Notes

Algo/Prog – DUT 1

Notes

Définir une procédure

- Programmer = définir des procédures et fonctions
- Programme = pyramide de fonctions avec `main` au sommet
- procédure définie → utilisable comme les autres
- Aucune différence entre vos fonctions et celles des bibliothèques (`lireInt()`, etc).

Notes

Définir une procédure

Syntaxe :

- mot clé `void` : déclare une procédure (fonction sans résultat)
- nom de la procédure
- paramètres entre parenthèses (même si aucun)
- corps de la fonction : instructions à réaliser à chaque appel de la méthode

```
void proc(paramètres) {

  instructions

}
```

Notes

Définir une procédure – Exemple

Pas d'argument

```
void proc() {

  ...

}
```

Deux arguments

```
void proc(int x, int y) {    Arguments formels (decl.)

  ... x ... y ...Argument formel (util.)

}
```

Notes

Définir une procédure – Exemple

```
void ecrireAdd(int x, int y) {
  ecrireInt(x);
  ecrireString(" + ");
  ecrireInt(y);
  ecrireString(" = ");
  ecrireInt(x+y);
  ecrireSautDeLigne();
}
```

P.S. Sans oublier le `#include"inout.h"`.

Notes

Exemple de procédure + tests

```
void ecrireAdd (int x, int y) {
  ecrireInt(x);                    paramètres formels
  ecrireString(" + ");
  ecrireInt(y);
  ecrireString(" = ");
  ecrireInt(x+y);
  ecrireSautDeLigne();
}

void main() {
  ecrireString("Début\n");    procédure externe
  ecrireAdd(8,6);        procédure définie ci-dessus
  ecrireAdd(7,13);    paramètres réels
  int a = lireInt();    procédure externe
  int b = lireInt();
  ecrireAdd(a,b);
  ecrireString("Fin\n");
}
```

Notes

Exemple de procédure + test (mieux)

```
void ecrireAdd (int x, int y) {
  ecrireInt(x);
  ecrireString(" + ");
  ecrireInt(y);
  ecrireString(" = ");
  ecrireInt(x+y);
  ecrireSautDeLigne();
}

void testecrireAdd() {
  ecrireString("Début test ecrireAdd\n");
  ecrireAdd(8,6);
  ecrireAdd(7,13);
  int a = lireInt();
  int b = lireInt();
  ecrireAdd(a,b);
  ecrireString("Fin test ecrireAdd\n");
}

void main() {
  testecrireAdd();
}
```

Notes

Portée des variables

```
void ecrireAdd(int x, int y) {
  ecrireInt(x);
  ecrireString(" + ");
  ecrireInt(y);
  ecrireString(" = ");
  ecrireInt(x+y);
  ecrireSautDeLigne();
}

void testecrireAdd() {
  ecrireString("Début test ecrireAdd\n");
  ecrireAdd(8,6);
  ecrireAdd(7,13);
  int a = lireInt();
  int b = lireInt();
  ecrireAdd(a,b);
  ecrireString("Fin test ecrireAdd\n");
}
```

paramètres **locaux**

x y non visibles

X Y non visibles

Déclarations **locale**

Notes

Portée des variables

```
void ecrireAdd(int x, int y) {
  ecrireInt(x);
  ecrireString(" + ");
  ecrireInt(y);
  ecrireString(" = ");
  ecrireInt(x+y);
  ecrireSautDeLigne();
}

void testecrireAdd() {
  ecrireString("Début test ecrireAdd\n");
  int x = 5;
  int y = 2;
  ecrireAdd(x,y);
  ecrireAdd(7,13);
}
```

Rien à voir!

Notes

Portée des variables

Règle générale très importante

Seules informations transmises à une procédure/fonction lors de l'appel : **valeurs (~~noms~~)** des paramètres

P.S. Sauf ~~variables globales~~ mais c'est mal (et interdit ici)

Parce-que

Notes

Définir une fonction

Syntaxe :

- type de la valeur retournée (~~**void**~~)
- nom de la fonction
- paramètres
- corps de la fonction : instructions à réaliser à chaque appel de la méthode
  + **Valeur à retourner**

```
int f(paramètres) {

  instructions
  return ...;
}
```

Notes

Définir une fonction

```
int f(paramètres) {

  instructions
  return ...;
}
```

doit être un **int**

Notes

Définir une fonction – Exemples

```
int oppose (int x) {
  return -x;
}
```

```
int x2plus2xyplusy2 (int x,int y) {
  return x*x + 2*x*y + y*y;
}
```

```
int max (int x, int y) {
  if (x>y) { return x; }
  else { return y; }
}
```

Notes

Définir une fonction – Exemples

```
int somme3 (int x,int y,int z) {
  int res = x;
  res = res + y;
  res = res + z;
  return res;
}

void testsomme3 () {
  ecrireInt(somme3(2,7,11));
  int x = somme3(5,6,7);
  ecrireInt(x);
}
void main() {
  testsomme3();
}
```

Notes

Conditionnelle

Problème :

- spécifier différents cas possibles à l'exécution (≪ conditions ≫)
- spécifier un comportement différent pour chaque cas

⇒ tester une condition

- deux séquences d'instructions différentes en fonction du résultat du test

```
if (cond) {
  ... // instructions si test vrai
}
else {
  ... // instructions si test faux
}
```

≪ tester ≫ une condition ≠ ≪ tester ≫ une fonction

Notes

Conditionnelle

```
if (cond) {
  ... // instructions si test vrai
}
else {
  ... // instructions si test faux
}
```

- On peut omettre le `else` si séquence vide d'instruction
- Conseil : Mettez toujours les accolades

Notes

Conditionnelle

```
int x;
x = lireInt();
if (x >= 0) {
  ecrireString("nombre positif ou nul.");
}
else {
  ecrireString("nombre strictement négatif.");
}
ecrireString("\n");
```

- évaluation de la condition (`x >= 0`).
- si condition vraie premère séquence d'instructions
- sinon deuxième séance
- après 3. ou 4. on exécute de toute façon la suite

Notes

Test

`x <= 0`

- expression ($\neq$ instruction)
- à l'exécution retourne une valeur
- valeurs possibles ? « VRAI » ou « FAUX »
- n'existe pas en C ($\neq$ Java, C++, etc)
- entier !
- valeur 0 : FAUX
- valeur $\neq 0$: VRAI

Notes

 typage des tests

```
int x = 0;
if (x = 0) {
  ecrireString("Oui");
} else {
  ecrireString("Non");
}
```

« Non »

- `x = 0` ~~test~~ affectation !
- En C : affectation retourne la valeur de droite
- ici retourne 0 reconnu comme FALSE
- branche `else` exécutée

```
int x = 0;
if (x == 0) {
  ecrireString("Oui");
} else {
  ecrireString("Non");
}
```

« Oui »

Notes

Opérateurs de test

- `e1 == e2` et `e1 != e2`
- `e1 <= e2` et `e1 >= e2`
- `e1 < e2` et `e1 > e2`
- `!test`
- `test1 && test2` et `test1 || test2`

Notes

Tester le résultat d'une fonction

```
int add3(int x, int y, int z){
  return x + y + z;
}

void testerAdd3 () {
  int res = add3(2,3,4);
  if (res != 9) {
    ecrireString("Erreur sur add3!");
    ecrireString("Attendu: 9");
    ecrireString("Obtenu: ");
    ecrireInt(res);
    ecrireSautDeLigne();
  }
}

void main() {
  testerAdd3();
}
```

Notes

Notes

Notes