**Langage de programmation**

**HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG**

**Bloc 1**

Atelier 2 – Alternatives

Objectifs

* Distinguer les différents types de conditions
* Employer une alternative simple
* Employer une alternative double
* Employer une alternative multiple
* Tester un programme

[Introduction 1](#_Toc144795392)

[A. Qu'est-ce qu'une condition ? 2](#_Toc144795393)

[B. Alternatives simples 5](#_Toc144795394)

[C. Alternatives doubles 8](#_Toc144795395)

[D. Alternatives multiples 10](#_Toc144795396)

[E. Tester les programmes 11](#_Toc144795397)

# Introduction

Dans cet atelier, vous allez apprendre à modifier l’ordre dans lequel les instructions sont exécutées, appelé le **flux des instructions**, au moyen d’instructions de **contrôle de flux**. Les instructions de contrôle de flux permettent, par exemple, de faire un choix entre deux actions, répéter une ou plusieurs actions…

La première catégorie d’instructions de contrôle de flux est l’alternative.

Pour écrire des alternatives, il faut être d’abord capable d’écrire des conditions.

Dans ce document, plusieurs conventions sont utilisées :

* les mots gras désignent des termes de vocabulaire liés à l’**informatique en général**.
* les mots soulignés et gras désignent des termes de vocabulaire directement liés aux cours de **programmation**.
* le logo signifie que vous avez quelque chose à réaliser.
* le logo est associé aux cadres présentant certaines conventions.
* le logo est associé aux cadres présentant les éléments liés à la propreté/lisibilité du code (*clean code*).

# Qu'est-ce qu'une condition ?

Une **condition** est une expression booléenne, c’est-à-dire, comme vous l’avez vu dans le cadre du **module 1**, une expression dont la valeur est vraie ou fausse.

Ce type de valeur est le résultat d’une opération de comparaison. Pour comparer des valeurs, qu’elles soient littérales, mémorisées dans des variables, le résultat de l’appel d’une fonction… on utilise les **opérateurs de comparaison** que vous connaissez déjà. Les valeurs comparées peuvent être a priori de n’importe quel type (nombre ou caractère). Cependant, en C, il faut que les deux valeurs de la comparaison soient du même type !

## Condition simple

Une condition simple est une comparaison entre deux valeurs : un littéral, une valeur mémorisée dans une variable, une expression…

La comparaison de deux littéraux n’a pas vraiment de sens dans un programme, bien que vous en ayez testé quelques-unes pour comprendre la notion d’expression booléenne. En effet, puisque vous êtes capables de déterminer le résultat par vous-même, il est inutile de demander à la machine de le faire !

Créez un nouveau fichier source conditions.c et ajoutez-y la fonction principale. Écrivez (dans la fonction principale) les conditions qui correspondent aux propositions suivantes (en choisissant des noms de variables adéquats) :

* La température est plus grande que 15.
* La somme est égale à 120.
* Le nombre est différent de 0.
* Le nombre est pair.

N’hésitez pas à discuter de vos réponses avec d’autres étudiants…

Pour ne pas répéter une même condition plusieurs fois, on peut passer par une variable de manière à mémoriser le résultat de l’évaluation de la condition.

bool estMasculin = sexe == 'M';

Cette instruction déclare la variable estMasculin et lui affecte le résultat de l’évaluation de la condition, c’est-à-dire true si le sexe vaut 'M' et false sinon.

Cette écriture est possible car, souvenez-vous, dans la définition du langage C, les opérateurs de comparaison ont une priorité plus élevée que l’opérateur d’affectation.

Faites-en sorte de mémoriser le résultat de chacune des conditions que vous avez déjà rédigée dans une variable en prenant à nouveau soin de choisir un nom de variable non ambigu.

Le résultat de ces deux étapes devrait ressembler à ce qui est proposé ci-dessous.

bool plusDe15Degres = temperature > 15;

bool sommeEgaleA120 = somme == 120;

bool nonNul = nombre != 0;

bool pair = nombre % 2 == 0;

Complétez ce code pour que chacune des variables utilisées dans les expressions de droites soient déclarées et initialisées avec une valeur de votre choix… au bon endroit ! Ajoutez ensuite, pour chacune des expressions de gauches, l’instruction suivante en la complétant correctement.

printf("La condition est evaluee a %s", … ? "vrai" : "faux");

Vous devriez avoir le programme suivant :

void main(void) {

double temperature = 17;

int somme = 80;

int nombre = 5;

bool plusDe15Degres = temperature > 15 ;

bool sommeEgaleA120 = somme == 120 ;

bool nonNul = nombre != 0;

bool pair = nombre % 2 == 0;

printf("La condition est evaluee a %s",   
 plusDe15Degres ? "vrai" : "faux");

printf("La condition est evaluee a %s",   
 sommeEgaleA120 ? "vrai" : "faux");

printf("La condition est evaluee a %s", nonNul ? "vrai" : "faux");

printf("La condition est evaluee a %s", pair ? "vrai" : "faux");

}

## Condition composée

Enfin, certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées grâce à une simple comparaison. En effet, dans certaines conditions, se cachent plusieurs conditions. On parle de conditions composées. Les conditions simples y sont reliées par des **opérateurs logiques[[1]](#footnote-1)** : ET, OU et NON.

En C, leur syntaxe est particulière : &&, || et !. En effet les opérateurs & et | sont déjà utilisés pour les opérations sur les nombres au niveau binaire qui sont vues dans le cours d’« Architecture des ordinateurs ».

Si on veut vérifier qu’une cote est entre 16 et 18 inclus, on écrit l’expression suivante :

cote >= 16 && cote < 18

Voici à nouveau le tableau des prio rités mis à jour :

|  |  |
| --- | --- |
| -(unaire) ++ -- (*type*) | ← |
| \* / % | → |
| + -(binaire) | → |
| < <= > >= | → |
| == != | → |
| && | → |
| || | → |
| ?: | ← |
| = += -= \*= /= %= | ← |

Créez un nouveau fichier source informations.c et ajoutez-y la fonction principale. Écrivez (dans la fonction principale) les conditions qui correspondent aux propositions suivantes (en choisissant des noms de variables adéquats) :

* L’âge est positif et inférieur à 18.
* Le sexe est soit 'M', soit 'F'.
* L’année est bissextile.

Si vous ne connaissez pas les propriétés d’une année bissextile, allez chercher sur le Net, mais ne commencez pas à recopier n’importe quoi !

Complétez ce fichier source pour qu’il demande et récupère l’âge et le sexe de l’utilisateur, et qu’il indique, après chaque obtention, si ces informations sont valides au moyen des conditions que vous avez écrites ci-avant.

Vous devriez avoir un bout de code qui correspond à ce qui est proposé ci-dessous.

int age;

char sexe;

printf("Age : ");

scanf\_s("%d", &age);

printf("%s", age > 0 && age < 18 ? "Valide" : "Invalide");

printf("Sexe : ");

scanf\_s("%c", &sexe, 1);

printf("%s", sexe == 'M' || sexe == 'F' ? "Valide" : "Invalide");

Dans ce même fichier, ajoutez les instructions qui permettent de demander et de récupérer une année et qui indique si c’est une année bissextile.

# Alternatives simples

Maintenant que vous savez écrire des conditions, vous allez pouvoir les utiliser.

En effet, une alternative est une instruction qui permet de choisir quelles instructions seront exécutées après l’évaluation d’une condition.

Il existe plusieurs formes d’alternatives : les alternatives simples, les alternatives doubles et les alternatives multiples.

La forme la plus simple d'alternative est :

**SI** *condition* **ALORS** *bloc d'instructions*

Cela signifie que, après l’évaluation de la condition, si sa valeur est true, le bloc d’instructions qui suit le **ALORS** est exécuté. En revanche, dans le cas où cette valeur est false, le bloc d’instructions est ignoré.

Dans les deux cas, les instructions situées juste après l’alternativeseront exécutées.

Comme vous l’avez vu en « Principes de programmation », le diagramme d’actions correspondant est le suivant.

┌── **if** (*condition*)  
│ *instructions si la condition est vraie*

└──

En C, l’alternative simple est composée d’une **instruction d’en-tête**, celle qui contient if suivi de la condition entre parenthèses ().

if (condition)

{

// bloc d’instructions si true

}

Voici quelques instructions qui permettent de déterminer si, sur base de la cote sur 20 reçue à un examen, vous êtes en échec.

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote < 10)   
{

printf("Aie, c'est un echec !");

}

Écrivez ces instruction dans un nouveau fichier source cote.c, sans oublier d’ajouter les éléments nécessaires à son exécution.

Exécutez ce programme et choisissez des valeurs afin de vérifier tous les cas envisageables, en sachant qu’on suppose que l’utilisateur entre bien un entier. Il ne s’agit pas de vérifier le format de la valeur, mais juste de la comparer à 10.

Ici, il semble en effet utile de vérifier le cas d’un entier inférieur à 10, d’un entier égal à 10 afin de s’assurer que ce cas particulier est bien géré, et une valeur supérieure à 10. La notion de « test » est approfondie un peu plus loin dans cet atelier. N’hésitez pas à revenir à ce point lorsque vous en saurez plus sur les tests…

Indentation

Il est maintenant temps de reparler de la notion d’**indentation** rapidement abordée dans le **module O**. En effet, vous avez vu que toutes les instructions d’un même bloc d’instructions sont « encadrées » par des accolades. Jusque maintenant le seul bloc auquel vous avez eu affaire est le bloc de la fonction principale main. Maintenant que vous pouvez ajouter des blocs, il est nécessaire de savoir que chaque séquence d’instructions doit être décalée d’une tabulation par rapport à l’instruction d’entête du bloc.

void main (void)

**{**

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote < 10)

**{**

printf("Aie, c'est un echec !");

}

}

Une autre manière d'écrire est de mettre l’accolade ouvrante sur la même ligne que l’instruction d’entête.

void main (void) **{**

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote < 10) **{**

printf("Aie, c'est un echec !");

}

}

Ces deux notations sont équivalentes, vous pouvez donc choisir celle qui vous convient le mieux. En revanche il est inélégant, dans le même programme, d’utiliser ces deux notations conjointement… Soyez cohérents ! Pour une question de mise en page, c’est la seconde notation qui y est adoptée dans les exemples.

Une attention toute particulière est portée à l’indentation de vos programmes lors des évaluations car c’est un des aspects principaux de la lisibilité de votre code.

Il est à noter que, dans le cas des instructions de contrôle de flux, s’il n’y a qu’une seule instruction qui en dépend, on peut omettre les accolades.

void main (void) {

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote < 10)

printf("Aie, c'est un echec !");

}

Veillez à n’utiliser cette notation que lorsque c’est justifié et pas juste par « paresse ».

Dans le fichier source d20.c, dans la fonction principale, écrivez les instructions qui permettent de demander le résultat d’un lancer de dé à 20 faces à l’utilisateur.   
Si le résultat du lancer de dé est supérieur ou égal à 10, on considère que c’est un bon résultat. Dans ce cas, le message "Beau lancer ! " est affiché, sinon rien n’est affiché.

Attention, lisez bien l’énoncé ! Le but n’est pas de vérifier que la valeur est bien entre 1 et 20 (inclus), c’est une condition supposée déjà respectée par l’utilisateur.

Commencez par faire un diagramme d’actions ou par écrire la structure de votre programme sur une feuille afin de ne pas travailler par essai/erreur. N’hésitez cependant pas à vérifier que votre programme fait bien ce que vous espériez. Cela ne devrait pas poser de soucis si vous avez commencé par réfléchir à la logique.

## Pourquoi ça fonctionne ?

Afin de vous rendre compte de l’importance des noms de variables et des messages affichés, essayez de comprendre ce que ce morceau de code est supposé faire !

tot = x \* y ;

if (x > 10)

printf("Plus grand que 10");

tot = x \* 0.9;

printf("Total : %.2f", tot);

Si vous ne comprenez pas, c’est normal… Ce morceau de code est erroné et incompréhensible !

Remarquez que, sur la version ci-dessous, les noms des variables correspondent à la valeur qu’elles contiennent. Vous devriez avoir un peu plus facile à comprendre ce dont il s’agit. Le message est également adapté. Cependant, il reste une erreur algorithmique !

Assurez-vous d’avoir compris la notion de bloc d’instructions. Sans exécuter ce bout de code, pensez-vous que la réduction est appliquée à bon escient ? Pourquoi ?

prixTotal = nbRepas \* prixRepas;

if (nbRepas > 10)

printf("Reduction pour plus de 10 repas");

prixTotal = prixTotal \* 0.9;

printf("A payer : %.2f euros", prixTotal);

Si vous n’arrivez pas à trouver l’erreur, écrivez le tout dans nouveau fichier source reductionRepas.c :

* Ajoutez-y les instructions nécessaires à son exécution ainsi que celles permettant de demander à l’utilisateur le nombre de repas et le prix d’un repas, et de les récupérer dans les variables adéquates.
* Exécutez le programme en soumettant un nombre de repas supérieur à 10. Le montant final devrait être correct.
* Essayez maintenant avec un nombre de repas inférieur ou égal à 10. Que constatez-vous ?
* Mettez-vous à la place du restaurateur et imaginez sa réaction…
* Il y a deux caractères à ajouter, lesquels ? Corrigez le programme afin qu’il fasse ce qu’il est supposé réaliser.

Une fois de plus assurez-vous d’avoir compris « pourquoi ça fonctionne » et pas juste « comment ça fonctionne ».

# Alternatives doubles

Comme vous l’avez lu ci-avant, il existe plusieurs formes d’alternatives : les alternatives simples, les alternatives doubles et les alternatives multiples.

La deuxième forme d'alternative envisagée est l’alternative double :

**SI** *condition* **ALORS** *bloc d'instructions* **SINON** *bloc d’instructions*

Cela signifie que, après l’évaluation de la condition, si sa valeur est true, le bloc d’instructions qui suit le **ALORS** est exécuté. En revanche, dans le cas où cette valeur est false, c’est le bloc d’instructions qui suit le **SINON** qui est exécuté.

Dans les deux cas, les instructions situées juste après l’alternative seront exécutées.

Comme vous l’avez vu en « Principes de programmation », le diagramme d’actions correspondant est le suivant.

┌── **if** (*condition*)  
│ *instructions si la condition est vraie*

├── **else**

│ *instructions si la condition est fausse*

└──

En C, l’alternative double est composée de deux parties :

* la première partie est introduite par l’**instruction d’en-tête**, celle qui commence par le mot réservé if suivi de la condition entre parenthèses (). Le bloc d’instructions qui suit cette instruction d’entête est exécuté dans le cas d’une évaluation de la condition à true.
* la deuxième partie est introduite par le mot réservé else. Le bloc d’instructions qui suit le else est exécuté dans le cas d’une évaluation de la condition à false.

if (condition)

{

// bloc d’instructions si true

}

else

{

// bloc d’instructions si false

}

Le but des instructions qui suivent est de déterminer, sur base de la cote sur 20 reçue à une évaluation, si vous avez réussi ou si vous êtes en échec.

Une façon de faire est d’écrire ce qui suit :

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote >= 10) {

printf("Super ! Tu as reussi !");

}

if (cote < 10) {

printf("Aie, c'est un echec !");

}

Cette façon de faire ne correspond pas du tout à ce qu’on attend de vous !

Rappelez-vous que les conditions pour qu’un code soit considéré comme « valide » n’est pas juste le fait qu’il tourne, mais il faut qu’il soit aussi propre et efficace.

À votre avis quelle est la qualité que ce code ne respecte pas et pourquoi ?

La façon correcte de réaliser ce qui est demandé dans ce cas-ci est d’écrire l’alternative comme montré ci-dessous.

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote >= 10) {

printf("Super ! Tu as reussi !");

} else {

printf("Aie, c'est un echec !");

}

Modifiez le fichier source cote.c de manière à intégrer cette modification. Exécutez ce programme et choisissez des valeurs afin de vérifier que vous tenez compte de tous les cas envisageables. La notion de « test » est approfondie un peu plus loin dans cet atelier. N’hésitez pas à revenir à ce point lorsque vous en saurez plus sur les tests…

Normalement dans le fichier source d20.c, vous devriez avoir notamment écrit les instructions suivantes.

if (resultat >= 10) {

printf("Beau lancer !")

}

Ajoutez à ce fichier les instructions qui permettent d’afficher le message « Relancez le de... » si le résultat du lancer n’est pas supérieur ou égal à 10.

Exécutez le programme et vérifiez à nouveau que le message affiché correspond à vos attentes :

* Si ce n’est pas le cas, pourquoi ? Ne rajoutez pas n’importe quoi juste pour que ça marche ! Analysez le code afin de comprendre pourquoi il ne donne pas le résultat attendu.
* Si c’est le cas, vérifiez que ce code est propre et lisible. Avez-vous utilisé la bonne structure de contrôle ?

Attention ! N’oubliez pas de lire ce qui est demandé et de répondre à la question plutôt que de faire ce que vous avez envie de faire.

# Alternatives multiples

Reprenez le fichier cote.c et modifiez-le comme ci-dessous.

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote > 10)

{

printf("Super ! Tu as reussi !")

}

if (cote < 10)

{

printf("Aie, c'est un echec !")

}

else

{

printf("Ouf ! De justesse...")

}

Lisez bien ce programme pour essayer de comprendre ce qu’on en attend. Pensez-vous que les instructions proposées vont correspondre à ce qu’on attend ?

Exécutez ensuite le programme et vérifiez à nouveau que le message affiché correspond à vos attentes. Entrez d’abord une cote inférieure à 10, ensuite une cote égale à 10 et enfin une cote supérieure à 10. Les deux premiers cas ne devraient pas poser problème. Mais que se passe-t-il lorsque vous proposez une cote supérieure à 10 ? Pourquoi ?

La solution la plus adéquate pour éviter cette erreur est d’utiliser des *alternatives imbriquées*. Plusieurs solutions s’offrent à vous dans le choix de l’ordre des conditions vérifiées. En voici une :

int cote;

printf("Cote : ");

scanf("%d", &cote);

if (cote > 10) {

printf("Super ! Tu as reussi !")

} else {

if (cote < 10) {

printf("Aie, c'est un echec !")

} else {

printf("Ouf ! De justesse...")

}

}

Le deuxième if correspond au bloc d’instructions à réaliser lorsque la première condition cote > 10 est fausse, c’est-à-dire que cote <= 10. Faites bien attention à l’indentation !

Testez à nouveau ce programme et assurez-vous de comprendre le flux des instructions.

Modifier une dernière fois le fichier source d20.c pour, qu’en plus de ce qui a déjà été fait, le message

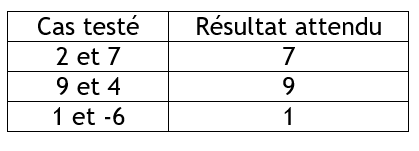
* « Echec critique » s’affiche dans le cas d’un résultat à 1, et
* « Reussite critique » s’affiche dans le cas d’un résultat à 20.

# Tester les programmes

Tant que vous êtes dans l’aspect « test de programme », il semble important de préciser quelques éléments à ce sujet !

## Des tests ne vont jamais remplacer une spécification précise

Si c’est vrai qu’on peut utiliser des tests pour expliciter ce qu’on attend du code, seuls ils ne suffiront jamais à décrire précisément ce que le code doit faire. Il pourra toujours rester un doute.  
Par exemple, supposez qu’on vous demande d’écrire les instructions qui sur base de deux entiers nb1 et nb2 garnissent la variable resultat de façon à correspondre aux cas de test suivants.



Les deux implémentations suivantes passent avec succès ces trois tests.

### Implémentation #1 : donne le maximum parmi nb1 et nb2

if (nb1 > nb2) {

resultat = nb1;

} else {

resultat = nb2;

}

### Implémentation #2 : donne nb1 s’il est impair, sinon nb2

if (nb1 % 2 == 1) {

resultat = nb1;

} else {

resultat = nb2;

}

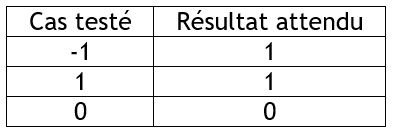
Cet exemple montre bien que les tests seuls ne suffisent pas pour déterminer précisément ce qu’on veut que les instructions fassent. Seul un énoncé clair et précis permet de lever toute ambiguïté. Il est donc important de bien réfléchir avant d’écrire n’importe quoi et de tenir compte de ce qui vous est enseigné en « Principes de programmation » et en « Outils mathématiques pour l’informatique ».

## Ce n’est pas parce que tous les tests réussissent que le code est correct

Si les tests révèlent un problème, c’est qu’il y a un bug quelque part dans le code… Cependant, l’inverse n’est pas vrai : un programme qui passe avec succès tous les tests pourrait quand même avoir des bugs. Les tests peuvent révéler certains bugs, mais ils ne permettent pas de garantir que le code est correct !

Par exemple, on vous demande d’écrire les instructions qui déterminent la valeur absolue d’un entier x et la mettent dans resultat.

Un jeu de tests acceptable pourrait être le suivant (on teste un cas positif, un cas négatif, et on n’oublie pas le cas du zéro).



Seulement, les trois implémentations suivantes passent avec succès tous ces tests… Or, seule l’une d’entre elles remplit bien le contrat de départ, à savoir déterminer la valeur absolue d’un entier !

### Implémentation #1 : calcule |x|

if (x < 0) {

resultat = x \* -1 ;

} else {

resultat = x ;

}

### Implémentation #2 : calcule x2

resultat = x \* x

### Implémentation #3 : indique si x est non nul (0 pour faux, 1 pour vrai)

if (x == 0) {

resultat = 0;

} else {

resultat = 1;

}

La principale leçon à retenir est que, même si les tests peuvent être utiles et déceler des bugs, ils ne remplacent pas la réflexion qui doit avoir lieu lors de la conception du programme. Bien penser la structure du programme pour que tous les cas y soient traités correctement est la meilleure manière d’arriver à un code correct !

Relisez les différents énoncés ci-dessus et vérifiez que les programmes ont été bien pensés et que les tests que vous avez effectués sont judicieux.

1. Ils n’ont pas exactement la même signification que les connecteurs logiques vus en logique des propositions dans le cadre du cours de « Outils mathématiques pour l’informatique ». Plus d’informations à ce sujet vous seront transmises en temps voulu. Pour les curieux il s’agit de l’évaluation paresseuse… [↑](#footnote-ref-1)