Linguagens e Ambientes de Programação (2018/2019)

Teórica 12 (15/abr/2019)

Operadores.
Avaliação de expressões.
Tipos derivados.
Passagem de parâmetros.
Apontadores e sua utilidade.
Apontadores: Parâmetros de saída de funções.
Apontadores: Manipulação de vetores e relação entre vetores e apontadores.
Apontadores: Manipulação de memória a baixo nível.

Operadores

Eis a lista completa dos operadores do C que podem ser usados em expressões:

```
aritméticos:    +  -  *  /  %
lógicos:        !  &&  ||
relacionais:    <  >  <=  >=  ==  !=
bits:           >> << & ^ |~
atribuição:     =  +=  -=  *=  /=  %=  &=  |=  ^=  <<=  >>=
condicional     ?:
cast:           (type)
inc/dec:        expr++  ++expr  expr--  --expr
sequenciação:   ,
sizeof:         sizeof
apontadores:    *  &  ->  []
field:          .              (para acesso a campos de registos e uniões)
agrupamento:    (expr)
```

Note que em C (tal como em Java), a atribuição produz um resultado e por isso é considerada uma expressão, não um comando. O valor da expressão `v = exp` é o valor que fica na variável `v` depois da expressão `exp` ter sido avaliada e depois da atribuição ter sido concretizada.

O C suporta sobrecarga (overloading) de alguns operadores. Por exemplo, o operador + é usado para denotar três operações diferentes: a soma inteira; a soma real; a soma entre um apontador e um inteiro.

Precedências e associatividades

Precedências dos operadores por ordem decrescente de prioridade:

```
()
[] -> . expr++ expr--                          esq
! ~ ++ -- - (type) * & sizeof ++expr --expr    dir
* / %                                          esq
+ -                                            esq
<< >>                                          esq
< <= > >=                                      esq
== !=                                          esq
&                                              esq
^                                              esq
|                                              esq
&&                                             esq
||                                             esq
?:                                             esq
= += -= etc.                                   dir
,                                              esq
```

Exemplo. A complexa expressão logica abaixo não precisa de nenhum parêntesis:

```
if( year % 4 == 0 && year % 100 != 0 || year % 400 == 0 )
    printf("Ano bissexto\n") ;
else
    printf("Ano comum\n") ;
```

Avaliação de expressões

Ordem de avaliação

O compilador tem a liberdade de rearranjar as expressões por forma a otimizar a eficiência da sua avaliação mesmo que esta envolva efeitos laterais. Os parêntesis podem mesmo não ser respeitados. O que é importante é que não se viole nenhuma das regras da álgebra. O compilador também ter a liberdade de avaliar os argumentos nas chamadas das funções por qualquer ordem.

Portanto a ordem de avaliação não está geralmente definida e devemos evitar escrever expressões cujos efeitos ou resultados dependam da ordem de avaliação. Exemplos:

```
i = i++ ;          /* o valor final de i não está definido */
f(i++, i++) ;      /* o valor dos argumentos não está definido, mas o valor final de i não tem problema */
f(*p1++, *p2++) ;  /* o valor dos argumentos não está definido no caso dos dois apontadores referirem a mesma posição de memória */
f() + g()          /* qualquer das funções pode ser executada em primeiro lugar */
```

O último exemplo só é problemático se as duas funções produzirem efeitos laterais que sejam dependentes da ordem de avaliação.

Pontos de sequenciação

Mas nem tudo está indefinido na ordem de avaliação de expressões. Temos os **pontos de sequenciação** para nos ajudar. São pontos dentro das expressões que garantem que os efeitos laterais das expressões anteriores já foram completamente concretizados.

Os pontos de sequenciação do C estão associados aos seguintes operadores:

```
,  &&  ||  ?:
```

Hierarquia dos tipos numéricos

Os tipos numéricos podem ser livremente misturados em expressões. Quando isso acontece, são efetuadas promoções automáticas de tipo de acordo com a seguinte hierarquia:

```
char short
int
unsigned int
long int
unsigned long int
long long int
unsigned long long int
double
long double
```

Conversões automáticas de tipos numéricos

Aplicam se sempre sucessivamente as seguintes regras na avaliação de uma expressão:

1. char, short, enum -> int; float -> double
2. Para cada operando binário com operandos de tipo diferente, o menos importante é convertido no tipo do mais importante, antes de se efetuar a operação.
3. Nas atribuições (v = exp) o tipo do valor da direita é convertido num valor do tipo da esquerda antes de se fazer a atribuição. Muitas vezes isso implica uma despromoção de tipo e uma truncagem de valor.

Exemplos:

```
5 + 2.0 * 'a'
(5.3 + 5) + 7
int i = 200 * 'a'
```

Tipos derivados

Há 4 variedades de tipos derivados:

- Vetores (arrays)
- Registos (structures)
- Uniões
- Apontadores

Podem ser usados diretamente, como na seguinte definição de variável

```
struct
{
    double re, im ;
} myvar ;
```

mas muitas vezes usa-se a construção **typedef** para lhes associar um nome. Exemplos:

```
typedef struct
{
    double re, im ;
} Complex;

typedef union
{
    int x;
    char c;
} IntChar;

typedef int Vetor[5];

typedef int Matriz[2][3];
```

```
typedef char String[256];

typedef void *Pointer;

typedef int *IntPointer;

typedef IntPointer *PointerToIntPointer;

typedef int **PointerToIntPointer;

typedef int IntFunction(void);

typedef IntFunction IntFunctionArray[100];
```

Eis algumas definições de variáveis usando os tipos definidos anteriormente:

```
Complex z;
IntChar u;
Vetor vetor;
Matriz matriz;
String str;
Pointer v;
IntPointer pt;
IntFunctionArray ops ;
```

Um tipo soma

Agora exemplo maior, que combina diversos tipos derivados e que mostra como é que se definem habitualmente **tipos soma** em C. Analise bem.

```
#define MAX_SHAPES   200

typedef enum {
    LINE, CIRCLE, RECTANGLE
} ShapeKind ;

typedef struct {
    double x, y;
} Point;

typedef struct {
    Point p1, p2;
} Line;

typedef struct {
    Point centre;
    int radius;
} Circle;

typedef struct {
    Point top_left;
    double width, height;
} Rectangle;

typedef struct {   // Isto é um tipo SOMA, programado com a ajuda duma UNION
    ShapeKind kind;
    int color;
    union {
        Line line;
        Circle circle;
        Rectangle rectangle;
    } u;
} Shape;

typedef Shape Shapes[MAX_SHAPES];
```

Passagem de parâmetros para funções

Tipos primitivos e tipos registo

Os parâmetros de tipos primitivos e de tipos-registo são passados por "valor", sendo portanto sempre **parâmetros de entrada**: os valores dos parâmetros circulam apenas de fora da função para dentro da função.

Do ponto de vista técnico, os parâmetros de tipos primitivos e de registos são implementados como simples variáveis locais que têm a particularidade de serem inicializadas no momento da chamada da função. Se, porventura, dentro duma função se fizer uma atribuição a um desses parâmetro, está-se apenas a alterar a variável local; nada está a ser alterado no exterior da função.

Tipos vetores

Sobre a passagem de vetores como parâmetro há duas particularidades especiais. Para se trabalhar em C com vetores, é preciso realmente assimilar estas duas ideias:

1. Quando se define um parâmetro de tipo vetor numa função, **nunca se deve indicar o tamanho da primeira dimensão**, pois as funções do C aceitam vetores em que essa primeira dimensão pode ter um tamanho qualquer. O tamanho da primeira dimensão é normalmente passado num

argumento inteiro, ao lado do vetor.

Esta opção da linguagem C tem a seguinte justificação: assim a função fica mais geral e torna-se útil em mais situações.

2. Os parâmetros de tipo vetor são **parâmetros de entrada e saída**. Dentro duma função, um parâmetro de tipo vetor representa sempre o vetor original que foi passado, porque a passagem é implementada usando um apontador.

Exemplo: Leitura e preenchimento integral dum vetor de tamanho `n`.

```
void read_values(int vector[], int n)
{
    int i;
    for( i = 0 ; i < n ; i++ ) {
        printf("vector[%d] = ", i);
        scanf("%d", &vector[i]);
    }
}
```

Apontadores

Introdução: Variáveis e apontadores

Os programas processam dados armazenados na memória do computador. O mecanismo mais simples que permite aceder a essa memória são as **variáveis**. Cada variável tem um nome e um tipo e representa um pedaço da memória do computador onde podem ser guardados valores desse tipo.

Através da utilização de variáveis, é possível ir muito longe na escrita de programas em C. Mas há algumas situações em que o uso de variáveis não é suficiente e é necessário usar mecanismo mais flexível de manipulação da memória:

- Trata-se do mecanismo dos **apontadores**!

Conceitos básicos sobre apontadores

Portanto os apontadores constituem uma segundo mecanismo de acesso à memória:

- Em vez de se usar um nome para identificar um pedaço da memória, usa-se diretamente o endereço dessa zona de memória para a identificar e chama-se a esse endereço um **apontador**. Diz-se que o apontador **aponta** para uma determinada zona de memória.

Normalmente os apontadores são guardados em **variáveis de tipo apontador**.

Em C há tipos específicos para representar apontadores. O tipo dos apontadores que apontam para valores de tipo T escreve-se:

```
T *
```

Para exemplificar, eis a definição duma variável de tipo apontador para inteiro:

```
int *pt ;
```

Em C, há duas operações principais para manipular apontadores: o operador **`&`** permite obter um apontador para uma variável qualquer, assim

```
&Variável
```

e o operador **`*`** permite aceder ao valor apontado por um apontador, assim:

```
*Apontador
```

Vejamos um exemplo. Abaixo, define-se uma variável inteira normal `v`. A seguir define-se uma variável de tipo apontador para inteiros `x` e fazemo-la apontar para a variável v. Depois, usamos o apontador para colocar o valor 42 na zona de memória apontada por `x`.

```
int v = 0 ;
int *x = &v ;
*x = 42
```

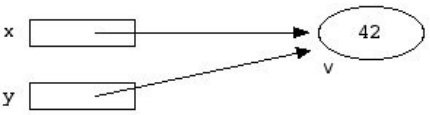
A seguinte figura, ilustra a situação após a atribuição do valor 42 a `*x`.

Repare que a variável `v` pode ser acedida de duas formas: (1) usando o nome `v`; (2) usando a expressão `*x`.

Para perceber melhor as possibilidades dos apontadores vamos definir agora um segundo apontador, `y` a fazê-lo também apontar para a variável `v`:

```
int *y = x ;
```

Obtém-se a seguinte situação:



Agora o conteúdo da variável `v` pode ser acedido de três formas diferentes: (1) usando o nome `v`; (2) usando a expressão `*x`; (3) usando a expressão `*y`.

O operador `&` chama-se **operador endereço**. O operador `*` chama-se **operador de desreferenciação**.

Repare na seguinte curiosa equivalência, que é válida em C para qualquer variável `v`:

```
*&v == v
```

Falta ainda uma referência à constante predefinida de tipo apontador, **`NULL`**. Garante-se que este apontador constante não aponta para sítio nenhum. Pode ser atribuído a uma variável de tipo apontador, por exemplo assim:

```
int *z = NULL ;
```

Neste caso serve para assinalar que a variável `z` não está a apontar para sítio nenhum, de momento.

O apontador NULL também pode ser usado em testes, assim:

```
if (z == NULL) ...
```

O apontador NULL não pode ser desreferenciado, pois não aponta para sítio nenhum.

Apontadores para registos

O seguinte tipo registo permite representar datas:

```
typedef struct {
    int day, month, year ;
} Date ;
```

Vamos definir agora uma variável de tipo `Date` e coloquemos um apontador de tipo `Date *` a apontar para a primeira:

```
Date d = {25, 12, 2008};
Date *p = &d;
```

Para aceder, através do apontador, ao ano da data `d`, podemos escrever:

```
(*p).year
```

Mas a utilização de apontadores para registos em C é tão frequente, que foi criada uma notação mais compacta e sugestiva para fazer isso: o operador `->`. A seguinte expressão é equivalente à anterior:

```
p->year
```

Em geral, a seguinte notação geral permite aceder a campos de registos através de apontadores:

```
Apontador->Etiqueta
```

Compatibilidade entre apontadores

Em C, tipos de apontadores que apontem para valores de tipos diferentes são incompatíveis entre si. Com uma exceção: o tipo especial `void *` - o tipo `void *` é compatível com todos os tipos de operadores.

```
int *pti;
double *ptd ;
void *v ;

pti = ptd ;             /* Errado */
pti = (int *)ptd ;      /* Válido por causa do cast */

v = ptd ;               /* Válido porque se trata de void * */
pti = v ;               /* Válido porque se trata de void * */
```

Utilidade dos apontadores

Os exemplos anteriores são interessantes, mas não mostram ainda as situações práticas em que a utilização de apontadores é essencial na linguagem C. As situações práticas em que se usam apontadores em C são as seguintes:

1. **Implementação de parâmetros de saída nas funções.**
2. **Manipulação de vetores.**
3. **Manipulação de memória a baixo nível.**
4. **Manipulação de variáveis criadas dinamicamente usando a função de biblioteca `malloc`.**
5. **Programação genérica através de apontadores de tipo `void *`.**

Apontadores: Parâmetros de saída de funções

Vamos tentar programar uma função para trocar o valor de duas variáveis inteiras. Este é um exemplo clássico que ilustra a necessidade de suportar **parâmetros de saída** na linguagem C.

Esta primeira tentativa não funciona:

```
void swap(int a, int b)   /* Não funciona!!!! */
{
    int temp = a;
    a = b;
    b = temp;
}
```

A chamada de `swap(x,y)` não muda nada, porque os parâmetros das funções são implementados usando variáveis locais, inicializadas com cópias dos valores que aparecem na chamada. A função `swap` faz a troca das cópias locais, mas não troca o conteúdo das variáveis originais. Diz-se que os parâmetros `a` e `b` são **parâmetros de entrada**, porque eles permitem apenas transferir dados de fora para dentro da função.

Para resolver este problema, temos de usar apontadores. A função `swap` precisa de aceder às variáveis originais através dos apontadores para efetuar a troca. Fica assim:

```
void swap(int *a, int *b)   /* Funciona!!!! */
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Agora a chamada escreve-se `swap(&x,&y)` e a troca é realmente efetuada. Diz-se que os parâmetros `a` e `b` são **parâmetros de saída**, porque eles permitem passar dados de dentro para fora da função.

Repare que agora ficámos a conhecer duas maneiras de fazer uma função produzir dados para o seu exterior:

1. Através do seu resultado.
2. Através de parâmetros de saída (usando apontadores).

O seguinte exemplo mostra uma função com dois *resultados*, sendo os *resultados* implementados usando parâmetros de saída. A função faz o seguinte: dado um vetor de reais e o respetivo comprimento, a função calcula o máximo e o mínimo do vetor, ao mesmo tempo:

```
void maxMin(double v[], int n, double *max, double *min) /* precondição: n > 0 */
{
    double lmax = v[0];
    double lmin = v[0];
    int i;
    for( i = 1 ; i < n ; i++ ) {
        if (v[i] > lmax) lmax = v[i] ;
        if (v[i] < lmin) lmin = v[i] ;
    }
    *max = lmax;
    *min = lmin;
}
```

Exemplo de chamada:

```
double vect[] = {1.0, 2.9, 34.6, 44.2,  0.01};
double max, min;
maxMin(vect, 5, &max, &min);
```

Algumas funções de biblioteca também usam parâmetros de saída. Por exemplo, é o caso da função de biblioteca `scanf`. Veja um exemplo de utilização:

```
scanf("%d %lf %c", &i, &r, &c);
```

Apontadores: Manipulação de vetores e relação entre vetores e apontadores

Provavelmente você vai ficar surpreendido(a), mas em C o nome duma variável de tipo vetor representa um apontador constante para a primeira componente do vetor guardado na variável.

Considere o seguinte vetor

```
int vetor[100];
```

Para aceder ao primeiro elemento do vetor, normalmente nós escrevemos:

```
vetor[0]
```

Mas também podemos escrever o que está a seguir, pois o resultado é exatamente o mesmo.

```
*vetor
```

A linguagem C também permite a seguinte atribuição:

```
int *pt = vetor;
```

e, inclusivamente, permite-se a aplicação de operações sobre vetores a argumentos de tipo apontador. Por exemplo as seguintes expressões são legítimas:

```
pt[0]
pt[5]
pt[99]
```

Note que, quando de passa um vetor como parâmetro para uma função, o que realmente se passa é um apontador para a primeira componente do vetor. Portanto um parâmetro de tipo vetor é sempre um parâmetro de saída, apesar de não ser explicitamente declarado como apontador.

Aritmética de apontadores

Do ponto de vista dum apontador de tipo T*, toda a memória apontada é um grande vetor de valores de tipo T.

Os operadores + e - podem ser aplicados a apontadores para T e inteiros nos seguintes casos:

- **pt + i** - o resultado é um apontador que referência um valor de tipo T que esteja i posições depois de pt. Cada posição tem o tamanho de T.
- **pt - i** - o resultado é um apontador que referência um valor de tipo T que esteja i posições antes de pt. Cada posição tem o tamanho de T.
- **pt1 - pt2** - o resultado é um inteiro que indica o numero de valores de tipo T entre pt1 e pt2. Estes dois apontadores têm de ser do mesmo tipo T*.

Para o vetor abaixo são verdadeiras as equivalências indicadas:

```
Vetor v ;

v[0] == *v
v[1] == *(v+1)
v[-1] == *(v-1)
&v[0] == v
&v[1] == v + 1
```

Fazer `v = v + 1` é proibido pois v é um apontador constante.

Exemplo - Duas formas diferentes de copiar vetores (neste caso bidimensionais)

```
#define SIZE    20

int i1[SIZE][SIZE], i2[SIZE][SIZE] ;
int *pt1, *pt2, *ptEnd ;
int i, j;

/* Forma 1 */
for( i = 0 ; i < SIZE ; i++ )
    for( j = 0 ; j < SIZE ; j++ )
        i2[i][j] = i1[i][j] ;

/* Forma 2 */
for( pt1 =i1 , pt2 = i2, ptEnd = pt2 + SIZE * SIZE ;
            pt2 < ptEnd ;
                *pt1++ = *pt2++ ) ;
```

Repare que a primeira forma envolve um ciclo embutido noutro e que, durante a execução, é necessário fazer muitas contas para repetidamente determinar quais as localizações correspondentes às expressões `i2[i][j]` e `i1[i][j]`.

Quanto à segunda forma, envolve um único ciclo e evita a necessidade de se fazerem as contas atrás referidas.

A segunda forma é mais difícil de perceber, mas é um pouco mais eficiente do que a primeira.

Apontadores: Manipulação de memória a baixo nível

Para interpretar uma zona de memória como um valor dum tipo `T`, basta fazer um apontador de tipo `T*` apontar para lá e ler.

Por exemplo, no código abaixo, define-se um bloco de memória chamado `b`, e usa-se um apontador de tipo `double*` para ler um valor real que se encontra guardado a partir do byte 5.

Repare como a variável `pt` é inicializada: obtém-se o endereço do byte 5 (esse endereço é um apontador de tipo `Byte*`) e depois aplica-se um cast para o converter num apontador de tipo `double*`.

```
#define BLOCK_SIZE 32
typedef unsigned char Byte;
typedef Byte Block[BLOCK_SIZE];

Block b;                        // um bloco
double *pt = (double *)&b[5];  // conversão de tipo de apontador
double d = *pt;                // leitura do valor através do apontador pt
```

#100