-using nouveau\_nom = type\_existant; ou type\_existant [tableau] Permet de donner un alias à un type déjà existant. On pourra alors utiliser ce dernier pour définir des variables au lieu du type originale. Aide simplement à la visibilité. Permet aussi de changer rapidement le type, à un seul endroit, de plusieurs variables ayant le même alias. On peut donner un alias à un alias.  
  
-enum Exemple {1, 2, 3, etc} Type énuméré permet de créer un type de variable qui ne pourra prendre que les valeurs incluses. On doit ensuite l’assigner à une variable, elle ne pourra prendre que les valeurs possible mise dans le tableau du type. Les valeurs qui sont inclus dans le type sont considéré comme étant des entiers, comme dans un tableau. Par défaut, la première valeur du type sera donc un 0 dans la mémoire de l’ordinateur. Cela veut dire qu’elles peuvent être incluse dans un switch. Dans le code il faut tout de même utiliser les noms affectés. On peut associer la valeur entière que l’on souhaite à chaque possibilité de notre type de la manière suivante : {AA = 100, A = 90, B = 80, C, D = 60, E} Si on ne spécifie pas la valeur, elle sera n+1. Donc C vaudrait 81 et E vaudrait 61. Les comparaisons sont faites à partir des entiers associés à chaque valeur. On peut forcer la transformation d’un entier en un notre variable énuméré : C notre variable entière et Ak notre variable énuméré ce qui donnera Ak(C).  
EX : 1. Définition type enum Direction { NORD, SUD, EST, OUEST, INCONNU}; 2. Définition variable Direction cap; Affectation cap = NORD;   
Il se peut que deux types aient des valeurs possibles ayant le même nom. Il faut alors que notre type devienne une class.   
EX : 1. Définition type enum class RVB { ROUGE, VERT, BLEU}; enum class SPECTRE {VIOLET, INDIGO, BLEU, VERT, JAUNE, ORANGE, ROUGE}; 2. Définition variable RVB rvb; SPECTRE spectre; 3. Affectation rvb = RVB::ROUGE; spectre = SPECTRE::ROUGE;  
Puisque rvb est associé au type RVB, elle pourra prendre que des valeurs du même type. Même chose pour la variable spectre qui ne pourra que prendre des valeurs possibles du type SPECTRE.  
  
- struct Exemple {int x; int y; string doing, urmom}; Le type structuré permet de faire un type dont les variables pourront encapsuler plusieurs informations. Ces informations peuvent être chacune de type différent. Permet finalement de créer plusieurs variables facilement et rapidement. Si deux variables ont le même type, elle peuvent alors être équivalente. Par exemple, Exemple c = Exemple d; Pour afficher ou bien utiliser une variable contenant plusieurs valeurs, il est important de spécifier quelle partie de la variable on souhaite utiliser. On peut inclure un type structuré à l’intérieur d’un nouveau type structuré.  
EX : 1. Association à une variable Exemple c; 2. Donner des valeurs c.x=10; c.y=15; 3. PS peut aussi être écrit ainsi c = { 10 , 15};   
EX Tableau: 1. Création du tableau Exemple lol[10] 2. lol[0].x = 10; lol[0].y = 15; lol[0].doing = “i had sex”; lol[0].urmom = “with your mom”; 3. PS peut aussi être écrit ainsi lol[0] = { 10, 15, “i had sex’’, ‘’with your mom’’};  
  
-setcolor(Color ::voir CVM header pour toutes les possibilités) Permet de changer la couleur des caractères qui seront affichés. La couleur de base est setcolor(Color ::wht);   
  
-show(bool statement) Permet de faire affiché ou disparaitre le curseur de la console  
  
-setwsize(largeur y, largeur x) Permet de modifier les dimensions de la console  
  
-void nom\_de\_la\_fonction(Type param\_1 , Type param\_2 , Type param\_N){} Une fonction est un outil de rangement pour nommer et regrouper les instructions d’une tâche spécifique. Seul la fonction main est obligatoire mais la décomposition du programme en plusieurs fonctions est nécessaire pour faciliter sa lisibilité, son entretien et son évolution. L’écriture d’une nouvelle fonction ce fait de la même manière que la fonction main(). L’utilisation du type void est obligatoire pour le moment. Les parenthèses après le nom sont aussi obligatoires. Un appel à la fonction consiste à exiger son exécution et par conséquent d’exécuter ses instructions. Une fonction peut appeler une infinité de fonction.  
Une fonction a aussi des paramètres. Les paramètres sont des variables locales spéciales (des contenants) Les informations contenues dans ces paramètres seront données par la fonction appelante. Ces informations seront ensuite utilisées par les instructions de la fonction. En ordre, les paramètres seront créés avant l’exécution des instructions de la fonction. Les paramètres seront ensuite initialisés avec. Les valeurs données lors de l’appel. Les paramètres seront alors utilisés par les instructions de la fonction. Les paramètres seront détruits à la fin de la fonction. La valeur des paramètres peut être modifier par la fonction qui l’appel. Alors que de simples variables présentent dans la fonction ne pourront pas être modifier par la fonction qui l’appel. Les paramètres disponible à la fonction son inclus dans entre les parenthèses, il faut y indiquer les type. Les valeurs que l’on veut donner à ces paramètres sont inclus encore une fois entre les parenthèses lorsque la fonction appelante demande à utiliser la fonction externe.   
  
-Type\_retour nom\_de\_la\_fonction (){} Permet de conserver le résultat d’une fonction à l’intérieur d’une variable de la façon suivante : Type\_retour var; var = nom\_de\_la\_fonction();   
Ex : int somme(int a, int b)  
{  
 int total = a+b;  
 return total;  
}

int main(void)  
{  
 int x = 10, y = 20, z;  
 z = somme(x,y);  
}  
Dans ce cas-ci, somme équivaudra au résultat de la variable à laquelle return est lié, soit total.  
  
- Type\_retour nom\_fonction (Type param, …) ; La déclaration d’une fonction est une instruction pour annoncer l’existence d’une fonction. La déclaration d’une fonction contient seulement le nom, le type de retour et la liste de paramètres, sans le bloc {} d’instructions. Une fois la fonction déclarée, elle peut être utilisé sans avoir de code à l’intérieur de cette dernière. On peut donc mettre la définition de notre fonction où on le souhaite. Dans la déclaration, il est possible de simplement mettre le type des paramètres sans nécessairement leur donner de nom. On peut appeler nos paramètres par valeur (soit une valeur qui ne sera pas modifier dans le programme mais seulement dans la fonction), par référence(soit une valeur qui sera modifier dans le reste du programme, impossible de prendre une valeur temporaire), et par adresse/pointeur(qui ne prendra alors qu’un pointeur, une adresse, pour faire ses calculs, la syntaxe est la même que pour l’utilisation normal d’un pointeur lors de la création du paramètre. La valeur présente à l’adresse ne sera modifier que dans la fonction et non dans le programme).  
  
-Type & variable\_référence = variable\_référée; Définir une référence consiste à créer un autre nom, un alias, pour accéder au contenant d’une autre variable. Le nom de cette référence devient alors une alternative pour lire ou modifier le contenu de cette autre variable. Il est permis de définir plusieurs références vers la même variable. Le & indique au compilateur qu’il s’agit d’une référence. Il est possible d’utiliser des références à l’intérieur des paramètres d’une fonction, les variables inclus dans les paramètres en tant que référence seront alors modifier, contrairement à ci on ne fait que déclarer une variable.  
  
- Beep (Frequence, Durée) Permet de faire faire un bip audible par l’ordinateur. La fonction prend deux int, le premier pour la fréquence et le deuxième pour la durée en millisecondes.  
  
-tm est une structure de windows pour ranger les informations temporelles. Il faut donc qu’il soit associé à une variable comme suit : date.tm\_mday / date.tm\_mon (il faut rajouter +1 à la valeur) / date.tm\_year (il faut rajouter +1900 à la valeur) /date.tm\_hour / date.tm\_min / date.tm\_sec . Le seul qui diffère est date.tm\_wday qui redonne un chiffre entre 0 et 6, représentant les jours de la semaine de dimanche à samedi  
  
-setcp(1252) permet de d’utiliser des lettres avec accents sans inclure le caractère ascii de la dite lettre  
  
-Fonction surchargé, plusieurs fonctions peuvent porter le même nom, car le compilateur regarde les paramètres pour établir si elles sont différentes. Ces fonctions peuvent avoir le même nombre de paramètre tant que leurs types varient. Si les différentes fonctions ont les mêmes types de paramètres, on peut en changer l’ordre pour les différencier. Le type de donné que la fonction se doit de retourner ne permet pas au compilateur de différencier entre les différentes fonctions portant le même nom. Le fait de rajouter une référence à un paramètre ne permet pas au compilateur de différencier les fonctions.  
  
-double taxer(double montant, double taxe\_1 = TPS, double taxe\_2 = TVQ) Paramètres par défaut, il est possible de donner des valeurs par défaut à nos paramètres, tant quelles sont du type requis par la fonction. Si aucune autre valeur n’est spécifiée lors de l’appel, les valeurs par défaut du paramètre seront utilisées. On ne peut pas donner une valeur par défaut à un paramètre si le paramètre suivant n’en a pas une. Si on veut inclure des paramètres par défaut, il est prioritaire de les inclure dans la déclaration de la fonction s’il y a lieu, et non dans la fonction en elle-même.  
  
-#include <fstream> fstream est une bibliothèque qui nous permet de travailler avec des fichiers. Pour l’utiliser, il est important d’aussi inclure iostream.  
  
-fstream est un type de variable qui nous permet de stocker le nom d’un fichier. Ex : fstream fileName;  
  
-Il existe plusieurs façons d’ouvrir un fichier (Garder en tête string fname = nomFile.txt) cela associe le nom de notre fichier à notre variable fstream :  
1. fileName.open(fname, ios ::out); Le out ou output crée ou recrée un nouveau fichier vide. Pour écrire dans le fichier.  
2. fileName.open(fname, ios ::app); Le happening place notre curseur à la fin du fichier, ne le détruit, ne fait que l’ouvrir.  
3. fileName.open(fname, ios ::in); Le in ouvre le fichier et place le curseur au début du fichier sans en détruire le contenu. Permet de lire le fichier.   
4. fileName.open(fname, ios ::beg); Le beginning ouvre le fichier et place le curseur au début du fichier, encore une fois sans en affecter le contenu.  
5. fileName.open(fname, ios ::binary); Permet d’ouvrir le fichier sans que certain caractère tel que le ‘\n’ soit transformer en un autre.  
6. fileName.open(fname, ios ::trunc); Utile seulement pour recréer une base donnée, car il supprime l’ancienne. Agit un peu de la même manière que ios ::out, mais a besoin qu’un fichier existe déjà. Il supprime le contenu du fichier et non le fichier en soit.  
7. fileName.open(fname, ios :: end) Permet, sans modifier le contenu du fichier, de placer le curseur à la fin du dit fichier.   
  
(PS : Mettre | entre différent ios :: permet d’ouvrir le fichier en utilisant plusieurs mode. C’est-à-dire que si on fait ios ::in|ios ::out le fichier sera ouvert pour recevoir et envoyer de l’information. Lorsqu’utiliser en tandem, ios ::in et out, ios ::out ne supprimera pas le fichier, mais on peut tout de même supprimer le contenu du fichier en incluant ios ::trunc)  
  
-fileName.close(); Permet de fermer le fichier, est une bonne pratique à avoir.  
  
-Pour lire un fichier mot par mot, on utilisera le nom de notre variable fstream qui contient le nom de notre fichier. On peut alors l’utiliser tel un cin >>. Dépendant de la variable utiliser pour stocker l’information du fichier il sera lu différemment. Mais normalement, il lira jusqu’au premier espace, comme un cin. On peut utiliser notre fstream dans un while pour lire tout notre fichier jusqu’à ce qu’il y est un fail. Ex : while(fileName >> myText) myText étant un string.  
  
-Pour lire un fichier ligne par ligne, on utilisera getline(fileName, myText); myText étant un string. Il lira alors le fichier ligne par ligne, dans le fond jusqu’au premier changement de ligne qui apparaitra. On peut changer jusqu’à quel caractère on souhaite qu’il lise en faisant getline(fileName, myText, ‘’,’’); S’il ne trouve pas le nouveau délimiteur, il retournera l’entièreté du fichier. On peut l’utiliser à l’intérieur d’un while pour qu’il agisse jusqu’à la fin du fichier. Ex : while(getline(fileName, MyText)){cout << myText;} myText étant un string.  
  
-fileName.get(nom\_char); Permet de lire un fichier caractère par caractère. Peut être utiliser dans un while(), si l’action de get est possible retourner une valeur true et si l’action n’est pas possible on elle retournera false.  
  
-Pour écrire du contenu dans un fichier, on fait simplement utiliser notre variable fstream tel un cout. Sauf que du coter droit on peut retrouver une autre variable qui contiendra de l’information qui sera alors stocker dans le fichier associé à notre variable fstream.  
  
-skipws et noskipws Font ce que leur nom implique, le premier permet de sauter les espaces vides lors de la lecture d’un fichier texte, la deuxième retourne tout, les espaces vides ainsi que les changements de ligne. Ne sera respecter que lors de get.  
  
-fileName.is\_open(); Permet de savoir si le fichier où l’on souhaite travailler est ouvert  
  
-fileName.fail(); Retourne une valeur boolenne pour savoir si on a finit de lire le fichier. Peut être utiliser dans un while().  
  
-fileName.clear(); Puisque la lecture de fichier agit tel un cin, il est important de clear la mémoire lorsqu’il y a une erreur/fail.  
  
-fileName.write((char\*)&variable, sizeof(structure)); Permet d’écrire la variable voulu dans un fichier dans un espace prédéfini dans un fichier. La première partie de write demande l’adresse de la variable que l’on veut écrire, il faut que cette variable soit un char, c’est pour ça que l’exemple convertie notre adresse en char. La deuxième partie de write est la taille que la variable prend en mémoire dans notre programme, et donc la taille qu’elle prendra dans le fichier. Si on connait la taille que prend chaque variable dans le fichier, on peut se déplacer d’une variable à l’autre dans le fichier grâce à seekg. Utiliser dans des fichiers binaires (.db).  
  
-fileName.read((char\*)&variable, sizeof(contenant)); Fonctionne de la même manière que write, mais nous permet de lire des variables dans un fichier, et donc de les inscrire dans la mémoire de notre programme. On connait l’espace que prend chaque variable grâce à sizeof(). La première partie de read est l’adresse de la variable qui va contenir ce qui aura été lu. Le deuxième est la taille que prend la variable dans le fichier et en mémoire. À cause de l’organisation du fichier, on peut utiliser seekg pour se déplacer dans le fichier et lire la partie du fichier que l’on souhaite. Utiliser dans des fichiers binaires (.db).   
  
-fileName.seekg(déplacement, emplacement\_de\_départ); Permet de déplacer le curseur dans le fichier. Ex : fileName.seekg(0, ios ::beg); La première partie de la fonction est la grandeur du saut qui devra être fait dans le fichier. La deuxième partie est l’emplacement à partir d’où le saut dans le fichier se fera. On cherche donc à savoir combien d’octet on va sauter pour se rendre au prochain endroit où se trouve la section d’information voulu. Pour cela on a besoin de savoir combien d’objet il y a entre nous et notre destination ainsi que la taille de ces objets.   
  
-fileName.tellg() Retourne l’emplacement du curseur dans les octets du fichier. En l’utilisant avec seekg, on peut déterminer la taille du fichier. On commence par placer le curseur à la fin du fichier avec   
fileName.seekg(0, ios ::end); ensuite on utilise fileName.tellg() qui nous offre la taille du fichier. On peut alors la convertir en unsigned long long (size\_t) pour pouvoir la manipuler. (size\_t)fileName.tellg()  
  
-remove() Permet de supprimer un fichier du dossier ou se situe l’exécutable   
  
-sizeof() Permet de connaitre la taille en octet d’une variable ou même d’une structure.  
  
////////////////////////////////////////Old Info, but still kinda valid//////////////////////////////////////////////////  
-ofstream MyFile (‘’filename.txt’’); Permet de créer et d’écrire dans des fichiers. Pour y écrire on inscrit le nom du fichier suivit de <<, un peu comme un cout. Il est bonne pratique de fermer le fichier une fois que l’on finit d’y inscrire de l’information, on peut faire cela avec la ligne MyFile.close();.  
  
-ifstream MyReadFile(‘’filename.txt’’); ou fstream MyReadFile(‘’filename.txt’’); Indique au programme dans quel fichier se trouve l’information que l’on cherche et l’associe à une variable. Pour afficher l’information on peut utiliser un while ainsi qu’un getline pour lire le fichier ligne par ligne comme suit : while(getline (MyReadFile, myText)) { cout << myText;} myText étant un string, le texte sera alors affiché ligne par ligne. Si on veut stocker l’information dans une variable, il suffit d’utiliser le nom de la variable qui représente notre fichier et de l’utiliser tel un cin >>. Lorsque l’on fait cela, on peut l’inclure dans un while(), lorsqu’aucun caractère ou mot ne pourra être inclus, l’action fera alors un fail, retournant un false.  
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////  
  
-#include <cstring> permet d’utiliser string.c\_str() qui transforme le string en const char. Est utile surtout pour utiliser des strings avec la fonction remove().  
  
-NomVariable.resize(Valeur Numérique) Permet de changer la taille d’un string. Si le string a 20 charactères, on peut le resize à 10. Le string original n’aura maintenant que 10 charactères.  
  
-Option de modifications de string : string.replace(position, quantité, remplacement) Remplace la quantité de charactère indiqué à partir de la position donnée avec le remplacement offert. string.substr(position, quantité) Retire la quantité de charactère à partir de la position. string.erase(position, quantité) Efface la quantité de charactères indiqué à partir de la position indiqué. string.find(‘’ ‘’) Cherche pour la suite de charactère indiqué entre parenthèse à partir du début du string, retourne la position des dit charactères. string.rfind(‘’ ‘’) Chercher pour la suite de charactère indiqué entre parenthèse à partir de la fin du string, retourne la position du dit charactères.  
  
-stringstream ss; Type de contenant qui permet de gérer un stream de caractères. Fonctionne tel un cin ou un cout. Permet d’inclure des données quelconques dans un string, les transforment pour qu’elles puissent être inclus dans un string. Doit avoir #include <sstream>. Pour mettre les valeurs dans un string ss.str(NomDuString); Prière d’utiliser .clear() après chaque utilisation pour réinitialiser le tampon, otherwise on ne peut pas utiliser le même stringstream de suite. Il est possible de convertir un string en int grâce à cette fonction.   
Ex : string temp; int test; getline(cin,temp); ss.str(‘’’’); ss<<temp; ss>>test>>ws; ss.clear(); Si une valeur autre qu’un int est inséré, la valeur de test sera 0. .str() permet de réinitialiser le tampon à une chaine de charactère vide. ws permet de retirer les espaces blancs si présent.   
Prière de garder en compte que stringstream arrête au première espace rencontré, si on veut inclure ce qui vient après le première espace, il faut répéter la commande stringstream jusqu’à avoir vidé le tampon.  
  
-&NomDeMaVariable & est un opérateur unaire qui permet d’obtenir l’adresse en mémoire de la dite variable. À la différence d’une référence, le symbole se situe à gauche du nom de la variable préalablement déclaré, alors qu’une référence à le symbole & à droite du type de la variable.   
  
-TypeDeLaVariableDontOnVeutConserverL’adresse\* NomDeMonPointeur Un pointeur permet de conserver l’adresse de notre variable. C’est donc un pointeur du même type que la variable dont il conserve l’adresse. Un pointeur ne peut pas changer de type. Lorsqu’on initialise un pointeur, la valeur qui s’y trouve est aléatoire. On peut cependant l’initialiser. Ex : double\* p; p = nullptr; ou p = null; (Usage recommandé pour initialiser un pointeur à null, s’il n’est pas associer à une variable, s’assure qu’il pointe à rien du tout, il n’y aura donc pas daccess violation) on peut aussi l’initialiser à p = 0; Mais ce n’est pas recommandé. Un pointer peut servir de prédicat pour un test, puisque si l’adresse est nul, la valeur retourner sera FALSE.  
  
-\* Opérateur unaire, opérateur de déréférencement qui permet d’aller à une adresse et d’en voir le contenu.   
Ex : int v = 25; int \*ptr = &v; cout << ptr; (affichera l’adresse) cout << \*ptr; (affichera le contenu de l’adresse) Si on fait \*ptr = 10, cela changera le contenu de l’adresse. Exemple pour te déplacer dans un tableau : N[1] est équivalent à \*(N+1), car N est un pointeur et représente l’adresse du début de mon tableau, +1 lui additionne l’espace que prend le type de N, on arrive donc à la deuxième valeur de notre tableau. Pour afficher les variables d’une struct grâce à son pointeur il faut prendre la priorité des opérations en comptes. Ex : struct Produit { int no; double prix; string description;} Produit lait = { 1023, 3.25, ‘’lait 2%’’}; Produit \*p = &lait; cout << (\*p).no; Ps : L’écriture cout<<\*p.no; ne fonctionnera pas car . à priorité dans l’ordre des opérations sur \*.  
  
- -> Opérateur raccourci qui permet de simplifier la syntaxe lorsqu’on tente d’atteindre les membres d’une stuct par exemple. Voir ex précédent pour contexte : cout<<(\*p).no; == cout << p->no; ou même cout<<(\*p).description; == cout<<p->description;  
  
-new Permet de créer un contenant du type désiré. Ex : new double; Il est alors important de sauvegarder l’adresse, car l’on a autrement aucune manière de la référencé. Donc, si double\* p; pour sauvegarder l’adresse de mon contenant on fait p = new double{40.45};. Il faut que le pointeur soit du même type que le contenant dont on sauvegarde l’adresse. On n’aura alors accès au contenant qu’avec l’adresse. new permet d’allouer dynamiquement un tableau : size\_t size = 10; cin>>size; int\* T = nullptr; T = new int[size];. Pour allouer des valeurs à notre tableau on peut faire T[0] = 100; ou même \*(T+1) = 200; qui lui ira dans la deuxième case de notre tableau.   
  
-delete Permet de supprimer un contenant, libère donc l’adresse qui lui était allouer. Si on se base sur l’exemple précédent, on ferait delete p; Pour supprimer le contenu de l’adresse. p contiendra toujours l’adresse, il n’y aura par contre plus de contenant à cette adresse. delete ne peut être utiliser que sur des pointeurs. Permet au final de libérer de l’espace en mémoire lorsque l’on a plus besoin de la variable. new et delete permette donc en quelque sorte de créer des variables temporaires. Après avoir delete le contenu d’une adresse, il est bonne pratique de mettre l’adresse contenu dans p à nul donc faire p=nullptr; Pour supprimer un tableau, il est important d’indiquer au programme que c’est un tableau, autrement il n’y aura que la première case du tableau qui sera supprimé. Pour ce faire : delete [] Tableau;  
  
  
**Concept de stockage de données**  
[  
-stack Une pile d’appelle qui prend en priorité le dernier arrivé en file. Un peu de la même manière qu’une fonction va exécuter en priorité la fonction qu’elle appelle avant de continuer son propre code. C’est un concept commun en programmation qui permet de stocker des données d’une manière spécifique.   
Explication : Il y a un pointeur important (top) qui pointe toujours vers l’item du haut de la pile. Au départ, la pile est vide et le pointeur (top) est nul (top = nullptr). L’ajout (push) d’un item se fait au-dessus de la pile et le pointeur (top) est ajusté. Le retrait (pop) d’un item se fait au-dessus de la pile et pointeur (top) est ajusté. Le dernier item ajouté sera toujours le premier item retiré (last in, first out). Chaque item possède un pointeur (below) vers l’item du dessous, qui sera nul pour le premier item de la pile. Un parcours de la pile débute au (top) et se poursuit en suivant le pointeur below de chaque item, il est donc impossible de rendre à un item directement. L’usage d’un pointeur temporaire d’item (it) peut s’avérer utile lors des manipulations de la pile. Les items retirés de la sont recyclés (delete).  
  
-queue Un peu de la même manière que l’on fait la file dans la vraie vie. On rajoute donc nos items à la fin de la queue. On traite donc en priorité le premier item qui a été ajouté. Premier arrivé, premier servi. C’est un concept commun en programmation qui permet de stocker des données d’une manière spécifique.  
Explication : Il y a deux pointeurs importants, soit front qui pointe toujours vers le premier item de la file et back qui pointe toujours vers le dernier item de la file. Au départ, la file est vide et les pointeurs (front) et (back) sont nuls (nullptr). L’ajout (push) d’un item se fait à l’avant (front) de la file, et le pointeur (front) est ajusté. Si la file contient un seul item, les pointeurs (back) et (front) pointent les deux sur celui-ci. Le premier item ajouté (push) sera toujours le premier item retiré (pop) (first in, first out). Le parcours dans la file débute à l’avant (front) et se poursuit en suivant le pointeur next de chaque item il est donc impossible de se rendre à un item directement. L’usage d’un pointeur temporaire d’item (it) peut ‘avérer utile lors des manipulations de la file. Les items retirés (pop) de la file sont recyclés (delete).  
  
-push et pop sont des termes qui représentent le déplacement d’objet dans un stack ou dans une queue. Push est l’action de rajouter un objet, au début pour un stack et à fin pour une queue. Pop est l’action de retirer un objet du stack ou de la queue, on ne peut retirer que l’objet qui est tout en haut du stack soit le dernier item qui a été ajouté, alors que pour une queue on ne peut retirer que le premier objet qui y avait été ajouté. Le dernier objet ajouter est celui sur le dessus de la pile pour le stack, et celui à la fin de la file pour une queue. C’est un concept commun en programmation qui permet de gérer les données stocker dans un stack ou une queue. Les deux méthodes sont similaires, mais la manière dont on gère l’information est simplement différente, de par son ordre.   
  
-liste Un peu comme une queue, à la seule différence qu’on peut rajouter et retirer des éléments où on le souhaite. On peut aussi s’y déplacer facilement.   
Explication : Il y a deux pointeurs importants pour la liste soit un pointeur (last) vers le dernier item. Un pointeur (first) vers le premier item. Chaque item possède un pointeur (next) vers l’item suivant de la liste, à moins d’petre le dernier. Au départ, la liste est vide et les deux pointeurs (first) et (last) sont nuls (nullptr). L’ajout (push) d’un item peut se faire au début comme à la fin de la liste. Le retrait (pop) d’un item se fait au début de la liste. Le parcours de la liste débute au début (first) et se poursuit en suivant le pointeur next de chaque item, il est donc impossible de rendre à un item directement. L’usage d’un pointeur temporaire d’item (it) peut s’avérer utile lors des manipulations de la liste. Les items retirés (pop) de la liste sont recyclés (delete).  
  
-liste doublement chainée Un peu comme une liste, à la seule différence que l’on peut aller dans la direction que l’on veut à l’intérieur de la liste.   
Explication : Il y a deux pointeurs importants pour la liste : (first) et (last). Chaque item possède deux pointeurs (prev) et (next) vers ses items adjacents dans la liste. Au départ, la liste est vide et les deux pointeurs (first) et (last) sont nuls (nullptr). L’ajout et le retrait d’un item peut se faire n’importe où dans la liste. Si la liste contient un seule item, les pointeurs (first) et (last) pointent sur celui-ci. Parcourir la liste peut se faire dans les deux directions : first 🡪 last ou last 🡪 first, il est impossible de se rendre à un item directement. L’usage d’un pointeur auxiliaire d’item (it) peut s’avérer utile lors des manipulations de la liste. Les items retirés de la liste sont recyclés (delete).  
  
PS : Les derniers points sont simplement des options possibles pour arranger des données dans un tableau dont la taille change constamment. Il est optimal de faire ce qui est mieux selon notre situation et d’adapter les différents arrangements selon nos besoins. Aller voir l’exemple du prof pour comment lui le code et des exemples photos de l’arrangement de données.  
]  
  
-#include permet d’avoir les déclarations présentent dans d’autres fichiers .h, de manière qu’il ne soit pas nécessaire de réécrire les déclarations dans chaque nouveau fichier où on veut l’inclure. Est un copieur de texte, il copie le texte du fichier inclut dans la commande. Le # indique que c’est une commande de pré-compilation.   
  
-#pragma once s’assure que les include ne fasse qu’une seul fois, de manière à ne pas importer plusieurs fois la même bibliothèque\header.   
  
-Il est possible de ranger nos fonctions dans différent cpp tant que celui-ci est inclus dans le projet. Permet d’utiliser des fonctions récurrentes sans avoir à les réécrire dans différent projet.   
  
-Un module est une unité de rangement, soit une combinaison d’un fichier cpp et un fichier h. En règle générale, on inclut les définitions dans le fichier cpp et on inclut les déclarations ou bien les prototypes dans le fichier h. On peut déclarer des fonctions, des types, des struct, des variables, tout ce que l’on souhaite en soit. On peut inclure d’autre header à l’intérieur d’un nouveau header qui aurait besoin de quelque chose qui serait uniquement disponible dans ce dit autre fichier. Si un header à besoin d’information provenant d’un autre header, mais que par exemple main, n’a pas besoin de ce dit premier fichier, on peut simplement inclure le deuxième header qui importera indirectement l’information donc il a besoin pour fonctionner dans main. Il est important que le cpp soit dans le même fichier que son header. On garde généralement tous les fichiers cpp et h dans un même fichier.   
  
-Processus de compilation en C++ : La première étape est la précompilation, de vérification ainsi que de la génération du code machine. On doit donc commencer par transformer le fichier cpp en cpp+ en intégrant le code que va inclure les commandes de précompilation tel que le include. Il doit ensuite vérifier s’il y a des erreurs dans le fichier. S’il n’y a aucune erreur il se doit de traduire le fichier cpp en code machine dans un fichier obj. La deuxième étape est l’édition des liens et l’assemblage de l’exécutable. Il doit donc s’assurer que tous les liens entre les différents fichiers soit valide. S’il n’y a aucune erreur, le compilateur peut enfin créer l’exécutable.  
  
-Une fonction privée est une fonction qui ne peut être utiliser quand dans le cpp originale où elle a été définie. On peut mettre le mot static devant le type de notre fonction pour s’assure que la définition soit invisible dans les autres cpp. Pour rendre une fonction publique, il faut inclure la déclaration dans le fichier h associer au dit cpp. Si on n’inclut pas le mot static devant le type de la fonction et que la déclaration n’est pas incluse dans le header, on peut tricher et déclarer la fonction que l’on veut utiliser dans le cpp qui l’utilisera.   
  
  
  
-extern permet de déclarer une variable globale Ex : extern int G; Devrait se retrouver dans le point h. extern est le contraire de static. extern=public alors que static=private. Donner une valeur à la variable déclarer, transforme la déclaration en définition. Le mot extern est sous-entendu pour toute les variables globales. Les constantes sont, au contraire, par défaut static.  
  
-Lorsque l’on met static devant une variable locale à l’intérieur d’une fonction, elle devient alors globale. C’est-à-dire que la variable ne sera pas détruite et créer à chaque utilisation de la fonction. Elle reste toujours en mémoire et ne pourra être vu que par la fonction dans laquelle elle a été créée.   
  
-Un tableau de char à besoin d’un caractère butoir pour qu’il finisse correctement soit ‘\0’. À utiliser pour la lecture de fichier binaire pour le moment.   
  
-Pour convertir une variable d’un type à un autre il faut mettre le type de la nouvelle variable entre parenthèse, suivit du nom de la variable à convertir. Ex : double test; (int)test;  
  
-Récursivité. La forme générale d’une fonction récursive est f(n) = n op f(n-1) où n est l’exemplaire à évaluer, où op est une opération quelconque, complexe ou non, où f(n-1) est une solution pour l’exemplaire plus petit (n-1). Si j’ai une solution pour f(n-1), alors je peux obtenir une solution pour f(n) en appliquant n et une opération sur f(n-1). Si on trouve une solution pour le plus petit problème, alors trouver une solution pour le plus gros sera très simple grâce à une opération. Un exemple de fonction récursive est la factorielle. La factorielle d’un entier naturel n (n!) est le produit des nombres entiers strictement positif inférieurs ou égaux à n. La formule en soit est n!=n\*(n-1)\*(n-2)\*…\*1;, mais en simplifiant on arrive à n!=n\*(n-1)!; on sait aussi que 0!=1; Pour le calculer de manière récursif, on utilise généralement la forme suivant :  
void fct\_recursive(size\_t n)  
{  
 if(n == 0 || n==1) //Une condition qui permettra d’arrêter le rappel récursif  
 {  
 return 1;  
 }  
 else  
 {  
 return n \* fct\_recursive(n – 1);  
 }   
}  
La fonction s’appelle donc jusqu’à arriver à 1, pour ensuite envoyer cette information à l’appelle de la fonction précédente jusqu’à revenir à l’original pour trouver notre résultat. Permet finalement d’éviter le stack overflow que l’on pourrait avoir avec une simple boucle for qui aurait été la solution itérative.   
Au final la récursivité est d’utiliser une fonction qui s’appelle elle-même à répétition pour alléger le calcul.