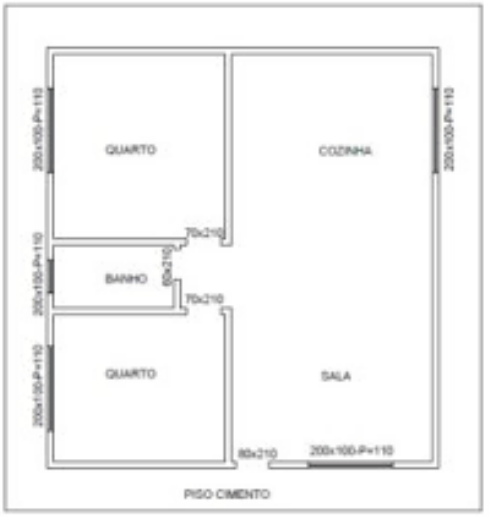
É importante entender e utilizar a linguagem gráfica na representação da instância de uma variável *struct* e a sua associação com a manipulação de campos

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 baixa;



O modelo da struct é descrito pelos seus campos: struct teste{ campoA;
campoB;
};

A casa é construída em um endereço no logradouro;

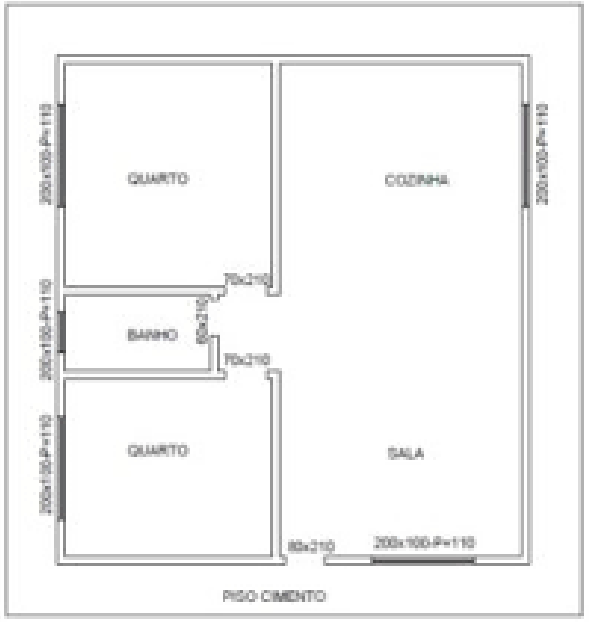
A struct é instanciada em endereço de memória;

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;
};
```

não instancia a struct apenas define seu modelo de dados

Isso também **não instancia** a struct apenas instancia três apontadores do tipo struct teste

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

```
P=(struct teste*) malloc(sizeof(struct teste));
```

P é o endereço da struct (casa)

Alocação dinâmica: uma maneira de instanciar a struct

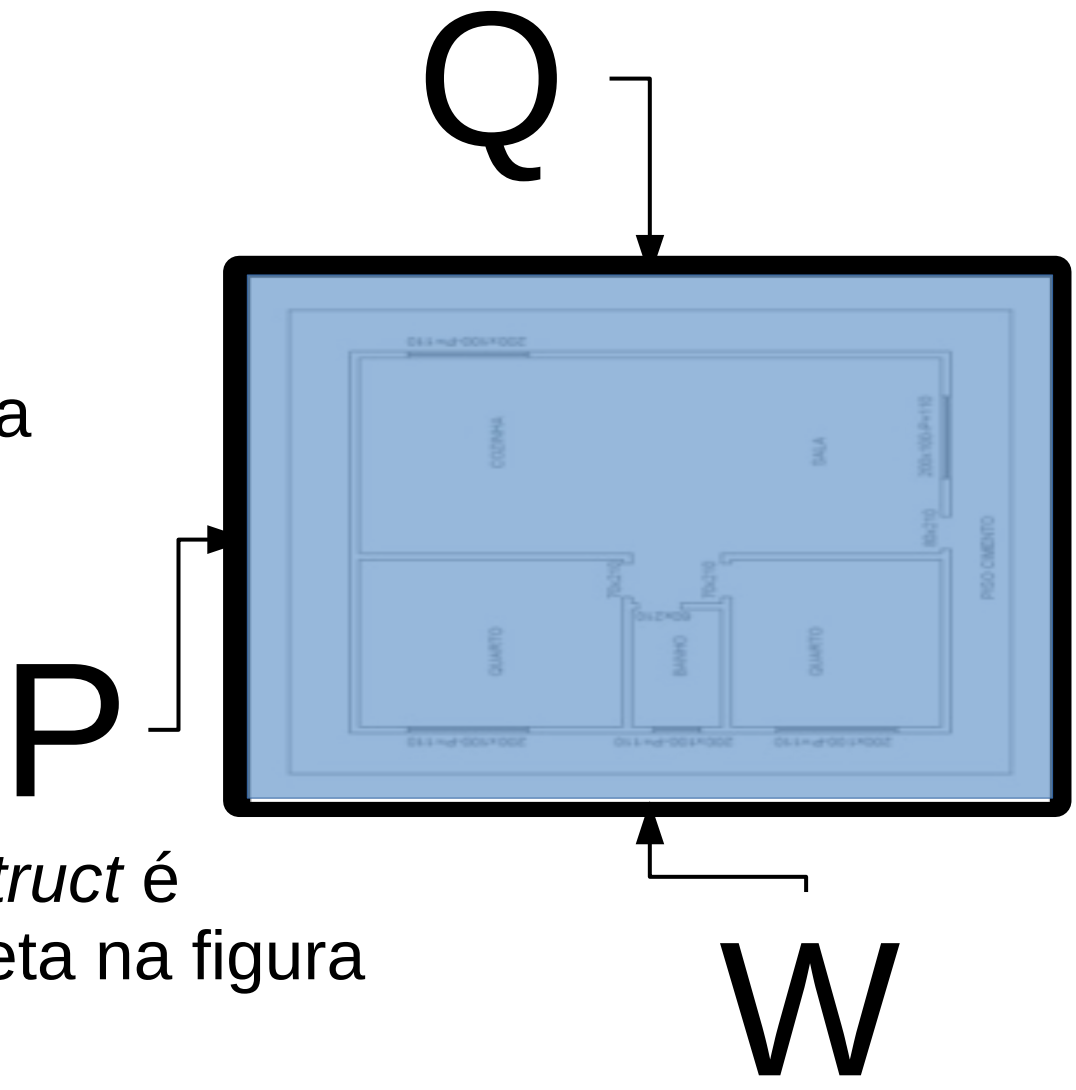
```
struct teste Z;
```

Alocação estática: outra maneira de instanciar a struct

A variável Z é a própria *struct* (*casa)*

W=Q=P;

P, Q e W contêm o endereço da mesma *struct* (casa)



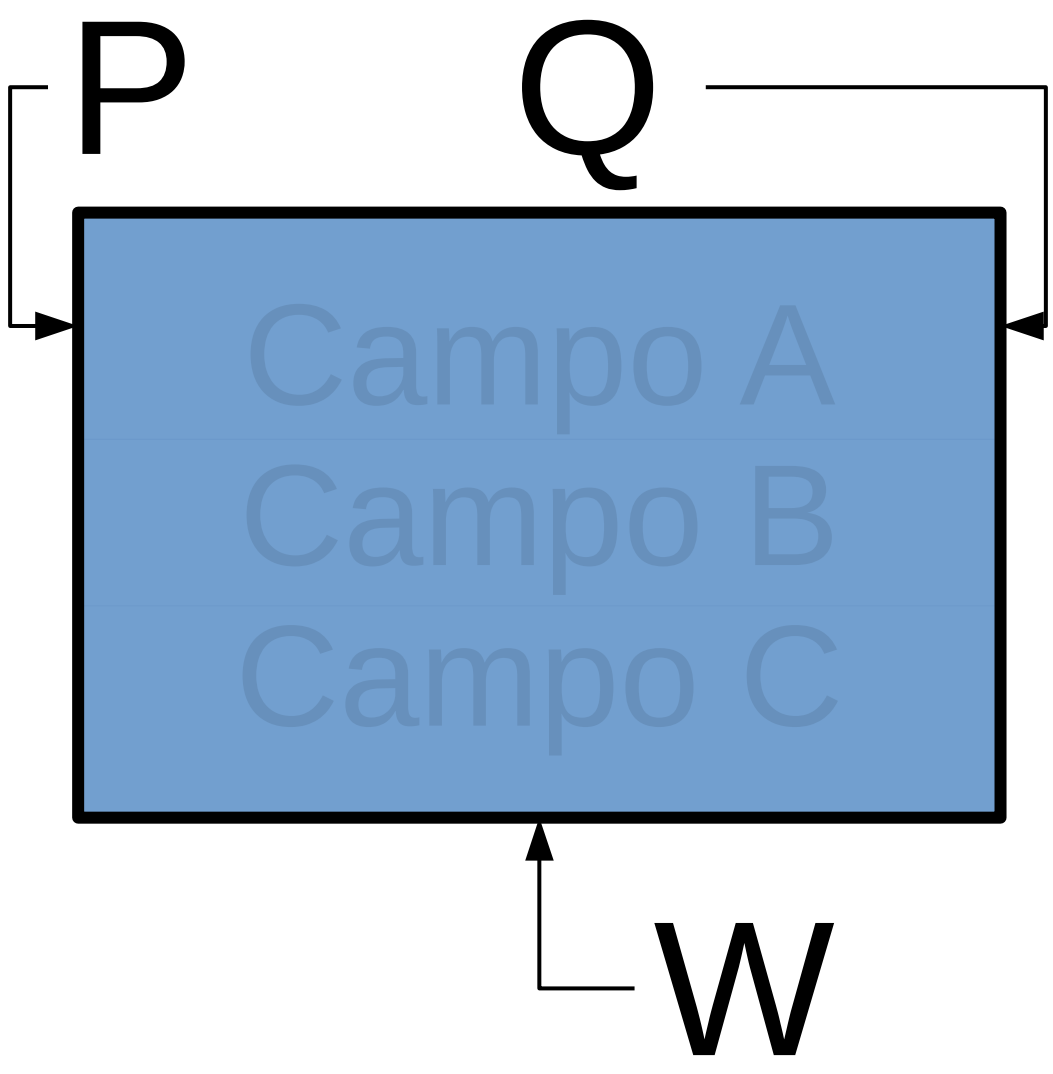
Graficamente, a instância da *struct* é representada pela moldura preta na figura (parede externa da casa);

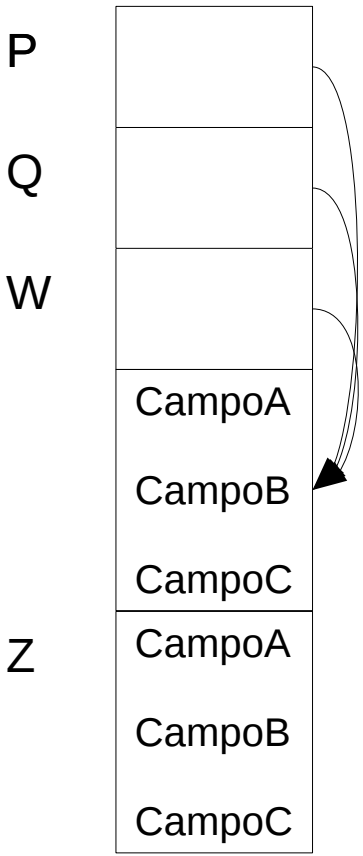
Resultado:
P, **Q** e **W**

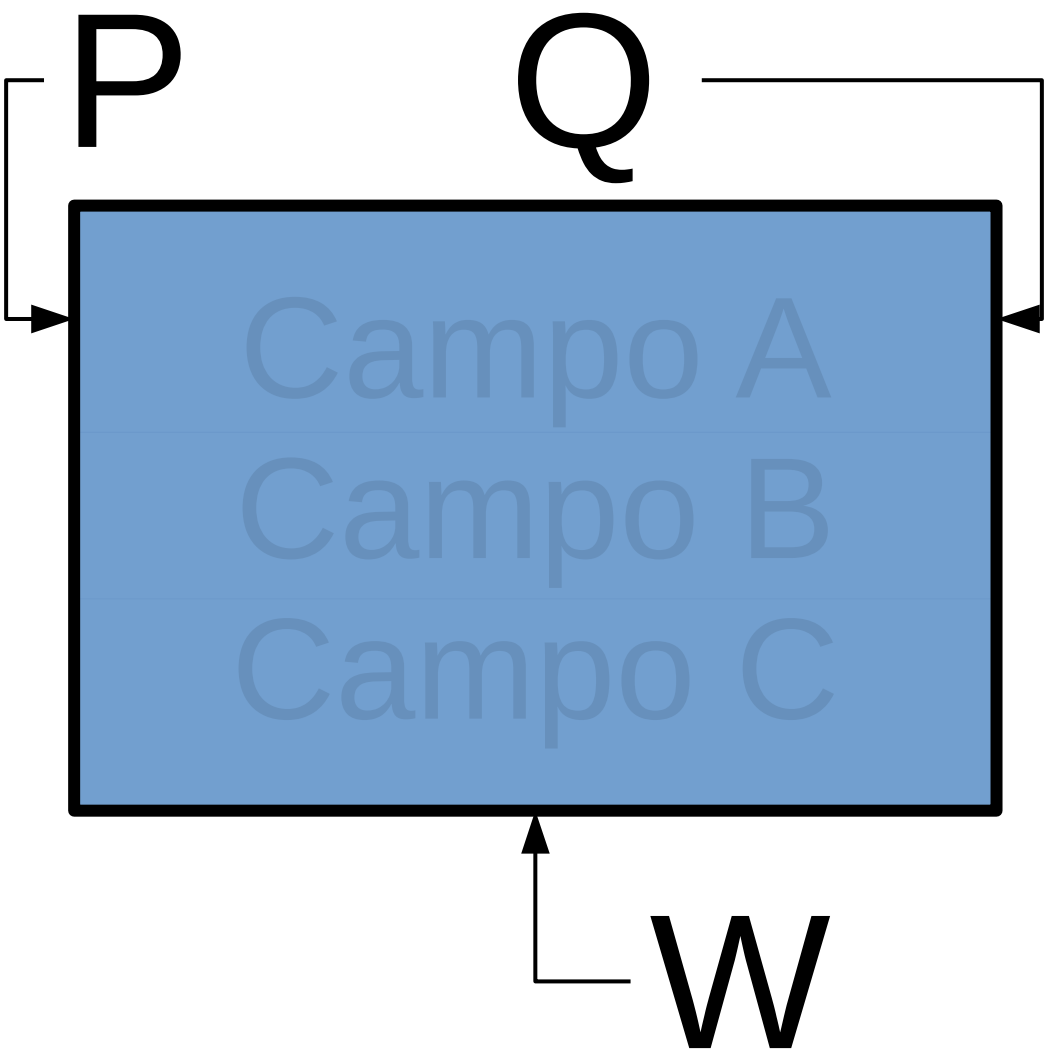
- **Apontam** para o endereço da mesma **instância** de *struct teste* (na verdade o endereço do primeiro byte da *struct*)

  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 **não** podem ser acessados **diretamente** pelos seus identificadores;

Acesso aos campos:

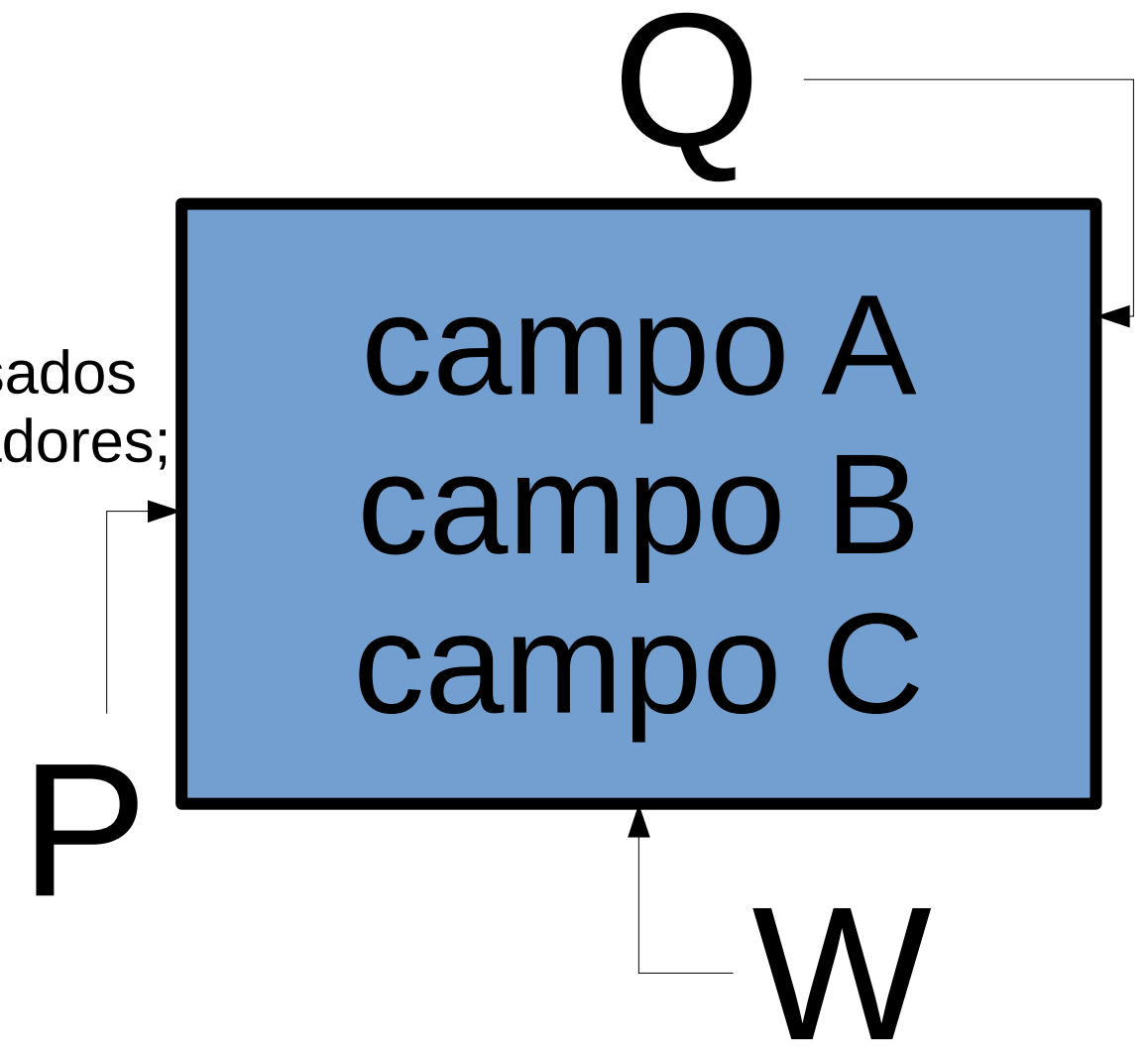
ponteiroInicializado->campo

P->campoC

Ou

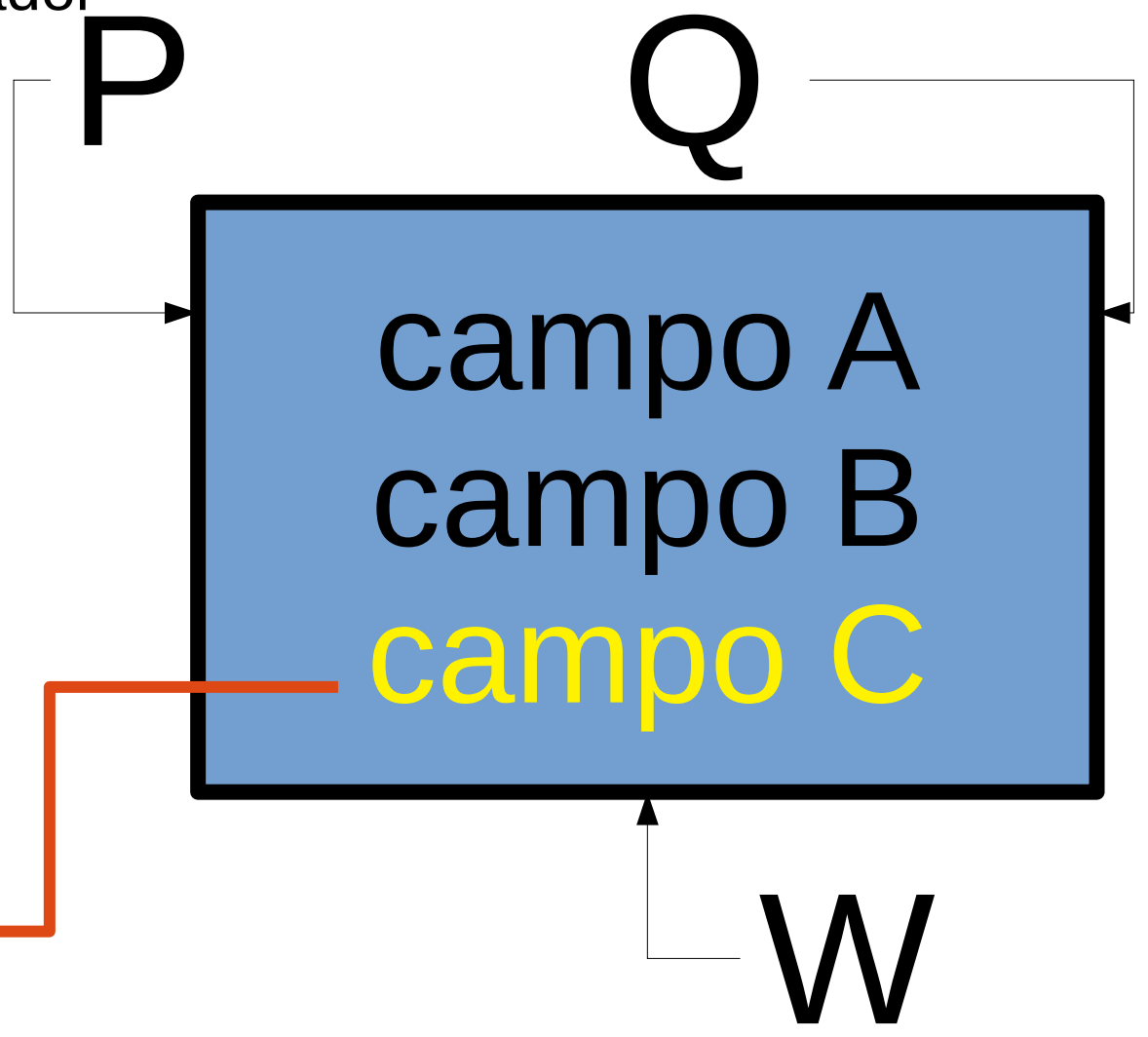
VariávelStruct.campo

Z.campoA



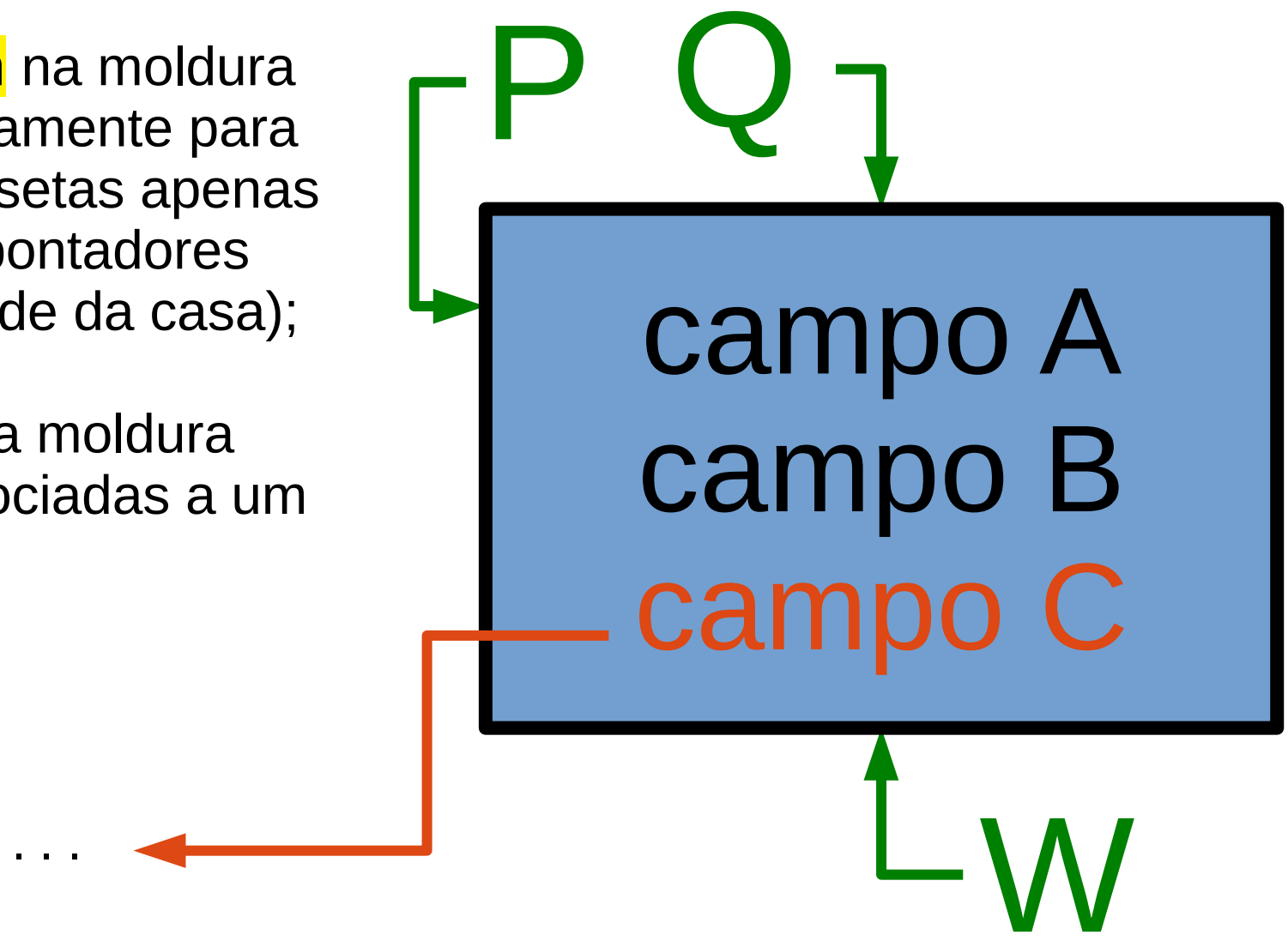
Um campo pode ser um apontador para uma *struct* cujo modelo é Igual ao que o contém.

```
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;

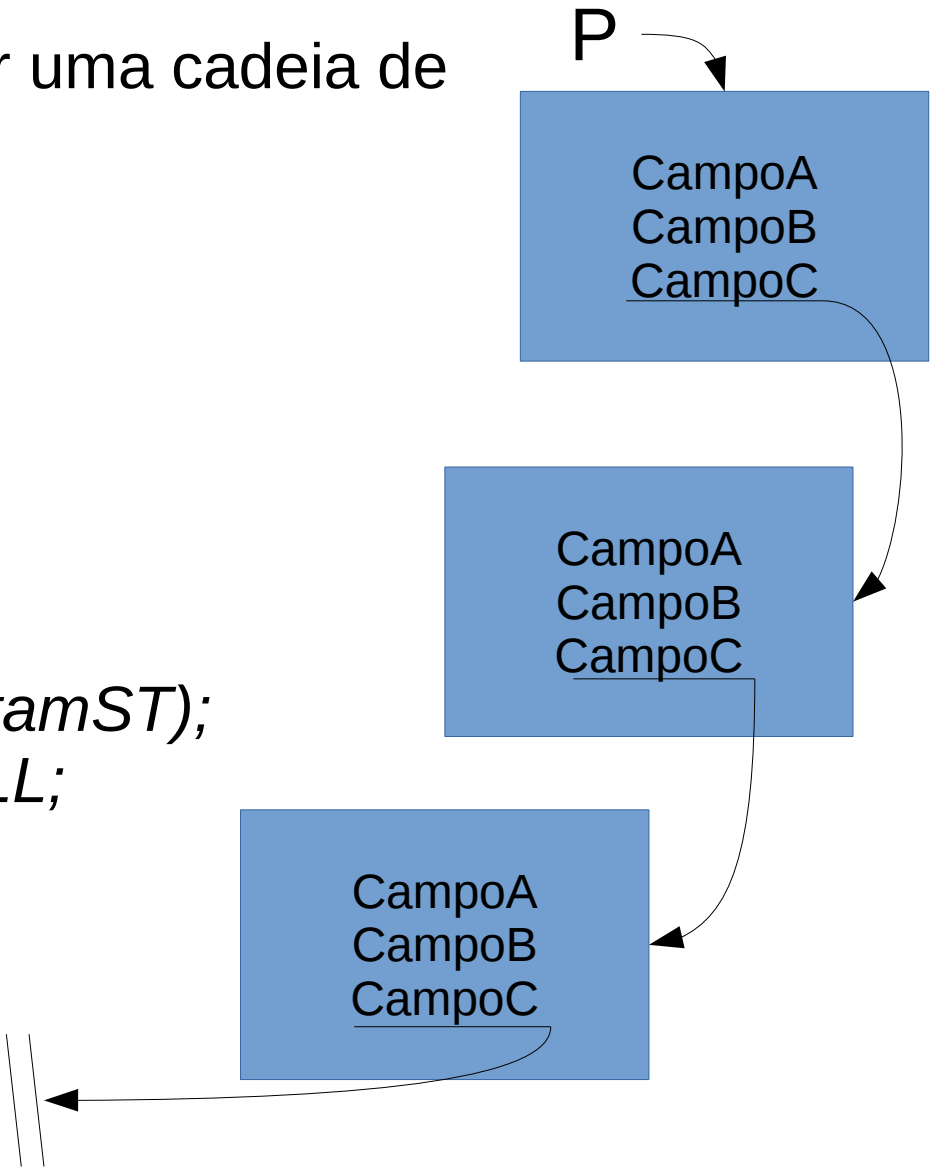


O campoC pode ser utilizado para criar uma cadeia de instancias da *structs teste:*

*typedef struct teste * ptST;*

tamST=sizeof(struct teste)

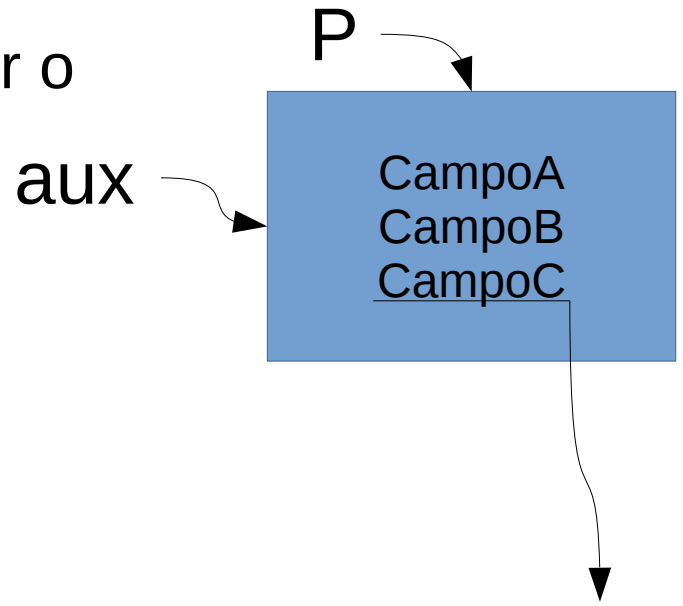
P=(ptST)malloc(tam);
P→campoC= (ptST)malloc(tamST);
P→campoC→campoC=(ptST) malloc(tamST);
P→campoC→campoC→campoC=NULL;



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

```
ptST aux;

P=(ptST)malloc(tamST);

aux=P;

Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

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



ptST aux;

P=(ptST)malloc(tamST);

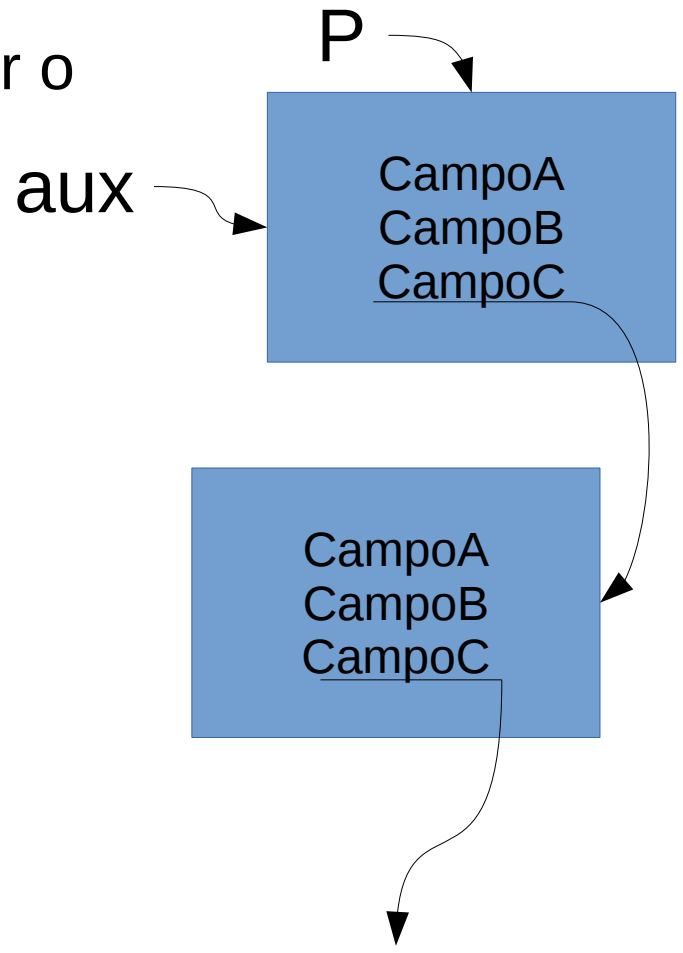
aux=P;

```
Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

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

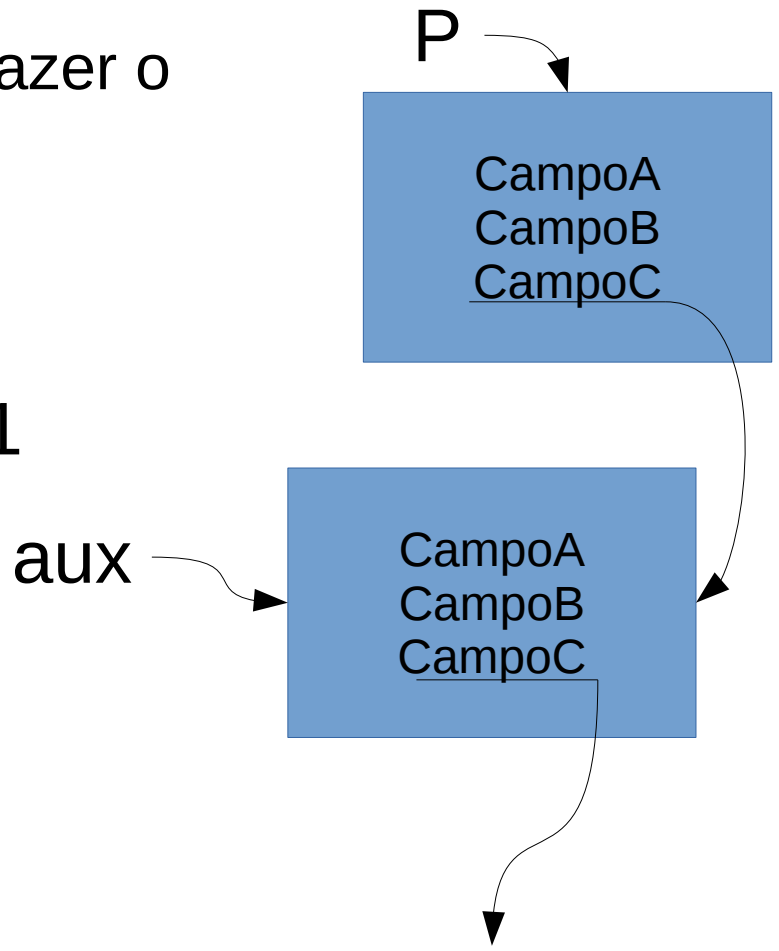
```
ptST aux;

P=(ptST)malloc(tamST);

aux=P;

Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```

i=1



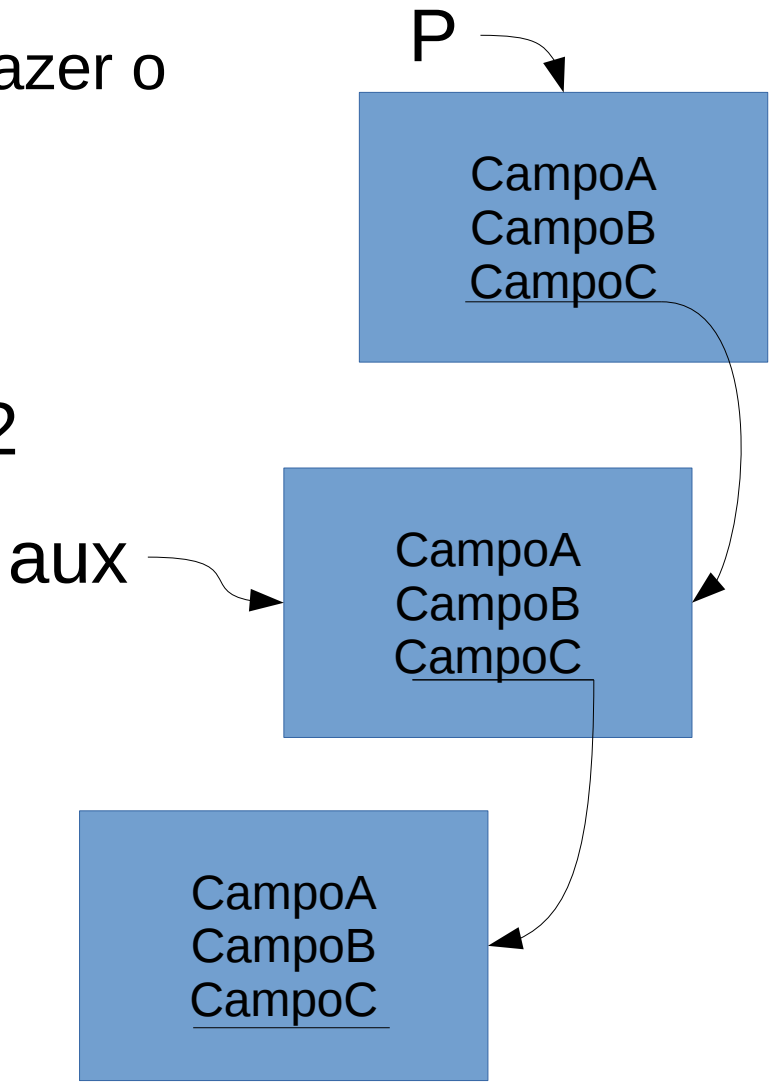
Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

```
ptST aux;

P=(ptST)malloc(tamST);

aux=P;

Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



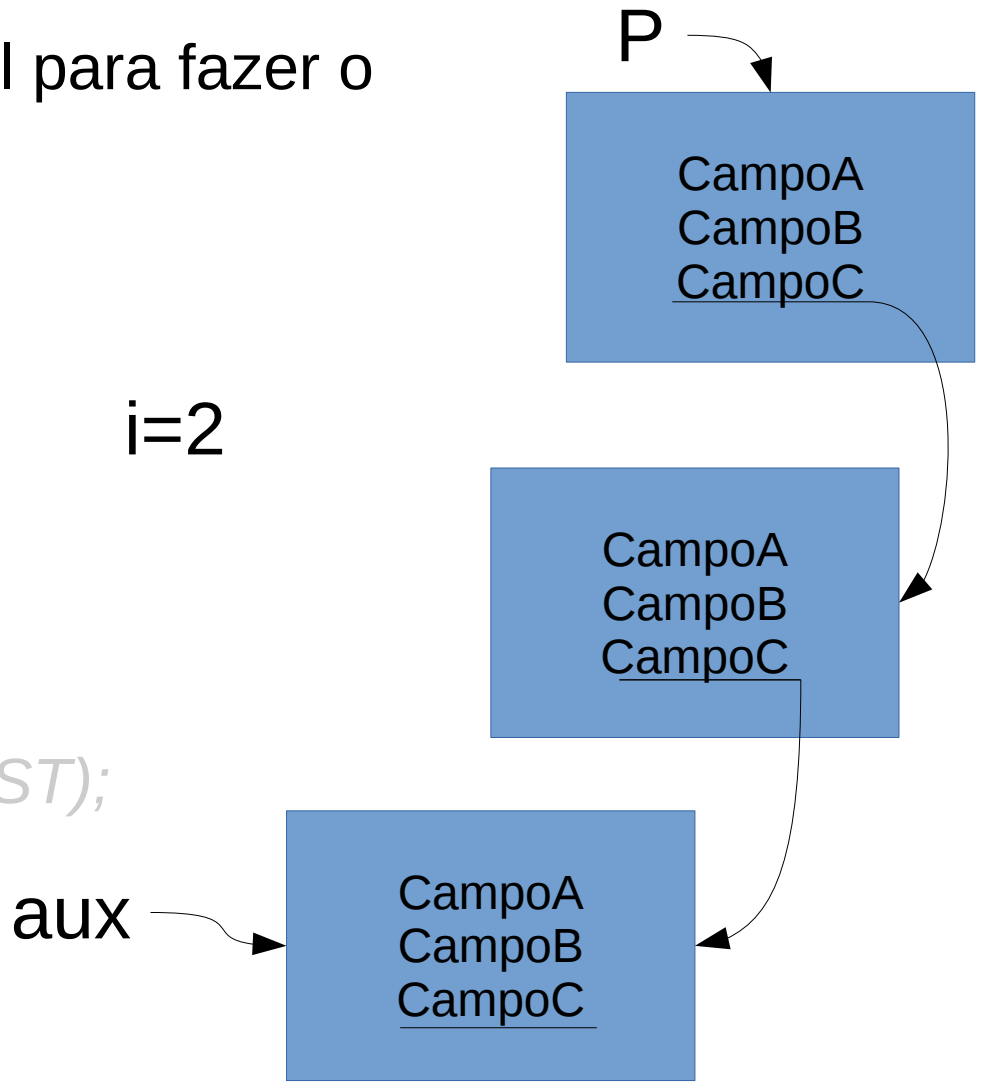
Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

```
ptST aux;

P=(ptST)malloc(tamST);

aux=P;

Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



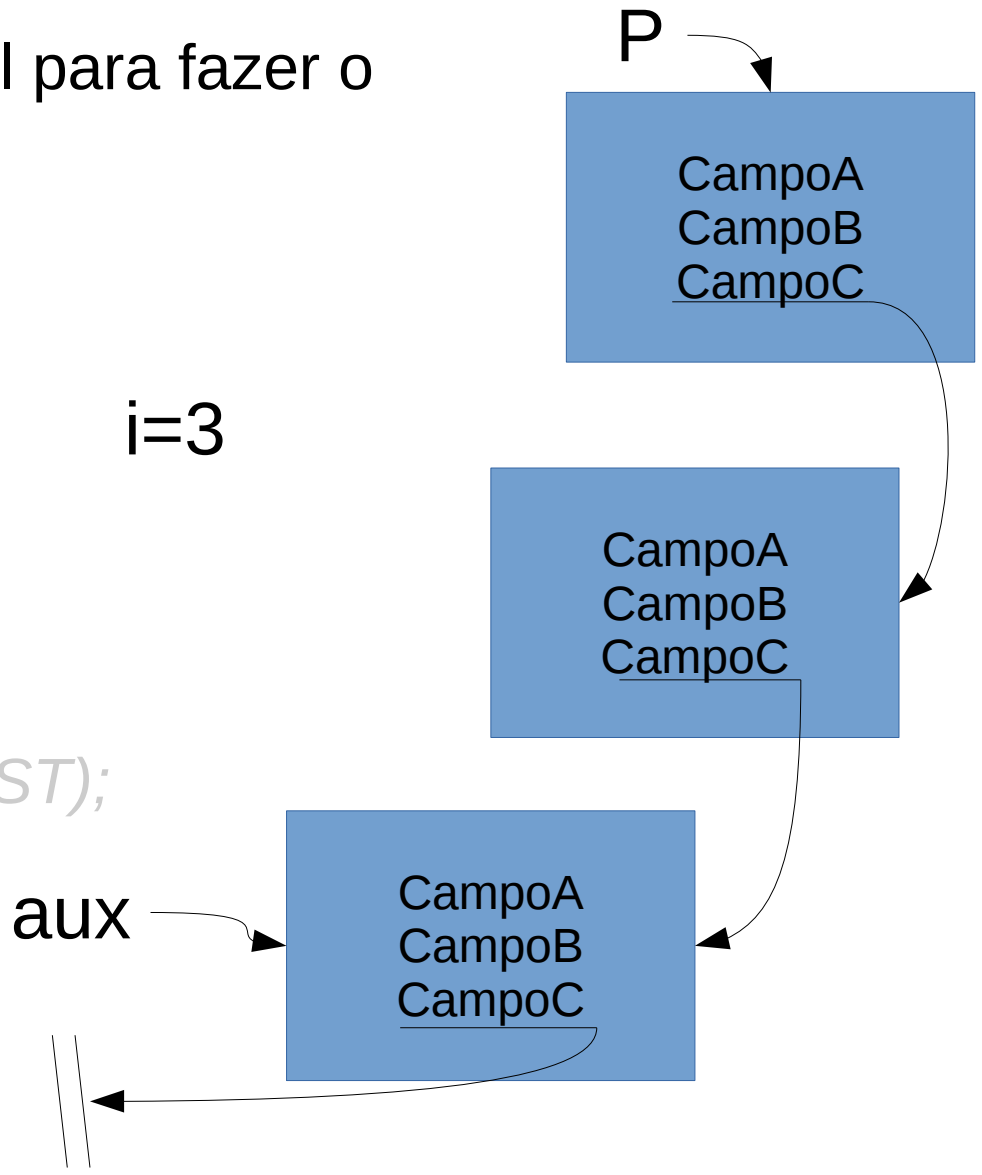
Ponteiro auxiliar: uma forma mais ágil para fazer o mesmo (e muito mais...)

```
ptST aux;

P=(ptST)malloc(tamST);

aux=P;

Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



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

```
ptST aux;

P=(ptST)malloc(tamST);

aux=P;

Para (i=1;i<=2;i++)
    aux→campoC= (ptST)malloc(tamST);
    aux=aux→campoC;

aux→campoC=NULL;
```



- No arquivo codigosRevisaoC.c:

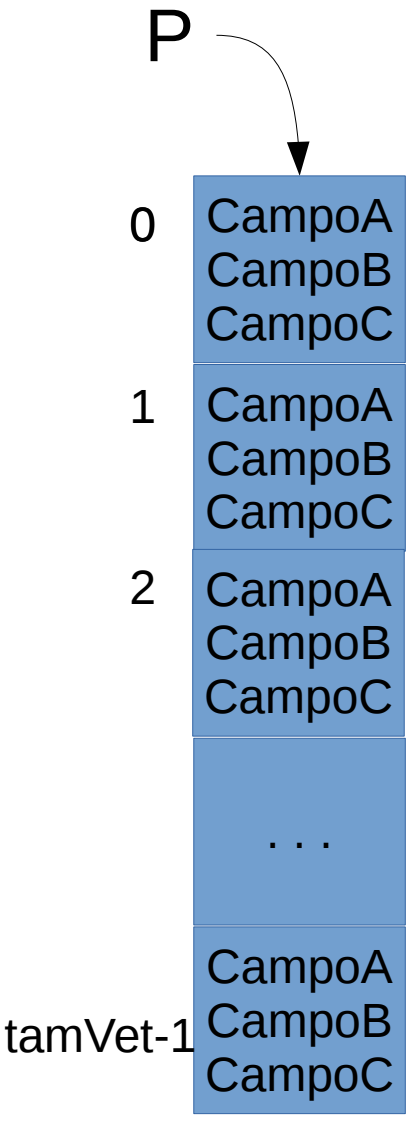
  Faça “#define TEST 120”, salve, compile, execute e analise os “prints” exibidos.
- Os resultados são conforme se espera?

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

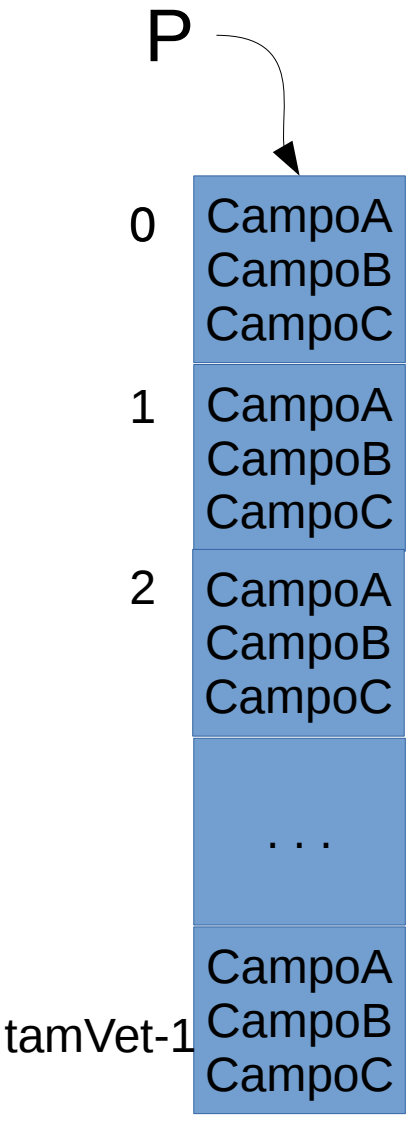


Vetor de structs

```
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;
};
P=(ptST) 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