

3.2. Configuration de Visual Studio Code

Maintenant que le pont est établi, configurons VS Code pour qu'il comprenne notre projet Amiga.

Étape 1 : Ouvrir le dossier du projet

Dans VS Code, allez dans `File > Open Folder...` et sélectionnez le dossier de votre projet, c'est-à-dire `C:\AmigaDev\Work\Projects\QuizGame`. L'explorateur de fichiers de VS Code affichera un dossier vide.

Étape 2 : Créer le premier fichier source

Cliquez sur l'icône "New File" dans l'explorateur de VS Code et créez un fichier nommé `main.c`.

Étape 3 : Configurer l'IntelliSense (le chemin d'inclusion)

C'est l'étape la plus importante et la plus "magique". Pour que VS Code puisse vous aider avec l'auto-complétion des fonctions AmigaOS, il doit savoir où se trouvent les fichiers d'en-tête (`.h`) de SAS/C. Ces fichiers sont sur votre disque dur Amiga émulé, dans le dossier `sc:include`.

La méthode la plus simple pour déclencher la configuration est de créer une erreur intentionnelle. Dans votre fichier `main.c`, tapez la ligne suivante :

```
#include <exec/types.h>
```

L'extension C/C++ soulignera cette ligne d'un zigzag vert, indiquant qu'elle ne trouve pas le fichier. Une ampoule jaune va apparaître.

Cliquez sur l'ampoule, puis sur `Edit "includePath" setting`. Cela ouvrira le fichier de configuration JSON de l'extension C/C++ (`c_cpp_properties.json`) dans un onglet.

Dans ce fichier, trouvez la section `includePath`. Vous devez y ajouter le chemin d'accès au dossier `include` de SAS/C. Mais attention, ce chemin doit être le chemin vu depuis Windows, à travers le partage que nous avons configuré.

Si vous avez installé SAS/C sur votre disque dur virtuel `DH0:`, ce chemin n'est pas directement accessible. La solution est de copier le dossier `include` de SAS/C vers notre dossier partagé.

24. Dans l'émulateur Amiga, ouvrez un Shell (CLI).
25. Copiez l'intégralité du dossier d'includes de SAS/C vers votre volume partagé. La commande serait :

```
copy sc:include Work:include ALL CLONE
```

Maintenant, sur votre PC Windows, vous aurez un dossier `C:\AmigaDev\Work\include` contenant tous les fichiers `.h` nécessaires.

Retournez dans VS Code, dans le fichier `c_cpp_properties.json`. Modifiez la section `includePath` pour y inclure ce nouveau dossier. Le chemin doit utiliser des barres obliques `/` ou des doubles barres obliques inverses `\\`.

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**",
                "C:/AmigaDev/Work/include/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "compilerPath": "C:/MinGW/bin/gcc.exe", // Ce chemin n'est pas important
pour l'IntelliSense Amiga
            "cStandard": "c11",
            "cppStandard": "c++17",
            "intelliSenseMode": "windows-gcc-x86"
        }
    ],
    "version": 4
}
```

Sauvegardez le fichier `c_cpp_properties.json`. Fermez et rouvrez votre fichier `main.c`. Le zigzag vert sous `#include ` devrait avoir disparu. Mieux encore, si vous commencez à taper le nom d'une structure ou d'une fonction Amiga, l'IntelliSense devrait se déclencher !

3.3. Écrire, Compiler et Exécuter votre premier programme

Maintenant que tout est en place, suivons le cycle complet de développement.

Étape 1 : Écrire le code "Hello, World!"

Dans VS Code, écrivez ce petit programme dans `main.c`. Il utilise une fonction de la `dos.library` pour afficher un message dans la console.

```
#include <proto/dos.h>
#include <stdio.h>

/*
 * Pour utiliser les fonctions de dos.library, nous avons besoin
 * de son pointeur de base. Nous le déclarerons en global.
 */
struct Library *DOSBase;

int main(void)
{
    /*
     * Avant d'utiliser une bibliothèque, il faut l'ouvrir.
     * La bibliothèque "exec.library" est la seule qui est toujours ouverte.
     * Nous n'avons pas besoin de l'ouvrir, mais nous avons besoin de son
     * pointeur de base (ExecBase) pour appeler OpenLibrary.
     * Sous SAS/C, ExecBase est automatiquement disponible.
     */
    DOSBase = OpenLibrary("dos.library", 34); // Version 34 pour Kickstart 1.3

    if (DOSBase)
    {
        // La bibliothèque est ouverte, nous pouvons utiliser ses fonctions.
        Printf("Bonjour, Amiga depuis VS Code!\n");

        // Il est crucial de fermer les bibliothèques que l'on a ouvertes.
        CloseLibrary(DOSBase);
        return 0; // RETURN_OK
    }

    // Si la bibliothèque n'a pas pu être ouverte, on retourne une erreur.
    printf("Impossible d'ouvrir dos.library\n");
    return 20; // RETURN_FAIL
}
```

> **Note sur le code :** Ce code utilise `Printf` de la `dos.library` de l'Amiga, qui est la manière standard d'écrire dans la console. Il utilise aussi `OpenLibrary` et `CloseLibrary` de `exec.library`, qui sont fondamentales pour toute programmation AmigaOS. Nous reviendrons en détail sur ces concepts.

Étape 2 : Préparer le projet pour SAS/C

SAS/C utilise un système de "makefile" pour gérer la compilation de projets. Avant de pouvoir compiler, nous devons initialiser le projet avec une commande SAS/C.

26. Basculez sur la fenêtre de l'émulateur WinUAE.
27. Ouvrez un Shell (CLI).

28. Naviguez jusqu'au dossier de votre projet :

```
cd Work:Projects/QuizGame
```

29. Lancez la commande de configuration de SAS/C :

```
scsetup
```

Cette commande va créer quelques fichiers de configuration pour SAS/C dans votre dossier de projet.

Étape 3 : Créer le Makefile et Compiler

Toujours dans le Shell Amiga :

30. **Créer le Makefile :** SAS/C fournit un outil pratique, `mkmk`, pour générer un makefile simple. La syntaxe est `mkmk NOM_EXECUTABLE=fichiers_sources`. Dans notre cas :

```
mkmk main=main.c
```

Cela crée un fichier nommé `Makefile` dans votre dossier.

31. **Compiler :** Maintenant, nous pouvons lancer la compilation avec `smake` (la version de `make` de SAS/C) :

```
smake
```

SAS/C va compiler votre `main.c` en `main.o` (fichier objet) puis le lier pour créer l'exécutable final, nommé `main`.

Étape 4 : Gérer les erreurs de compilation

Si votre code contient des erreurs, `smake` les affichera dans la console. Il peut être difficile de les lire dans la petite fenêtre du Shell. Une astuce consiste à rediriger la sortie vers un fichier texte, que nous pourrons ensuite consulter confortablement dans VS Code.

Basculez sur VS Code. Un nouveau fichier `erreurs.txt` sera apparu dans votre projet. Vous pouvez l'ouvrir pour analyser les erreurs, corriger votre code dans `main.c`, puis relancer `smake` dans l'émulateur.

Étape 5 : Exécuter le programme

Une fois la compilation réussie, un exécutable nommé `main` se trouve dans votre dossier. Pour le lancer depuis le Shell, tapez simplement son nom :

Le message "Bonjour, Amiga depuis VS Code!" devrait s'afficher dans la console !

Félicitations ! Vous avez mis en place un flux de travail de développement C pour Amiga complet et fonctionnel. Vous pouvez maintenant vous concentrer sur l'apprentissage des spécificités de la programmation Amiga.

4. Les Fondamentaux du Langage C pour l'Amiga

Programmer en C sur Amiga est différent de la programmation C sur un système moderne comme Linux ou Windows. L'architecture unique de l'Amiga et son système d'exploitation imposent des concepts et des pratiques spécifiques. Cette section plonge au cœur de ces spécificités.

4.1. Les Spécificités de l'AmigaOS

Le Multitâche Préemptif sans Protection Mémoire

L'Amiga a été l'un des premiers ordinateurs personnels à offrir un système d'exploitation multitâche préemptif. Cela signifie que le système d'exploitation (le noyau Exec) est responsable de l'ordonnancement des tâches, leur donnant à tour de rôle un petit temps de processeur. Du point de vue du programmeur, cela donne l'impression que plusieurs programmes tournent simultanément .

Cependant, les Amiga classiques (basés sur les processeurs 680x0) ne disposent pas d'une unité de gestion mémoire (MMU). Il n'y a donc **aucune protection de la mémoire**. Un programme peut lire ou écrire n'importe où en mémoire, y compris dans l'espace d'un autre programme ou même du système d'exploitation.

> **Conséquence pour le développeur :** La rigueur est absolue. Une seule erreur de pointeur (un pointeur nul, un dépassement de tampon) peut corrompre la mémoire d'une autre tâche ou du système, menant à un comportement erratique ou à un crash complet de la machine (le fameux écran "Guru Meditation"). Des outils de débogage comme MungWall ou Enforcer peuvent aider à détecter ces erreurs, mais la meilleure défense reste un code propre et prudent.

La Coopération pour les Ressources

Puisque plusieurs tâches coexistent, elles doivent partager les ressources du système (périphériques, mémoire, fenêtres, etc.). L'AmigaOS fournit des mécanismes pour arbitrer l'accès à ces ressources. En tant que programmeur, vous avez la responsabilité de :

32. **Demander** une ressource au système avant de l'utiliser (ex: allouer de la mémoire, ouvrir un fichier, ouvrir une fenêtre).
33. **Vérifier** que la ressource vous a bien été accordée (l'appel peut échouer si la ressource n'est pas disponible).
34. **Libérer** la ressource dès que vous n'en avez plus besoin, pour que d'autres tâches puissent l'utiliser.

Le système ne nettoie pas automatiquement derrière vous à la fermeture de votre programme. Oublier de libérer une ressource (une "fuite") la rendra indisponible pour tout le système jusqu'au prochain redémarrage.

4.2. Le Cœur du Système : Les Bibliothèques (.library)

La quasi-totalité des fonctionnalités de l'AmigaOS est accessible via des **bibliothèques partagées dynamiques**. Ce sont des fichiers, généralement situés dans le dossier `LIBS:`, qui contiennent des ensembles de fonctions liées. Par exemple, `dos.library` pour les opérations sur les fichiers, `intuition.library` pour l'interface graphique, `graphics.library` pour le dessin, etc. .

L'avantage est qu'une seule copie de la bibliothèque est chargée en mémoire, peu importe le nombre de programmes qui l'utilisent, ce qui économise la précieuse RAM.

Le Rituel d'Utilisation d'une Bibliothèque

Pour utiliser une fonction d'une bibliothèque, vous devez suivre un rituel en trois étapes, orchestré par la bibliothèque maîtresse, **`exec.library`**, qui est toujours disponible.

35. **Déclarer un pointeur de base :** Chaque bibliothèque ouverte est représentée par un pointeur vers sa structure de base en mémoire. Par convention, ce pointeur est nommé en ajoutant "Base" au nom de la bibliothèque (ex: `DOSBase`, `IntuitionBase`).

```
struct Library *DOSBase;
```

36. **Ouvrir la bibliothèque :** On utilise la fonction `OpenLibrary()` d'Exec. On lui passe le nom de la bibliothèque et la version minimale requise.

```
            /* Sous SAS/C, le pointeur vers la base d'Exec, ExecBase, est global
et disponible.
   Pour les cross-compilers, il faut souvent le récupérer autrement. */
extern struct ExecBase *SysBase; // SysBase est un autre nom pour ExecBase

// On ouvre dos.library, en demandant au minimum la version 34 (Kickstart 1.3)
DOSBase = OpenLibrary("dos.library", 34);

// TOUJOURS vérifier si l'ouverture a réussi !
if (DOSBase == NULL)
{
    // Gérer l'erreur (par ex. quitter le programme)
}
```

37.

Fermer la bibliothèque : C'est une étape cruciale. Avant de quitter votre programme, vous devez fermer toutes les bibliothèques que vous avez ouvertes avec `CloseLibrary()`.

```
            if (DOSBase)
{
    CloseLibrary(DOSBase);
}
```

Appeler les Fonctions de la Bibliothèque

Une fois la bibliothèque ouverte et son pointeur de base obtenu, vous pouvez appeler ses fonctions. Les compilateurs C pour Amiga (comme SAS/C) utilisent un système de "macros de stubs" ou de "fichiers de prototypes" (`proto/dos.h`, par exemple) qui masquent la complexité des appels.

En coulisses, chaque appel de fonction est un saut indirect via le pointeur de base de la bibliothèque. Par exemple, appeler `Printf()` revient à faire quelque chose comme `JSR -offset(DOSBase)`, où `offset` est la position de la fonction `Printf` dans la table des vecteurs de la bibliothèque . Heureusement, en tant que programmeur C, vous n'avez qu'à inclure le bon fichier d'en-tête et appeler la fonction normalement.

```
// Inclure les prototypes pour dos.library
#include <proto/dos.h>
#include <proto/exec.h>

struct Library *DOSBase;

int main(void)
{
    DOSBase = OpenLibrary("dos.library", 34);
    if (DOSBase)
    {
        // Appel simple de la fonction Printf()
        Printf("Ceci est un appel à une fonction de dos.library !\n");
        CloseLibrary(DOSBase);
    }
    return 0;
}
```

Voici une table des bibliothèques les plus courantes que vous utiliserez :

Nom de la Bibliothèque	Pointeur de Base (Convention)	Rôle Principal
`exec.library`	`SysBase` ou `ExecBase`	Noyau : gestion des tâches, mémoire, ports, interruptions.
`dos.library`	`DOSBase`	Système de fichiers, entrée/sortie console, exécution de commandes.
`intuition.library`	`IntuitionBase`	Interface utilisateur : écrans, fenêtres, requêtes, événements.
`graphics.library`	`GfxBase`	Dessin 2D : lignes, polygones, texte, manipulation de bitmaps.
`gadtools.library`	`GadToolsBase`	Création simplifiée de gadgets (boutons, sliders, etc.) pour l'interface.
`asl.library`	`AslBase`	Boîtes de dialogue standard (sélection de fichier, de police).
`audio.device`	(C'est un "device", pas une "library")	Gestion du son. L'ouverture est légèrement différente.

4.3. Gestion de la Mémoire : Chip RAM vs. Fast RAM

L'une des plus grandes particularités de l'architecture Amiga est sa division de la mémoire vive en (au moins) deux types .

- **Chip RAM :** C'est la seule mémoire accessible à la fois par le CPU (le processeur 68000) **et** par les puces custom (Agnus, Denise, Paula). Par conséquent, toutes les données qui doivent être lues ou écrites par ces puces (données graphiques pour l'affichage, données sonores pour la lecture, listes d'instructions pour le Copper et le Blitter) **doivent impérativement résider en Chip RAM**. C'est une ressource précieuse et souvent limitée (512 Ko ou 1 Mo sur un A500 standard).

- **Fast RAM :** C'est une mémoire accessible **uniquement** par le CPU. Comme son nom l'indique, l'accès y est plus rapide car il n'y a pas de contention avec les puces custom qui accèdent à la Chip RAM. On y stocke généralement le code du programme, les variables, les piles et toutes les données qui n'ont pas besoin d'être vues par le matériel custom.

Allocation de Mémoire en C

La fonction standard C `malloc()` ne vous donne aucun contrôle sur le type de mémoire allouée. Pour programmer sur Amiga, il faut utiliser les fonctions de `exec.library` : `AllocMem()` et `FreeMem()`.

`AllocMem()` prend deux arguments : la taille en octets à allouer, et des attributs qui spécifient le type de mémoire.

```
#include <proto/exec.h>

// ... à l'intérieur d'une fonction

void *myFastMemoryBlock;
void *myChipMemoryBlock;

// Allouer 1024 octets de Fast RAM
// MEMF_ANY : n'importe quel type de mémoire, mais le système privilégie la Fast RAM.
// C'est le choix par défaut pour le code et les données du programme.
myFastMemoryBlock = AllocMem(1024, MEMF_ANY);

// Allouer 32000 octets de Chip RAM pour un écran (par exemple)
// MEMF_CHIP : demande explicitement de la Chip RAM.
// MEMF_CLEAR : demande au système de mettre la mémoire allouée à zéro.
myChipMemoryBlock = AllocMem(32000, MEMF_CHIP | MEMF_CLEAR);

// TOUJOURS vérifier le retour de AllocMem() !
if (myChipMemoryBlock == NULL)
{
    Printf("Plus de Chip RAM disponible !\n");
}

// ... utiliser les blocs mémoire ...

// Libérer la mémoire dans l'ordre inverse de l'allocation (bonne pratique)
if (myChipMemoryBlock)
{
    FreeMem(myChipMemoryBlock, 32000);
}
if (myFastMemoryBlock)
{
    FreeMem(myFastMemoryBlock, 1024);
}
```

Attention : `FreeMem()` requiert la taille exacte du bloc à libérer. Fournir une taille incorrecte peut corrompre le gestionnaire de mémoire du système. Il est donc essentiel de stocker la taille des blocs que vous allouez.

4.4. Accès Direct au Matériel (Hardware Banging)

Bien que l'AmigaOS fournisse des bibliothèques de haut niveau pour la plupart des tâches, la clé de la performance et des effets spectaculaires de la demoscene réside dans la programmation directe des puces custom. C'est un sujet vaste et complexe, mais voici une introduction aux concepts

de base.

Les puces custom sont contrôlées via un ensemble de registres mappés en mémoire, à des adresses fixes. En C, on y accède généralement via un pointeur vers une structure globale nommée `custom` .

```
#include <hardware/custom.h>

// Le compilateur (via amiga.lib) fournit cette structure globale
// qui pointe vers la zone mémoire des registres custom.
extern struct Custom custom;

void changeBackgroundColor(int color)
{
    // COLOR00 est le registre de la couleur de fond.
    // On écrit directement dedans.
    custom.color[0] = color;
}
```

> Avant d'accéder directement au matériel, un programme "bien élevé" doit demander au système d'exploitation la permission de le faire, pour éviter les conflits dans un environnement multitâche. Cela se fait en "verrouillant" la ressource via des fonctions comme `OwnBlitter()` ou en prenant le contrôle total du système avec `Forbid()`/`Permit()`. Pour des programmes simples ou des démos qui prennent tout l'écran, on peut souvent ignorer cette étape, mais c'est une mauvaise pratique pour des applications Workbench.

Le Copper : Le Co-processeur Graphique

Le Copper (ou coprocesseur) est une petite merveille. C'est un processeur simple qui s'exécute en parallèle du CPU principal. Sa seule tâche est de lire une liste d'instructions (une "Copper list") et de les exécuter en parfaite synchronisation avec le balayage de l'écran par le faisceau d'électrons .

Le Copper ne connaît que trois instructions :

- **`WAIT` :** Attend que le faisceau atteigne une position (X, Y) spécifique de l'écran.
- **`MOVE` :** Écrit une valeur dans un des registres des puces custom.
- **`SKIP` :** Saute l'instruction suivante si le faisceau a déjà dépassé une certaine position.

Grâce à cela, le Copper peut changer les couleurs, les palettes, les pointeurs de bitplanes, et bien d'autres choses, à la volée, au milieu d'une image, sans aucune intervention du CPU ! C'est ainsi que les démos Amiga pouvaient afficher des centaines de couleurs alors que le mode graphique n'en supportait officiellement que 32.

Créer une Copper list en C implique de créer un tableau de `UWORD` (entiers 16 bits non signés) en Chip RAM et de le remplir avec les instructions `MOVE` et `WAIT` encodées.

```
/*
 * Copper List — Dégradé de gris sur le fond
 *
 * FORMAT DES INSTRUCTIONS COPPER :
 *  MOVE : IR1 = adresse du registre custom (bit 0 = 0)
 *         IR2 = valeur à écrire
 *  WAIT : IR1 = (vpos[7:0] << 8) | (hpos[7:1] << 1) | 0x0001  (bit 0 = 1)
 *         IR2 = 0xFFFE = masque (comparer tous les bits V et H,
 *               ne pas attendre le Blitter)
 *
 * FORMAT DES COULEURS AMIGA (registres COLOR00-COLOR31) :
 *  Format 12 bits : 0x0RGB — chaque canal (R, G, B) = 4 bits → valeurs 0..15.
 *  Pour une couleur grise : R = G = B. Ex : gris moyen = 0x0777.
 *
 * La Copper List DOIT résider en Chip RAM (__chip sous SAS/C).
 */
#include <hardware/custom.h>
#include <hardware/dmabits.h>

/* 32 lignes × (2 mots WAIT + 2 mots MOVE) + 2 mots de terminateur */
#define GRADIENT_LINES    32
#define COPPER_LIST_SIZE  (GRADIENT_LINES * 4 + 2)

/* Première ligne visible en PAL : 0x2C = 44 */
#define FIRST_LINE  0x2C

/* Doit être en Chip RAM ! */
UWORD __chip copperList[COPPER_LIST_SIZE];

extern struct Custom custom;

void createGradientCopperList(void)
{
    int    i;
    UWORD *cptr = copperList;

    for (i = 0; i < GRADIENT_LINES; i++)
    {
        UBYTE vpos = (UBYTE)(FIRST_LINE + i);

        /* --- Instruction WAIT ---
         * Attendre le début de la ligne vpos (position horizontale = 0).
         * IR1 : (vpos << 8) | 0x01  →  bit 0 = 1 signale une instruction WAIT.
         * IR2 : 0xFFFE              →  comparer tous les bits, ignorer le Blitter.
         */
        *cptr++ = ((UWORD)vpos << 8) | 0x0001;
        *cptr++ = 0xFFFE;

        /* --- Instruction MOVE vers COLOR00 (offset 0x0180) ---
         * IR1 : 0x0180  →  bit 0 = 0 signale une instruction MOVE.
         * IR2 : couleur grise.
         *
         * ATTENTION : chaque canal de couleur Amiga fait 4 bits (0..15).
         * i va de 0 à 31 : on divise par 2 (i >> 1) pour rester dans [0, 15].
         * Sans cette division, les bits de poids fort débordent dans les canaux
         * voisins et produisent des couleurs incorrectes.
         */
        {
            UWORD canal = (UWORD)(i >> 1);              /* 0..15 */
            UWORD grey  = (canal << 8) | (canal << 4) | canal; /* 0x0RGB gris */
            *cptr++ = 0x0180;   /* MOVE → COLOR00 */
            *cptr++ = grey;
        }
    }
```

```
    /* --- Terminateur de Copper List ---
     * WAIT sur une position impossible (ligne 0xFF, colonne 0xFF)
     * → le Copper ne peut jamais atteindre cette position → arrêt effectif.
     */
    *cptr++ = 0xFFFF;
    *cptr++ = 0xFFFE;
}

void installCopperList(void)
{
    /* Pointer COP1LC vers notre liste en Chip RAM */
    custom.cop1lc = (ULONG)copperList;

    /* Activer le DMA du Copper (DMAF_COPEN = bit 7 de DMACON) */
    custom.dmacon = DMAF_SETCLR | DMAF_COPEN;

    /* Forcer le rechargement immédiat de la Copper List au prochain pixel */
    custom.copjmp1 = 0x0000;
}
```

Le Blitter : Le Copieur de Blocs

Le Blitter (Block Image Transferrer) est un autre co-processeur matériel dédié aux opérations sur la mémoire, principalement en Chip RAM. Il est extrêmement rapide pour :

- Copier des blocs de mémoire (beaucoup plus vite que le CPU).
- Dessiner des lignes.
- Remplir des surfaces (area fill).
- Effectuer des opérations logiques sur jusqu'à 3 sources de données (A, B, C) pour produire un résultat (D). C'est la base des "bobs" (Blitter Objects), des sprites logiciels.

Utiliser le Blitter implique de configurer une vingtaine de registres (pointeurs vers les sources et la destination, tailles, modulos, etc.) puis de lancer l'opération. Il faut ensuite attendre que le Blitter ait terminé son travail.

```
/*
 * BLITTER — Copie de blocs en Chip RAM
 *
 * CORRECTION waitBlitter() :
 *   Le registre DMACONR ($DFF002) indique l'état du DMA en général,
 *   mais le bit "Blitter busy" se lit dans DMACONR bit 14 (DMAF_BLTDONE).
 *   Cependant, la méthode CORRECTE et portable est de lire le registre
 *   ADKCONR ($DFF010) bit 14 (ADKF_BLTDONE n'existe pas) — en réalité
 *   le bit "Blitter busy" est le bit 14 de DMACONR, mais il faut lire
 *   dmaconr (en lecture) et non dmacon (en écriture).
 *   Sur Amiga OCS/ECS la convention correcte est :
 *     while (custom.dmaconr & (1 << 14)) {}   ← BBUSY = bit 14 de DMACONR
 *   La constante DMAF_BLTDONE n'existe pas dans hardware/dmabits.h standard.
 *   On utilise le masque numérique ou la constante BBUSY de hardware/blit.h.
 *
 * BLTCON0 — Minterm D = A (simple copie sans masque) :
 *   Bits [11:8] = USE_A, USE_B, USE_C, USE_D → on n'utilise que A et D : 0x09xx
 *   Non : pour D = A, on utilise USE_A=1, USE_D=1 et minterm 0xCA = 1100 1010
 *   → BLTCON0 = (USE_A | USE_D) << 8 | 0xCA = 0x09CA
 *   Le commentaire "0x0FCA" est incorrect (0x0F active USE_A+B+C+D mais
 *   sans source B ni C ce n'est pas nécessaire — la valeur correcte est 0x09CA).
 */
#include <hardware/custom.h>
#include <hardware/blit.h>   /* Pour les masques BLTCON0 */

extern struct Custom custom;

/* Attendre que le Blitter ait terminé son opération précédente.
 * DMACONR bit 14 = BBUSY (Blitter BUSy) — 1 = occupé, 0 = libre.
 * On DOIT appeler cette fonction avant de configurer le Blitter.
 * Ne jamais modifier les registres du Blitter pendant qu'il est actif ! */
void waitBlitter(void)
{
    /* Lire DMACONR ($DFF002) — registre en lecture seule.
     * Bit 14 = BBUSY : le Blitter est occupé tant que ce bit est à 1. */
    while (custom.dmaconr & (1 << 14)) { /* attendre */ }
}

/*
 * Copie un rectangle de 64×64 pixels d'une source vers une destination,
 * dans un écran de 320 pixels de large (40 octets par ligne, 1 bitplane).
 *
 * source : adresse du premier pixel source en Chip RAM
 * dest   : adresse du premier pixel destination en Chip RAM
 */
void blitRectangle(APTR source, APTR dest)
{
    waitBlitter(); /* Toujours attendre avant de toucher au Blitter */

    /* BLTCON0 — Registre de contrôle 0 du Blitter :
     *   Bits [15:12] : SHIFT de la source A (0 = pas de décalage)
     *   Bits [11:8]  : USE_A, USE_B, USE_C, USE_D (quelles sources activer)
     *   Bits [7:0]   : minterm (opération logique entre sources)
     *
     * Pour une copie simple D = A (sans masque, sans source B ou C) :
     *   USE_A = 1, USE_D = 1, USE_B = 0, USE_C = 0 → bits [11:8] = 0x9
     *   Minterm 0xCA = 1100 1010 → D = A (résultat = source A directe)
     *   → BLTCON0 = 0x09CA
     */
    custom.bltcon0 = 0x09CA;
    custom.bltcon1 = 0x0000; /* Pas de fill, pas de line mode */

    /* Masques de premier et dernier mot — 0xFFFF = pas de masquage */
    custom.bltafwm = 0xFFFF;
```

```
    custom.bltalwm = 0xFFFF;

    /* Pointeurs vers la source A et la destination D (Chip RAM obligatoire) */
    custom.bltapt = source;
    custom.bltdpt = dest;

    /* Modulo : nombre d'OCTETS à ajouter à l'adresse après chaque ligne.
     * Formule : modulo = (largeur_totale_en_octets) - (largeur_blit_en_octets)
     * Écran   : 320 px → 320/8 = 40 octets par ligne
     * Blit    : 64 px  →  64/8 =  8 octets par ligne (= 4 mots de 16 bits)
     * Modulo  : 40 - 8 = 32 octets
     */
    custom.bltamod = 40 - (64 / 8); /* 32 octets */
    custom.bltdmod = 40 - (64 / 8); /* 32 octets */

    /* BLTSIZE — Lance le Blitter et définit la taille :
     *   Bits [15:6] = hauteur en lignes (64 lignes → 64 << 6)
     *   Bits [5:0]  = largeur en mots de 16 bits (64px / 16 = 4 mots)
     * L'écriture dans BLTSIZE démarre immédiatement le Blitter. */
    custom.bltsize = (64 << 6) | 4;
}
```

Le Son avec Paula

La puce Paula gère le son et les entrées/sorties. Elle dispose de 4 canaux sonores 8 bits DMA. Cela signifie que vous pouvez lui donner un pointeur vers un échantillon sonore (un "sample") en Chip RAM, sa longueur, et Paula le jouera automatiquement sans intervention du CPU .

On peut contrôler Paula directement via ses registres (`AUDxLCH`, `AUDxLEN`, `AUDxPER`, `AUDxVOL`) ou, de manière plus portable et coopérative, via l'`audio.device`.

```
/*
 * Son avec Paula — accès direct aux registres (pour programmes standalone)
 *
 * FORMAT DES SAMPLES :
 *   Paula utilise des échantillons 8 bits SIGNÉS (valeurs -128 à +127).
 *   Le format est parfois appelé PCM 8-bit signed ou .raw.
 *
 * FORMULE DE LA PÉRIODE (registre ac_per) :
 *   Fréquence (Hz) = Horloge_PAL / Période
 *   Horloge PAL    = 3 546 895 Hz
 *   Exemples :
 *     Période 424 → 3546895 / 424 ≈  8363 Hz  (fréquence standard des samples MOD)
 *     Période 214 → 3546895 / 214 ≈ 16574 Hz
 *     Période 120 → 3546895 / 120 ≈ 29557 Hz  (aigu)
 *   Plus la valeur est FAIBLE, plus le son est AIGU.
 *   Valeur minimale recommandée : 124 (limite hardware Paula).
 *
 * ORDRE D'INITIALISATION (obligatoire) :
 *   1. Désactiver le DMA du canal avant toute modification des registres.
 *   2. Configurer ac_ptr, ac_len, ac_per, ac_vol.
 *   3. Activer le DMA → Paula commence à lire le sample.
 */
#include <hardware/custom.h>
#include <hardware/dmabits.h>

extern struct Custom custom;

/* Sample en Chip RAM — OBLIGATOIRE pour que Paula puisse y accéder via DMA.
 * La taille doit être un multiple de 2 (Paula lit par mots de 16 bits). */
BYTE __chip my_sound_sample[1000]; /* BYTE = 8 bits signé, comme attendu par Paula */
/* ... (charger le contenu du sample ici, ex. depuis un fichier avec dos.library) */

void playSound(void)
{
    /* Étape 1 : désactiver le DMA audio du canal 0 AVANT de modifier les registres.
     * (écriture dans DMACON sans DMAF_SETCLR = désactivation des bits spécifiés) */
    custom.dmacon = DMAF_AUD0;

    /* Étape 2 : configurer les registres du canal 0 */
    custom.aud[0].ac_ptr = (UWORD *)my_sound_sample; /* Adresse du sample (Chip RAM)
*/
    custom.aud[0].ac_len = 1000 / 2;                /* Longueur en MOTS (pas en
octets) */
    custom.aud[0].ac_vol = 64;                       /* Volume : 0 (silence) à 64
(max) */

    /* Période : détermine la fréquence de lecture.
     * 424 ≈ 8363 Hz en PAL, fréquence standard des samples de musique MOD.
     * Valeur 120 est trop élevée pour du son de qualité — utiliser 424 par défaut.
*/
    custom.aud[0].ac_per = 424; /* ≈ 8363 Hz en PAL */

    /* Étape 3 : activer le DMA du canal 0 → Paula démarre la lecture */
    custom.dmacon = DMAF_SETCLR | DMAF_AUD0;
}
```

Lecture des Entrées (Joystick/Souris)

Les ports joystick/souris sont lus via les registres `JOY0DAT` (`$DFF00A`) pour le port gauche (port 0, souris) et `JOY1DAT` (`$DFF00C`) pour le port droit (port 1, joystick Player 1). Ces registres contiennent des **compteurs quadrature encodés en code de Gray à 2 bits**. Le décodage des

directions nécessite une opération XOR, et non une simple lecture de bits.

```
/*
 * Lecture correcte du joystick Amiga — Port 1 (port droit = Player 1)
 *
 * STRUCTURE DU REGISTRE JOY1DAT (même structure pour JOY0DAT) :
 *   Bits [15:8] : compteur Y (vertical)   — bit 9 = poids fort, bit 8 = poids faible
 *   Bits  [7:0] : compteur X (horizontal) — bit 1 = poids fort, bit 0 = poids faible
 *
 * DÉCODAGE PAR CODE DE GRAY (formules correctes) :
 *   Haut   : XOR(bit8,  bit9) = (joy >> 8) ^ (joy >> 9)  → masqué à 1
 *   Bas    : bit9 seul        = (joy >> 9)                → masqué à 1
 *   Droite : bit1 seul        = (joy >> 1)                → masqué à 1
 *   Gauche : XOR(bit0, bit1)  = joy ^ (joy >> 1)          → masqué à 1
 *
 * BOUTON FIRE :
 *   Le bouton fire n'est PAS dans JOY1DAT.
 *   Il se lit dans le registre CIA-A PRA à l'adresse $BFE001 :
 *     Bit 7 = /FIR1 (port 1, actif BAS)  : fire = !(CIAA_PRA & (1<<7))
 *     Bit 6 = /FIR0 (port 0, actif BAS)  : fire = !(CIAA_PRA & (1<<6))
 */
#include <hardware/custom.h>
#include <exec/types.h>

extern struct Custom custom;

/* Adresse directe du registre CIA-A PRA */
#define CIAA_PRA  (*(volatile UBYTE *)0xBFE001)

void checkJoystickPort1(void)
{
    UWORD joy = custom.joy1dat;   /* Lire le registre JOY1DAT */

    /* Axe Y — décodage code de Gray sur les bits [9:8] */
    BOOL up   = (BOOL)(((joy >> 8) ^ (joy >> 9)) & 1);
    BOOL down = (BOOL)((joy >> 9) & 1);

    /* Axe X — décodage code de Gray sur les bits [1:0] */
    BOOL right = (BOOL)((joy >> 1) & 1);
    BOOL left  = (BOOL)((joy ^ (joy >> 1)) & 1);

    /* Bouton Fire : bit 7 du CIA-A PRA, logique ACTIVE BAS (0 = appuyé) */
    BOOL fire = (BOOL)(!(CIAA_PRA & (1 << 7)));

    if (up)    Printf("Haut\n");
    if (down)  Printf("Bas\n");
    if (left)  Printf("Gauche\n");
    if (right) Printf("Droite\n");
    if (fire)  Printf("Feu !\n");
}
```

Cette section n'a fait qu'effleurer la surface de la programmation matérielle. C'est un monde fascinant qui demande beaucoup de pratique et de lecture des manuels de référence (comme le Amiga Hardware Reference Manual). Pour notre jeu de quiz, nous nous appuierons principalement sur les bibliothèques de plus haut niveau comme Intuition et GadTools.

5. Projet Pratique : Création d'un Jeu de Quiz

Mettons en pratique ce que nous avons appris ! Nous allons créer un jeu de quiz simple, mais complet. Il lira des questions depuis un fichier, les affichera à l'écran avec quatre réponses possibles sous forme de boutons, et tiendra le score. Ce projet nous forcera à utiliser `exec.library`, `dos.library`, `intuition.library` et `gadtools.library`.

5.1. Conception et Structure du Jeu

Logique du jeu (Game Loop)

Le déroulement du jeu sera le suivant :

38. Initialisation : ouvrir les bibliothèques, créer l'interface (écran, fenêtre, gadgets).
39. Charger les questions depuis un fichier texte.
40. Boucle principale :

    - Afficher la question et les réponses actuelles.
    - Attendre une action de l'utilisateur (clic sur un bouton de réponse).
    - Vérifier si la réponse est correcte.
    - Mettre à jour le score et afficher un feedback (Correct/Incorrect).
    - Passer à la question suivante.

41. Fin du jeu : afficher le score final.
42. Nettoyage : fermer l'interface, libérer la mémoire, fermer les bibliothèques.

Structure des données

Nous avons besoin d'une structure pour stocker chaque question.

```
#define MAX_QUESTION_LEN 256
#define MAX_ANSWER_LEN 64
#define NUM_ANSWERS 4

typedef struct {
    char question[MAX_QUESTION_LEN];
    char answers[NUM_ANSWERS][MAX_ANSWER_LEN];
    int correct_answer_index; // 0 à 3
} QuizQuestion;
```

Nous lirons les questions depuis un fichier texte simple. Chaque question sera un bloc de 6 lignes :

```
La capitale de la France ?
Paris
Londres
Berlin
Rome
0
---
Quel processeur équipe l'Amiga 500 ?
Intel 8086
Zilog Z80
Motorola 68000
MOS 6502
2
---
```

La première ligne est la question, les quatre suivantes sont les réponses, la sixième est l'index de la bonne réponse (de 0 à 3), et un séparateur `---` indique la fin d'une question.

5.2. L'Interface Graphique avec Intuition et GadTools

Créer une interface graphique sur Amiga implique d'ouvrir un "Screen" (un espace de dessin avec sa propre palette de couleurs) et une "Window" à l'intérieur de cet écran. Les éléments interactifs comme les boutons sont appelés "Gadgets". La `gadtools.library` simplifie grandement la création de gadgets standards.

Ouverture de l'écran et de la fenêtre

On utilise `OpenScreen()` et `OpenWindow()` de `intuition.library`. Ces fonctions prennent en paramètre des structures `NewScreen` et `NewWindow` où l'on définit toutes les propriétés (taille, position, titre, couleurs, types de gadgets système, etc.).

Création des gadgets

Notre interface aura besoin de :

- Un gadget texte pour afficher la question.
- Quatre gadgets boutons pour les réponses.
- Un gadget texte pour le score.
- Un gadget texte pour le feedback (Correct/Incorrect).

Avec `gadtools.library`, on crée un contexte visuel (`VisualInfo`), puis on utilise `CreateGadget()` pour chaque gadget. C'est plus simple que de créer les structures de gadgets manuellement.

5.3. Gestion des Fichiers avec la dos.library

Pour lire notre fichier `questions.txt`, nous utiliserons les fonctions de la `dos.library` :

- **`Open()` :** Pour ouvrir le fichier en mode lecture. Prend le nom du fichier et le mode (`MODE_OLDFILE`). Renvoie un "file handle".
- **`FGets()` :** Pour lire une ligne du fichier dans un tampon. Similaire à `fgets()` en C standard.
- **`Close()` :** Pour fermer le fichier. Essentiel pour libérer le "lock" sur le fichier.

5.4. Implémentation de la Logique du Jeu

La boucle d'événements

Le cœur d'une application GUI Amiga est la boucle d'événements. Le programme se met en attente d'un message sur le port de communication de sa fenêtre (le fameux IDCMP).

```
ULONG signal, window_signal_mask;
struct IntuiMessage *message;
BOOL running = TRUE;

// Obtenir le masque de signal pour le port de la fenêtre
window_signal_mask = 1L << MyWindow->UserPort->mp_SigBit;

while(running)
{
    // Attendre un signal (un message est arrivé)
    signal = Wait(window_signal_mask);

    if (signal & window_signal_mask)
    {
        // Récupérer tous les messages en attente
        while (message = (struct IntuiMessage *)GetMsg(MyWindow->UserPort))
        {
            // Traiter le message
            switch (message->Class)
            {
                case IDCMP_CLOSEWINDOW:
                    running = FALSE;
                    break;

                case IDCMP_GADGETUP:
                    // Un bouton a été cliqué !
                    // L'ID du gadget est dans message->Code
                    handle_answer(message->Code);
                    break;
            }
            // Indiquer qu'on a traité le message
            ReplyMsg((struct Message *)message);
        }
    }
}
```

La fonction `handle_answer()` contiendra la logique pour vérifier la réponse, mettre à jour le score et passer à la question suivante.

5.5. Code Source Complet et Commenté

Voici une implémentation possible de notre jeu de quiz. Pour une meilleure organisation, nous pourrions le diviser en plusieurs fichiers (`main.c`, `gui.c`, `quiz_logic.c`), mais pour la simplicité de ce guide, nous allons tout mettre dans `main.c`.

> Ce code est long et détaillé. Prenez le temps de lire les commentaires pour comprendre le rôle de chaque partie.

```
/*
 * ============================================================================
 *  Amiga Quiz Game - main.c
 *
 *  Un jeu de quiz simple pour AmigaOS 1.3+
 *  Développé en C avec SAS/C 6.58
 *
 *  Ce code démontre :
 *  - L'ouverture et la fermeture des bibliothèques AmigaOS.
 *  - La création d'une interface graphique avec Intuition et GadTools.
 *  - La gestion des événements via un IDCMP.
 *  - La lecture de fichiers avec la dos.library.
 *  - L'affichage de texte.
 * ============================================================================
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <intuition/intuition.h>
#include <dos/dos.h>
#include <graphics/gfx.h>

// Fichiers de prototypes pour SAS/C
#include <proto/exec.h>
#include <proto/dos.h>
#include <proto/intuition.h>
#include <proto/gadtools.h>
#include <proto/graphics.h>

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

// --- Constantes et Structures ---

#define MAX_QUESTIONS 50
#define MAX_QUESTION_LEN 256
#define MAX_ANSWER_LEN 64
#define NUM_ANSWERS 4

// Structure pour une question
typedef struct {
    char question[MAX_QUESTION_LEN];
    char answers[NUM_ANSWERS][MAX_ANSWER_LEN];
    int correct_answer_index;
} QuizQuestion;

// Structure pour notre interface graphique
struct GUI {
    struct Screen *screen;
    struct Window *window;
    struct Gadget *gadgets[6]; // 1 question, 4 réponses, 1 score
    struct VisualInfo *visual_info;
};

// IDs pour nos gadgets
enum {
    GID_QUESTION_TEXT = 0,
    GID_ANSWER_1,
    GID_ANSWER_2,
    GID_ANSWER_3,
    GID_ANSWER_4,
    GID_SCORE_TEXT
};
```

```
// --- Variables Globales ---

// Pointeur de base des bibliothèques
struct Library *IntuitionBase = NULL;
struct Library *GfxBase = NULL;
struct Library *GadToolsBase = NULL;
struct Library *DOSBase = NULL;

// Données du jeu
QuizQuestion questions[MAX_QUESTIONS];
int num_questions = 0;
int current_question_index = 0;
int score = 0;

// --- Fonctions ---

// Fonction de nettoyage
void cleanup(struct GUI *gui) {
    if (gui) {
        if (gui->window) {
            // Libérer les gadgets
            if (gui->gadgets[0]) {
                FreeGadgets(gui->gadgets[0]);
            }
            CloseWindow(gui->window);
        }
        if (gui->visual_info) {
            FreeVisualInfo(gui->visual_info);
        }
        if (gui->screen) {
            CloseScreen(gui->screen);
        }
    }

    // Fermer les bibliothèques
    if (GadToolsBase) CloseLibrary(GadToolsBase);
    if (GfxBase) CloseLibrary(GfxBase);
    if (IntuitionBase) CloseLibrary(IntuitionBase);
    if (DOSBase) CloseLibrary(DOSBase);
}

// Initialisation de l'interface graphique
BOOL init_gui(struct GUI *gui) {
    /*
     * CORRECTION : l'écran était déclaré avec un commentaire "Lores"
     * mais avec la résolution 640x200 et le flag HIRES, ce qui est contradictoire.
     *
     * RÈGLE : la résolution et le flag ViewModes doivent être cohérents :
     *   640 pixels de large → flag HIRES obligatoire
     *   320 pixels de large → pas de flag (ou 0)
     *
     * Ici on choisit HIRES (640×200, 2 bitplanes = 4 couleurs),
     * adapté à une interface avec du texte fin.
     */
    struct NewScreen ns = {
        0, 0, 640, 200, 2, /* HIRES, 2 bitplanes = 4 couleurs */
        0, 1,              /* DetailPen (avant-plan), BlockPen (arrière-plan) */
        HIRES,             /* ViewModes : HIRES — cohérent avec la largeur 640 */
        CUSTOMSCREEN,
        NULL,
        "Amiga Quiz !",
        NULL, NULL
    };
```

```
    gui->screen = OpenScreen(&ns);
    if (!gui->screen) {
        printf("Impossible d'ouvrir l'ecran\n");
        return FALSE;
    }

    // Obtenir les informations visuelles pour GadTools
    gui->visual_info = GetVisualInfo(gui->screen, TAG_DONE);
    if (!gui->visual_info) {
        printf("Impossible d'obtenir VisualInfo\n");
        return FALSE;
    }

    // Création des gadgets
    struct NewGadget ng;
    struct Gadget *first_gadget = NULL;
    int top = 15;

    // Gadget pour la question
    ng.ng_VisualInfo = gui->visual_info;
    ng.ng_LeftEdge = 20;
    ng.ng_TopEdge = top;
    ng.ng_Width = 600;
    ng.ng_Height = 30;
    ng.ng_GadgetText = "Chargement des questions...";
    ng.ng_GadgetID = GID_QUESTION_TEXT;
    ng.ng_Flags = NG_HIGHLABEL;
    ng.ng_TextAttr = NULL; // Police par défaut
    ng.ng_UserData = NULL;
    first_gadget = CreateGadget(TEXT_KIND, first_gadget, &ng, TAG_DONE);
    if (!first_gadget) return FALSE;
    gui->gadgets[GID_QUESTION_TEXT] = first_gadget;
    top += 40;

    // Gadgets pour les réponses
    // CORRECTION : on chaîne les gadgets via un pointeur 'prev_gadget' initialisé
    // au premier gadget (question), évitant de passer un pointeur non initialisé.
    {
        struct Gadget *prev_gadget = first_gadget;
        int i;
        for (i = 0; i < NUM_ANSWERS; i++) {
            ng.ng_LeftEdge = 40;
            ng.ng_TopEdge = top;
            ng.ng_Width = 560;
            ng.ng_Height = 15;
            ng.ng_GadgetText = "Reponse";
            ng.ng_GadgetID = GID_ANSWER_1 + i;
            ng.ng_Flags = 0;
            prev_gadget = CreateGadget(BUTTON_KIND, prev_gadget, &ng, TAG_DONE);
            if (!prev_gadget) return FALSE;
            gui->gadgets[GID_ANSWER_1 + i] = prev_gadget;
            top += 20;
        }
    }
    top += 10;

    // Gadget pour le score
    ng.ng_LeftEdge = 20;
    ng.ng_TopEdge = top;
    ng.ng_Width = 600;
    ng.ng_Height = 15;
    ng.ng_GadgetText = "Score: 0";
    ng.ng_GadgetID = GID_SCORE_TEXT;
```

```
    ng.ng_Flags = NG_HIGHLABEL;
    gui->gadgets[GID_SCORE_TEXT] = CreateGadget(TEXT_KIND,
gui->gadgets[GID_ANSWER_4], &ng, TAG_DONE);
    if (!gui->gadgets[GID_SCORE_TEXT]) return FALSE;

    // Définition de la fenêtre
    struct NewWindow nw = {
        0, 0, 640, 200,
        0, 1, // Couleurs
        IDCMP_CLOSEWINDOW | IDCMP_GADGETUP, // Événements à écouter
        WFLG_DRAGBAR | WFLG_DEPTHGADGET | WFLG_CLOSEGADGET | WFLG_ACTIVATE |
WFLG_SMART_REFRESH,
        first_gadget, // Premier gadget de la liste
        NULL,
        "Amiga Quiz v1.0",
        gui->screen, // Attachée à notre écran
        NULL, 0, 0, 0, 0,
        CUSTOMSCREEN
    };

    gui->window = OpenWindow(&nw);
    if (!gui->window) {
        printf("Impossible d'ouvrir la fenetre\n");
        return FALSE;
    }

    // Attacher les gadgets à la fenêtre
    AddGList(gui->window, first_gadget, -1, -1, NULL);
    RefreshGList(first_gadget, gui->window, NULL, -1);

    return TRUE;
}

// Chargement des questions depuis le fichier
BOOL load_questions(const char *filename) {
    BPTR file_handle;
    char buffer[MAX_QUESTION_LEN];
    int line_in_block = 0;

    file_handle = Open(filename, MODE_OLDFILE);
    if (!file_handle) {
        printf("Impossible d'ouvrir le fichier de questions: %s\n", filename);
        return FALSE;
    }

    num_questions = 0;
    while (FGets(file_handle, buffer, sizeof(buffer)) && num_questions <
MAX_QUESTIONS) {
        // Enlever le retour à la ligne
        buffer[strlen(buffer) - 1] = '\0';

        switch (line_in_block) {
            case 0: // Question
                strcpy(questions[num_questions].question, buffer);
                break;
            case 1: // Réponse 1
            case 2: // Réponse 2
            case 3: // Réponse 3
            case 4: // Réponse 4
                strcpy(questions[num_questions].answers[line_in_block - 1], buffer);
                break;
            case 5: // Index bonne réponse
                questions[num_questions].correct_answer_index = atoi(buffer);
                break;
```

```
            case 6: // Séparateur "---"
                num_questions++;
                line_in_block = -1; // Sera incrémenté à 0
                break;
        }
        line_in_block++;
    }

    Close(file_handle);
    return (num_questions > 0);
}

// Afficher la question actuelle
void display_question(struct GUI *gui) {
    if (current_question_index >= num_questions) {
        // Fin du jeu
        GT_SetGadgetAttrs(gui->gadgets[GID_QUESTION_TEXT], gui->window, NULL,
            GTTX_Text, "Quiz termine !", TAG_DONE);
        int i;
        for (i = 0; i < NUM_ANSWERS; i++) {
            GT_SetGadgetAttrs(gui->gadgets[GID_ANSWER_1 + i], gui->window, NULL,
                GTTX_Text, "", TAG_DONE);
        }
        return;
    }

    QuizQuestion *q = &questions[current_question_index];

    // Mettre à jour le texte des gadgets
    GT_SetGadgetAttrs(gui->gadgets[GID_QUESTION_TEXT], gui->window, NULL,
        GTTX_Text, q->question, TAG_DONE);

    int i;
    for (i = 0; i < NUM_ANSWERS; i++) {
        GT_SetGadgetAttrs(gui->gadgets[GID_ANSWER_1 + i], gui->window, NULL,
            GTTX_Text, q->answers[i], TAG_DONE);
    }
}

// Gérer la réponse de l'utilisateur
void handle_answer(struct GUI *gui, UWORD gadget_id) {
    if (current_question_index >= num_questions) return;

    int selected_answer_index = gadget_id - GID_ANSWER_1;
    QuizQuestion *q = &questions[current_question_index];
    char score_text[32];

    if (selected_answer_index == q->correct_answer_index) {
        score++;
        // Afficher un feedback visuel (par ex. flasher l'écran)
        ScreenFlash(gui->screen, 0);
    }

    // Mettre à jour le score
    sprintf(score_text, "Score: %d / %d", score, num_questions);
    GT_SetGadgetAttrs(gui->gadgets[GID_SCORE_TEXT], gui->window, NULL,
        GTTX_Text, score_text, TAG_DONE);

    // Passer à la question suivante
    current_question_index++;
    display_question(gui);
}

// --- Fonction Principale ---
```

```
int main(void) {
    struct GUI gui = {0};
    BOOL running = TRUE;

    /*
     * 1. Ouvrir les bibliothèques dans l'ordre des dépendances.
     *    Chaque bibliothèque est vérifiée immédiatement après son ouverture.
     *    En cas d'échec, on appelle cleanup() qui ferme ce qui a déjà été ouvert.
     */
    DOSBase = OpenLibrary("dos.library", 34);
    if (!DOSBase) {
        printf("Impossible d'ouvrir dos.library\n");
        return 20;
    }

    GfxBase = OpenLibrary("graphics.library", 33);
    if (!GfxBase) {
        Printf("Impossible d'ouvrir graphics.library\n");
        cleanup(&gui);
        return 20;
    }

    IntuitionBase = OpenLibrary("intuition.library", 33);
    if (!IntuitionBase) {
        Printf("Impossible d'ouvrir intuition.library\n");
        cleanup(&gui);
        return 20;
    }

    /* CORRECTION : GadToolsBase doit être ouvert AVANT tout appel
     * à GetVisualInfo() ou CreateGadget(). La version minimale
     * requise est 37 (Kickstart 2.0) pour gadtools.library. */
    GadToolsBase = OpenLibrary("gadtools.library", 37);
    if (!GadToolsBase) {
        Printf("Impossible d'ouvrir gadtools.library (Kickstart 2.0 requis)\n");
        cleanup(&gui);
        return 20;
    }

    // 2. Initialiser l'interface
    if (!init_gui(&gui)) {
        cleanup(&gui);
        return 20;
    }

    // 3. Charger les questions
    if (!load_questions("questions.txt")) {
        GT_SetGadgetAttrs(gui.gadgets[GID_QUESTION_TEXT], gui.window, NULL,
            GTTX_Text, "Erreur: Fichier 'questions.txt' introuvable ou vide.",
TAG_DONE);
    } else {
        display_question(&gui);
    }

    // 4. Boucle d'événements
    {
        ULONG signals;
        ULONG window_signal_mask = 1L << gui.window->UserPort->mp_SigBit;
        struct IntuiMessage *message;

    while (running) {
        signals = Wait(window_signal_mask);
```

```
        if (signals & window_signal_mask) {
            while (message = (struct IntuiMessage *)GetMsg(gui.window->UserPort)) {
                /*
                 * CORRECTION CRITIQUE : il faut lire les champs du message
                 * AVANT d'appeler ReplyMsg(). Après ReplyMsg(), la structure
                 * IntuiMessage est rendue au système et son contenu peut
                 * être écrasé à tout moment → accès interdit.
                 */
                ULONG  msg_class  = message->Class;
                UWORD  msg_code   = message->Code;
                UWORD  gadget_id  = 0;

                if (msg_class == IDCMP_GADGETUP) {
                    gadget_id = ((struct Gadget *)message->IAddress)->GadgetID;
                }

                /* On peut maintenant rendre le message en sécurité */
                ReplyMsg((struct Message *)message);

                switch (msg_class) {
                    case IDCMP_CLOSEWINDOW:
                        running = FALSE;
                        break;
                    case IDCMP_GADGETUP:
                        if (gadget_id >= GID_ANSWER_1 && gadget_id <= GID_ANSWER_4) {
                            handle_answer(&gui, gadget_id);
                        }
                        break;
                }
            }
        }
    }
    } /* fin du bloc de déclaration des variables de boucle d'événements */

    // 5. Nettoyage
    cleanup(&gui);
    return 0;
}
```

Pour utiliser ce code, vous devez créer un fichier `questions.txt` dans le même dossier que votre exécutable, avec le format décrit précédemment. Compilez avec `mkmk main=main.c` puis `smake`, et lancez `main` depuis le Shell.

6. Alternative Avancée : La Cross-Compilation avec VBCC/GCC

La méthode hybride avec SAS/C est excellente pour sa compatibilité, mais elle requiert de jongler entre VS Code et l'émulateur pour chaque compilation. Les développeurs modernes préfèrent souvent une chaîne d'outils de cross-compilation qui s'intègre mieux à leur IDE.

6.1. Introduction à VBCC

VBCC (Volker Barthelmann C Compiler) est un compilateur C moderne, hautement optimisé et multi-cibles, très populaire pour le développement Amiga. Il s'exécute sur Windows, Linux

ou macOS et produit des exécutables pour le 68k de l'Amiga .

Installation rapide de VBCC sur Windows

La manière la plus simple d'installer un environnement VBCC complet est d'utiliser un package pré-compilé.

43. Téléchargez le package "amiga-dev" depuis le dépôt GitHub de cahirwpz.
44. Extrayez l'archive dans `C:\amiga-dev\`.
45. Ajoutez `C:\amiga-dev\bin\` à votre variable d'environnement `PATH` système.
46. Créez une variable d'environnement système nommée `VBCC` et donnez-lui la valeur `C:\amiga-dev\targets\m68k-amigaos`.

Après un redémarrage de votre terminal, vous devriez pouvoir compiler un fichier C pour Amiga directement depuis la ligne de commande Windows avec la commande `vc`.

6.2. Une Solution Intégrée : L'extension "Amiga C/C++" pour VS Code

Pour une expérience de développement ultime, il existe une extension pour VS Code qui automatise presque tout : **Amiga C/C++ Compile, Debug & Profile** (une fourche de l'extension originale de Bartman/Abyss) . Cette extension est une véritable révolution, transformant VS Code en un IDE Amiga complet et autonome.

Fonctionnalités Clés

Cette extension est un guichet unique pour le développement Amiga, incluant :

- **Chaîne de compilation complète :** Elle embarque un compilateur **GCC** récent (versions 13+ ou même 15+) entièrement configuré pour la cross-compilation vers le processeur m68k de l'Amiga. Plus besoin d'installer et de configurer VBCC ou SAS/C séparément.
- **Intégration transparente avec WinUAE :** L'extension est capable de piloter WinUAE en arrière-plan. Une simple commande lance la compilation, démarre l'émulateur et exécute votre programme.
- **Débogage de niveau professionnel :** C'est sa plus grande force. Vous pouvez placer des points d'arrêt directement dans votre code C/C++ dans VS Code, inspecter les variables, parcourir la pile d'appels, visualiser la mémoire et les registres du CPU, le tout pendant que le code s'exécute dans WinUAE .
- **Outils de profilage avancés :** Elle inclut des outils pour analyser les performances de votre code, comme un profileur d'images (Frame Profiler), un débogueur graphique pour analyser les opérations du Blitter et du Copper, et un profileur de taille pour optimiser la taille de l'exécutable.

- **Assistants intégrés :** L'extension propose des outils d'aide, comme le "Gradient Master" pour créer des dégradés pour le Copper, un "Image Tool" pour convertir des images, et même une "BLTCON Cheat Sheet" pour concevoir des opérations Blitter .
- **Support multi-configuration :** Il est très simple de cibler différents modèles d'Amiga (A500, A1200, A4000) en modifiant une simple ligne dans le fichier de configuration `launch.json`.

Mise en route rapide

Le flux de travail avec cette extension est d'une simplicité déconcertante :

47. Installez l'extension **"Amiga C/C++ Compile, Debug & Profile"** depuis le Marketplace de VS Code.
48. Installez également l'extension **C/C++** de Microsoft si ce n'est pas déjà fait.
49. Créez un dossier de projet vide et ouvrez-le dans VS Code (`File > Open Folder...`).
50. Ouvrez la palette de commandes (`Ctrl+Shift+P`) et tapez **`Amiga: init Project`**. Cela crée un projet de base avec un `main.c`, un `Makefile` et les fichiers de configuration nécessaires.
51. Ouvrez le fichier `.vscode/launch.json` qui vient d'être créé. Localisez la ligne `"kickstart"` et remplacez la valeur par le chemin d'accès complet à votre fichier de ROM Kickstart 1.3.
52. **Appuyez sur F5.** C'est tout. L'extension va automatiquement compiler votre code, lancer WinUAE, monter l'exécutable et le lancer pour vous. Si vous placez un point d'arrêt dans `main.c`, l'exécution s'arrêtera à cet endroit.

Cette approche est de loin la plus productive pour le développement Amiga en 2025. Elle élimine la complexité de la configuration manuelle et offre des outils de débogage puissants qui étaient inimaginables à l'époque, rendant le processus de création plus accessible et beaucoup plus agréable.

7. Tester sur du Matériel Réel

Émuler, c'est bien. Voir son code tourner sur un véritable Amiga 500, c'est encore mieux ! Le défi est de transférer l'exécutable que nous avons compilé de notre PC vers l'Amiga.

7.1. Méthodes de Transfert de Fichiers

Plusieurs méthodes existent, des plus anciennes aux plus modernes.

Méthode 1 : CrossDOS et disquettes PC

L'Amiga, à partir du Workbench 2.x, inclut un système de fichiers appelé CrossDOS qui lui permet de lire et d'écrire des disquettes formatées en MS-DOS (720 Ko). Si vous avez un lecteur de disquette USB sur votre PC, vous pouvez :

53. Formater une disquette 720 Ko en FAT12 sur votre PC.
54. Copier votre exécutable `main` dessus.

55. Sur votre Amiga (avec WB 2.x+), activer le driver `PC0:` dans `Storage/DOSDrivers`.
56. Insérer la disquette. Une icône `PC0:????` devrait apparaître, vous permettant de copier le fichier sur votre disque dur Amiga .

Cette méthode est difficile avec un Amiga 500 de base (Kickstart 1.3) qui ne supporte pas CrossDOS nativement.

Méthode 2 : Câble Série Null-Modem

C'est la méthode classique. Elle nécessite un câble série null-modem (DB9 femelle vers DB25 mâle pour connecter un PC à un Amiga).

57. Connectez le PC et l'Amiga avec le câble.
58. Sur le PC, utilisez un logiciel comme Amiga Explorer (inclus dans Amiga Forever).
59. Sur l'Amiga, vous devez lancer un petit programme serveur (souvent transféré une première fois par disquette).
60. Amiga Explorer vous donnera alors un accès de type explorateur de fichiers à vos disques Amiga, vous permettant de glisser-déposer votre exécutable .

Méthode 3 : Adaptateur Compact Flash / Carte SD

C'est la méthode la plus pratique aujourd'hui. On peut utiliser un adaptateur IDE vers Compact Flash (ou SD) pour remplacer le disque dur d'un Amiga (plus courant sur A600/A1200, mais possible sur A500 avec des extensions).

61. Vous préparez la carte CF/SD sur votre PC en utilisant WinUAE pour y installer le Workbench et vos fichiers.
62. Vous pouvez ensuite monter cette carte CF/SD comme un disque dur dans WinUAE, y copier votre exécutable compilé.
63. Enfin, vous placez la carte CF/SD dans l'adaptateur de votre vrai Amiga et vous démarrez. Votre fichier est là.

Méthode 4 : Le Gotek

Un émulateur de lecteur de disquette Gotek remplace le lecteur de disquette physique de l'Amiga. Il lit des images de disquettes (`.adf`) depuis une clé USB.

64. Sur votre PC, utilisez un logiciel pour créer un fichier ADF vierge.
65. Montez cet ADF dans WinUAE et copiez-y votre exécutable.
66. Copiez ce fichier ADF sur une clé USB.
67. Insérez la clé USB dans le Gotek de votre Amiga et "chargez" la disquette virtuelle.

8. Conclusion et Ressources Supplémentaires

Ce voyage à travers le développement C sur Amiga vous a, je l'espère, donné les clés pour

commencer vos propres projets. Nous avons couvert la mise en place d'un environnement de développement hybride confortable, exploré les concepts fondamentaux de l'AmigaOS et de son matériel, et réalisé un projet concret de bout en bout.

La programmation sur Amiga est un domaine vaste et gratifiant. La clé est la curiosité, la patience et la lecture. N'ayez pas peur d'expérimenter, de faire planter votre émulateur, et de plonger dans la documentation.

Ressources Incontournables

- **Amiga Developer Docs (ADCD) :** La bible. Contient les manuels de référence du matériel (HRM), des bibliothèques (RKM) et des périphériques. Indispensable.
- **English Amiga Board (EAB) :** Le plus grand forum de la communauté Amiga. La section "Coding" est une mine d'or d'informations et d'entraide.
- **Aminet :** La plus grande archive de logiciels pour Amiga. Vous y trouverez des compilateurs, des outils, des exemples de code et des bibliothèques.
- **Dépôt de l'extension vscode-amiga-debug :** Pour ceux qui veulent utiliser l'approche de cross-compilation moderne.
- **Start with C programming on Amiga (Medium) :** Un excellent article qui explore en détail l'approche par cross-compilation avec VBCC.

Le monde du développement Amiga vous est maintenant ouvert. Amusez-vous bien !