Plan du cours

- Les structures
- Les tableaux
- Les fonctions
- Les pointeurs

Les structures

Renommage des types

- Un type de votre programme peut être renommé.

- Utilisation du mot clé **typedef**

- Exemple: renommage du type `float`


```
typedef float reel; //Nommage
reel a,b,c; //Déclaration de variables
```

Utilité du renommage

- Faire évoluer le type partout sans avoir à l'écrire à nouveau partout.

```
//Amélioration de la précision
typedef double reel;
```

- Renommage des types structurés et des ensembles énumérés.

Ensembles énumérés (1/2)

- Pour décrire des ensembles finis de valeurs.
- Utilisation du mot-clé **enum**
- Exemple: les couleurs

```
//déclaration directe
enum color_t {rouge, vert, bleu} p,q;

//déclaration avec typedef
typedef enum {rouge, vert, bleu} color_t;
color_t p,q;
color_t v = bleu;
```

Ensembles énumérés (2/2)

```
typedef enum {rouge, vert, bleu} color_t;
```

- À chaque nom est attribué automatiquement une valeur commençant par 0:
  - rouge=0
  - vert=1
  - bleu=2
- Utilisation comme des entiers:

```
cout<<(int)bleu; //Affichage de 2
```

- On peut également forcer les valeurs:

```
typedef enum
{rouge=1, vert, bleu, jaune=0, violet} color_t;
```

  - vert=2
  - bleu=3
  - violet=1

Les types structurés (1/2)

- Ensemble fini de variables représentant un objet plus complexe
  - Permettent de définir des types personnalisés
- Utilisation du mot-clé **`struct`**
- Exemples:

```
//Les nombres complexes
typedef struct {float a,b;} complexe_t;

//Les dates
typedef struct {int jour, mois, an;} date_t;

//Déclarations
complexe_t c1,c2;
date_t d1, d2,d3;
```

Les types structurés (2/2)

- Opérateurs d'accès:
  - Le . pour accéder aux différents champs
  - Le = pour l'affectation
- Exemples:

```
typedef struct {int jour, mois, an;} date_t;
date_t d1, d2, d3;

d1.jour=14; //Affectation champ par champ
d1.mois=7;
d1.an=1789;

d3={4,8,1789}; //Affectation globale

d2=d1; //Copie de d1 dans d2
```

Exemple de synthèse

```
typedef enum {
janvier=1, fevrier, mars, avril, mai, juin,
juillet, aout, septembre, octobre, novembre, decembre
} mois_t;          //Ensemble énuméré des mois

typedef struct {
        int jour;
        mois_t mois;
        int an;
} date_t;          //Type structuré de dates

typedef struct {
        char* nom;          //Nom de la personne
        date_t ddn;         //Date de naissance
} personne_t;

int main(void)
{
        personne_t p;
        p.nom="toto";
        p.ddn.jour=1;
        p.ddn.mois=janvier;
        p.ddn.an=2000;
        return 1;
}
```

Les tableaux statiques

Les tableaux à une dimension
Déclaration

- Regroupement sous un seul nom de plusieurs variables de même type.

- *Déclaration:*

```
type Nom[Nombre d'éléments];
```

- Exemples:

```
int T[6];     //tableau de 6 entiers
date_t D[10]; //Tableau de 10 dates
```

Les tableaux à une dimension
Utilisation

- Chaque case contient une valeur et est désignée par un indice
- Indices allant de 0 à taille-1
- On accède à la valeur de la (i-1)ème case du tableau en écrivant: T[i]
- *Exemple: La 2ème case d'indice 1 contient la valeur 14 désignée par l'expression T[1]*

Les tableaux à une dimension
Initialisation

Plusieurs techniques:

- Initialisation en une fois

```
int T[6] = {12,14,5,20,6,18};
```

- Avec une boucle (initialisation à 0):

```
for(i=0;i<6;i++)
  {T[i]=0;}
```

- Saisie par l'utilisateur:

```
date_t D[10];
for(i=0;i<10;i++)
  {cin >> D[i].jour;
   cin >> D[i].mois;
   cin >> D[i].annee;}
```

Les tableaux à une dimension
Précautions

- Pas de contrôle de dépassement d'indice dans les tableaux
  - On peut écrire `T[20]` sur un tableau qui ne contient que 10 éléments

- Impossible de copier un tableau directement dans un autre
  - `t1=t2;` copie seulement l'adresse de t2

Les tableaux à une dimension

Techniques de programmation

- **Effectuer la somme des éléments d'un tableau à n éléments**
- **Gérer une taille de tableau variable**
- **Chercher une valeur dans un tableau**

Les tableaux à plusieurs dimensions

- Un tableau peut avoir plusieurs dimensions (2 voire plus)

- Déclaration d'un tableau à 15 éléments
  (5 lignes x 3 colonnes)

```
int T[5][3];
```

- Elément situé à la i-ème ligne et la j-ème colonne : T[i+1][j+1]

Les tableaux à plusieurs dimensions
Techniques de programmation

- **Parcourir un tableau à deux dimension (affichage)**

Les fonctions

La librairie mathématique cmath

```
#include<cmath>
```

- Valeur absolue: `fabs(-3);  -> 3`
- Arrondi:
  - `floor(32.4);  -> 32`
  - `ceil(32,4);  -> 33`
- Puissance: `pow(4,2);`
- Racine carrée: `sqrt(100);  ->10`
- `sin  cos  tan  exp  log  log10 …`

Définir nos propres fonctions

- Nécessité de rendre un programme modulaire
  - découpage en fonctions

- Syntaxe:

```
TypeRetour MaFonction(parametres)
{
// Instructions
}
```

- Le main est une fonction
- Nous pouvons en définir autant que l'on veut

- Mot-clé **`return`** pour renvoyer le résultat.

Exemple de fonction

```
#include<iostream>

int factorielle(int n)          //Définition
{
        int i, res;
        res = 1;
        for(i=n; i > 0; i--)
        {
                res = res*i;
        }
        return res;
}

int main(void)
{
 cout << factorielle(5) << endl; //Appel de la fonction
}
```

Prototypage

- Déclarer les fonctions n'importe où.
  - une fonction doit toujours être déclarée avant d'être utilisée
  - Utilisation d'un prototype (signature de la fonction)

```
#include<iostream>
int factorielle(int);  //Prototype (déclaration)

int main(void)                    //Utilisation
{       cout << factorielle(5) << endl;}

int factorielle(int n)            //Définition
{
        //voir transparent précédent
}
```

Portée des variables

- Toute variable déclarée en dehors des fonctions est globale
  - Valeur modifiable par toutes les fonctions
- Toute variable déclarée dans une fonction est locale à celle-ci
  - Valeur modifiable dans la fonction seulement

```
#include<iostream>
float pi = 3.14;        //pi: Variable globale (à éviter)

float aire(float r)     //r: Variable locale
{       return pi * r * r;  }

int main(void)
{
        float rayon;    //rayon: variable locale
        cin >> rayon;
        cout << aire(rayon);
}
```

Les fonctions récursives

- On peut écrire une fonction qui fait appel à elle-même:
  - l'appel se fait en général sur une valeur plus petite

- Exemple: fonction factorielle récursive
  - même prototype mais contenu différent

```
int factorielle(int n)
{
        if(n==0)                    //Condition d'arrêt
        {return 1;}
        return n * factorielle(n-1); //Récursion
}
```

Arguments multiples

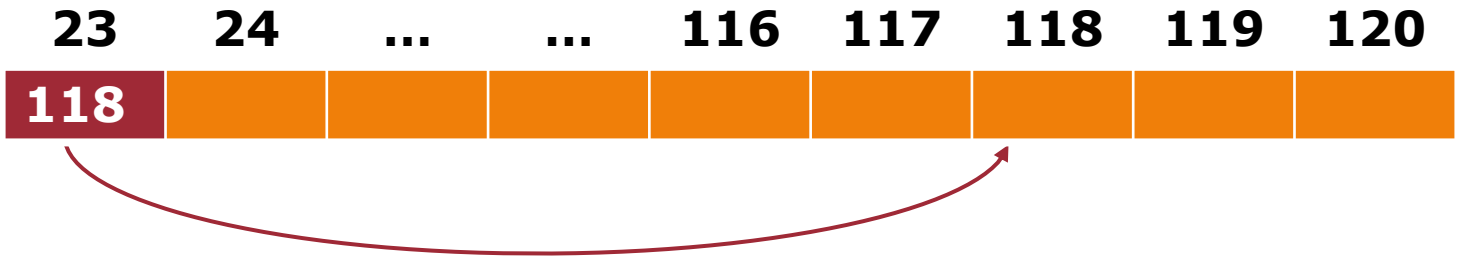
- Il est possible d'écrire des fonctions qui comportent plusieurs arguments:
- Exemple: fonction puissance récursive

```
int puissance(int x, int n)
{
  if(n==0)
      return 1;
  if(n==1)
      return x;
  return x * puissance(x, n-1);
}
```

Les pointeurs

Définition

- Variable contenant l'adresse d'une autre variable d'un type donné



Intérêt des pointeurs

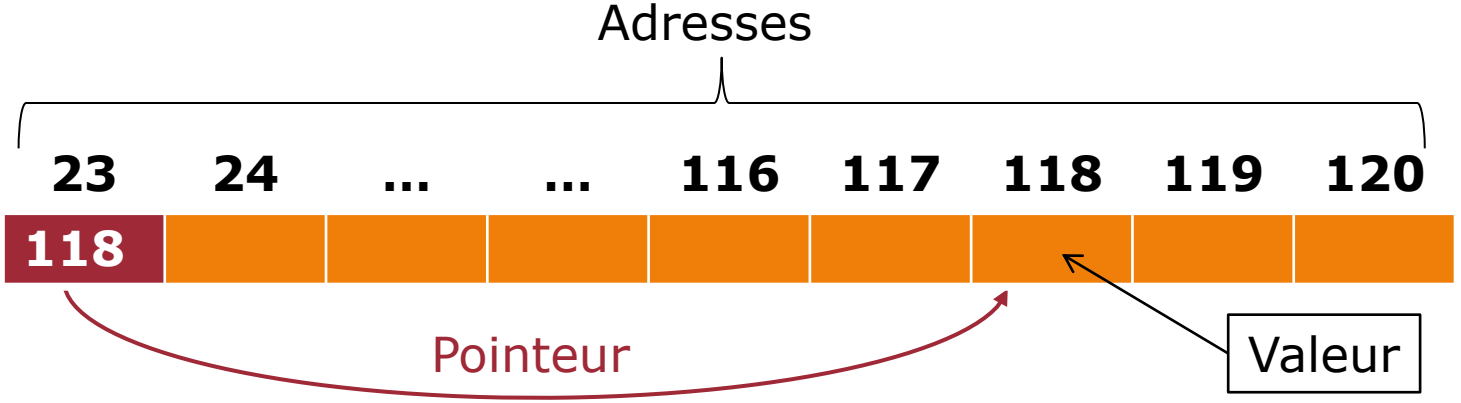
- Possibilité de manipuler des données de grande taille très simplement

- Créer des structures dynamiques
  - Taille évoluant au cours du temps
  - ≠ des tableaux

- Créer des structures complexes chainées (listes, arbres…)

Notion d'adresse

- Chaque bloc de la mémoire est identifié par un numéro (son adresse)
- Toute variable prend un nombre précis de cases mémoires (selon le type)
- Stockage de l'adresse de la variable dans un pointeur



Adresse d'une variable

- Pas besoin de manipuler directement l'adresse d'une variable

- Syntaxe pour connaitre l'adresse d'une variable:

```
&Nom_de_variable
```

Déclaration des pointeurs

- Syntaxe de la déclaration:

```
type* pointeur;
```

- Le type peut être un type de base (`int`, `char`), ou alors un type complexe (`struct`…)

Initialisation d'un pointeur

- Etape nécessaire car par défaut pointe n'importe où…

- Syntaxe:

```
Nom_du_pointeur = &variable_pointee;
```

```
int a = 4;
int* p1 = &a;
```

Accès à une variable pointée

- Pour accéder au contenu:

```
*pointeur
```

```
*p1=10;
```

- Dans une expression plus complexe:

```
a= (*p1)+3;
```

Utilisation des pointeurs

Passage d'argument à une fonction **par valeur**

```
int Ajout2(int a)
{ a +=2;
  return a; }

int b = 3;
b = Ajout2(b)
```

Passage d'argument **par adresse**

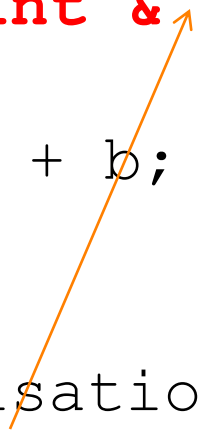
```
void Ajout2(int * a)
{ *a +=2; }

int b = 3;
Ajout2(&b);
```

Passage d'argument **par référence**

```
int Ajout2(int &);

int Ajout2(int & a)
{ a +=2; }

int b = 3;
Ajout2(b);
```

Passage d'argument par référence

```
/*Prototype*/
Ajout(int &, int);

/*Fonction*/
Ajout(int & a, int b)
{
  a = a + b;
}

/*Utilisation*/
int a = 3;
Ajout(a,4); /*Ajoute 4 à la variable a*/
```



Les références

- Utilisation en dehors du passage de paramètres à une fonction:

```
int e1;
int e2 = 10;

//Initialisation de la référence obligatoire
int& ref_e1=e1;

//Change la valeur de ref_e1 et e1 à 25.
ref_e1=25;

//Ne change pas la référence vers e1 mais seulement la
  valeur.
ref_e1= e2;

//Toutes les valeurs sont à 10
cout << e1 << e2 << ref_e1 << ref_e2;
```

Tableaux et pointeurs

- L'identificateur d'un tableau sans les [] est un pointeur

```
int t[10];
```

- `t` est un pointeur vers la première case du tableau
  - même chose que `&t[0]`
  - de type `int*`
- `t+i` est un pointeur vers la (`i+1`)ème case du tableau (`&t[i]`)
- Initialisation d'un tableau

```
for(int i=0; i<10;i++) {*(t+i)=0;}
```

Passage d'un tableau en argument d'une fonction

- Un tableau est automatiquement passé comme un argument par adresse d'une fonction
  - ses valeurs sont modifiables par la fonction

```
void init(int t[], int n, int val) //autre écriture: int *t
{
        int i;
        for(i=0; i<n; i++)
        {
                t[i]=val;
        }
}
```

Cas des tableaux à deux dimensions

- L'identificateur d'un tableau à deux dimension est aussi un pointeur

```
int t[3][4];
```

- C'est un tableau à trois éléments qui sont des tableaux de 4 entiers.

- `t[0]` est un pointeur vers le premier entier du premier tableau (`&t[0][0]`)