Quelques préliminaires :

Le langage C++, comme tous les langages est une succession d'instructions, exécutées du haut vers le bas (très très vite).
Le rôle du compilateur est de transformer le code C++ en une suite d'instructions processeur.

Si vous voulez coder directement en langage processeur (langage machine), cela s'appelle de l'assembleur, compliqué, mais en environ 2,5 fois plus rapide que le C++
(je veux dire, aujourd'hui, on s'en fou un peu avec la vitesse des machines mais pour tout ce qui est instruction qui demande beaucoup de charge CPU il y a moyen de coder rapide même en C++).

Si vous voulez avoir des infos sur l'assembleur et son fonctionnement, demandez moi, j'en ai fait sur calculette (même que gameboy classique) et ordi. Car il existe un assembleur par type de processeur.

Heureusement que les langages compilés comme le C++ marchent pour tous processeurs (évidemment car à la compilation le compilateur va faire un langage machine compatible à la machine utilisée).
Donc si vous voulez que ça marche sur mac ou linux, il faut compiler avec un mac ou linux...
Après quand vous avez 18 ans d'expérience comme moi, vous commenceriez à parler « beauté du code ».
Et effectivement, sans rentrer dans les détails, il n'y a pas d'avenir pour un code « moche »

J'utilise moi-même l'IDE (le programme pour écrire , compiler …) CodeBlocks Mingw et vous donnerez comment utiliser cet IDE ci. Prenez bien la version Mingw !

Dernière chose, si vous êtes pressé de commencer par la partie pour gérer le clavier, la musique et les graphismes. La Partie 8 vous montrera comment installer les librairies nécessaires. Je (evilsourcil@gmail.com) vous fournirez les codes surpuissants à moi pour vous aider dans ces taches. Ensuite, retourné à la Partie 1 ou bien commencez par le début...

Des choses de base à savoir :

Chaque lignes d'instructions C++ se terminent par un ;

Un beau code est au minimum bien indenté (pour cela utilisé la touche aux dessus de MAJ Lock)

```
    L'indentation vous
        sera expliqué
            ultérieurement.
```

Partie 1 : La mémoire

Je commence par la mémoire car dans un programme, le reste n'est que gestion de cette mémoire (ou presque).

Dans ce cours je parle de mémoire RAM et non du disque dur dont je ferais une section plus tard car moins important.
La mémoire RAM donc contenu dans les barrettes de RAM et est une succession d'octet (8 bits soit 8 zéros ou un) qui se suit. C'est donc pour tous types de données la même chose et ces octets se suivent et ont donc une adresse (0 , 1 , 2 , ...etc). On dit adresse mémoire.

Dans un programme, la mémoire peut être utilisée de plusieurs manières.

- Une variable est ce qu'il y a de plus basique et on les utilise tout le temps

Les variables ont un type qui définit ce que l'on peut stocker dedans
Les variables peuvent être déclarées ou on veut dans le code ainsi :
La plupart des variables sont déclaré au début afin d'être utilisé dans le code en dessous (le reste du programme)
Mais certaine vont être déclaré à la volée durant le code et sont généralement utilisés de manière isolé et ne sont pas censé contenir des données importantes pour tout le programme.
Utiliser des variables à la volée de manière intelligente peut accélérer grandement la vitesse d'exécution et rend le code plus beau.
Voici les types utilisables et leur façon d'être déclaré
(tout ce qui suit « // » dans un code C++ est un commentaire qui ne sera pas compilé mais aide la compréhension du code par un humain)(existe aussi /* ...commentaires */)

```
        char ici_le_nom_de_la_variable_1;
// une variable de type « char » du nom de  « ici_le_nom_de_la_variable_1 »
        int ici_le_nom_de_la_variable_2;
        float ici_le_nom_de_la_variable_3;
```

petite liste

char : un nombre entier entre -128 et 127 (codé sur un octet dans la mémoire)
unsigned char : un nombre entier entre 0 et 255 (unsigned rend la variable non signée, c'est-à-dire qu'elle ne peut pas stoker de nombres négatifs)
char vient de caractère car la table ASCII est une association de nombre entre 0 et 255 aux différents caractères textuels du clavier et autre. Aujourd'hui il y a tellement de caractère qu'il faut plus de mémoire par caractère pour tous les coder. (je t'invite à regarder la table ASCII sur internet)
int : un nombre entier entre -beaucoup et +beaucoup (codé sur 4 octets dans la mémoire)
unsigned int : de 0 à +2* plus beaucoup
float : un nombre à virgule flottante (exemple : 15.4823)(codé sur 4 octets dans la mémoire)
Important, les virgules dans un nombre en C++ se font avec le point .
le unsigned float doit exister mais à quoi bon ???
Important, dans un calcul avec des nombres flottants n'omettez jamais la virgule . Car sinon l'ordinateur va calculer un nombre entier (2.14 * 5 n'est pas égal à 2.14 * 5.0)

le type « bool » (de boolean = booléen) ne peut contenir qu'une valeur soit 0 soit 1 (faux ou vrai) mais je calme tout enthousiasme. Un Bool est stocké sur un octet entier. (et pas 8 bool par octet).
Utiliser le type bool plutôt que char n'a pour but que la lisibilité et la beauté du code (ce n'est pas des conneries).

Après on peut faire des opérations simples avec ces variables :
Le + , - , * , / font les opérations respectives plus , moins , fois , divisé

```
X = Y ; // prend Y et le met dans X
// X ne peut être qu'un nom de variable alors que Y peut être un calcul
X++; // Incrémente X de 1 ( X = X + 1 )
X--; // Décrémente X de 1 ( X = X - 1 )
X+=Y; // Augmente X de Y ( X = X + Y )
X-=Y; // Diminue X de Y ( X = X - Y )
X*=Y; // Multiplie X par Y ( X = X * Y )
X/=Y; // Divise X par Y ( X = X / Y )
```

En C, il est possible de créer son propre type de donnée contenant plusieurs types de variables.
C'est des structures. Soit :

```
struct le_nom_que_tu_veux_du_nouveau_type
{
   bool _1; // les noms de variables peuvent être ce qu'on veut mais attention aux caractères interdits
   int _2;
   float _3;
};
// après avoir déclaré la structure on peut créer une variable de ce nouveau type. Soit :
le_nom_que_tu_veux_du_nouveau_type Le_nom_de_ma_variable ;
// ensuite dans le code, on peut accéder aux divers éléments de notre variable comme suit :
Le_nom_de_ma_variable._1 ;
Le_nom_de_ma_variable._2 ;
Le_nom_de_ma_variable._3 ; // Le caractère qui fait ça est donc le « . »
```

Je veux vous parler de quelque chose et je ne sais pas ou le mettre alors voilà
si dans un code on met la ligne

```
#define bidule 200
```

ou

```
#define machin 0.0753
```

après tous les « bidule » seront remplacés à la compilation par 200 et « machin » par 0.0753
ceci n'est pas une déclaration de variable. En fait l'ordinateur va faire comme si on avait écrit 200 à la place de bidule. C'est plus rapide d'exécution et très bon pour tout ce qui est paramètre qui ne change pas dans le programme.
Exemple :

```
#define Resolution_X 1024
#define Resolution_Y 768
```

Maintenant nous allons parler des tableaux. (Ce n'est pas la fin y reste beaucoup après)
Dans la RAM toute mémoire qui stocke plusieurs data (données) et non une seule est un buffer (tampon)
Les tableaux permettent de stocker plusieurs données à la fois d'un type bien précis.
Exemple : le pourcentage de vie de 200 ennemis
Il y a deux façons de déclarer les tableaux et c'est là que ça devient compliqué.

1. La déclaration statique

```
int vie_des_ennemis[ 200 ];
```

ou bien

```
#define nombre_d_ennemis 200
int vie_des_ennemis[ nombre_d_ennemis ];
```

et voilà ! Maintenant nous avons 200 variables de vie de type int pour chacun des ennemis

et pour accéder aux différents points de vie des ennemis il suffit d'utiliser « [» et «] »
comme ceci :

```
vie_des_enemis[ 0 ];
vie_des_enemis[ 43 ];
vie_des_enemis[ 199 ];
```

Il faut juste savoir qu'un tableau de taille 200 a des index de 0 à 199. (de 0 à taille – 1)
C'est toujours le cas pour tous tableaux et de n'importe qu'elle taille.

la déclaration statique est la méthode la plus simple pour définir un tableau, elle est également très rapide d'exécution.
malheureusement cela ne marche que pour les tableaux dont on connais d'avance la taille maximale et aussi, cette taille ne peut varier pondant l'exécution.

2. La déclaration à la volée (ou dynamique)

Pour parler de déclaration dynamique de tableau en C++
Nous devons avant voir les pointeurs (COURAGE !)
Un pointeur se déclare de cette manière

```
int* vie_des_ennemis ;
```

Le type de données ... le nom du tableau
Il y a juste un petit * qui suit le type de données.
Le principe est simple, vie_des_ennemis n'est pas une variable mais un pointeur.
Le pointeur contient une adresse mémoire.
C'est-à-dire un nombre entier positif qui correspond à un index d'un octet dans la RAM (pfou)
On peut faire différentes opérations avec un pointeur.

- Opérations sur l'adresse mémoire

```
vie_des_ennemis++; // le pointeur avance d'un index de int (de 4 octets)
// C'est pour cela qu'on indique le type de donnée pointé par notre pointeur
// Comme cela il peut savoir quel genre de donné il pointe et aide pour cette opération
vie_des_ennemis--; // le pointeur recule d'un index du type
vie_des_ennemis+= X ; // avance de X index
vie_des_ennemis-= X ; // recule de X index
```

- Opérations sur ce que pointe le pointeur (la valeur de ce qui est stocké dans la RAM à l'adresse mémoire contenu dans le pointeur)

Ces opérations agissent sur la data dont l'adresse est pointée par le pointeur

```
*vie_des_ennemis = Y ; // prend Y et le met dans là où pointe  vie_des_ennemis
*vie_des_ennemis++; // Incrémente là où pointe vie_des_ennemis
*vie_des_ennemis--; // Décrémente là où pointe vie_des_ennemis
*vie_des_ennemis+=Y; // Augmente là où pointe vie_des_ennemis de Y
*vie_des_ennemis-=Y; // Diminue là où pointe vie_des_ennemis de Y
*vie_des_ennemis*=Y; // Multiplie là où pointe vie_des_ennemis par Y
*vie_des_ennemis/=Y; // Divise là où pointe vie_des_ennemis par Y
```

En fait il suffit juste d'ajouter * avant le nom du pointeur.

Sachez ceci, il est possible de créer des pointeurs de pointeurs (et même je crois récursivement infiniment)
Avant d'y réfléchir et de vous péter le cerveau, laisser ça pour le jour où vous en auraient besoin.

Petite astuce, pour récupérer l'adresse mémoire d'une variable, on dit, l'adresse physique
on ajoute un & avant la variable, soit

```
int* pointeur = &ma_variable // pointeur pointe sur ma_variable
```

et pour les tableaux c'est plus simple :

```
int* pointeur = &mon_tableau[ 0 ] ;
int* pointeur = mon_tableau ; // c'est mieux
```

Parce-que, il ne faut pas le dire, mais les tableaux sont en fait des pointeurs.
Un pointeur pointant sur un tableau peut donc également accéder aux index avec [ ]

```
int* pointeur = mon_tableau ;
pointeur[ 4 ] ;
```

Maintenant, passons à la création dynamique de tableaux.

```
Int* Tableau_de_int ;
Tableau_de_int = new int[ taille_du_tableau ];
```

Le langage C++ nous offre deux merveilleuses instructions (avant c'était plus chiant) :
new qui alloue un certain nombre d'index consécutif d'un type bien précis (ici int)
et delete qui supprime le buffer alloué précédemment. On dit qu'il libère la mémoire

```
delete[] Tableau_de_int; // ce fait sans même connaître la taille du tableau spécifié
```

si vous n'utilisez pas delete après avoir alloué avec new. La RAM utilisée ne sera plus utilisable pour l'ordinateur et tout se passera comme si vous avez de moins en moins de mémoire RAM (cela s'appelle une fuite de mémoire)

L'intérêt de créer un tableau dynamiquement est que cela peut-être fait à n'importe quel endroit du code, par exemple vous récupérez dans un fichier le nombre d'ennemis pour votre jeu, vous pouvez alors créer un tableau de la taille de ce nombre. Vous voulez Ajouter un ennemi à un moment. Plusieurs opérations sur les pointeurs vont vous permettre de le faire.

Une fois le tableau créé dynamiquement, on accède aux index de la même manière qu'avec un tableau statique.

```
        Tableau_de_int[ 0 ];
        Tableau_de_int[ 43 ];
        Tableau_de_int[ 199 ];
```

les index allant toujours de 0 à taille du tableau – 1

Partie 2 : Les conditions et boucles conditionnelles

Aucun rapport avec les conditions, tout votre programme doit se trouver

```
int main(int argc, char **argv)
{
	// Il doit se trouver ici
	return 0 ; // Lorsque l'exécution arrive ici, le programme est terminé
}
```

Peut-être la première ligne peut changer des fois en fonction du système ou autres mais c'est globalement ça. Vous comprendrez mieux cette partie quand vous serez à la partie sur les fonctions.

Autre aucun rapport
Je vais parler maintenant de l'indentation comme je l'avais annoncé au début
En C++ il arrive souvent que l'on ait des choses sous cette forme

```
{
	// avec des fois des { } à l'intérieur d'autres, et cela de nombreuses fois
	// exemple
	{
		{
		}
		{
			{
			}
		}
	}
}
```

L'indentation ci-dessous doit être respectée scrupuleusement c'est-à-dire une indentation par accolade
Si cela n'est pas fait, le programme va compiler mais le code va vite devenir incompréhensible.
ASTUCE : Maj + TAB décrémente et TAB incrémente.
Ceci étant dit, passons aux conditions
Déjà je vais vous donner les différents calculs conditionnels (oui pour l'ordinateur une condition est un calcul.)
Le résultat : FAUX = 0 et VRAI = tout autre valeur (en général 1)
D'ailleurs les mots true écrit dans le code vaut 1 et false 0, cela permet de garder une certaine logique pour séparer ce qui est nombre de ce qui est condition.

X == Y	égal	SI X égal Y retourne 1 (vrai) sinon 0 (faux)
X != Y	différent de	SI X différent de Y retourne 1 (vrai) sinon 0 (faux)
!X	non	Faux retourne vrai vrai retourne faux
X && Y	et	SI X et Y vrai retourne vrai sinon faux
X || Y	ou	SI X ou Y vrai retourne vrai sinon faux

X > Y	Supérieur à	SI X Supérieur à Y retourne vrai sinon faux
X >= Y	Supérieur ou égal à	SI X Supérieur ou égal à Y retourne vrai sinon faux
X < Y	Inférieur à	SI X Inférieur à Y retourne vrai sinon faux
X <= Y	Inférieur ou égal à	SI X Inférieur ou égal à Y retourne vrai sinon faux

Autre chose, les conditions étant des calculs, les parenthèses ( ) peuvent tout changer et sont à réfléchir. Je sais que ça va vous prendre la tête.
Une astuce de lisibilité : séparé les contenus par des espaces. Exemple :

```
if ( ( a == 5 ) || ( b != ( ( c == 10 ) && ( d >= e ) ) ) )
// c'est déjà suffisamment compliqué comme ça... alors ne rendez pas ça illisible en plus.
```

Une fois qu'on conné les calculs conditionnels
L'instruction If va nous permettre d'exécuter quelque chose en fonction d'une condition.

```
If ( condition )
{
        // ceci sera exécuté si condition = vrai
}       // si condition = faux, l'exécution va directement sauter à la fin ici
```

Il existe aussi la version de if pour le cas où on voudrait exécuter quelque chose quand condition vaut faux.

```
If ( condition )
{
        // ceci sera exécuté si condition = vrai
}else{
        // ceci sera exécuté si condition = faux
}
```

Évidemment si vous ne voulez exécuter quelque chose que si condition vaut faux, il vaut mieux faire ainsi :

```
If ( ! condition )
{
        // ceci sera exécuté si condition = faux
}
```

Il existe aussi un système conditionnel élégant :

```
switch( V ) // V la variable à comparer
{
        case 1:
                // à exécuter si V vaut 1
        break;  // à mettre pour couper l'exécution est sortir du switch
        case 2:
                // à exécuter si V vaut 2
        break;
        default:
                // à exécuter si V a une valeur non prévue ci-dessus
        break;
}
```

Voilà pour les conditions qui sont indispensables à tout programme.
Parlons maintenant des boucles conditionnelles.

Un programme qui se respecte utilise nécessairement des boucles (qui sont toutes conditionnelles)
Il y en a même une qui a un nom : La boucle principale
Évidemment si le programme ne bouclé pas, il s'éteindrait après avoir parcouru une seule fois le code.
On utilise donc une boucle dont l'ordinateur ne sortira que quand l'utilisateur le demande.

Il existe trois types de boucle en C++ :

La boucle While (utilisé le plus souvent comme boucle principale)

```
while( condition ) // ici si condition vaut faux, sort de la boucle
{

} // remonte à while et fait la condition
```

Il existe la variante qui fait un premier tour de boucle sans tester la condition d'abord (donc même si condition vaut faux). La boucle DoWhile

```
do{

}while( condition ) ; // Le ; EST IMPORTANT !
```

Et enfin la boucle qui permet de faire tout le reste,
La boucle For

```
for( int variable = 0 ; variable < 10 ; variable++ ) // ici plusieurs instructions se suivent
// déclaration d'une variable temporaire c'est-à-dire qu'elle disparaît en dehors de la boucle
// de type int
// et l'initialise à 0
// fait le teste conditionnel (ici  variable < 10)
// si le résultat est faux sort de la boucle sinon continu
// exécute le code en troisième position (ici variable++)
{

} // remonte
```

Si vous cherchez et comprenais ce qui se passe quand l'ordinateur exécute cela :
L'exécution va faire ici 10 tours de boucle en exécutant 10 fois ce qu'il y a dans la boucle.
Notre variable « variable » Va successivement faire toutes les valeurs de 0 à 10 et arriver à 10, la condition « variable < 10 » fera sortir de la boucle.

Je vous attire l'attention sur le fait que les tableaux et les boucles For se complète très bien.
Ainsi le code :

```
#define nombre_d_ennemis 200
int vie_des_ennemis[ nombre_d_ennemis ];
for( int a = 0 ; a < nombre_d_ennemis ; a++ )
{
        vie_des_ennemis[ a ];
}
```

Parcours tout le tableau vie_des_ennemis en accédant à chaque index successivement.

C'est avec cela que tout programmes et jeu possédant des data sous forme de tableaux sont possibles à réaliser.

Partie 3 : Un peu d'organisation

Alors là je vous avance tout de suite, cette partie n'est pas juste pour faire beau, c'est indispensable(réellement) pour la suite.
Votre code va nécessairement être sur plusieurs fichiers de code. Je vous laisserais le soin d'organiser tout ça dans Windows avec des dossiers.
Il y a deux types de fichiers de code : les .cpp qui contiennent le code C++ et les .h qui contiennent les entêtes.
Le fichier qui contient la boucle principale peut s'appeler main.cpp en référence à la ligne

```
int main(int argc, char **argv)
```

pour demander au compilateur d'intégrer le code d'un autre fichier de code .cpp ou .h dans un autre,

```
#include "niveau.cpp" // niveau.cpp en adresse relative
```

attention si vous découpé votre main.cpp en plusieurs fichiers (ce que j'aime bien faire) il faudra indiquer dans votre IDE (ici codeblocks) de ne pas les inclure !!

Sinon il arrive tout le temps que l'on veuille utiliser ce qui a déjà été développé par d'autre programmeurs avant soi. On aura donc besoin de l'instruction du précompilateur #include pour ce faire. Ps le précompilateur, c'est lui qui remplace les #define vu avant

```
#include <SDL.h> // EXEMPLES
#include <math.h>
#include <iostream>
#include <vector>
```

C'est dans la prochaine partie que je vous expliquerai comment organiser votre code.

Partie 4 : Les fonctions

Les fonctions... c'est un gros morceau... Allons-y !

Vous vous en êtes peut-être rendu compte, vous ne savez rien faire pour l'instant qui se rapproche d'un jeu ou d'un logiciel.
Soyons réaliste, rien n'est possible sans les fonctions !

A la base, une fonction peut être écrite pour ne pas avoir à recopier plusieurs fois les mêmes lignes de code. Une fonction est facilement reconnaissable au fait qu'elle est toujours précédé des parenthèses.
La fonction doit être déclarée :

```
void ma_fonction()
{
	// ce que fait la fonction
}
```

et peut alors s'utiliser de la manière suivante

```
ma_fonction() ; // on utilise la fonction pour la première fois
ma_fonction() ; // on utilise la fonction pour la deuxième fois
ma_fonction() ; // on utilise la fonction pour la troisième fois

for( int a = 0 ; a < 10 ; a++ ) // on utilise 10 fois la fonction
{
	ma_fonction() ;
}
```

Si c'était juste ça, ce ne serait pas terrible terrible

- Il faut savoir cela, la fonction peut renvoyer une variable, ou pas

Le type de la variable renvoyé s'écrit au début de la définition (ci dessus le « void »)
Vous pouvez remplacer le void par int ou float ou char … La fonction retournera une valeur encodée avec ce type. Si vous mettez void, la fonction ne renvoie rien.
Il faut alors imaginer que l'appel de cette fonction est comme un calcul que l'on peut utiliser :

```
int resultat = ma_fonction() ;
// ou
if ( ma_fonction() * 2 + 4 > 80 ) ;
```

Quand la fonction renvoie une valeur, la ligne :

```
return valeur ; // stoppe l'exécution de la fonction et renvoie valeur
```

on peut donc faire

```
int ma_fonction()
{
	if ( X == 1 )
	{
		return 0 ;
	}else{
		return 3 ;
	}
}
```

Vous savez tout à propos des retours de fonction

- Une fonction peut également recevoir des paramètres

Une fonction peut recevoir des paramètres c'est-à-dire des variables ou pointeur pour travailler avec. Il n'y a pas à ma connaissance de nombre maximal de paramètres que l'on peut passer à une fonction.
Les paramètres se déclarent dans la définition de la fonction. Et s'il y en a plus de 1, ils sont séparé

par des virgules.

```
	void ma_fonction( int parametre_1 , int* parametre_2 , float parametre_3 , bool int parametre_4 )
	{
		// ce que fait la fonction avec ces paramètres
	}
```

Vous n'êtes pas obligés de passer des paramètres à une fonction. Si vous ne passez pas de paramètres il suffit de laisser les parenthèses vides.
On appelle ensuite la fonction :

```
	ma_fonction(1 , pointeur , 10.56 , true )
```

Il y a plusieurs choses à savoir sur les paramètres :
déjà la fonction fait une copie physique des variables à son appel. C'est-à-dire que modifier une variable passée en paramètre dans une fonction ne modifiera pas la variable en dehors :

```
	void fonc( int v )
	{
		v++ ; // ici avec fonc( v ) ; plus bas, v égal 2
	}

	int v = 1 ; // v égal 1
	fonc( v ) ;
	// v est toujours égal à 1
```

La variable ayant été copié à l'exécution de la fonction, ce n'est pas la même qui ici a été incrémenté.
Pour pallier ce problème, on utilise les pointeurs. Donner l'adresse physique d'une data à la fonction permet d'accéder à celle-ci durant l'exécution de la fonction. Le cas pratique devient :

```
	void fonc( int* v )
	{
		*v++ ;
	}

	int v = 1 ; // v égal 1
	fonc( &v ) ; // on passe l'adresse de v
	// v égal 2 !!!!!
```

Le fait de passer des pointeurs des grosses data permet d'éviter la copie de grosses données à chaque appel de fonctions. C'est donc plus rapide d'exécution. Je vous rassure, c'est clairement inutile de faire ça à tout bout des champs avec de simples variables, très utiles pour les tableaux par exemple.

Petit retour au début où je vous avais indiqué de mettre votre code dans

```
int main(int argc, char **argv)
```

On comprend maintenant que ceci est un appel de fonction effectué par le système d'exploitation au lancement du programme, celui-ci prend des paramètres et chose magique, retourne un entier :
Je ne vous fais pas lambiner, return 0 ; à la fin de l'application indique à windows que le programme s'est terminé correctement. Cela vous permet de faire apparaître les erreurs par windows.

Maintenant je vais vous le dire, ce n'est pas comme ça que l'on est censé écrire et intégrer les fonctions (même si ce que je vous ai dit marche quand même).
On doit tout découper en plusieurs fichiers.
La déclaration de la fonction doit se trouver dans un fichier .cpp spécialement créer pour recevoir les déclarations de fonction. Par exemple « bonhomme.cpp » qui contient

```
void creer_bonhomme() {  ...  }
```

Et pour chaque .cpp créer, il doit y avoir un .h soit ici : « bonhomme.h ».

Dans ce fichier .h, On va écrire ce qu'on appelle les prototypes de fonctions.
En gros c'est la première ligne de déclaration de la fonction avec juste le type de variable et pas les noms des variables plus un ; à la fin.
Par exemple :

```
void fonc_1( unsigned int , float);
char fonc_2();
```

Les différents fichiers doivent être inclus dans le projet (dans codeblocks)
mais seul le fichier .h contenant les prototypes sera inclus au fichier main.cpp principal avec #include. Si un prototype n'a pas été écrit pour une des fonctions que vous voulez utiliser, cela ne compilera pas. D'autres par votre code va aller de plus en plus à se trouver sur de nombreux fichiers, le fichier .h devra être inclus partout où les fonctions qu'il déclare seront utilisé. Le fichier .h devra également être inclus au .cpp respectif à lui.
Petit truc en plus à faire dans les .h

```
#ifndef UN_NOM_A_CHOISIR
#define LE_MEME_NOM_CHOISI
// ici on met les prototypes
#endif
```

Ceux-ci sont des instructions du précompilateur pour éviter la redondance (vous faites ça!)
Je vais vous donner maintenant un conseil de pro pour rendre l'utilisation des fonctions encore plus puissante et cela donnera tout son sens à l'utilisation des prototypes.
Dans les prototypes dans les .h (mais pas dans les définitions du .cpp) on peut préremplir les valeurs des variables passées en paramètres.
Soit :

```
void ptite_fonction( unsigned int , float = 1.0 , float = 1.0 ,  int = 50 );
```

Le résultat est simple, une fois ceci fait, on n'est pas obligé de passer en paramètres ceux se trouvant à partir de la droite. Les appels de fonctions suivantes sont donc valide :

```
ptite_fonction( 10 , 2 , 3 , 42 ); // ou
ptite_fonction( 10 , 2 , 3 ); // ou                 équivalent à ( 10 , 2 , 3 , 50 )
ptite_fonction( 10 , 2 ); // ou                    équivalent à ( 10 , 2 , 1 , 50 )
ptite_fonction( 10 ); // ou                       équivalent à ( 10 , 1 , 1 , 50 )
ptite_fonction(); // ceci n'est pas valide dans notre cas, car le premier unsigned int n'est pas prédéfini
```

Tout ceci fonctionne de droite à gauche, si vous prédéfinissez un paramètre, il faut que les autres paramètres à droite de celui-ci le soient. On ne peut pas écrire :

```
void ptite_fonction( unsigned int , float = 1.0 , float = 1.0 ,  int ); // pour déclarer
// ou
ptite_fonction( 10 ,  ,  , 42 ); // pour appeler
```

Du coup il devient important de classer les paramètres par utilité.
A quoi cela sert-il ?
C'est super-bien pour toutes les fonctions qui reçoivent de manière générale les mêmes paramètres.
Je peux vous assurer que ça sert !

Pour finir les bibliothèque « iostream » ou « math.h » (regroupant des fonctions)

```
#include <iostream>
#include <math.h>
```

Nous donne pleins de super fonction et type : pour les fichiers... les calculs...

```
sqrt( N ) // retourne la racine carrée de N
pow( N , P ) // retourne N à la puissance P
int( N ) // peut transformer un flottant (nombre à virgule flottante > float) N en entier
sizeof(int) // renvoie le nombre d'octets qu'un type utilise par élément
cos( N ), sin( N ), tan( N ) // pour cosinus, sinus et tangente de N
atan2( X , Y ) // retourne l'angle de la droite ayant pour abscisse et ordonné X et Y
M_PI // constante ayant pour valeur π
```

Pour info, toutes include de libraires se font au début du code avant l'int main(...) car ils ne contiennent pas de code à exécuter mais des fonctions() prêtes à être utilisées.

Sinon, sachez qu'un même nom de fonction défini avec des paramètres différents est considéré par le compilateur comme plusieurs fonctions distinctes :

```
void int fonction( int paramètre_1 , bool paramètre_2 , float paramètre_3 ) ;
void int fonction( bool paramètre_1 ) ;
// les paramètres passés à la fonction à l'appel vont définir laquelle de ces fonctions sera exécutée

fonction( 10 , false , 2.5 ) ; // appellera la première fonction
fonction( true ) ; // appellera la seconde
```

Même conseil que pour les conditions, séparé les paramètres d'une fonction (dans le prototype, la définition et l'appel de la fonction) par des espaces :

```
void int fonction( int paramètre_1 , bool paramètre_2 , float paramètre_3 ) ;
```

La lisibilité s'en trouvera amélioré.

Partie 5 : Les Fichiers

Bon je vais torcher cette partie parce que ça me saoule :

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

on inclut ça dans le .h (jamais dans le .cpp, tous les include de librairie sont dans le .h et le .h inclus dans le.cpp)
ensuite :

```
char phrase[100]; // va contenir la phrase du nom du fichier à ouvrir
sprintf(phrase,"mon_fichier.type_fichier"); // fonction pour mettre une phrase dans le tableau
                                            //phrase
FILE* fichier = NULL; // pointeur fichier du type FILE*
fichier = fopen( phrase , "rb" ); // on ouvre le fichier "rb" en lecture / "wb"en écriture

fread( &var_int , sizeof(int) , 1, fichier ); // avec ça, on li un int que l'on met dans  var_int
// &var_int >> l'adresse mémoire où écrire ce qui est lue ,
// sizeof(int) >> la taille de chaque élément (selon leur type) ,
// 1 >> le nombre d'éléments à lire ,
// fichier  >> le pointeur du fichier ouvert

fwrite( &var_int , sizeof(int) , 1, fichier ); // pareil mais là on écrit dans le fichier
// ici, var_int sera écrit dans fichier

fclose(fichier); // il faut fermer le fichier qui a été ouvert en lecture ou en écriture
// si vous le fermez pas, windows vous dira gentiment : fichier ouvert par un programme
```

Partie 6 : La POO (programmation orienté objet)

Voilà le grand mot lâché de tout langage digne de ce nom.
Un Objet est en fait un ensemble de variables (En POO : arguments) et de fonctions (méthodes)
Il va falloir que tout ce que je vous ai enseigné jusqu'à présent s'enrichisse de la POO.
En fait un objet est une sorte de structure car on instancie (on crée une sorte de variable qui est en fait l'objet) à partir d'une définition de l'objet. Cet objet possède ses propres donnés et ses propres fonctions.
On peut donc imaginer un objet personnage qui contient un argument Point_De_Vie de type int et une méthode Recevoir_Coup qui a pour paramètre (c'est une fonction) le nombre de PV à perdre ou encore une méthode guérir avec pour paramètre le nombre de PV a regagner...
Dans un programme, on crée des objets comme celui-là, très concrets. Mais d'autres beaucoup plus abstrait car il arrive que plusieurs objets par exemple contenant différents types de data se combinent pour faire quelque chose (de génial !).

Alors quand on veut créer un nouveau type d'objet, on se crée 2 nouveaux fichiers :

- Le .h qui contient la définition de l'objet
- Le .cpp qui contient les fonctions de l'objet (les méthodes)

Voilà un exemple de .h, par exemple personnage.h

```
#ifndef PERSONNAGE
#define PERSONNAGE
class personnage
{
	private:
		int point_de_vie;
		void changer_vie( int );
	public :
		bool sexe; // 0 > homme , 1 > femme

		personnage( bool ); // constructeur
		~personnage(); // le ~ est pour le destructeur

		void gagner_point_de_vie( int );
		bool perdre_point_de_vie( int ); // return true si le perso est mort

		inline int get_point_de_vie()
		{
			return point_de_vie;
		}

		static void charge_donnee();

		static int valeur;
}
#endif
```

Nous avons déjà vu les instructions du précompilateur commençant par #
Voici la façon d'écrire le .h
Le nom de l'objet suit class
Il y a deux parties :
la partie private : les arguments et méthode qui s'y trouvent ne peuvent être accessible que dans le code se trouvant dans les méthodes de cet objet (on ne peut pas y accéder par exemple dans la boucle principale)
et celle public : les arguments dans cette partie pourront être accessible dans tous les autres codes, les méthodes pourront être appelé (exécuté) aussi dans les autres codes du programme.

Ceci a pour intérêt que votre objet soit utilisable même si on ne sait pas, ou ne sait plus, son fonctionnement exact.
L'objet est pensé pour être autosuffisant et ce qu'on peut lui demander de faire est bien délimité.
Pour vous convaincre, imaginé un objet possédant des arguments qui, s'il sont modifiés par l'utilisateur de la class provoqueraient un bug. Avec ces arguments en private, plus de problèmes.

Ensuite on y trouve des variables (ici argument) pour ça pas de problèmes.
Il y a aussi des prototypes de fonctions.
Je vous donne deux instructions dont je ne vous ai pas encore parlé :
inline avant le prototype et contenant aussi ce que fait la fonction (alors que normalement c'est dans le .cpp) en fait copie le code de la fonction partout où vous l'utiliserez (l'exécution ne fait donc pas un saut comme normalement à l'appel d'une fonction). Cela a pour résultat d'augmenter le poids de votre programme mais gagne aussi en vitesse d'exécution (très peu). A n'utiliser que pour de très petites fonctions qui sont appelées souvent de préférence.

La fonction que j'ai écrit

```
inline int get_point_de_vie()
{
	return point_de_vie;
}
```

est-ce qu'on appelle un accesseur.
Le principe est simple, notre argument point_de_vie étant private, on ne peut donc y accéder. C'est pour ça qu'on crée une méthode en public pour obtenir la valeur de point_de_vie.
Les accesseur étant généralement que d'une seule ligne, utiliser inline pour eux est bien.
On peut aussi imaginer un accesseur pour mettre une valeur dans point_de_vie.
Ce que ça donnerait :

```
inline void set_point_de_vie( int pv )
{
	point_de_vie = pv;
}
```

Comme cela, les arguments sont « protégés » et tout est toujours sous contrôle.
Ensuite l'instruction static elle est très intéressante car les méthodes static permettent à la méthode d'être appelé dans le programme même si aucun objet de la class n'a été instancié.
Ici on peut imaginer que

```
static void charge_donnee();
```

va charger les images ou autres trucs nécessaires à la class personnage
et ceci même si aucun personnage n'a été créé
ou encore ne charger tout qu'une seule fois même si on a créé plusieurs personnages.
Static peut être également utilisé sur des arguments :

```
static int valeur;
```

L'argument valeur pourra être défini même sans aucun objet instancié, dans par exemple notre fonction charge_donnee(). Il faut bien comprendre que chaque objet instancié se réfèrent alors à la même variable valeur et non une variable valeur par objet comme normalement.
Comme aucun objet n'est instancié, il faut spécifier la class à l'appel de la méthode :

```
personnage::charge_donnee();
```

Pour finir voici deux choses que j'ai besoin de vous expliquer

```
personnage( bool ); // constructeur
~personnage(); // le ~ est pour le déstructeur
```

C'est chose sont des méthodes, elles sont obligatoires quelque soit l'objet que vous codez.
En effet les objets possèdent une méthode appelée le constructeur, et une autre le destructeur
La première sera appelée à la construction de l'objet,
de manière classique :

```
personnage Le_Heros( true ) ;
```

ou de manière dynamique

```
personnage* Le_Heros = new personnage( true ) ;
```

(et oui les objets et les variables se ressemblent. Il y a même des pointeurs d'objets)
(en fait un objet est une sorte de type de donnée)
Le destructeur, lui, est appelé à la destruction de l'objet.
Si l'objet a été instancié de manière classique, celui-ci sera détruit à la sortie de main() ou on peut appeler le destructeur nous-même.
Si l'objet a été instancié dynamiquement, on peut aussi appeler le destructeur, mais l'instruction delete marche aussi :

```
delete  Le_Heros ;
```

Le constructeur quand à lui ne sera pas appelé par vous car il l'est à la création de l'objet.

Donc pour utiliser notre objet personnage par exemple dans main.cpp

```
#include "personnage.h"
```

Une fois l'objet instancié, on peut accéder aux arguments et méthodes déclarées publiques
Si l'objet est instancié classiquement :

```
Le_Heros. sexe
Le_Heros. gagner_point_de_vie( 14 );
```

Si c'est un pointeur d'objet :

```
Le_Heros->sexe
Le_Heros->gagner_point_de_vie( 14 );
```

On utilise le tiré du 6 et le supérieur : ->

Maintenant, nous allons voir comment écrire le .cpp
Voici un exemple du « personnage.cpp » qui contiendrait les méthodes de notre « personnage.h » vu précédemment.

Personnage.cpp

```
#include "personnage.h"

int personnage::valeur; // pour les arguments
                                    static

personnage::personnage( bool _sexe)
{
        sexe=_sexe ;
        point_de_vie = 100 ;
}

personnage::~personnage()
{
}

void personnage::changer_vie( int v )
{
         point_de_vie += v ;
}

void personnage::gagner_point_de_vie( int v )
{
         changer_vie( v ) ;
// changer_vie étant private, elle ne peut être
// exécuté que dans les méthodes de l'objet
}

bool personnage::perdre_point_de_vie( int v )
{
         changer_vie( -v ) ;
         return ( point_de_vie <= 0 ) ;
}        // return true si le perso est mort

void personnage::charge_donnee()
{
         valeur++ ;
}
```

Personnage.h

```
#ifndef PERSONNAGE
#define PERSONNAGE
class personnage
{
        private:
                int point_de_vie;
                void changer_vie( int );
        public :
                bool sexe;

                personnage( bool );
                ~personnage();

                void gagner_point_de_vie( int );
                bool perdre_point_de_vie( int );
                // return true si le perso est mort

                inline int get_point_de_vie()
                {
                        return point_de_vie;
                }

                static void charge_donnee();

                static int valeur;
}
#endif
```

Voilà je ne pense pas avoir à expliquer plus avant, cela m'a l'air limpide. Bonne chance.
Remarquez juste le « nom_de_class :: », le compilateur en a besoin pour que tout marche.
Pour finir, sachez que de nombreuses librairies proposent des fonctions, des structures mais surtout des objets entièrement fonctionnels et intéressants.
Par exemple des objets tableaux : Vector et Array (pour que toutes les manipulations de tableaux soient simplifié)
Ou encore l'objet : String (chaîne de caractères) pour les textes.

Par ailleurs, dans les méthodes d'un objet, l'instruction this est un pointeur vers cet objet.

D'autres trucs à vous apprendre : la surcharge d'opérateur et les constructeurs de recopie

- La surcharge d'opérateur :

Les opérateurs, laissez moi vous rappeler :
, - , * , / , ++ , -- , += , -= , *= , /= , == , ! ,!= , || , && , > , >= , < , <= …
il y en a d'autres comme << ou >> mais bon faut bien s'arrêter quelque part.

La surcharge d'opérateur, c'est demander à l'ordinateur d'exécuter une méthode quand un objet se trouve à utiliser un opérateur. par exemple :
```
objet_1 = objet_2 + objet_3 ;
```
Par exemple, imaginez un objet qui contient une chaîne de caractères, on peut surcharger l'opérateur + pour qu'il mette l'objet_3 à la suite de l'objet_2 pour créer une nouvelle phrase : objet_1
Bien sur, des fois certains opérateurs n'ont aucune logique à être surchargés.
```
objet_2 * objet_3 ;   //???
objet_2 && objet_3 ;  // ???
// pour un objet : chaîne de caractères cela n'a aucun sens
```

Les surcharges d'opérateurs s'écrivent comme ceci :
```
Nom_Class operator+( Nom_Class const& a, Nom_Class const& b);
// l'opérateur suit « operator »
// const signifie « constante »
// qui veut dire que les données de a et b ne seront pas modifié dans la méthode.
// ( marche aussi avec les fonctions )
```
Notre opérateur peut renvoyer ce que vous voulez et c'est même possible que l'opérateur agisse sur deux objets de class différentes.
Exemple : objet_chaine_de_caractere + valeur_int
« perso » 1 donne « perso1 »

- Les constructeurs de recopie :

Les constructeurs, vous connaissez, c'est la méthode appelée à la création d'un objet.
Et bien de la même manière que l'on peut créer une variable en fonction d'une autre :
```
    int var_1 = var_2 ;
```
On peut créer un constructeur d'objet qui reçoit un pointeur d'un autre pour, peut-être copier ses data (en fait vous pouvez décider de faire ce que vous voulez)
Ce constructeur s'écrit :
dans le .h
```
    Nom_Class( const Nom_Class &pointeur );
```
dans le .cpp
```
    Nom_Class::Nom_Class( const Nom_Class &pointeur )
```
Et s'utilise :
```
    Nom_Class objet_1();
    Nom_Class objet_1_copie = Nom_Class( objet_1 ); // la méthode de recopie sera appelée ici
```

Je vais vous donner une utilisation du constructeur de recopie que j'ai faite (pour comprendre)
j'ai un objet data qui contient un objet 3d.
Je veux que cet objet apparaisse à différents endroits à la fois où ils se déformeront avec le temps.
Je charge donc mon objet
Donc à chaque fois que je veux faire apparaître mon objet qui se déforme quelque part.
Je copie cet objet de data et utilise la copie pour l'afficher en la déformant...
C'est un exemple mais il arrivera que vous ayez à l'utiliser.

Partie 7 : L'Héritage

Cette partie est faite pour les pros de pro et n'est utile que si vous voulez, par exemple, stocker dans un tableau de pointeur, toutes les entités (héros,ennemis,boss...) de votre jeu avec des objets bien distincts pour chacun.
Personnellement on peut s'arranger autrement au début pour vos premiers programmes, mais l'utilisation de l'héritage va rendre votre code parfait même si c'est plus long à développer.

Si vous voulez passer dès maintenant à la création de jeux ou logiciels, vous pouvez passer à la partie 8 mais peut-être un jour, l'héritage vous manquera...

Il arrive que plusieurs objets aient des arguments et des méthodes qui se ressemblent ou sont identiques. Par exemple : héros, araignée méchante et boss du jeu ont tous des coordonnées, des points de vie, peuvent tous perdre de la vie et mourir.
Le principe de l'héritage est simple, tous ces objets vont hériter des arguments et des méthodes d'un objet unique qui dans le cas ci-dessus pourrait être « entité ». L'objet qui va être utilisé pour que d'autres en héritent est appelé l'objet père. Ceux qui héritent les objets fils.
Il y a de nombreux intérêts a utiliser l'héritage mais je vais vous en donner trois :

- Ne pas avoir à réécrire le même code plusieurs fois.
- Pouvoir modifier le fonctionnement de plusieurs objets en une seule fois
- Mais surtout, un pointeur de type un objet père peut pointer vers un objet fils. Cela permet de stocker par exemple des pointeurs dans un tableau vers des objets qui ne sont pas de la même class. Donc dans notre cas, héros, araignée méchante et boss du jeu serons stockés dans le même tableau de pointeur alors qu'ils ne sont pas de la même class. Les pointeurs seront du type de l'objet père. Une boucle for permettra de tous les gérer en même temps.

Alors ce qu'il faut savoir :
Tout arguments et méthodes d'une class mère (mère, père … pareil) seront accessibles (attention aux private et public!) par les class filles (fils, fille … idem).
A l'appel du constructeur, le constructeur de la class mère sera appelé avant celui du fils.
Pareil pour le destructeur.

Maintenant que vous connaissez le principe, voici comment l'écrire :

Dans le .h

```
#include "Class_mère.h"
class Class_Fille : public Class_mère
```

Dans le .cpp
(par exemple avec trois flottants)

```
Class_Fille::Class_Fille(float x,float y,float z) : Class_mère(x,y,z)
```

c'est cela qui appel la class mère.

Partie 8 : Je veux faire du sons et lumières !!

On arrive à la fin de l'apprentissage de la base du C++
Avec ce que vous avez appris vous pouvez créer le fonctionnement de tout programme, et pour le reste, je compte vous donner tout ce qui sera nécessaire pour vous dans un premier temps. Soit gestion de l'affichage 2D / 3D et des chargements d'image , gestion des sons et musiques.
On dit merci qui ? :)

Pour gérer la fenêtre de votre programme, le clavier , la souris , ou bien des joysticks...
On va utiliser la bibliothèque SDL2
(déjà la 1 c'était bien mais celle-ci est géniale)(compatibles tous systèmes)
Pour charger les différents formats d'images
SDL_image 2
Pour les sons et musiques
SDL_mixer 2

Pour l'affichage nous allons plutôt utiliser la carte graphique plutôt que de demander au processeur de le faire. C'est plus malin et plus souple. Nous passerons donc par Opengl qui est une interface pour simplifier les demandes que l'on fait à la carte graphique(également compatible tous systèmes)

Donc vous cherchez sur le net les librairies SDL2 …
Vous téléchargez les Development Libraries: La version MinGW
32 ou 64 bit, en fonction de ce que vous avez choisi dans l'installation de CodeBlocks
et le copié dans le dossier « i686 … »
copier le contenu des dossiers dans les dossiers du même nom que vous trouverez dans le dossier d'installation de CodeBlocks dans MinGW. (bin, include, lib, share ...)
Les dll se trouvant dans bin seront également copiés dans le dossier de votre programme à coté de votre .exe

Dans les fichiers où vous voulez inclure les fonctions des librairies SDL, ajouté au début

```
#include <SDL.h>
#include <SDL_image.h>
#include <SDL_mixer.h>
```

Ensuite dans CodeBlocks, votre projet ouvert,
Project >> Build option >> nom du projet séléctionné à gauche (et pas Debug ou Release)
>> Linker setting
Entrez :

```
mingw32
SDL2main
SDL2.dll
C:\   « L'emplacement de votre CodeBlocks »   \CodeBlocks\MinGW\lib\libSDL2_image.dll.a
libopengl32
libglu32
SDL2_mixer
```

dans >> search directories >> compiler
Entrez :

```
C:\  « L'emplacement de votre CodeBlocks »   \CodeBlocks\MinGW\include\SDL2
```

dans >> search directories >> linker
Entrez :

```
C:\  « emplacement de CodeBlocks »   \CodeBlocks\MinGW\i686-w    « 32 ou 64 »    -mingw32\lib
```

Voilà, vous pouvez utiliser les librairies SDL, SDLimage et SDLmixer dans votre programme.

Demandez-moi, je vous fournirai des objets, structures, codes que j'ai moi-même créés pour utiliser simplement ces librairies.
Mon code se découpe en 4 parties, entete, initialisation, boucle principale, destruction et fin.
Et gérer la souris, le clavier, les joysticks ou même créer un menu pour définir les touches (comme dans tout bon jeu) est ultra-simple.
Les images à charger ou afficher pareil, Les sons .wav ou musique .mp3 pareils.
Gérer la fenêtre, la gestion des changements de sa taille et de fermeture PAREIL !!

Partie 9 : Pour finir

Il vous faudra beaucoup de jugeote et d'organisation pour créer la plus simple des applis.
Bien sur, personne ne naît avec ça. Beaucoup de galère et d'apprentissage vous aurai besoin.
Je me suis intéressé, moi, à l'âge de 10 ans, n'ai appris la POO que vers 17 et je vous le dis, elle est loin d'être obligatoire pour faire un jeu. Bien sur, aujourd'hui, le moindre moteur est une succession d'objet avec héritage multiple. Mais je pense que pour avoir la logique dans sa tête, la POO n'est vraiment pas une aide. J'ai fait des trucs de ouf sans ça. Et le jour où vous pourraient imaginer un algorithme que beaucoup n'arriveraient même pas à comprendre ou à se représenter, vous serez fort mes fils !!

Partie 10 : Et la 3D ?

J'espère ne pas vous casser toute votre volonté en vous disant que pour la 3D, il existe une telle quantité de chose à comprendre et mémoriser que s'en est flippant.
Arriver à utiliser un logiciel de 3D comme blender est déjà une prouesse, surtout pour apprendre à l'utiliser entièrement.
Je vous donne quelques noms, en pagaille, de choses qu'on ait censé connaître si on veut faire des applis 3D.

Vertex, normal, matrice de transformation, calculs matriciel, zbuffer, antialiasing (multisampling), index array, u/v de texture, color array, sphere mapping, interpolations de pixels, listes compilées, lumière ambiante, diffuse, specular, double buffering, alpha blending, front-facing polygons, depth mask, Ztest, mip mapping, shader, stencil, scissors...

Armez-vous de patience, le chemin est long, il l'a toujours été...

PS : en parlant de Blender, si vous voulez que je fasse un cours dessus (en omettant rien), dites-le-moi !

Sanselme Armand (Mr Lalou)