COURS 4 - STRUCTURES DE CONTRÔLE ET INTERACTIONS AVEC L'UTILISATEUR

G. Bianchi, G. Blin, A. Bugeau, S. Gueorguieva, R. Uricaru

2015-2016

Programmation 1 - uf-info.ue.prog1@diff.u-bordeaux.fr

université de BORDEAUX

EXPRESSION ET INSTRUCTION

- Une expression est évaluable et produit une valeur (donc typée)
  - Constante
  - Variable
  - Expression arithmétique
  - Appel de fonction

EXPRESSION ET INSTRUCTION

▷ Une expression est évaluable et produit une valeur (donc typée)

Opération	Description	Valeur produite
x()	Appel de fonction	Valeur retournée
()	Groupe	Evaluation du groupe
* / %	Multiplier, diviser, modulo (reste)	Résultat arithmétique
+ -	Addition, soustraction	Résultat arithmétique
> >=	Plus grand (ou égal)	Résultat du test
< <=	Plus petit (ou égal)	Résultat du test
== !=	Égalité, différence (non égalité)	Résultat du test
=	Affectation	Valeur affectée

▷ Utiliser les parenthèses pour définir les priorités

EXPRESSION ET INSTRUCTION

- Une instruction est une étape exécutable du programme se terminant par le caractère `';'` dans le code (qui ne produit rien)
  - Déclaration de variable
  - Structure conditionnelle (test)
  - Structure répétitive (boucle)
  - Retourner une valeur
  - Une expression
- Les instructions peuvent être regroupées dans un bloc délimité par { et }

Comportement impératif par défaut

▷ Un programme commence toujours par exécuter la première instruction de son point d'entrée (i.e., la fonction `main`)

▷ Par défaut, une fois cette instruction exécutée, il exécute la suivante, etc.

▷ On peut modifier ce comportement par défaut grâce à des structures conditionnelles et/ou répétitives

LES CONDITIONS

VALEURS LOGIQUES

▷ En C, par convention, `0` représente la valeur logique faux, tout le reste représente la valeur logique vrai

▷ Depuis C99, les constantes `true` et `false` sont introduites et définies dans la librairie `stdbool.h`

▷ Il existe un type booléen nommé `bool`

Opérateurs logiques

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(true&&false);//vaut false
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(4&&0);//vaut false
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(4&&6);//vaut true
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(4||6);//vaut true
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(0||0);//vaut false
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(!17);//vaut false
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=(!0);//vaut true
```

OPÉRATEURS LOGIQUES

- Le ET logique est représenté par `&&`
- Le OU logique est représenté par `||`
- Le NON logique est représenté par `!`

a	b	a&&b	a||b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

```
bool b=((!0)&&(4||0));//vaut true
```

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠ Ne fonctionne QUE pour les nombres

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠Ne fonctionne QUE pour les nombres

```
bool b=(1==2);//vaut false
```

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠Ne fonctionne QUE pour les nombres

```
bool b=(3==3);//vaut true
```

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠Ne fonctionne QUE pour les nombres

```
bool b=(1!=2);//vaut true
```

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠Ne fonctionne QUE pour les nombres

```
bool b=(1<=3);//vaut true
```

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠Ne fonctionne QUE pour les nombres

```
bool b=(1<1);//vaut false
```

Opérateurs de comparaison de nombres

- L'égalité est représentée par `==`
- L'inégalité est représentée par `!=`
- L'infériorité est représentée par `<` ou `<=`
- La supériorité est représentée par `>` ou `>=`
- ⚠ Ne fonctionne QUE pour les nombres

```
bool b=('A'<'C');//vaut true ⚠
```

De Morgan et la négation

- $\overline{A \vee B} = \overline{A} \wedge \overline{B}$
  - $\overline{\text{Dire oui} \vee \text{Dire non}} = \overline{\text{Dire oui}} \wedge \overline{\text{Dire non}}$
- $\overline{A \wedge B} = \overline{A} \vee \overline{B}$
  - $\overline{\text{Beurre} \wedge \text{Argent}} = \overline{\text{Beurre}} \vee \overline{\text{Argent}}$

De Morgan et la négation

- $\overline{A \vee B} = \overline{A} \wedge \overline{B}$
  - $\overline{\text{Dire oui} \vee \text{Dire non}} = \overline{\text{Dire oui}} \wedge \overline{\text{Dire non}}$
- $\overline{A \wedge B} = \overline{A} \vee \overline{B}$
  - $\overline{\text{Beurre} \wedge \text{Argent}} = \overline{\text{Beurre}} \vee \overline{\text{Argent}}$
- `bool b=((!(a||b))==((!a)&&(!b)));//vaut true`

De Morgan et la négation

- $\overline{A \vee B} = \overline{A} \wedge \overline{B}$
  - $\overline{\text{Dire oui} \vee \text{Dire non}} = \overline{\text{Dire oui}} \wedge \overline{\text{Dire non}}$
- $\overline{A \wedge B} = \overline{A} \vee \overline{B}$
  - $\overline{\text{Beurre} \wedge \text{Argent}} = \overline{\text{Beurre}} \vee \overline{\text{Argent}}$
- `bool b=((!(a||b))==((!a)&&(!b)));//vaut true`
- `bool b=((!(a&&b))==((!a)||(!b)));//vaut true`

EVALUATION PARESSEUSE

- Les opérateurs `&&` et `||` sont paresseux
  - L'évaluation des expressions s'effectue de gauche à droite
  - Elle s'arrête dès que possible
- `bool b=((a=0) && (b=c));//vaut false`
- `bool b=((a=1) || (b=c));//vaut true`
- Dans les deux cas, `b=c` n'est pas évaluée ⚠

EVALUATION PARESSEUSE

▷ Les opérateurs `&&` et `||` sont paresseux

▷ L'évaluation des expressions s'effectue de gauche à droite

▷ Elle s'arrête dès que possible

▷ `bool b=((a=0) && (b=c));//vaut false`

▷ `bool b=((a=1) || (b=c));//vaut true`

▷ Dans les deux cas, `b=c` n'est pas évaluée ⚠

▷ ⚠Recommandation: éviter de comparer des variables de types différents (pour l'instant, on en reparlera plus tard !)

STRUCTURES CONDITIONELLES

IF ... ELSE ...

```
if(expression){
    instructions1;
}else{
    instructions2;
}
```

▷ Si l'évaluation de l'`expression` produit une valeur vraie (i.e., différente de zéro) alors on exécute les `instructions1` sinon on exécute les `instructions2`

IF ... ELSE ...

▷ Exemple

```
int a=5, b=3, max=0;

if (a > b){
    max = a;
}else{
    max = b;
}
printf("Le max est: %d\n", max);
```

IF ...

▷ La partie ”sinon” est optionnelle

```
if(expression){
    instructions1
}
```

```
int a=5, b=3, max=b;

if (a > b){
    max = a;
}
printf("Le max est: %d\n", max);
```

IF ... ELSE ...

▷ L'imbrication est possible

```
int note_cc=10, note_exam=4, note_rattrapage=10;
if((note_cc+note_exam)/2>=10){
    printf("UE validee en session 1.");
}else{
    if((note_cc+note_rattrapage)/2>=10){
        printf("UE validee en session 2.");
    }else{
        printf("UE non validee.");
    }
}
```

IF ... ELSE IF ... ELSE ...

- L'imbrication est possible
- Syntaxe "`else if`"*

```
int note_cc=10, note_exam=4, note_rattrapage=10;
if((note_cc+note_exam)/2>=10{
    printf("UE validee en session 1.");
}else if((note_cc+note_rattrapage)/2>=10{
    printf("UE validee en session 2.");
}else{
    printf("UE non validee.");
}
```

* ⚠ moins lisible

STRUCTURES RÉPÉTITIVES

FOR ...

```
for(inst_init; expression; inst_evolution){
    instructions;
}
```

▷ Faire `inst_init`

▷ Tant que `expression` produit une valeur vraie (i.e., différente de zéro), faire

  ▷ `instructions`

  ▷ `inst_evolution`

FOR ...

▷ Exemple classique

```
int i;
for(i=0; i<10; i=i+1){
    printf("i=%d\n",i);
}
```

FOR ...

▷ Exemple avec double initialisation

```
int i, n;
for(i=0, n=get_max(); i<n; i=i+1){
    printf("i=%d\n",i);
}
```

FOR ...

▷ On peut omettre `instructions` mais pas les accolades

```
int tab[] = {1, 2, 3, 2, 5};
int i, size = 5, n=2;
for(i=size-1; (i>=0) && (tab[i]!=n);i=i-1){}
printf("last position of %d is %d\n",n,i);
```

▷ L'absence d'`expression` génère une boucle infinie

▷ Cela peut être souhaitable, mais il faut savoir arrêter une boucle infinie (détails fournis après)

ATTENTION À L'EXPRESSION

- ⚠ L'`expression` est évaluée à chaque tour de boucle
- Attention aux complexités cachées ...

```
int i;
for(i=0; i<fct_quadratique(); i=i+1){
    printf("%d",i);
}
```

  - Exécution en temps cubique

ATTENTION À L'EXPRESSION

- ⚠ L'`expression` est évaluée à chaque tour de boucle
- Attention aux complexités cachées ...

```
int i,n;
for(i=0, n=fct_quadratique(); i<n; i=i+1){
    printf("%d",i);
}
```

  - Exécution en temps quadratique

WHILE ... DO ...

```
while(expression){
    instructions;
}
```

▷ Tant que `expression` produit une valeur vraie (i.e., différente de zéro), faire

  ▷ `instructions`

▷ Seule l'expression est obligatoire

▷ Si l'`expression` produit une valeur fausse, on ne rentre pas dans la boucle

WHILE ... DO ...

```
int i=0;
int j=10;
while(i < j){
   printf("i is still lower than j\n");
   i=i+1;
   j=j-1;
}
```

▷ La partie **initialisation** se trouve avant la boucle

▷ La partie **evolution** se trouve à l'intérieur de la boucle

WHILE VS FOR

- Question de lisibilité

```
int i;
for(i=0; i<10; i=i+1){
    printf("i=%d\n",i);
}
```

```
int i=0;
while(i<10){
    printf("i=%d\n",i);
    i=i+1;
}
```

DO ... WHILE ...

```
do{
    instructions;
}while(expression);
```

▷ Faire `instructions`

▷ Tant que `expression` produit une valeur vraie (i.e., différente de zéro), faire

  ▷ `instructions`

▷ A utiliser quand on doit effectuer au moins une fois `instructions`

DO ... WHILE ...

```
int i=0;
do{
    printf("i=%d\n",i);
    i=i+1;
}while(i<10);
```

▷ ⚠le ';' après l'`expression` est obligatoire

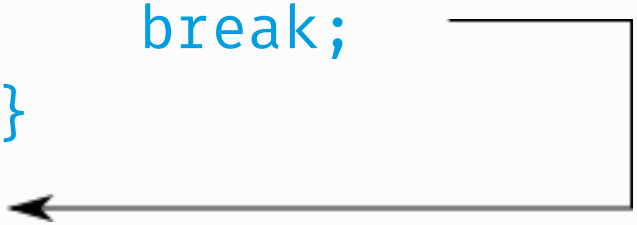
CONTRÔLE DU DÉROULEMENT

HAVE A BREAK, HAVE A KITKAT

▷ L'instruction **break** termine l'exécution de l'instruction **do**, **for**, **switch** ou **while** englobante la plus proche dans laquelle elle figure

▷ Utile dans certains cas d'erreur ou si le résultat d'un calcul est connu avant la fin

```
int tab[]={1, 2, 3, 2, 5};
int i, size=5, n=2;
for(i=0; i<size; i=i+1){
    if(tab[i]==n){
        break;
    }
}
printf("First position of %d is %d\n",n,i);
```



CONTINUE

▷ L'instruction **continue** passe à l'itération suivante de l'instruction **do**, **for**, ou **while** englobante la plus proche dans laquelle elle figure, en ignorant toutes les instructions restantes dans le corps de l'instruction répétitive

```
int tab[]={1, 2, 3, 2, 5};
int i=0, size=5, ignore=2, sum=0;;
while(i<size){
    i=i+1;
    if(tab[i-1]==ignore){
        continue;
    }
    sum=sum+tab[i-1];
}
printf("Sum ignoring %d is %d\n",ignore,sum);
```

CHOIX

▷ Les instructions `switch` et `case` aident à contrôler les opérations conditionelles sur des constantes entières

▷ Plus élégant que plein de `if ... else ...`

```
switch(expression){
    case const1: instructions1; break;
    case const2: instructions2; break;
    ...
    default : instructions;
}
```

▷ Si l'`expression` est égale à une constante particulière (e.g., disons `const2`) alors les instructions qui suivent le `case` correspondant sont exécutées

CHOIX

▷ Les instructions `switch` et `case` aident à contrôler les opérations conditionelles sur des constantes entières

▷ Plus élégant que plein de `if ... else ...`

```
switch(expression){
    case const1: instructions1; break;
    case const2: instructions2; break;
    ...
    default : instructions;
}
```

▷ Si l'`expression` n'est égale à aucune des constantes alors les instructions qui suivent `default` sont exécutées

CHOIX

▷ Les instructions **switch** et **case** aident à contrôler les opérations conditionelles sur des constantes entières

▷ Plus élégant que plein de **if ... else ...**

```
switch(expression){
    case const1: instructions1; break;
    case const2: instructions2; break;
    ...
    default : instructions;
}
```

▷ Sans **break** l'exécution ne s'interrompt pas entre deux **case**

▷ Cela peut permettre de factoriser des cas

CHOIX

```
char grade = 'B';
switch(grade){
    case 'A' :
        printf("Excellent!\n" );
        break;
    case 'B' :
    case 'C' :
        printf("Well done\n" );
        break;
    case 'D' :
        printf("You passed\n" );
        break;
    case 'F' :
        printf("Better try again\n" );
        break;
    default :
        printf("Invalid grade\n" );
}
printf("Your grade is  %c\n", grade );
```

INTERACTIONS AVEC L'UTILISATEUR

Lecture d'un caractère au clavier

▷ La fonction `int getchar(void)`, définie dans `stdio.h`, renvoit le code ascii du caractère lu sous la forme d'un `unsigned char` converti en `int` ou `EOF`[†] en cas d'erreur

```
#include <stdio.h>
int main (void){
    char c;
    int i;
    printf("Enter character: ");
    i = getchar();
    if(i!=EOF){
        c=(char)i;
    }
    return EXIT_SUCCESS;
}
```

[†] Constante égale à -1

Piège du type char

▷ `char` peut désigner, au choix du compilateur, un caractère signé (`signed char`) ou non-signé (`unsigned char`)

▷ La comparaison avec `EOF` échouera tout le temps sur une machine où il désigne implicitement `unsigned char`

▷ Extrait du manuel de gcc

▷ "Ideally, a portable program should always use `signed char` or `unsigned char` when it depends on the signedness of an object."

PRINTF

- Fonction d'affichage de la bibliothèque standard `stdio.h`
  - `printf(chaine de formatage, arguments)`
- Les expressions sont indiquées avec
  - `%d` : entier
  - `%f` : réel
  - `%c` : caractère
  - `%s` : chaîne de caractères

```
printf("average(%d,%d)=%f\n",i,j,(i+j)/2);
```

- Il doit y avoir autant d'expressions à afficher que de `%`

PRINTF

▷ L'affichage est bufferisé

▷ Uniquement affiché quand il y a un `'\n'`, quand le buffer est plein, quand le flot est fermé ou que le programme se termine correctement

```
char* msg="Computing...";
printf("%s",msg);
int i;
float f=0;
for(i=0; i<200000000; i=i+1){
    f=f+0.1;
}
printf("\nf=%f\n",f);
```

▷ Il y environ 3 secondes entre le lancement et l'affichage de `"Computing..."`

ARGUMENTS DE MAIN

```c
int main(int argc, char* argv[]){
    int i;
    for(i=0; i<argc; i++){
        printf("arg #%d=%s\n",i,argv[i]);
    }
    return EXIT_SUCCESS;
}
```

```
$>./a.out AA e "10 2"
arg #0=./a.out
arg #1=AA
arg #2=e
arg #3=10 2
```

▷ `argc` correspond au nombre de paramètres accessibles (le premier étant toujours le nom de l'exécutable)

▷ `argv` est un tableau de chaînes de caractères contenant les paramètres du programme

DOGGY BAG

TO TAKE AWAY ...

▷ Une expression produit une valeur alors qu'une instruction est exécutable

▷ `0` représente faux; le reste représente vrai

▷ Evaluation paresseuse des expressions booléennes

▷ `break` et `continue` ont un impact uniquement sur la structure répétitive englobante la plus proche

▷ Ne jamais utiliser le type `char` sans préciser `signed` ou `unsigned`

Questions?