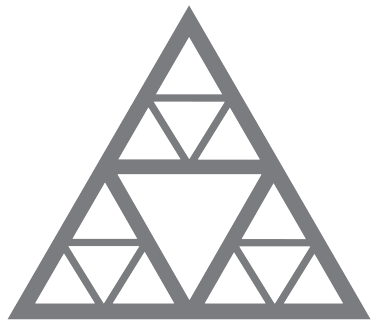
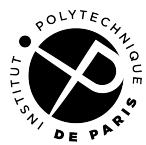
PRALG: pointeurs

Pascal Monasse
pascal.monasse@enpc.fr



ÉCOLE NATIONALE DES
PONTS
ET CHAUSSÉES



IP PARIS

Adresse des variables

- Pointeur = variable désignant une autre variable.
- On a déjà vu ça avec les références, mais les pointeurs sont plus généraux.
- Chaque variable a une taille en mémoire (# octets), fixée en fonction de son type. On peut l'obtenir par la fonction sizeof .
- Elle a aussi une adresse en mémoire, le numéro de son 1er octet (les autres suivent).
- Un pointeur est en fait cette adresse, qu'on peut obtenir avec l'opérateur *&var*, c'est donc une valeur numérique.

```
int x=1;
cout<< "taille:" << sizeof(int) << "-adresse:" << &x;
```

Exemple de résultat : "taille : 4 adresse :0x7ffe379ce7ac"
Cette adresse est affichée en hexadécimal (préfixe 0*x*).

Stocker l'adresse dans un pointeur

- On stocke l'adresse et la taille dans une variable "pointeur" : int* p = &x;
- p a bien une valeur numérique, mais son type indique que la variable pointée x est un int.
- On accède à la variable pointée par $*p$.
- *p=0; revient à faire $x = 0$.
- C'est une variable comme une autre, on peut changer sa valeur : int y=1; p=&y; cout << *p; affiche bien 1.
- p=0; ou p=nullptr; if (p!=0) *p=1; pointeur sur "rien".
- Ne pas confondre :
  1. *type** p (pointeur sur *type*), $*p$ (variable pointée par p)
  2. *type*& r=v (référence sur v), &v (adresse de v)
- Attention : int* p1,p2; $\Leftrightarrow$ int *p1,p2; $\Leftrightarrow$ int* p1; int p2; $\neq$ int *p1,*p2;
- int* p1 définit $*p1$ de type int, donc $p1$ est un pointeur sur int.

Allocation dynamique, tableaux

- new alloue de la mémoire et renvoie l'adresse (du 1er octet) :

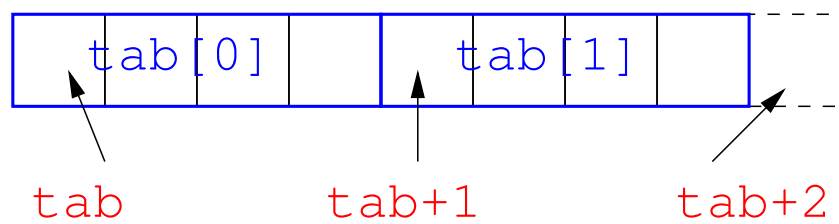
```
int* cree_int() { return new int; }
int* cree_int(int val) { return new int(val); }
int *p1=cree_int(), *p2=cree_int(2); //OK
delete p1; delete p2; // Pas delete p1,p2
```

- Avec tableau :

```
int* cree_tab(int n) { return new int[n]; }
int *t=cree_tab(5); t[0]=1; t[4]=-1;
delete [] t; // Ne pas oublier []
```

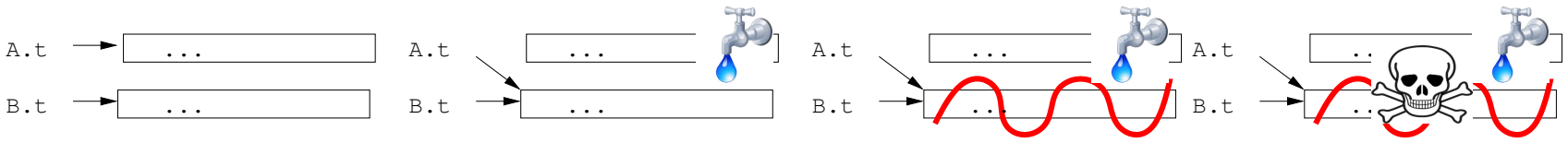
- Arithmétique des pointeurs :

```
int* tab = new int[2];
int* p;
for(p=tab; p!=tab+2; p++)
  *p = 0;
cout << p-tab; //2
delete [] tab;
```



Pointeur : puissant mais dangereux

```
struct Mat {int m,n; double* t;};
Mat cree(int,int); void detruis(Mat A);
```



Solution :

- Constructeur Mat::Mat(int nlig, int ncol) (new)
- Destructeur Mat::~Mat() (delete [])
- Constructeur par copie Mat::Mat(const Mat& M) (new)
- Affectation Mat& Mat::operator=(const Mat& M) (delete [] puis new)

```
Mat A(3,5); //Mat::Mat(int,int)
Mat B=A, C(A); //les deux appellent Mat::Mat(const Mat&)
C=B; //Mat::operator=
void print(Mat M); //Mat::Mat(const Mat&)+Mat::~Mat
Mat transpose(const Mat& M); //Mat::Mat(int,int)+Mat::Mat(const Mat&)+Mat::~Mat
```

Exemples

- Init tableau :

```
int* p=t;
for(int i=0; i<n; i++)
  *p++=0; //ou bien {*p=0; p++;}
```

- Copie de tableau :

```
int *p=t1, *q=t2;
for(int i=0; i<n; i++)
  *q++ = *p++;
```

- Notez : $*p{+}{+} \Leftrightarrow *(p{+}{+}) \neq (*p){+}{+}$

-
```
class A {public: int i; void f();};
A* a = new A;
*a.i = 0;    //Erreur, interprete comme *(a.i)
(*a).i = 0; //OK
a->i = 0;    //Equivalent mais mieux
*a.f();      //Erreur, interprete comme *(a.f())
a->f();      //OK
```

- `int* p; *p=0;` Ouch ! ecriture mémoire à adresse abritraire