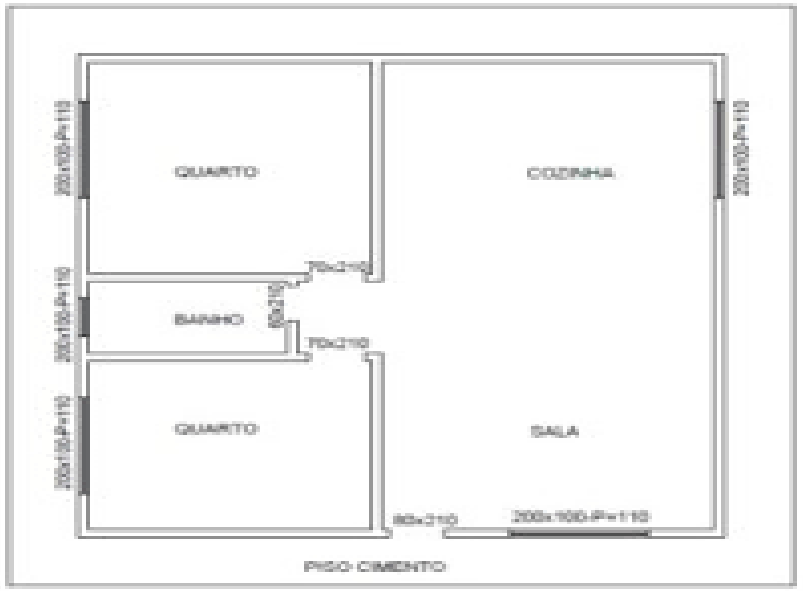
É importante entender e utilizar a linguagem gráfica na representação da instância de uma variável *struct* e a sua associação com a manipulação de campos

Para entender o significado de desenhos contendo operadores manipulando uma variável *struct*, vamos fazer uma analogia entre uma casa e uma variável desse tipo

O modelo da casa está na sua planta



O modelo da struct é descrito pelos seus campos:

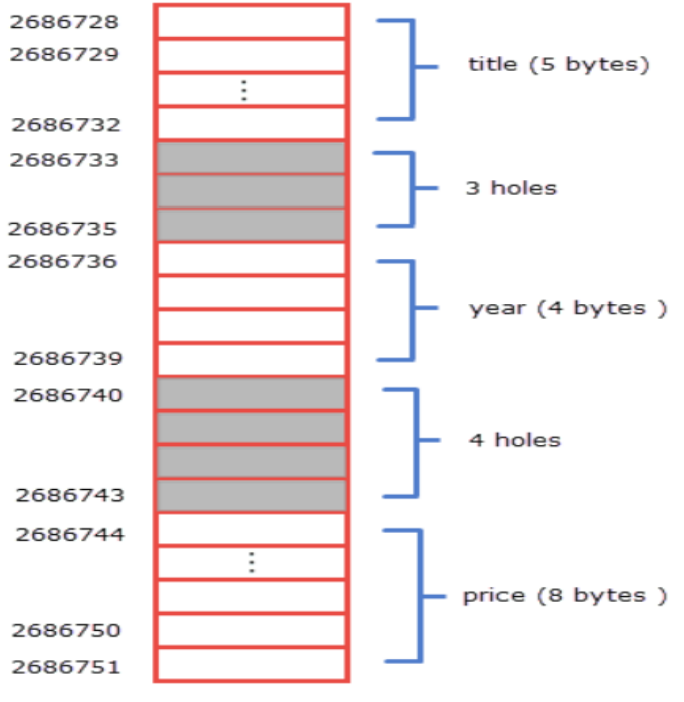
```
struct xxxx{     campoA;
                 campoB;
        };
```

A casa é construída em um endereço no logradouro;

A struct é instanciada em endereço de memória;

```
struct book
{
    char title[5];
    int year;
    double price;
};

struct book b1 =
{"Book1", 1988, 4.51};
```

```
Size of structure b1 =

name(5 bytes) + 3
holes + year(4 bytes)
+ 4 holes + price(8
bytes)

= 24 bytes
```



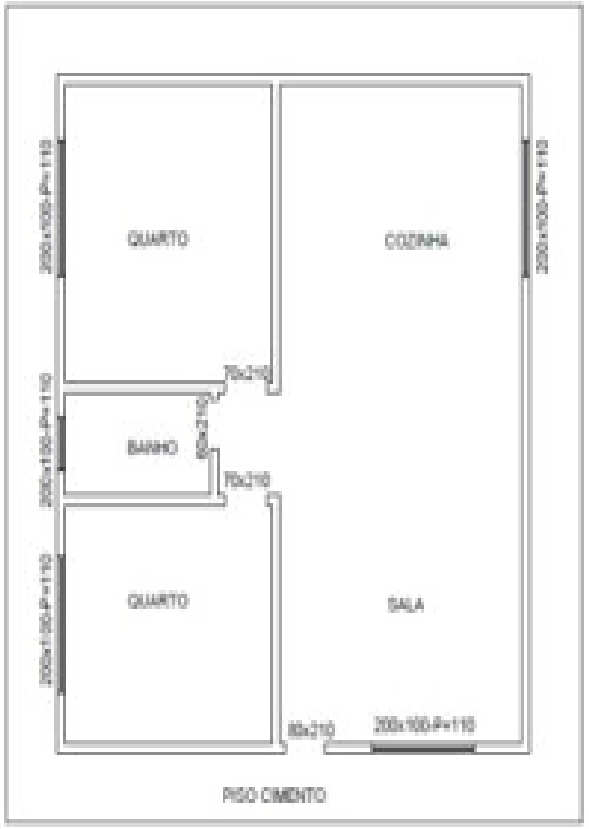
https://overiq.com/c-programming-101/structure-basics-in-c/

Casa → instância de uma struct

Cômodos na planta baixa → campos da *struct*

Cômodos não existem sem a casa estar construída → os campos não existem sem a *struct* instanciada em um endereço de memória

Cômodos não podem ser acessados sem o acesso prévio ao local da casa → campos não podem ser acessados sem o acesso à instância da *struct* via seu endereço de memória



```
struct teste{
    tipo campoA;
    tipo campoB;
    tipo campoC;
};

struct teste *P=NULL, *Q=NULL, *W=NULL;


P=(struct teste*) malloc(sizeof(struct teste));
//P é o endereço da struct (casa)

struct teste Z;

//A variável Z é a própria struct (casa)
```

A descrição da struct **não** a **instancia** apenas define seu modelo de dados

A declaração de ponteiros também **não instancia** a struct, no caso, foram instanciados Apontadores do tipo struct teste

Alocação dinâmica: uma maneira de instanciar a struct

Alocação estática: outra maneira de Instanciar a struct

Comando: *W=Q=P;*

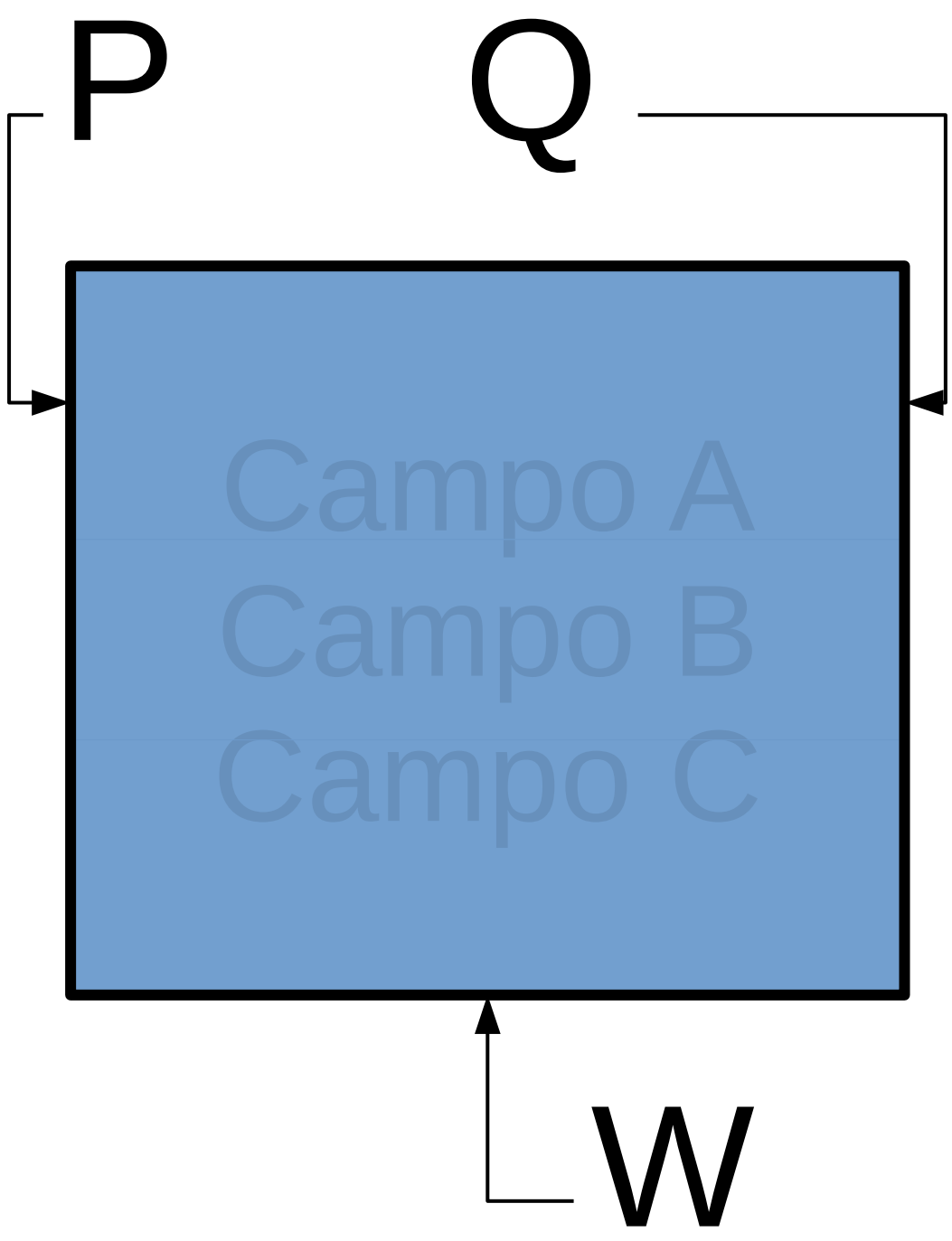
Resultado:

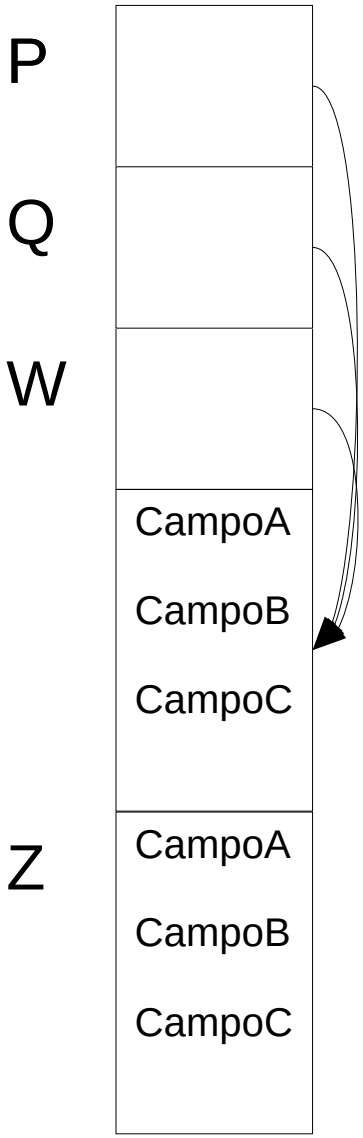
P, **Q** e **W**

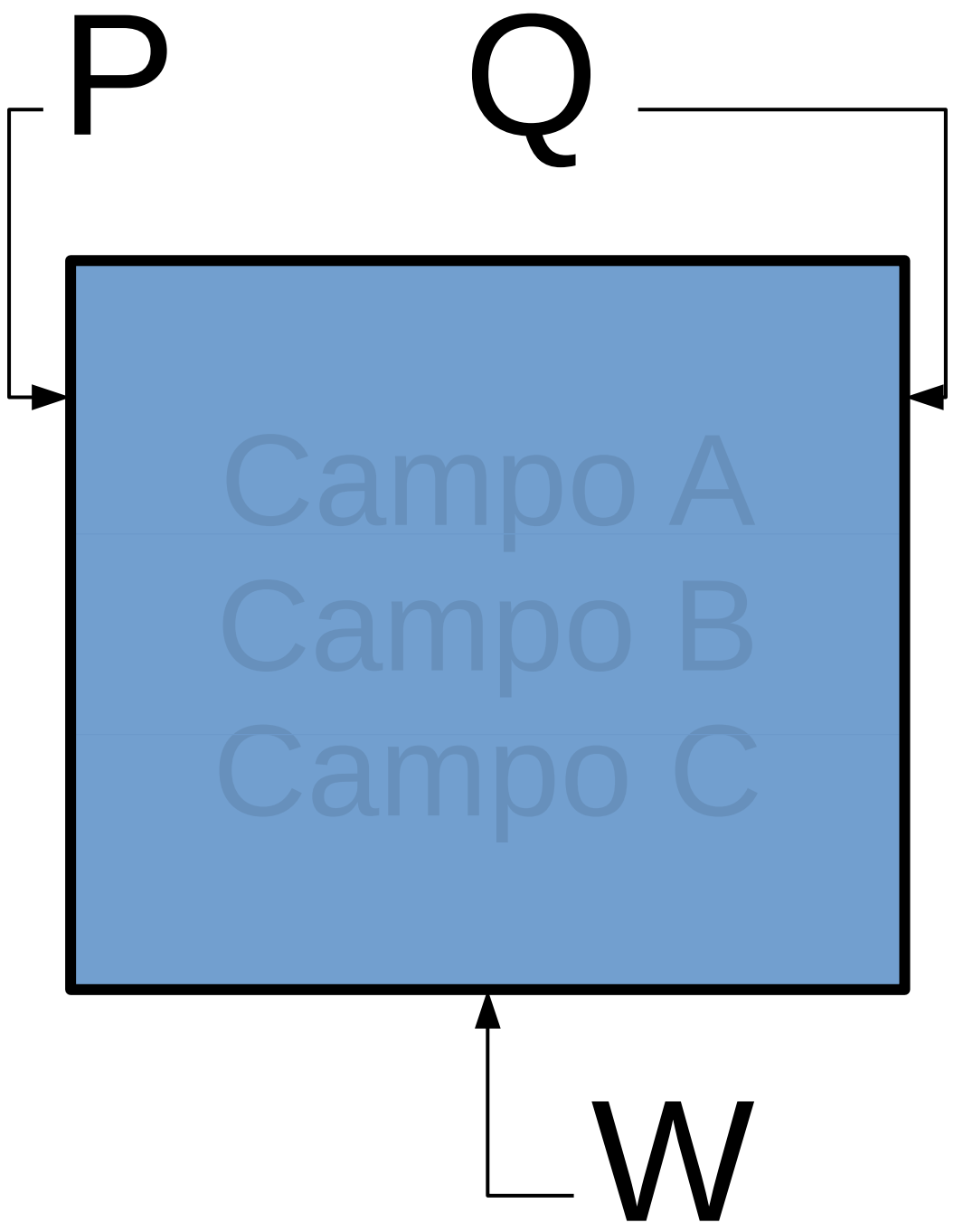
- **Apontam** para o endereço da mesma **instância** de *struct teste* (na verdade o endereço do primeiro byte da *struct*)

  Graficamente representado por setas chegando na **moldura** preta que simboliza a instância da *struct*

- **Não apontam** diretamente para nenhum **campo** da *struct teste*




P
Q
W
CampoA
CampoB
CampoC
Z
CampoA
CampoB
CampoC


P
Q
Campo A
Campo B
Campo C
W

```
struct teste{
    tipo campoA;
    tipo campoB;
    tipo campoC;
};
```

Os campos de uma struct **não** podem ser acessados **diretamente** pelos seus identificadores;

Acesso aos campos:

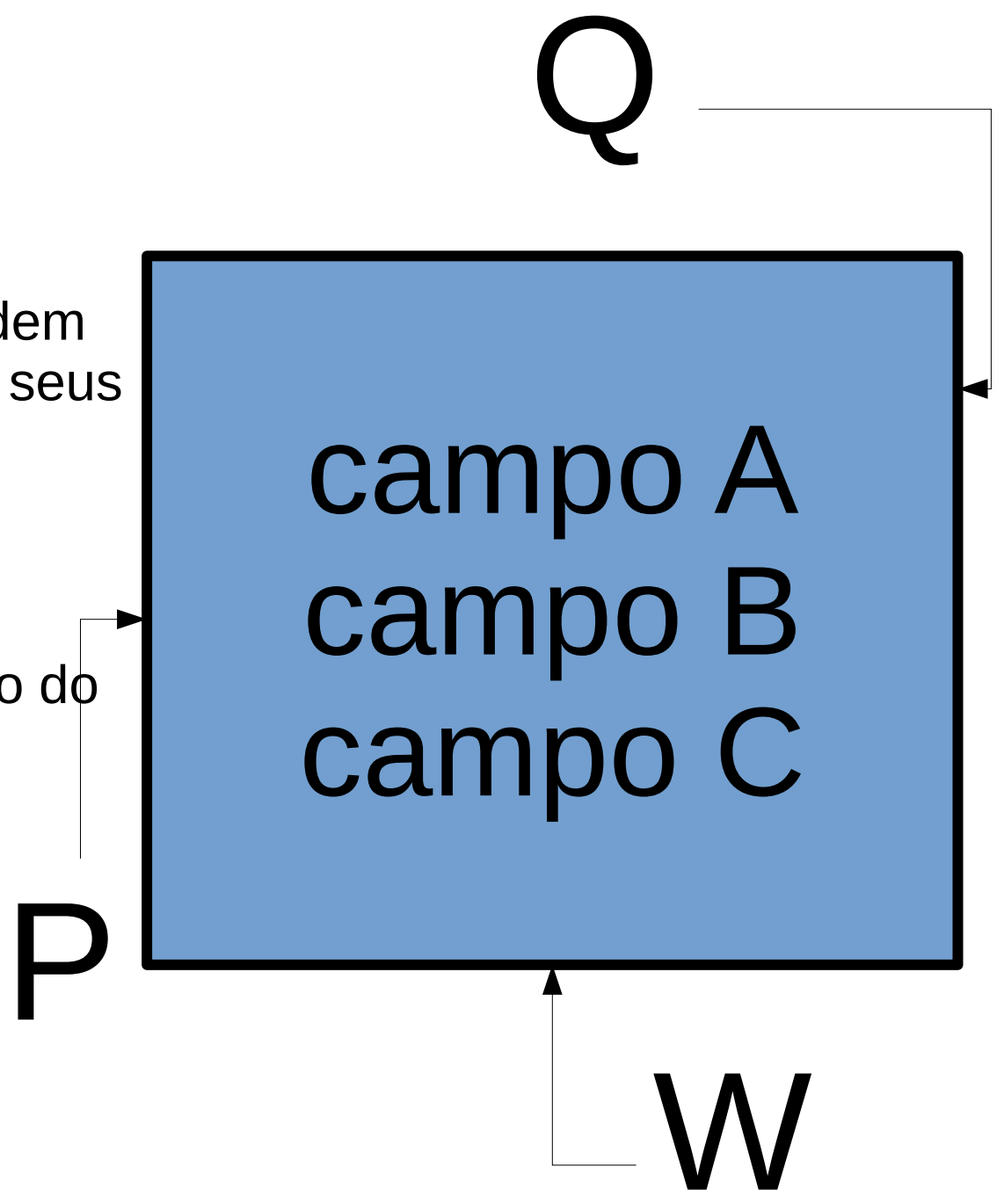
Uso do operador seta “->” por meio do endereço da struct

P->campoC

Uso do operador ponto

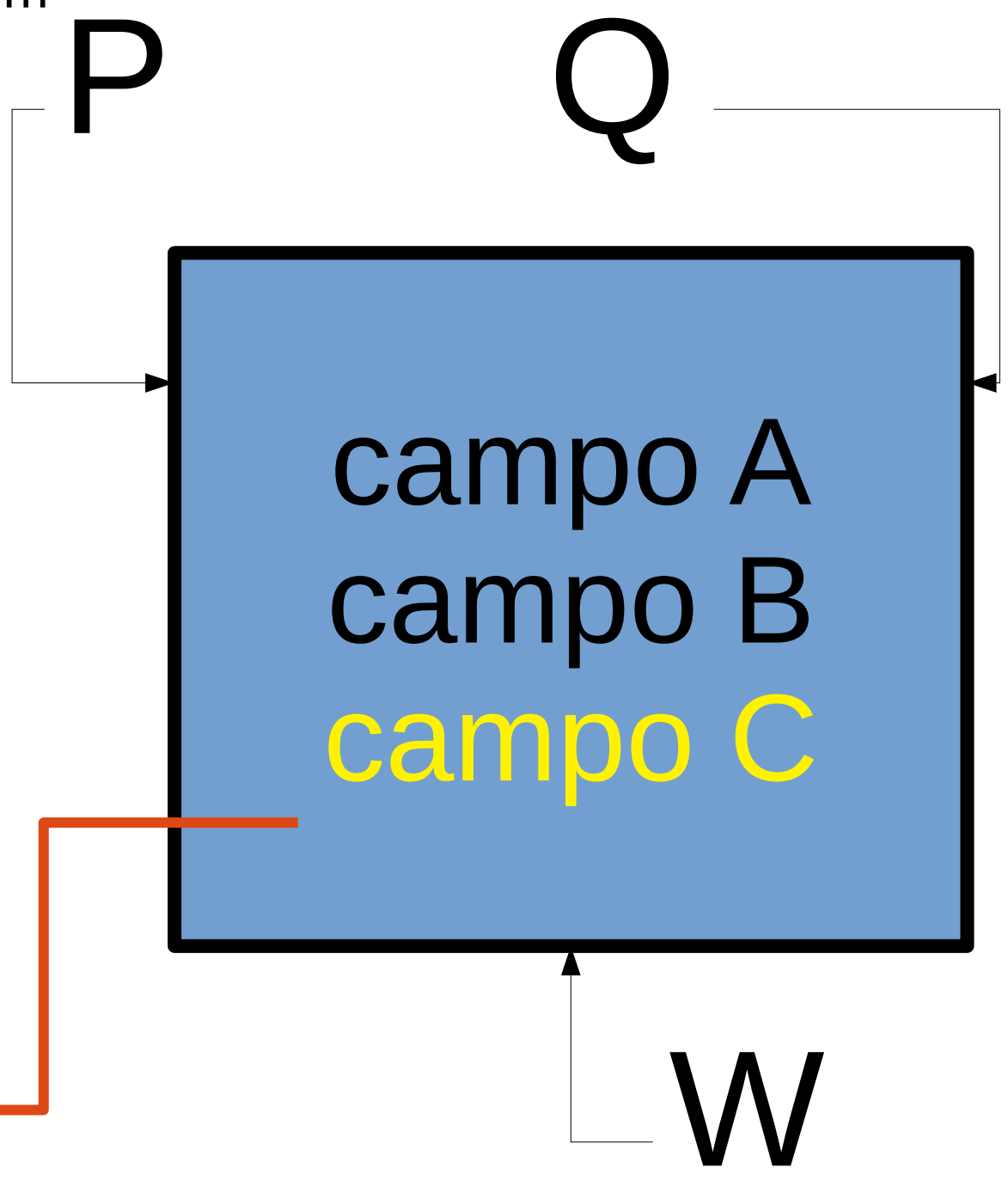
VariávelStruct.campo

Z.campoA



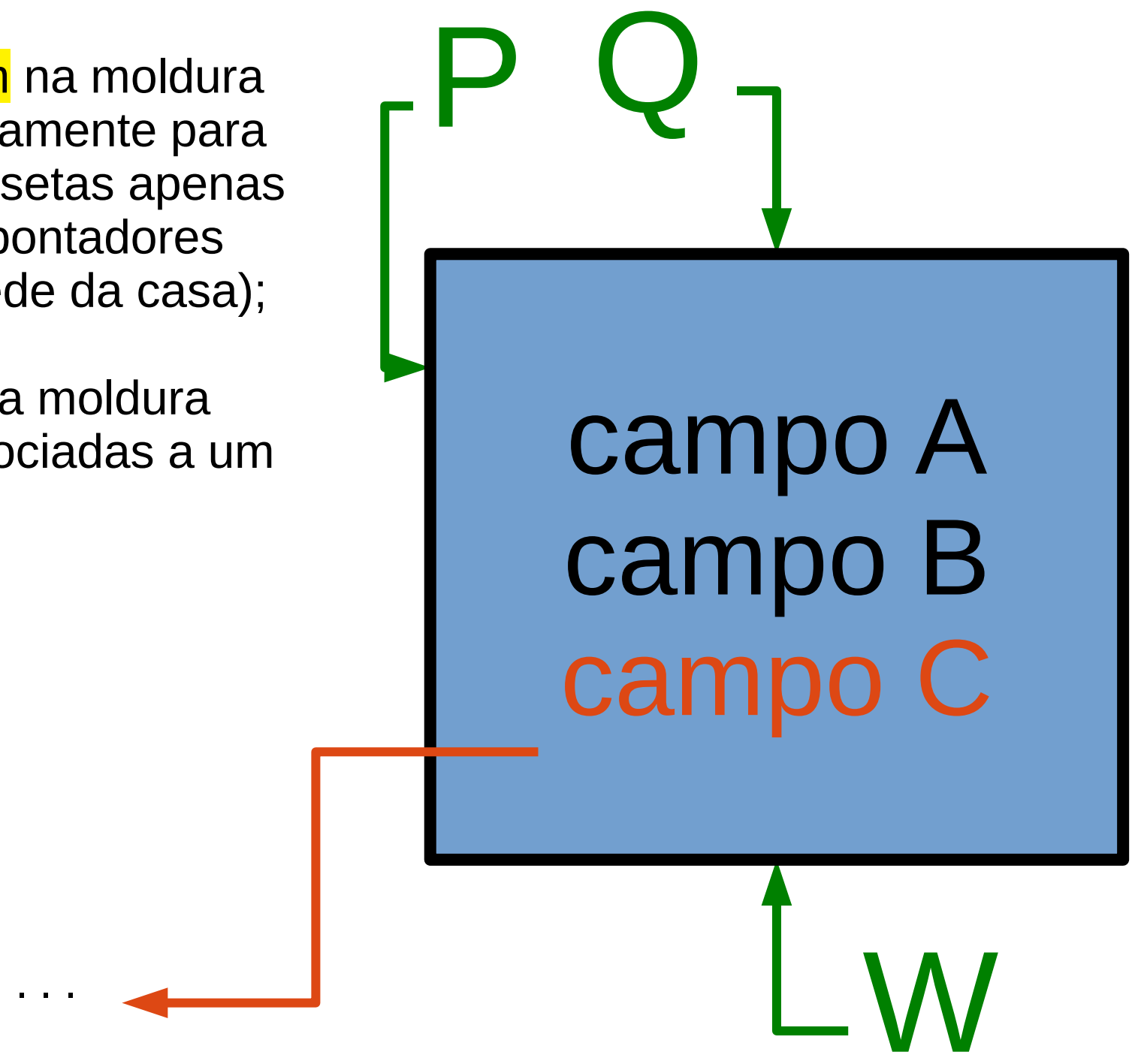
Um campo da struct pode ser um apontador para uma *struct* cujo modelo é Igual ao da strict que o contém.

```
struct teste{
    tipo campoA;
    tipo campoB;
    struct teste *campoC;
};
```



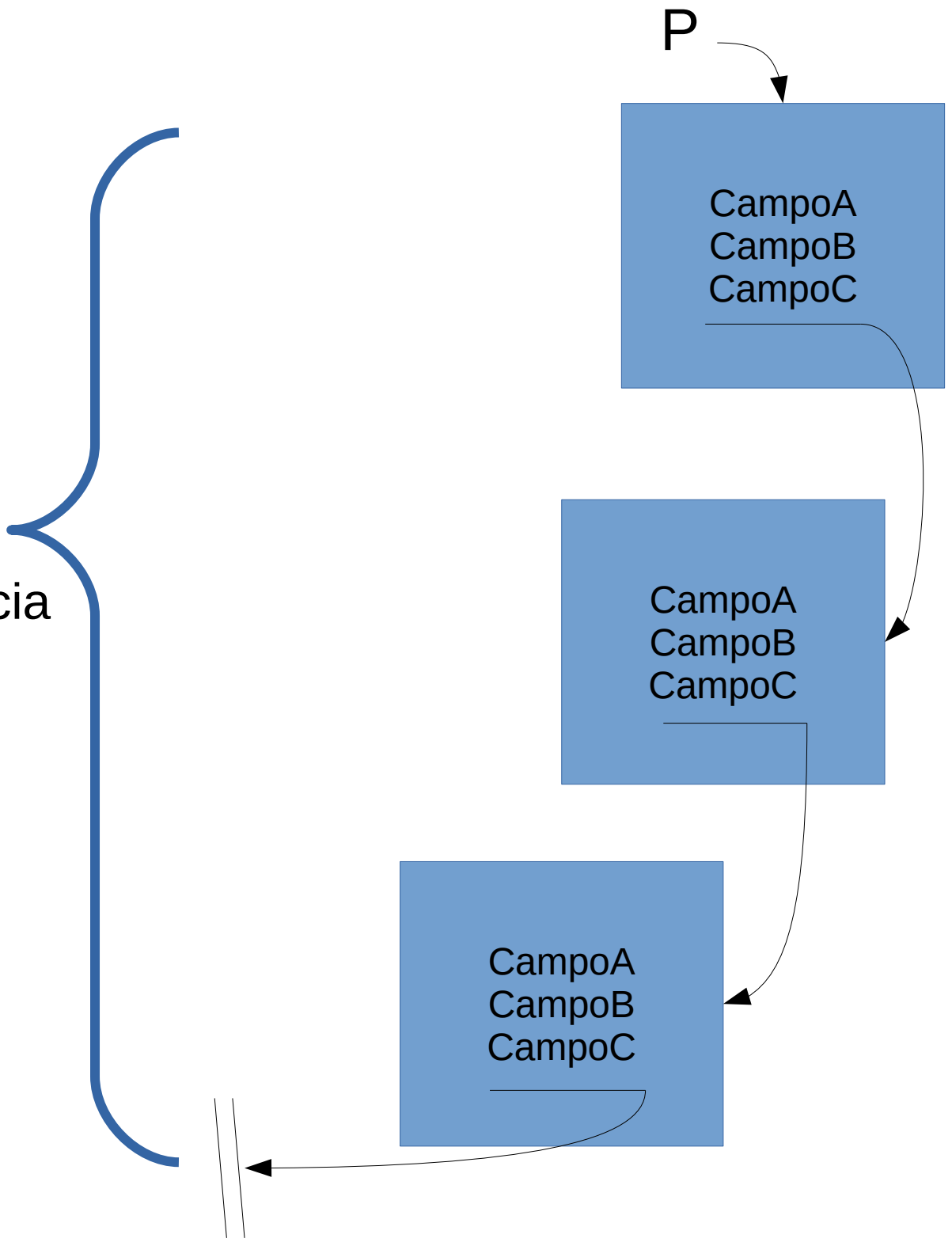
Setas que chegam na moldura não apontam diretamente para um campo, essas setas apenas representam os apontadores para a *struct* (parede da casa);

Setas que saem da moldura estão sempre associadas a um campo-apontador;



Tirando proveito do campo apontador:

Como construir uma sequência em cadeia de três instancias da *struct teste?*



O campoC pode ser utilizado para criar uma cadeia de instancias da *structs teste:*

*struct teste *inicio=NULL;*

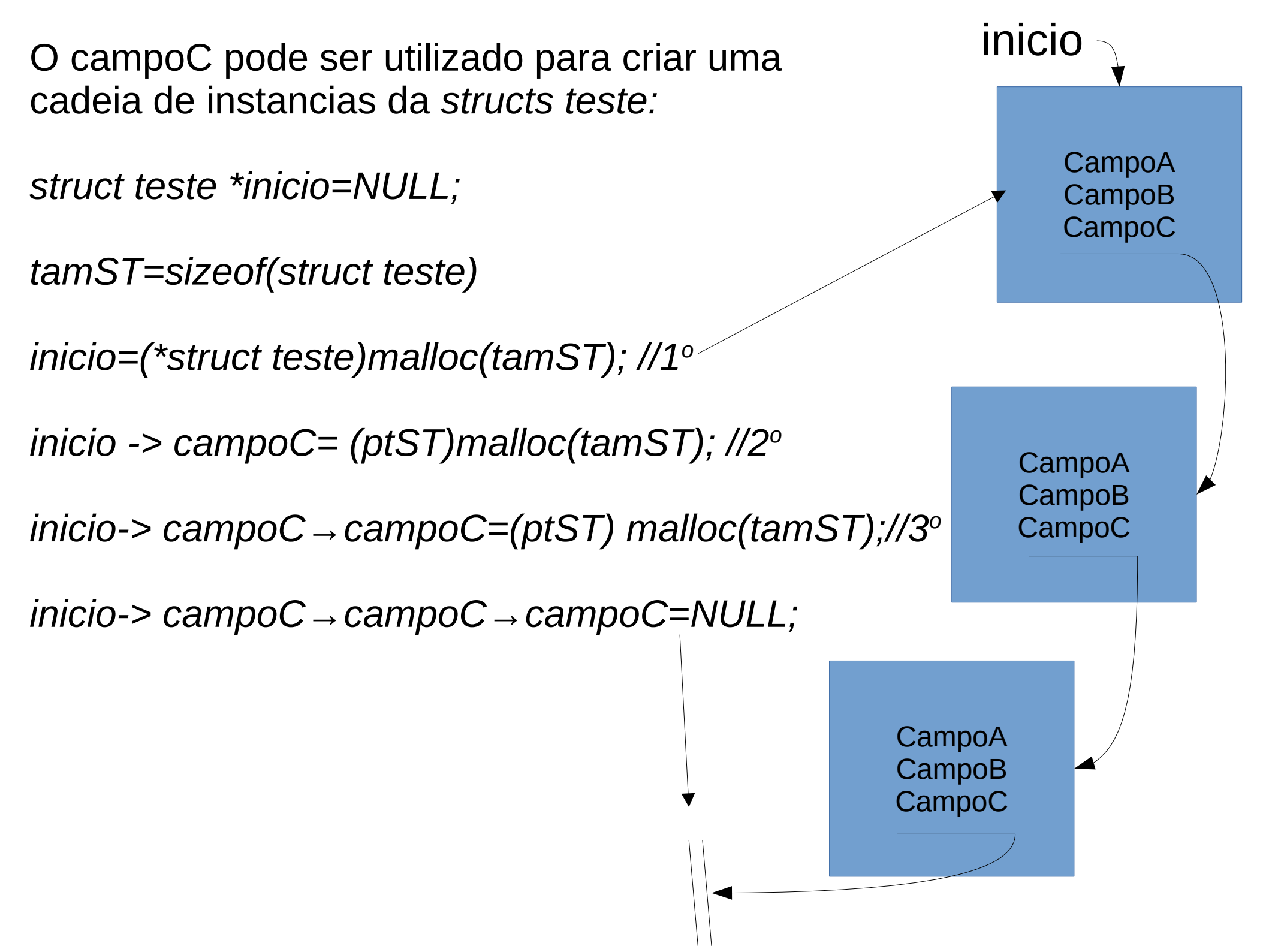
tamST=sizeof(struct teste)

*inicio=(*struct teste)malloc(tamST); //1º*

inicio -> campoC= (ptST)malloc(tamST); //2º

inicio-> campoC → campoC=(ptST) malloc(tamST);//3º

inicio-> campoC → campoC → campoC=NULL;

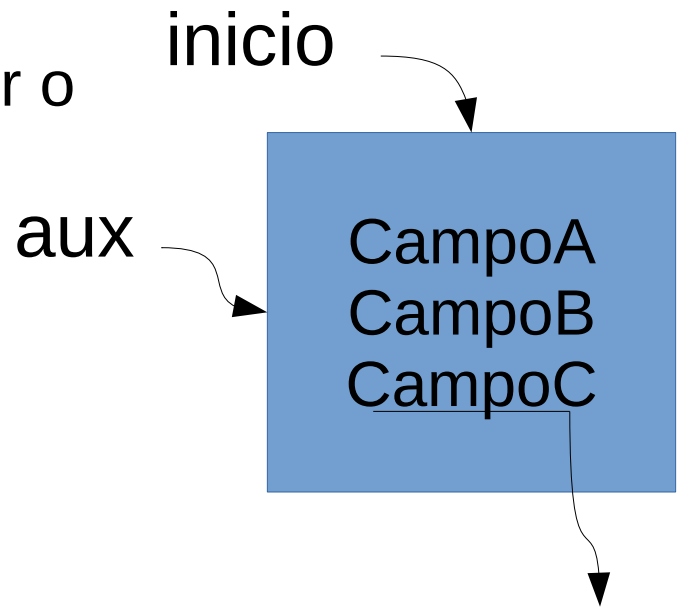


Forma mais inteligente de criar uma sequência de “N” elementos:

Usando um ponteiro auxiliar:

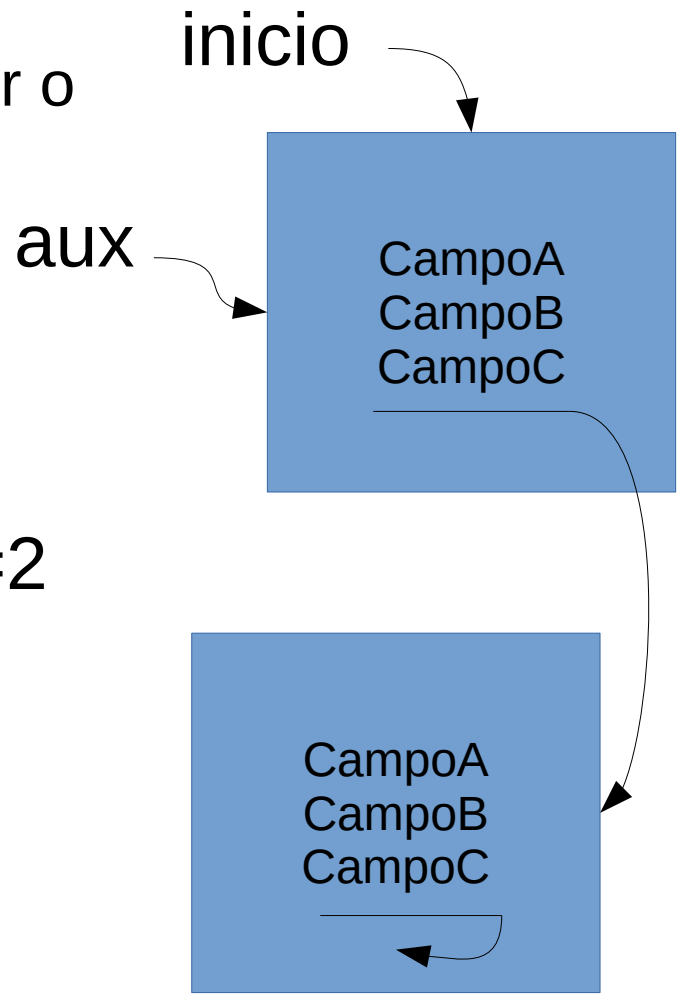
```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=1;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)



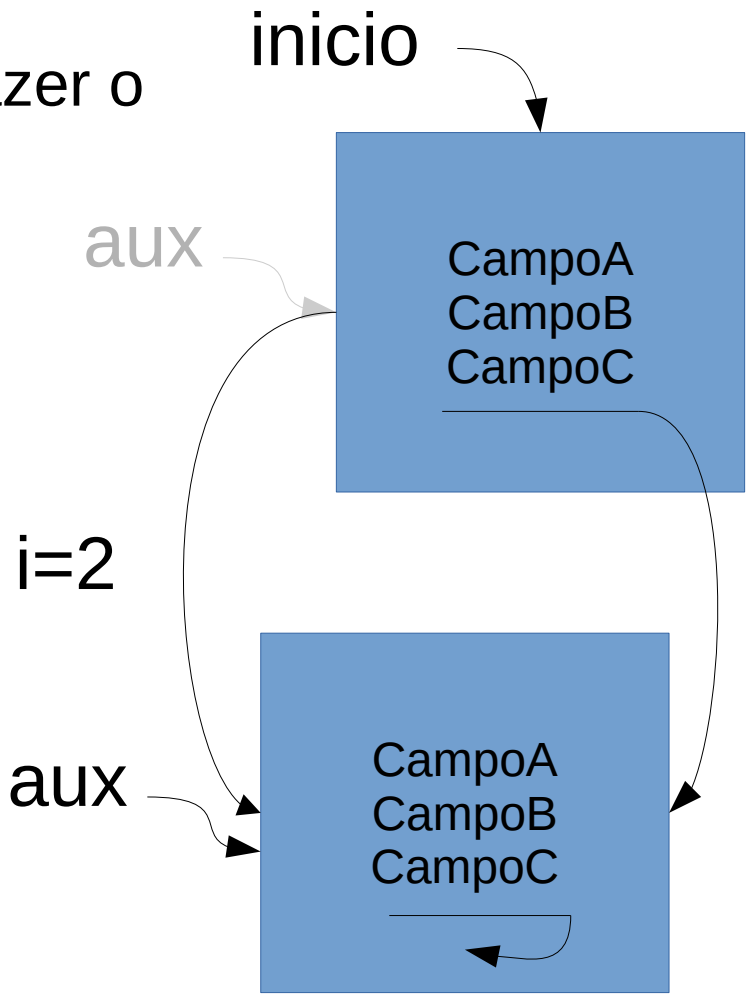

```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=2;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)




```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=1;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

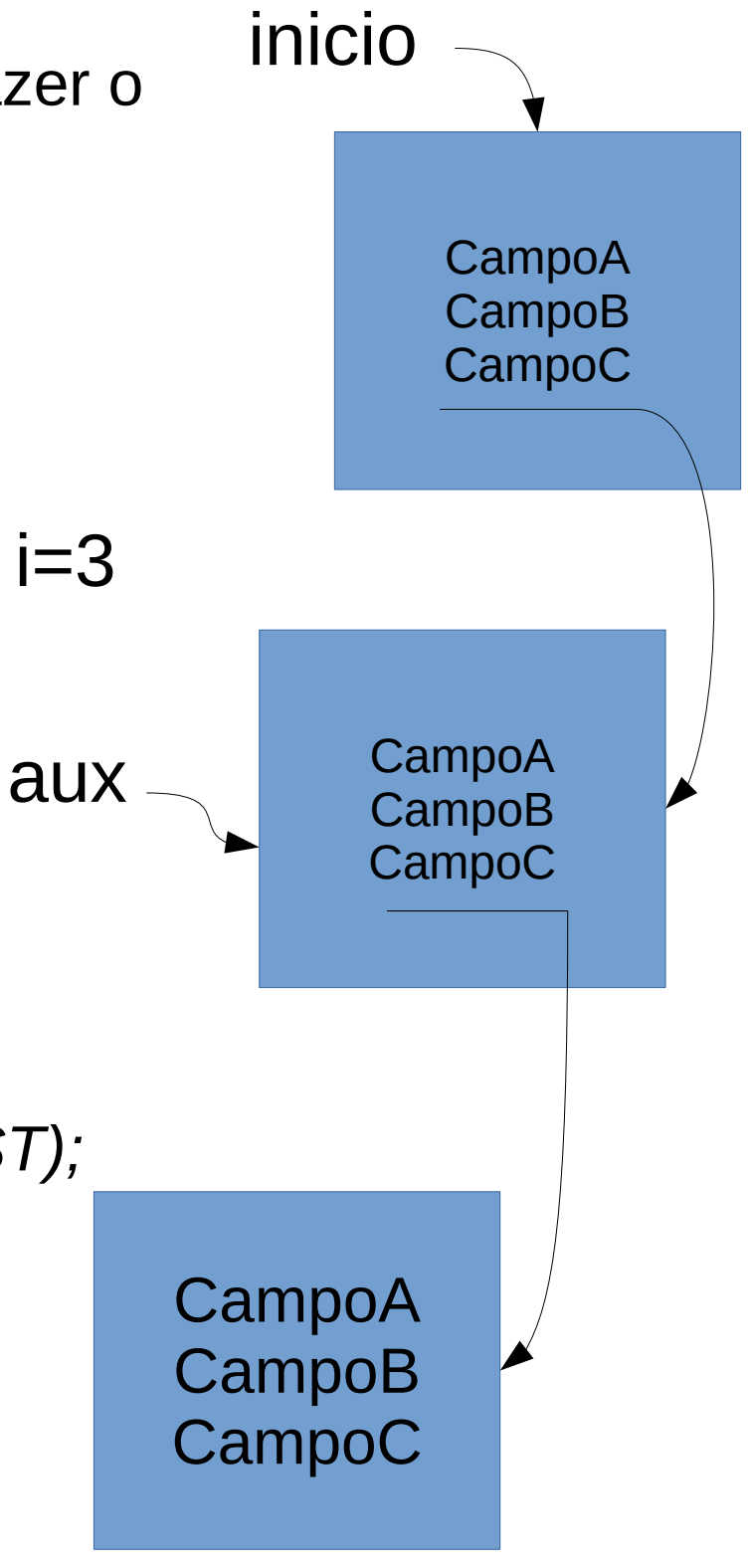
Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)




```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=2;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

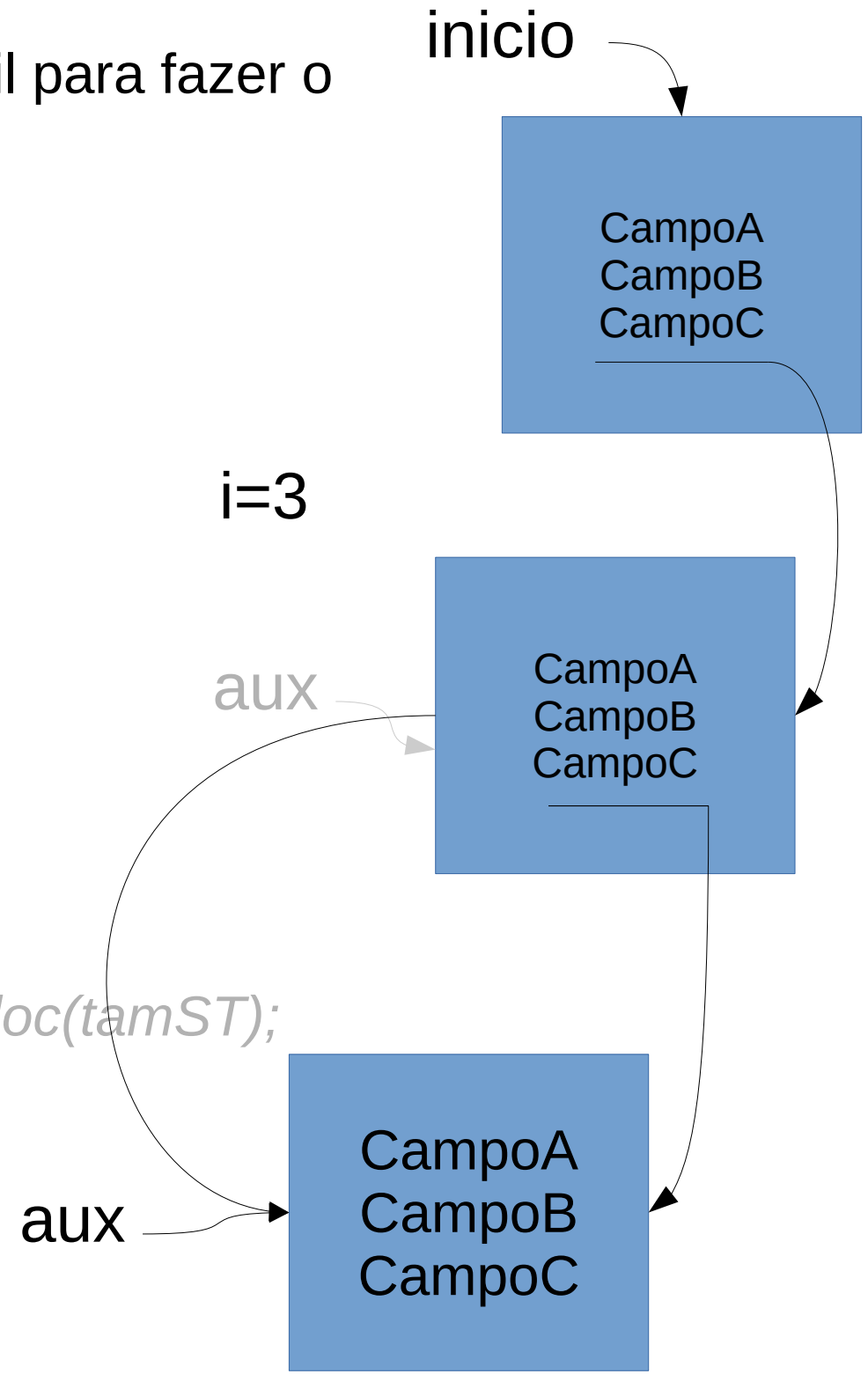
Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=2;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

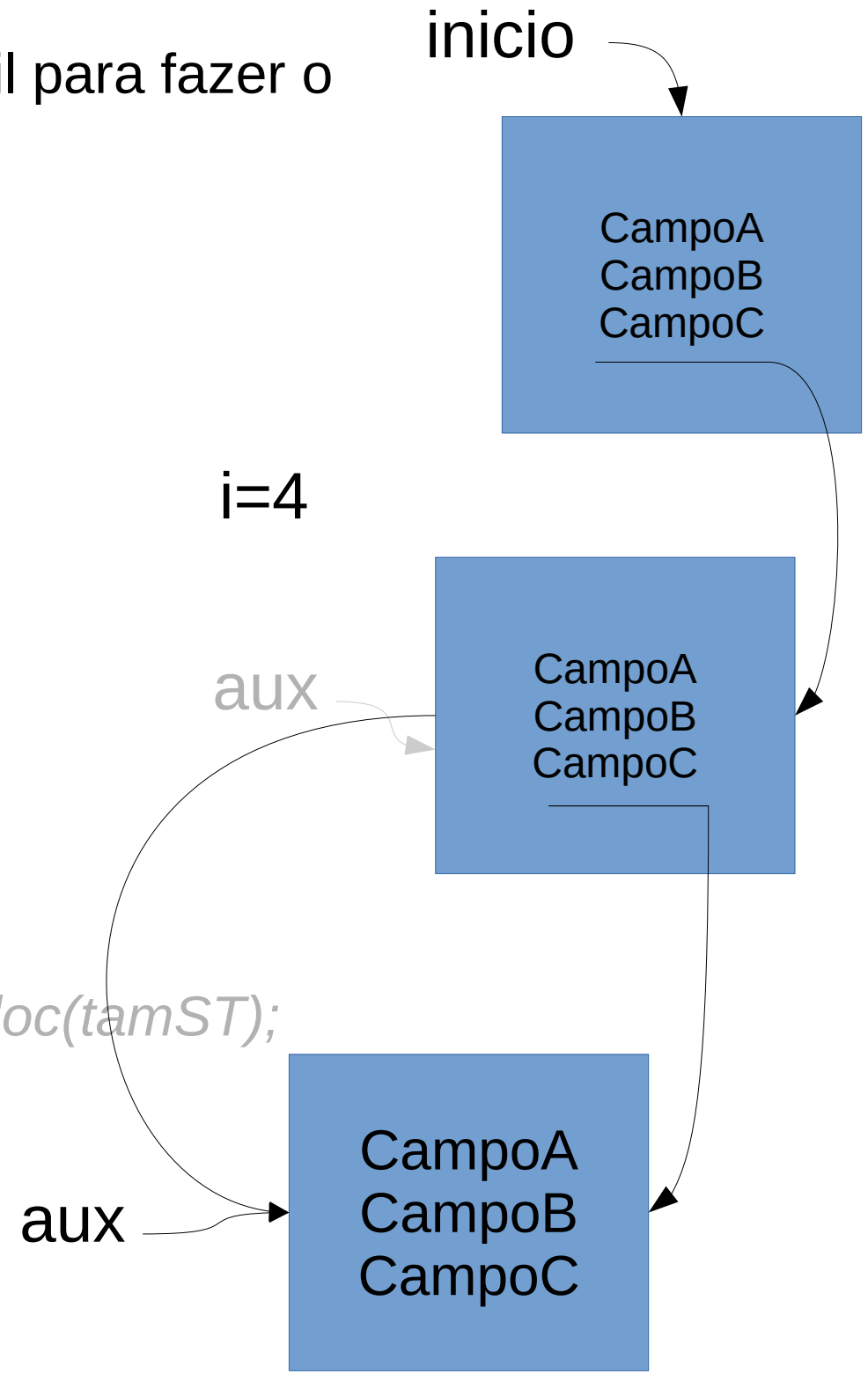
```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=2;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

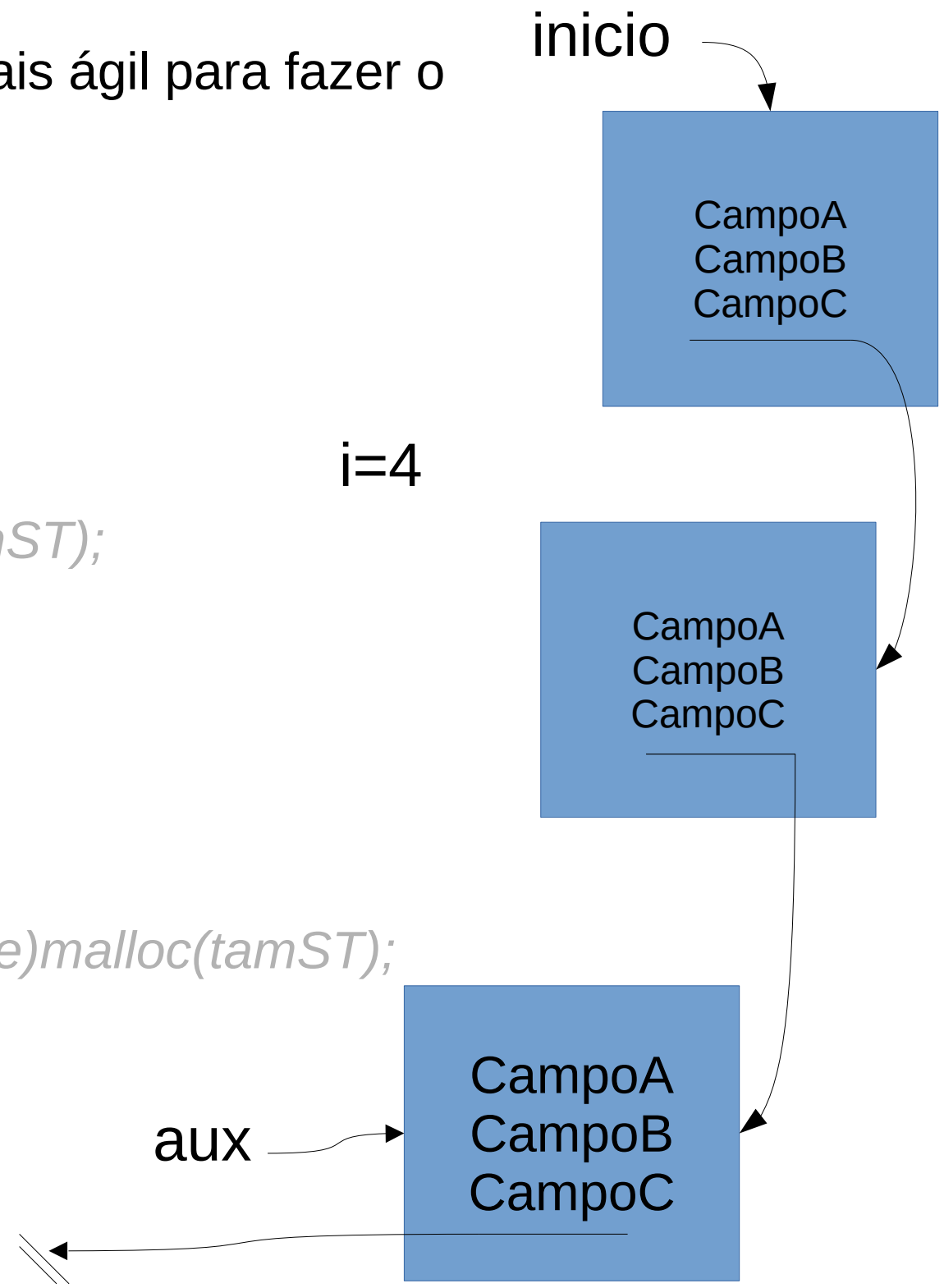
```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=2;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

Quebra laço
i>N



Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

```
struct teste *aux=NULL;

tamST=sizeof(struct teste);

inicio =(*struct teste)malloc(tamST);

aux=inicio;

N=3;

for (i=2;i<=N;i++)
    aux→campoC= (*struct teste)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



- No arquivo codigosRevisaoC.c:

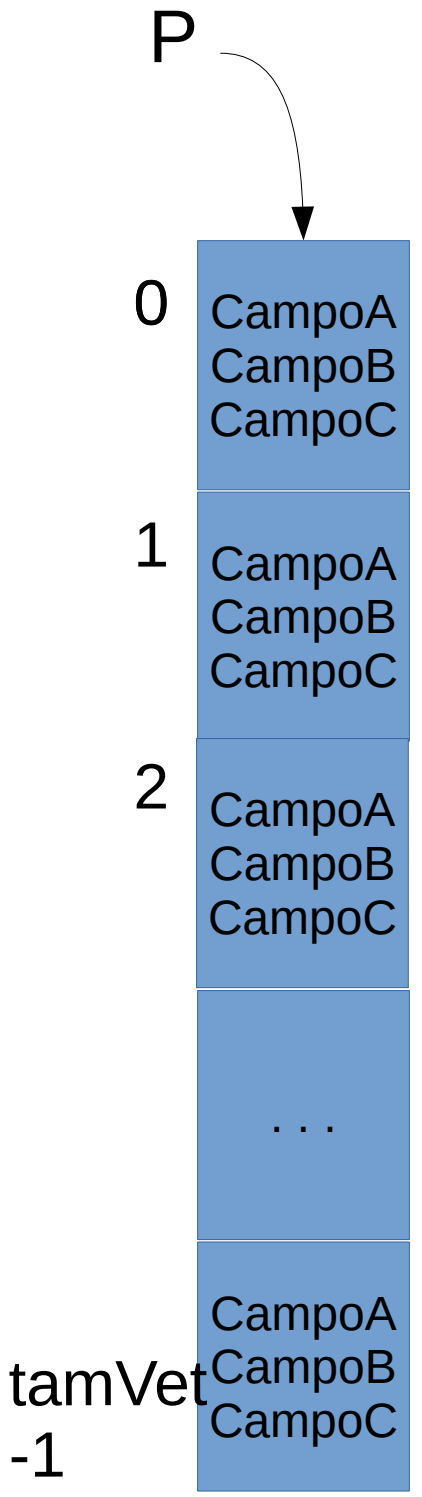
  Faça “#define TEST 120”, salve, compile, execute e analise os “prints” exibidos.
- Os resultados são conforme se espera?

Vetor de structs

- Ao invés de produzir uma casa por vez (sequência encadeada) podemos criar uma rua inteira de casas de uma única vez:
  - Vetor de structs.

Vetor de structs

```
struct teste{
      tipo campoA;
      tipo campoB;
      tipo campoC;
};

struct teste *P=NULL;
int tamVet;
puts("entre com o tamanho de vetor");
fscanf("%i",&tamVet);

P=(*struct teste) malloc(tamVet*tamST);
if (P != NULL)
{ /*...*/ }
else
{ ERRO }
```

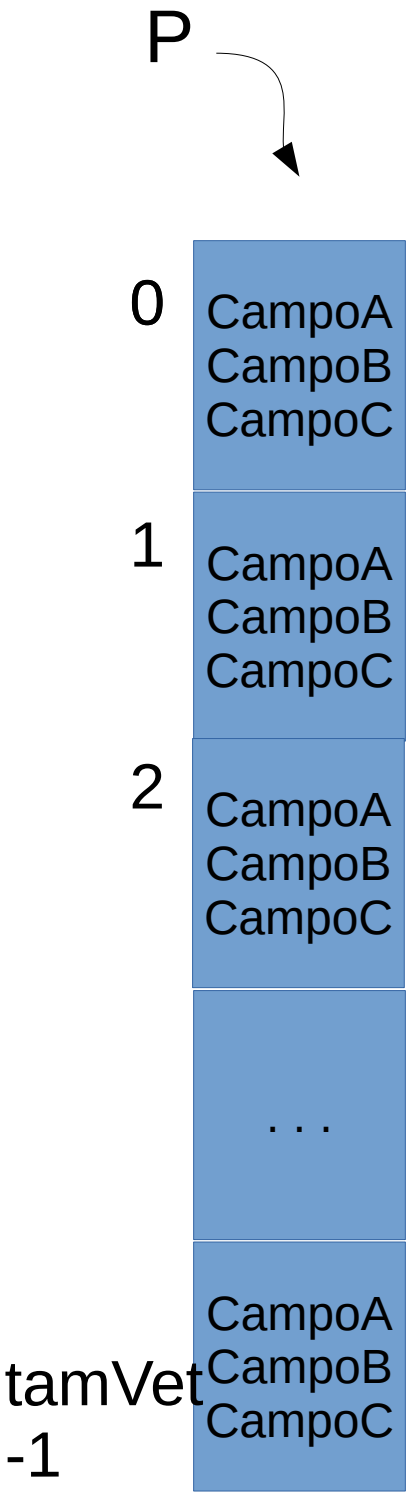


Vetor de structs

```
struct teste{
    tipo campoA;
    tipo campoB;
    tipo campoC;
};
```

```
for (i=0;i<tamVet;i++)
    P[i].campoA = a  /* (P+i)→campoA=a */
    P[i].campoB = b  /* (P+i)→campoB=b */
    P[i].campoC = c  /* (P+i)→campoC=c */
```

- *Desnecessário um campo de ligação entre os elementos;*
- *Encadeamento pré-determinado e implícito.*



Possibilidade: encadeamento explícito dentro do vetor

```
struct teste{
      tipo campoA;
      tipo campoB;
      int campoC;
};

Int tamVet=10;
Int tamST = sizeof(struct teste);
P=(*struct teste) malloc(tamVet*tamST);
if (P != NULL)
{ for (i=0;i<tamVet-1;i++)
                P→campoC=i++;
   P->campoC=-1;
}
else
{ ERRO }
```

CampoC é o elo de ligação no encadeamento