Programmation structurée

Vincent Archambault-B
IFT 2035 - Université de Montréal

Pour obtenir le code source de ce document

- https://github.com/archambaultv/IFT2035-UdeM
- vincent.archambault-bouffard@umontreal.ca

Programmation impérative

- Langage basé sur un modèle d'une machine
- Le programme indique la séquence d'opérations que la machine doit suivre

Points de contrôle

- Emplacement dans le code source entre deux opérations du programme

```
bool foo() {
①
  int c;
②
  c = 5;
③
  bool b = isOdd(c);
④
  return b;
⑤
}
```

C

Points de contrôle

Point d'exécution	Le point de contrôle où est rendu l'exécution
Flux de contrôle	La progression du point d'exécution (séquence de points de contrôle atteints)

Exécution séquentielle

- Après l’exécution d’une opération, le point d'exécution est le prochaine point de contrôle dans le code source
- En général l’ordre d’exécution par défaut

Exécution séquentielle

- L'ordre des énoncés est important.

```
bool foo() {
  int c;
  c = 5;
  bool b = isOdd(c);
  c++;
  return b;
}
```

C

≠

```
bool foo() {
  int c;
  c = 5;
  c++;
  bool b = isOdd(c);
  return b;
}
```

C

Énoncés de contrôle

- Pour un ordre d'exécution (flux de contrôle) autre que séquentiel
  - ex : goto, if, for, while

goto

- Énoncé de contrôle le plus primitif
- Lorsque le programme rencontre un énoncé goto, le prochain point de contrôle exécuté n’est pas le suivant dans le code source mais celui indiqué par le goto
- La destination est indiquée par une étiquette
- Correspond aux instructions de branchement en langage machine

goto

```
     x = 1;
     goto fin;
deb: x = x*2;
     printf ("%d\n", x);
fin: if (x<10)
       goto deb;
```

goto en C

```
x = 1;
while (x<10) {
  x = x*2;
  printf ("%d\n", x);
};
```

Code équivalent

goto => programme incompréhensibles

- L'utilisation abusive de goto mène à des programmes incompréhensibles
- Il est très difficile de visualiser la séquence d'opérations

Programmation structurée

- Le syntaxe doit aider à comprendre le flux de contrôle
- Un programme est structurée si le flux de contrôle est directement lié à sa structure syntaxique

Programmation structurée

Énoncé de contrôle Syntaxe définissant un flux de contrôle particulier

Énoncé bloc

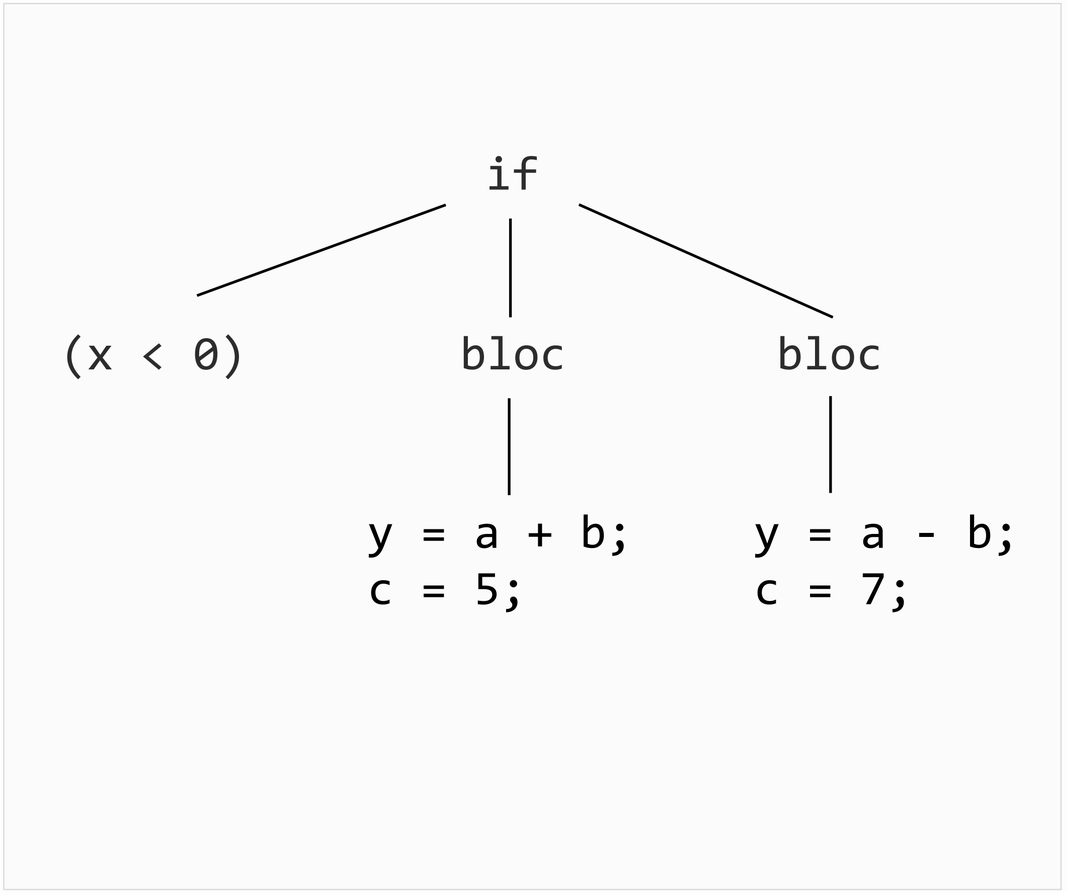
- Pour restaurer l'exécution séquentielle
- Utilisé lorsque la grammaire d'un énoncé de contrôle s'attend à avoir une seule instruction

```c
if (x < 0)
  {
  y = a + b;
  c = 5;
  }
else
  {
  y = a - b;
  c = 7;
  }
```

C

Énoncé bloc

- Pour restaurer l'exécution séquentielle
- Utilisé lorsque la grammaire d'un énoncé de contrôle s'attend à avoir une seule instruction



Arbre de syntaxe abstraite partiellement élaboré

Énoncé bloc

- Certains langages permettent d'introduire des variables locales au bloc

```c
if (x < 0)
  {
  int y = a + b;
  c = y + 5;
  }
else
  {
  int y = a - b;
  c = y + 7;
  }
```

C

Énoncé bloc en C

- Les accolades {} sont utilisées pour délimiter le bloc
- Le point virgule est utilisé comme terminateur d'énoncés

```c
if (x < 0)
  {
  int y = a + b;
  c = y + 5;
  }
else
  {
  int y = a - b;
  c = y + 7;
  }
```

C

Énoncé conditionnel

- Permet de conditionner l'exécution d'opérations en fonction du résultat d'un calcul
- Souvent le else est optionnel

```c
if (x < 0)
  {
  y = a + b;
  c = 5;
  }
else
  {
  y = a - b;
  c = 7;
  }
```

C

Énoncé de boucle while

- Nombre d'itérations inconnu avant d'entrer dans la boucle
- Exécute le corps de la boucle tant que le test ne retourne pas une valeur associée à false
- Le test est exécuté avant la première itération

```c
while(a < b)
  {
    a = exp1;
    b = exp2;
  }
```

C

Énoncé de boucle do … while

- Comme while, sauf que le test est exécuté après la première itération

```c
do
  {
    a = exp1;
    b = exp2;
  }
while(a < b)
```

C

Énoncé break

- Quitte la boucle courante

```c
while(a < b)
  {
    a = exp1;
    b = exp2;
    if (casSpecial(a, b))
       break;
  }
```

C

Énoncé break

- Évite d'avoir à écrire une condition de boucle compliquée
- Rompt avec la programmation structurée pure, l'énoncé de contrôle de la boucle a maintenant plusieurs points de sortie

```c
while(a < b)
  {
    a = exp1;
    b = exp2;
    if (casSpecial(a, b))
       break;
  }
```

C

```c
bool exit = false;
while(!exit && a < b)
  {
    a = exp1;
    b = exp2;
    if (casSpecial(a, b))
      exit = true;
  }
}
```

C

Énoncé continue

- Passer à la prochaine itération

```c
while(a < b)
  {
    a = exp1;
    if (casSpecial(a, b))
       continue;
    b = exp2;
  }
```

C

Énoncé continue

- Évite d'avoir à écrire plusieurs niveaux de if
- Rompt avec la programmation structurée pure, le corps de la boucle n'est pas toujours exécuté en entier

```c
while(a < b)
  {
    a = exp1;
    if (casSpecial(a, b))
       continue;
    b = exp2;
  }
```

C

```c
while(a < b)
  {
    a = exp1;
    if (!casSpecial(a, b))
      b = exp2;
  }
}
```

C

Énoncé de boucle for

- Nombre d'itérations connu avant d'entrer dans la boucle

```c
for(int i; i < 10; i++)
  {
    exp1;
    exp2;
  }
```

C

Énoncé de boucle for

- Une boucle for peut être réécrite avec un while
- La syntaxe de for sépare la logique de la boucle et la logique du corps de la boucle

```c
for(int i; i < 10; i++)
  {
    exp1;
    exp2;
  }
```

C

```c
int i;
while (i < 10)
  {
    exp1;
    exp2;
    i++;
  }
```

C

Énoncé de boucle for

- Il est possible en C d'omettre certains expressions de la boucle for

```c
for(; i < 10; i++)
  {
    exp1;
    exp2;
  }
```

C

```c
while (i < 10)
  {
    exp1;
    exp2;
    i++;
  }
```

C

Énoncé de boucle for

- Il est possible en C d'omettre certains expressions de la boucle for

```c
for(int i;; i++)
  {
    exp1;
    exp2;
  }
```

C

```c
int i;
while (true)
  {
    exp1;
    exp2;
    i++;
  }
```

C

Énoncé de boucle for

- Il est possible en C d'omettre certains expressions de la boucle for

```c
for(int i;i < 10;)
  {
    exp1;
    exp2;
  }
```

C

```c
int i;
while (i < 10)
  {
    exp1;
    exp2;
  }
```

C

Énoncé return

- Quitte immédiatement la procédure ou fonction courante

```c
int foo(int x){
  if (casSpecial(x))
    return -1;
  a = exp1;
  b = exp2;
  return a * b;
};
```

C

Énoncé de sélection par cas

- Sélectionne un énoncé parmi plusieurs
- Plusieurs syntaxes et sémantiques différentes d'un langage à l'autre

```c
switch(x){
  case 1:
    exp1;
    break;

  case 2:
    exp2;
    break;

  default:
    exp3
}
```

C

Énoncé switch en C

- Accepte un entier ou une valeur pouvant être traduite en entier (char, enum, etc).
- Les expressions du case doivent être constantes

```c
switch(x){
  case 1:
    exp1;
    break;

  case 2:
    exp2;
    break;

  default:
    exp3
}
```

C

Énoncé switch en C

- Le code à l'intérieur d'un switch est du code normal
- Les case sont comme des étiquettes de goto
- L'exécution du switch démarre au case correspondant à la valeur de l'expression testée

```c
switch(x){
  case 1:
    exp1;
    break;

  case 2:
    exp2;
    break;

  default:
    exp3
}
```

C

Énoncé switch en C

- Lorsqu'un cas est choisi, toutes les instructions du switch suivant le case sont exécutées
- Il faut donc explicitement mettre un break si on veut éviter de tout exécuter

```
switch(x){
  case 1:
    exp1;

  case 2:
    exp2;
    break;

  default:
    exp3
}
```

Si x == 1, exp1 et exp2 sont exécutés. C

Énoncé switch en C

- S'il n'y a pas de default et aucun cas n'est choisi, rien n'est exécuté. Ce n'est pas une erreur.

```
switch(x){
  case 1:
    exp1;

  case 2:
    exp2;
    break;
}
```

Si x == 1, exp1 et exp2 sont exécutés. C

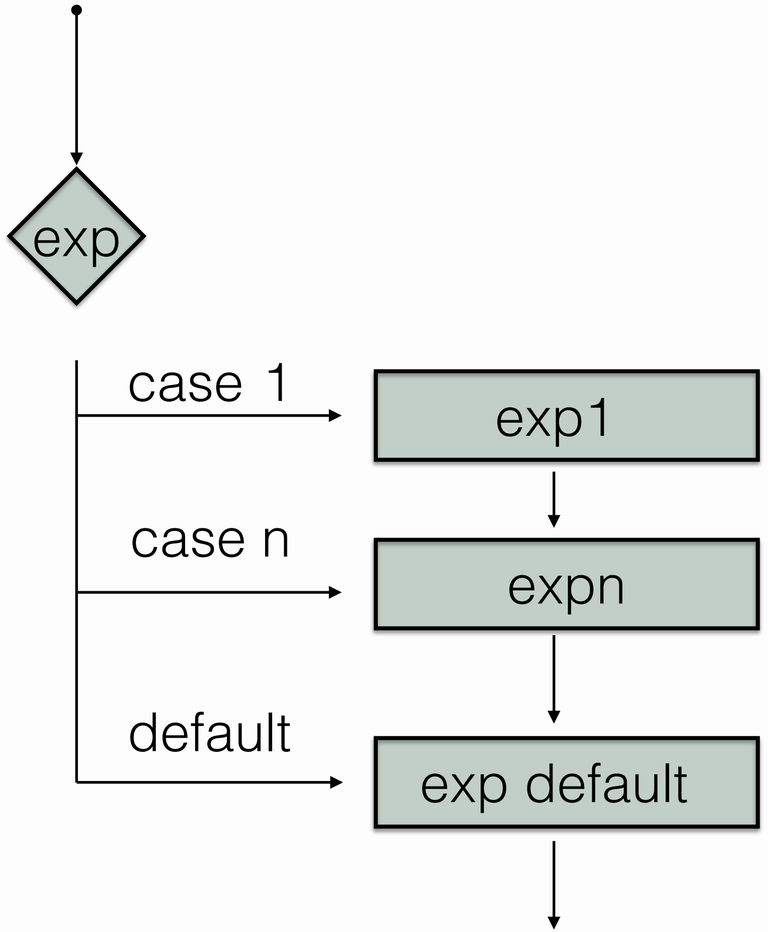
Énoncé switch en C

- Un énoncé peut contenir des sous-énoncés

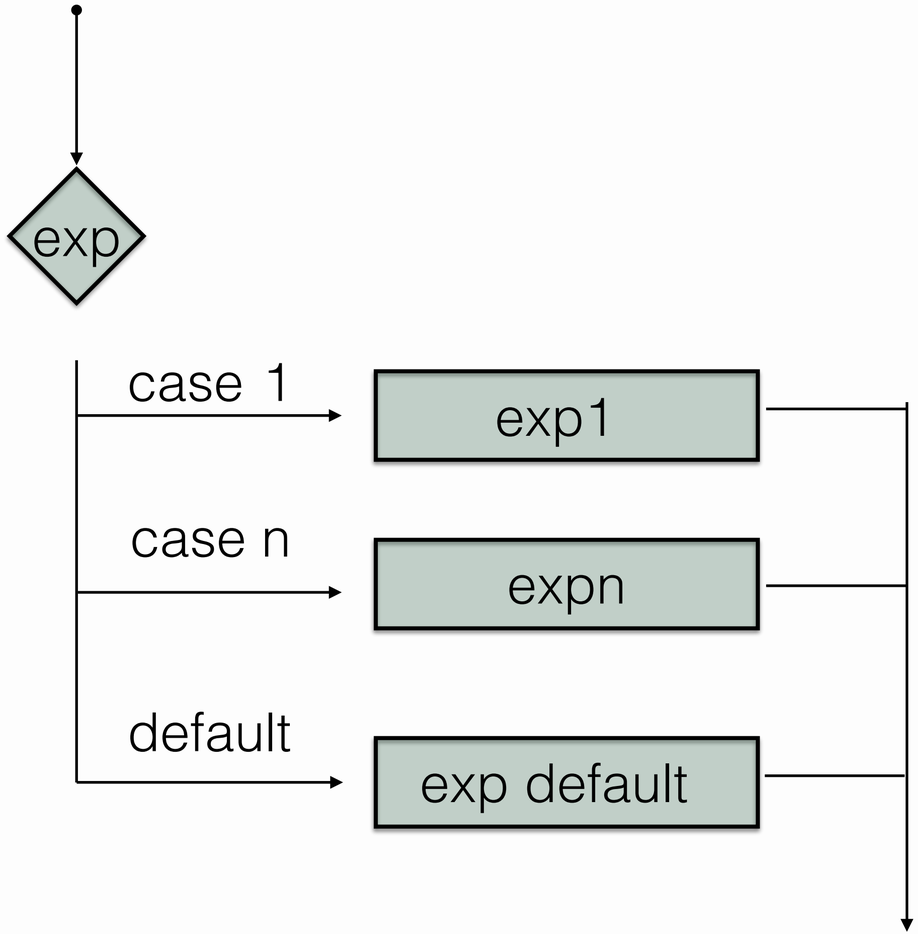
```c
switch(x){
  case 1:
    if (y==0);
      default: y = 1;
    else
      case 2: y = 2;
}
```

C

Énoncé switch en C



Énoncé switch dans la plupart des langages



Exceptions

- Un évènement exceptionnel qui demande au programme de changer son flux de contrôle normal
- Plusieurs causes

  erreurs de logique : *division par zéro, pointeur nul*

  manque de ressources : *mémoire épuisée, fichier inexistant*

  erreurs matérielles : *panne de réseau, papier coincé*

Exceptions

- L'approche la plus simple pour indiquer une erreur est d'utiliser une valeur de retour spéciale
  - malloc() retourne NULL si mémoire est épuisée
  - fopen() retourne NULL si fichier inexistant
  - scanf() retourne EOF s'il y a une erreur de lecture

Exceptions

- En C, errno est une variable globale qui contient un entier qui précise la nature de l'erreur

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>

extern int errno;

int main() {
  FILE *f;
  f = fopen("foo.txt", "r");
  if (f == NULL){
    printf("Error : %d\n", errno);
    return 1;
  }
  ...
}
```

C

Exceptions

- Ces approches nécessitent d'entremêler la gestion des exceptions et le code normalement exécuté
- Code difficile à lire et comprendre

```c
int foo () {
  char *t = malloc (1000);
  if (t != NULL) {
    FILE *f = open ("file.txt","r");
    if (f != NULL) {
       ...
    }
    else
      { free (t); return 1;}
  else
    return 1;
```

C

Exceptions

- Plusieurs langages offrent des énoncés de contrôle pour gérer les exceptions

  try : *le code qui s'exécute normalement*

  catch : *le code à exécuter s'il y a une exception*

  throw / raise : *générer une exception*

  finally : *le code qui s'exécute après un try même s'il y a eu une exception*