Métodos e Técnicas de Programação
.:: Tipos derivados, Estruturas e Funções básicas ::.

Prof. Igor Peretta

2018-2

Contents

1 Novos tipos

A partir dos tipos básicos do C, podemos derivar novos tipos com o auxílio do comando `typedef`.

Síntaxe:

```
typedef
    [...] definição do novo tipo a partir de primitivas e estruturas [...]
NOME_DO_TIPO_DERIVADO;
```

Com isso, podemos, a partir dessa abstração, definir tipos que auxiliem no processo de implementação de um algoritmo, além de facilitar a leitura/documentação de nosso código.

```
#include <stdio.h>
typedef
    int
Inteiro;
typedef
    unsigned char
Byte;
typedef
    int *
P_int;
int main(){
    Inteiro meunumero, i;
    P_int endereco;
    Byte byte;
    meunumero = 42;
    endereco = &meunumero;
    printf("No endereco %p, temos o dado %d (hex: %08X )\n",
endereco, meunumero, meunumero);
    for(i = 0; i < sizeof(Inteiro); i++) {
byte = ((Byte*) endereco)[i];
printf("No endereco %p, temos o byte %02X\n",
    ((Byte*) endereco)+i, byte);
    }
    printf("Inteiros sao armazenados em little-endian (byte a byte)\n"
   "e big-endian (bit a bit, no mesmo byte)!\n");
    return 0;
}
```

```
No endereco 0x7ffcb9d05f88, temos o dado 42 (hex: 0000002A )
No endereco 0x7ffcb9d05f88, temos o byte 2A
No endereco 0x7ffcb9d05f89, temos o byte 00
No endereco 0x7ffcb9d05f8a, temos o byte 00
No endereco 0x7ffcb9d05f8b, temos o byte 00
```

```
Inteiros sao armazenados em little-endian (byte a byte)
e big-endian (bit a bit, no mesmo byte)!
```

2 Estruturas

Estruturas são tipos de dados que encapsulam outros dados, garantindo que os mesmos estejam armazenados de forma consecutiva na memória.

2.1 Struct

Dentro de C, o formato para definir uma estrutura (*struct*) é:

```
struct NOME_DA_ESTRUTURA {
  tipo membro1;
  tipo membro2;
  ...
} NOME_DE_VARIÁVEL;
```

onde `NOME_DA_ESTRUTURA` se refere ao identificador da estrutura e os *membros* da estrutura são variáveis, cada qual com seu tipo. Note que a definição termina com um ";" porque poderia conter ainda o `NOME_DE_VARIÁVEL` (opcional).

Para criar uma instância da estrutura (uma nova variável), basta usar o `Nome_da_estrutura` como um novo tipo de dado, precedido de `struct`:

```
[no escopo de qualquer função]
struct NOME_DA_ESTRUTURA NOME_DE_VARIÁVEL;
```

Para acessar as variáveis contidas na estrutura, usamos o **ponto**:

```
NOME_DE_VARIÁVEL.membro1 = XXX;
```

Podemos usar ponteiros para indicar o endereço da estrutura, mas o acesso ao conteúdo se modifica, usamos o `->`:

```
[no escopo de qualquer função]
struct NOME_DA_ESTRUTURA * PONTEIRO_PARA_VARIÁVEL_DO_TIPO_STRUCT;
PONTEIRO_PARA_VARIÁVEL_DO_TIPO_STRUCT->membro1 = XXX;
```

Exemplo com `struct Ponto`:

```
#include <stdio.h>
struct Ponto {
  float x;
  float y;
};
int main() {
  struct Ponto A;
  struct Ponto *endereco = &A;
  A.x = 3.0;
  A.y = 5.0;
  printf("Ponto A, no endereco %p (com %d bytes),"
 " = (%f,%f)\n", endereco, sizeof(A),
 endereco->x, endereco->y);
  printf("End.x = %p; End.y = %p\n", &(A.x), &(A.y));
  return 0;
}
```

```
Ponto A, no endereco 0x7fffa6d02640 (com 8 bytes), = (3.000000,5.000000)
End.x = 0x7fffa6d02640; End.y = 0x7fffa6d02644
```

2.1.1 Tipo derivado

Para criar um `typedef` para estruturas, precisamos notar que enquanto não estiver definido o novo tipo, o mesmo não existe. Isso significa que, se houver um membro da estrutura do tipo novo, teremos que usar a definição original.

Exemplo:

```
#include <stdio.h>
typedef
    struct meuPonto {
      float x;
      float y;
    }
Ponto;
int main() {
  Ponto A;
  Ponto *endereco = &A;
  A.x = 3.0;
  A.y = 5.0;
  printf("Ponto A, no endereco %p (com %d bytes),"
 " = (%f,%f)\n", endereco, sizeof(Ponto),
```

```
 endereco->x, endereco->y);
  printf("End.x = %p; End.y = %p\n", &(A.x), &(A.y));
  return 0;
}
```

```
Ponto A, no endereco 0x7ffeb8ab1900 (com 8 bytes), = (3.000000,5.000000)
End.x = 0x7ffeb8ab1900; End.y = 0x7ffeb8ab1904
```

Outro exemplo:

```
#include <stdio.h>
typedef
    struct meuPonto {
float x;
float y;
struct meuPonto * vizinho;
    }
Ponto;
int main() {
    Ponto A, B;
    A.x = 3.0;
    A.y = 5.0;
    A.vizinho = &B;
    A.vizinho->x = 3.5;
    A.vizinho->y = 5.5;
    printf("Ponto A = (%f,%f)\n", A.x, A.y);
    printf("Vizinho ponto A = (%f,%f)\n",
    B.x, B.y);
    return 0;
}
```

```
Ponto A = (3.000000,5.000000)
Vizinho ponto A = (3.500000,5.500000)
```

2.2 Union

Uniões são como estruturas, exceto que todas as variáveis compartilhar o mesmo espaço de memória. Quando declarada, o compilador aloca memória suficiente para o maior tipo de dado (em bytes) na união.

```
union U {
```

```
  unsigned int ui;
  float fp;
};
union U un;
un.fp = -3.5f;
printf("UN tem %u bytes\n", sizeof(un));
printf("End.ui %p; End.fp %p\n", &(un.ui), &(un.fp));
printf("%X\n %g\n", un.ui, un.fp);

UN tem 4 bytes
End.ui 0x7ffce6ed9040; End.fp 0x7ffce6ed9040
C0600000
 -3.5
```

2.2.1 Tipo derivado

```
#include <stdio.h>
typedef
  union U {
    int Int;
    char Char[5];
  }
UNIAO;
int main() {
  UNIAO un;
  int i;
  strncpy(un.Char,"RATO",4);
  for(i = 0; i < 4; i++)
    printf("%c : ascii (hex) = %X\n", un.Char[i], un.Char[i]);
  printf("un.Int = %X\n", un.Int);
  return 0;
}

R : ascii (hex) = 52
A : ascii (hex) = 41
T : ascii (hex) = 54
O : ascii (hex) = 4F
un.Int = 4F544152
```

2.3 Enum

O tipo de dados conhecido por enumeração (enum) são uma forma especial de inteiros que seguem restrições de valor. São utilizados para aumentar a abstração dos programas.

```
enum meses { jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez };
enum meses calendario = jun;
switch(calendario) {
 case jan: printf("Janeiro\n"); break;
 case fev: printf("Fevereiro\n"); break;
 case mar: printf("Marco\n"); break;
 case abr: printf("Abril\n"); break;
 case mai: printf("Maio\n"); break;
 case jun: printf("Junho\n"); break;
 case jul: printf("Julho\n"); break;
 case ago: printf("Agosto\n"); break;
 case set: printf("Setembro\n"); break;
 case out: printf("Outubro\n"); break;
 case nov: printf("Novembro\n"); break;
 case dez: printf("Dezembro\n"); break;
}
```

```
Junho
```

2.3.1 Tipo derivado

```
typedef
  enum meubool { false = 0, true = 1 }
bool;
int main() {
  bool check = !true;
  printf("%s\n", (check)? "X true" : "Y false");
}
```

```
Y false
```

3 Funções

Funções são trechos de código que idealmente possuem uma única tarefa e que são destacados do corpo principal do programa, permitindo assim que sejam reutilizados.

3.1 Definição

Função principal é a função `main()` tem um significado especial nos programas em C, pois é a função que é inicialmente executada (em inglês, *entry point*). O termo `int` define a função main como sendo uma função que retorna um número inteiro. O termo void indica que a função não aceita parâmetros. A função main, normalmente aceita parâmetros, que são passado pela linha de comando. Os compiladores e sistemas operacionais atuais reconhecem as seguintes declarações de `main`:

```
void main() { ...; }
int main() { ...; return 0; }
int main(void) { ...; return 0; }
int main(int argc, char **argv) { ...; return 0; }
```

Uma função é definida pelo **tipo de seu retorno**, seu **identificador** e seus **argumentos** (cada qual com seu tipo). Além disso, as instruções que compõe a função são encapsuladas entre chaves. A última instrução de uma função é a `return` que deverá conter o valor da resposta com o tipo de retorno da função.

Exemplo:

```
int soma(int a, int b) {
    int s = a + b;
    return s;
}
int main() {
    printf("%d\n",soma(3,4));
    return 0;
}
```

3.2 Escopo de variáveis

O escopo de uma variável é definido pelas regiões (blocos) onde a variável pode ser utilizada. Por exemplo, as variáveis declaradas no início do corpo da função `main()` podem ser utilizadas em qualquer lugar dentro da **mesma** função, ou seja, não podem ser utilizadas em outra função. Variáveis declaradas no **mesmo escopo** precisam ter **nomes diferentes**, mas nomes podem ser "reaproveitados" em outros escopos.

Em linhas gerais, o escopo de uma variável é definido pela extensão de código que delimita sua existência.

```
int dobro(int x) {
  return x*2;
}
int main() {
  int x = 3;
  dobro(x);
  printf("%d\n", x);
  return 0;
}
```

```
3
```

```
int dobro(int x) {
  return x*2;
}
int main() {
  int x = 3;
  x = dobro(x);
  printf("%d\n", x);
  return 0;
}
```

```
6
```

```
void dobro(int * x) {
  *x *= 2;
}
int main() {
  int x = 3;
  dobro(&x);
  printf("%d\n", x);
  return 0;
}
```

```
6
```

3.3 Passagem de argumentos ou parâmetros

Passagem **por valor** é a forma mais comum utilizada para passagem de parâmetros. Os valores dos argumentos são copiados para novas variáveis dentro do escopo da função "isolando" a variável original.

```
int muda(int x) { x = x + 7;
  printf("(funcao) x = %d - %p\n", x, &x);
  return x; }
int main() {
  int x = 3, y;
  printf("(antes)  x = %d - %p\n", x, &x);
  y = muda(x);
  printf("(depois) x = %d, y = %d\n", x, y);
  return 0; }
```

```
(antes)  x = 3 - 0x7fff7fb68ae0
(funcao) x = 10 - 0x7fff7fb68acc
(depois) x = 3, y = 10
```

Na passagem **por referência**, não mais o valor do argumento é copiado, mas sim o endereço da memória onde a variável original se encontra. Com isso, tem-se acesso ao dado original e a função pode modificá-lo "remotamente". Um bom exemplo de uso de passagem por referência é quando desejamos criar uma função que retorne mais de um valor.

```
void muda(int *x) { *x = *x + 7;
  printf("(funcao) x = %d - %p\n", *x, x);
   }
int main() {
  int x = 3, y;
  printf("(antes)  x = %d - %p\n", x, &x);
  muda(&x); y = x;
  printf("(depois) x = %d, y = %d\n", x, y);
  return 0; }
```

```
(antes)  x = 3 - 0x7fffac979ae0
(funcao) x = 10 - 0x7fffac979ae0
(depois) x = 10, y = 10
```

Uma função pode ter nenhum, um ou mais argumentos.

```
#include <stdio.h>
#include <time.h>
typedef
    struct P { float x, y; }
Ponto;
```

```
int dado() {
    return rand() % 6 + 1;
}
int prox_potencia_2(long long int x) {
    int p;
    for(p = 1; x >>= 1; p++);
    return p;
}
float AXmaisY(int a, Ponto * p) {
    return ((float)a)*p->x + p->y;
}
int main() {
    srand(time(0));
    int i;
    long long x = 0xFFF;
    Ponto A;
    A.x = 3.14; A.y = -2.71;
    for(i = 0; i < 10; i++) printf("%d ", dado());
    printf("\n");
    printf("A potencia de 2 necessaria para representar "
   "o valor: %lld e igual a %d\n", x, prox_potencia_2(x));
    printf("%d * %g + (%g) = %g\n", 2, A.x, A.y, AXmaisY(2, &A));
    return 0;
}
```

```
2 2 1 4 6 4 3 6 2 6
A potencia de 2 necessaria para representar o valor: 4095 e igual a 12
2 * 3.14 + (-2.71) = 3.57
```

3.4 Retorno de função

Uma função pode ou não retornar um valor do seu tipo de retorno. Funções que não retornam valores são do tipo `void`.

É possível criar funções do tipo `void`. São funções que não retornam valores, portanto não é devido o uso do `return` para encerrá-las.

É necessário definir a função antes da função `main()` (depois dos *includes*) para que possa ser usada.

```
#include <stdio.h>
typedef
```

```
    struct P { float x, y; }
Ponto;
Ponto somapontos(Ponto * pA, Ponto * pB) {
    Ponto P;
    P.x =  pA->x + pB->x;
    P.y =  pA->y + pB->y;
    return P;
}
void soma(int a, int b, int * c) {
    *c = a+b;
}
int main() {
    int s;
    Ponto A, B, C;
    A.x = 1; A.y = 2; B.x = -1, B.y = 2;
    C = somapontos(&A, &B);
    printf("ponto = (%g, %g)\n", C.x, C.y);
    soma(3, 4, &s);
    printf("3 + 4 = %d\n", s);
    return 0;
}

ponto = (0, 4)
3 + 4 = 7
```

3.5 Vetor e retorno de função para tipos derivados e estruturas

Um vetor(*array*) ou um retorno de função para tipos derivados ou de estruturas funciona do mesmo jeito do que para os tipos básicos.

Exemplo de uso de vetor de estruturas e de retorno de função:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 100000
typedef
  struct meuPonto {
    float x;
    float y;
```

```
  }
Ponto;
Ponto centro_massa(Ponto * meusdados, int ND) {
  Ponto cm;
  int i;
  cm.x = cm.y = 0.0;
  for(i = 0; i < ND; i++){
    cm.x += meusdados[i].x;
    cm.y += meusdados[i].y;
  }
  cm.x /= ND; cm.y /= ND;
  return cm;
}
int main() {
  srand(time(NULL));
  Ponto dados[N]; Ponto cm;
  int i;
  for(i = 0; i < N; i++) {
    dados[i].x = rand()%100 + 1;
    dados[i].y = rand()%100 + 1;
  }
  cm = centro_massa(dados, N);
  printf("Centro de massa = (%g,%g)\n", cm.x, cm.y);
  return 0;
}

Centro de massa = (50.5049,50.632)
```

3.6 Função com argumento e retorno de struct

```
#include <stdio.h>
struct Ponto {
  float x;
  float y;
};
struct Ponto funcao(struct Ponto a) {
   struct Ponto resp;
   resp.x = 2*a.x;
   resp.y = 3*a.y;
   return resp;
```

```
}
int main() {
  struct Ponto b, c;
  int i;
  b.x = 2.f; b.y = -1.f;
  c = funcao(b);
  printf("( %f, %f ) \n", c.x, c.y);
  return 0;
}

( 4.000000, -3.000000 )
```

3.7 Ponteiro para função

Síntaxe para declaração de ponteiro para função:

```
void (*foo)(int)
```

Neste exemplo, a função a ser apontada retorna `void` e tem um único argumento do tipo `int`. O nome da variável ponteiro é `foo`.

Para iniciar um ponteiro de função, você deve dar o endereço de uma função no seu programa.

```
#include <stdio.h>
void minha_func(int x) {
    printf( "%d\n", x );
}
void outra_func(int x) {
    printf( "%d\n", x+1 );
}
int main() {
    void (*foo)(int);
    /* o '&' é opcional */
    foo = &minha_func;
    /* o uso é direto */
    foo(3);
    /* alternativamente */
    foo = outra_func;
    (*foo)(4);
    return 0;
}
```

```
3
5
```

```
int soma(int x, int y) {
  return x+y;
}
int sub(int x, int y) {
  return x-y;
}
int main() {
  int (*ptrfunc)(int, int);
  ptrfunc = soma;
  printf("%d\n", (*ptrfunc)(4,5));
  ptrfunc = sub;
  printf("%d\n", (*ptrfunc)(4,5));
  return 0;
}
```

```
9
-1
```

Note que ponteiros para funções são flexíveis, uma vez que o próprio nome da função já é um ponteiro (similar ao nome de *arrays* e de *strings*). Ou seja, você pode trabalhar com `&` e `*`, mas também pode omiti-los.