

Funções

Funções (também chamadas de *rotinas* ou *procedimentos*) são trechos de código situados fora do programa principal. As funções são identificadas por um nome. Este nome é chamado (ou invocado) toda vez que se deseja executar o trecho de código da função.

Motivos para usar funções

Suponha o seguinte programa: para dois alunos, o programa lê o valor das notas de três provas. A média é calculada somente com a nota das duas melhores provas. Deseja-se descobrir qual dos dois alunos obteve a melhor média.

Um abordagem simplista, em C, seria:

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

int main(int argc, char argv[]) {
    // Declarar notas e médias dos alunos A e B
    float notaA1, notaA2, notaA3, mediaA;
    float notaB1, notaB2, notaB3, mediaB;

    // Ler notas dos alunos A e B
    scanf("%f %f %f", &notaA1, &notaA2, &notaA3);
    scanf("%f %f %f", &notaB1, &notaB2, &notaB3);

    // Calcular a média para o aluno A
    if ((notaA1 <= notaA2) && (notaA1 <= notaA3)) {
        mediaA = (notaA2 + notaA3) / 2.0f;
    } else if ((notaA2 <= notaA1) && (notaA2 <= notaA3)) {
        mediaA = (notaA1 + notaA3) / 2.0f;
    } else {
        mediaA = (notaA1 + notaA2) / 2.0f;
    }

    // Calcular a média para o aluno B
    if ((notaB1 <= notaB2) && (notaB1 <= notaB3)) {
        mediaB = (notaB2 + notaB3) / 2.0f;
    } else if ((notaB2 <= notaB1) && (notaB2 <= notaB3)) {
        mediaB = (notaB1 + notaB3) / 2.0f;
    } else {
        mediaB = (notaB1 + notaB2) / 2.0f;
    }

    // Imprimir resultado
    if (mediaA > mediaB) {
        printf("Aluno A obteve melhor desempenho");
    } else if (mediaA < mediaB) {
        printf("Aluno B obteve melhor desempenho");
    } else {
        printf("Os alunos A e B tem a mesma media");
    }

    return 0;
}
```

Consulte: `Funcoes\Notas01\Notas01.vcproj`

Primeiro, declara-se as variáveis que armazenarão as notas e as médias dos alunos A e B. O programa lê então as três notas de cada aluno e armazena-as nas variáveis `notaA1`, `notaA2`, `notaA3`, `notaB1`, `notaB2`, `notaB3`.

Em seguida, o programa calcula a média das duas melhores notas para o aluno A. Se a nota 1 for a pior entre as três (*notaA1* é menor que *notaA2* e menor que *notaA3*), então a média é calculada utilizando-se somente as outras duas notas (*notaA2* e *notaA3*). Senão, o programa verifica outra possibilidade: nota 1 e 3 são as duas melhores (portanto, *notaA2* é menor que *notaA1* e menor que *notaA3*). Caso contrário, sabemos que sobra apenas uma única alternativa: nota 1 e 2 são as melhores.

Para o aluno B, o programa realiza os mesmos testes, mas trocando, respectivamente, *notaA1*, *notaA2*, *notaA3* e *mediaA* por *notaB1*, *notaB2*, *notaB3* e *mediaB*.

No fim, as duas médias são comparadas e o programa imprime uma mensagem adequada.

Inconveniências

O código apresentado possui algumas inconveniências, que serão discutidas agora.

Primeiro, note que o código **repete as mesmas instruções** para calcular a média das duas melhores notas tanto para o aluno A como para aluno B. Observe que o código, na realidade, **repete o mesmo algoritmo**, mas para *variáveis diferentes*.

Caso seja necessário alterar uma instrução do algoritmo, o programador precisa garantir que as duas cópias do código sejam mudadas da mesma maneira, mas é **difícil manter a consistência**. Facilmente alguém esquecerá de alterar algum detalhe em uma das repetições do código.

Não obstante, **o código é extenso**, mesmo executando sempre o mesmo algoritmo. E é também **conceitualmente confuso**, pois ele mistura dois algoritmos em um único programa: o algoritmo de calcular a média das duas melhores notas e o algoritmo para comparar médias.

Por fim, é difícil reaproveitar tanto o algoritmo do cálculo de média como o de comparação de notas em outro programa, a não ser copiando o código.

O conceito de funções alivia esses problemas.

O que é uma função em C

O nome *função* lembra as funções que conhecemos na matemática, mas, em C, é um conceito muito mais amplo.

Em C, a palavra **função** deve ser entendida como sinônimo de **algoritmo**.

Uma função implementa um algoritmo, e, portanto, apresenta os mesmos conceitos do algoritmo: entrada, saída, código finito, sempre parando ao executar.

Uma função é um bloco de código que:

- Realiza uma determinada tarefa específica. Normalmente, uma função implementa um algoritmo.
- Pode ser executado a qualquer momento pelo programa.
- Recebe dados de entrada do programa (ou do teclado, arquivo, etc).
- Retorna um resultado (ou escreve algo na tela, em arquivo, etc).

Um programa pode executar uma função diversas vezes, em momentos diferentes, com valores diferentes para as variáveis.

Assim, quando temos uma mesma tarefa que precisamos executar em diversos pontos diferentes do programa, podemos **definir uma função** que implementa esta tarefa. Quando necessitamos executar esta tarefa, basta **chamar a função**, ao invés de repetir todo seu código. Os valores que a função deve utilizar naquele momento são dados de entrada que são **passados como parâmetros**.

No exemplo anterior poderíamos definir uma função específica para calcular a média das duas melhores notas. Chamariamos a função duas vezes, uma vez com as notas do o aluno A e outra vez com as notas do aluno B. Desta forma, evitaríamos a repetição do código, escrevendo-o apenas uma vez na definição da função. Caso seja necessário alterar o algoritmo do cálculo da média, basta alterar o código que define a função.

Chamando uma função

Quando a execução chega a uma instrução contendo uma chamada de função, a execução do programa que chamou é interrompida temporariamente e o computador passa a executar as instruções contidas no algoritmo da função. Veja na Figura 1 a representação do fluxo de execução. Terminada a execução desse algoritmo, a execução continua na mesma posição em que o programa foi interrompido.

O código que realiza a chamada de função é denominado **código chamador**. A função é denominada **função chamada**.

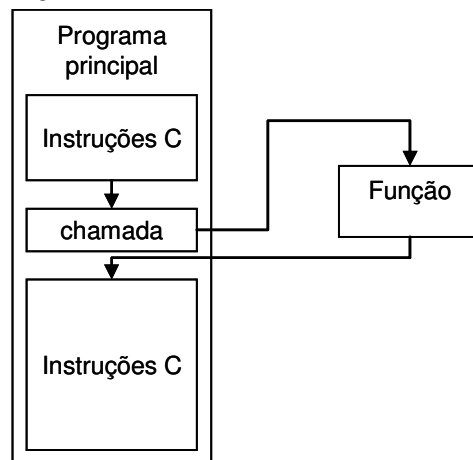


Figura 1 – Chamada de função e desvio de execução

Exemplo

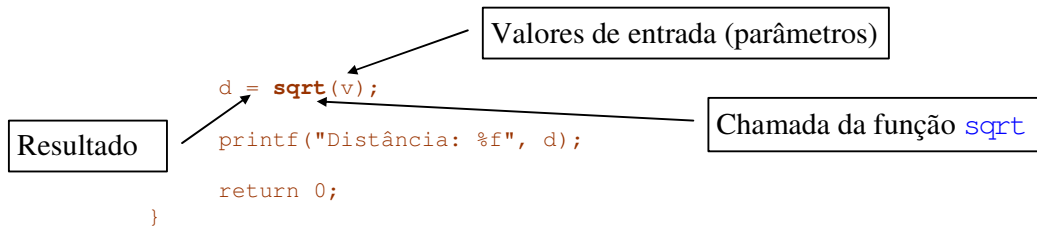
Considere um programa que lê duas coordenadas (x1, y1) e (x2, y2) de pontos no plano Euclidiano e imprime a distância entre os dois pontos. Este cálculo exige a extração da raiz quadrada. A linguagem C não possui operador primitivo para extrair a raiz quadrada. Felizmente, existe uma função chamada `sqrt` que realiza este cálculo. Ela recebe como entrada um número (como um `double`) e após sua execução, retorna a raiz quadrada deste número (também como um `double`).

A chamada da função está no código a seguir:

```
#include <math.h>
int main(int argc, char *argv[]) {
    double x1, x2, y1, y2, d;
    double v;

    scanf("%lf, %lf, %lf, %lf", &x1, &y1, &x2, &y2);

    v = (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
```

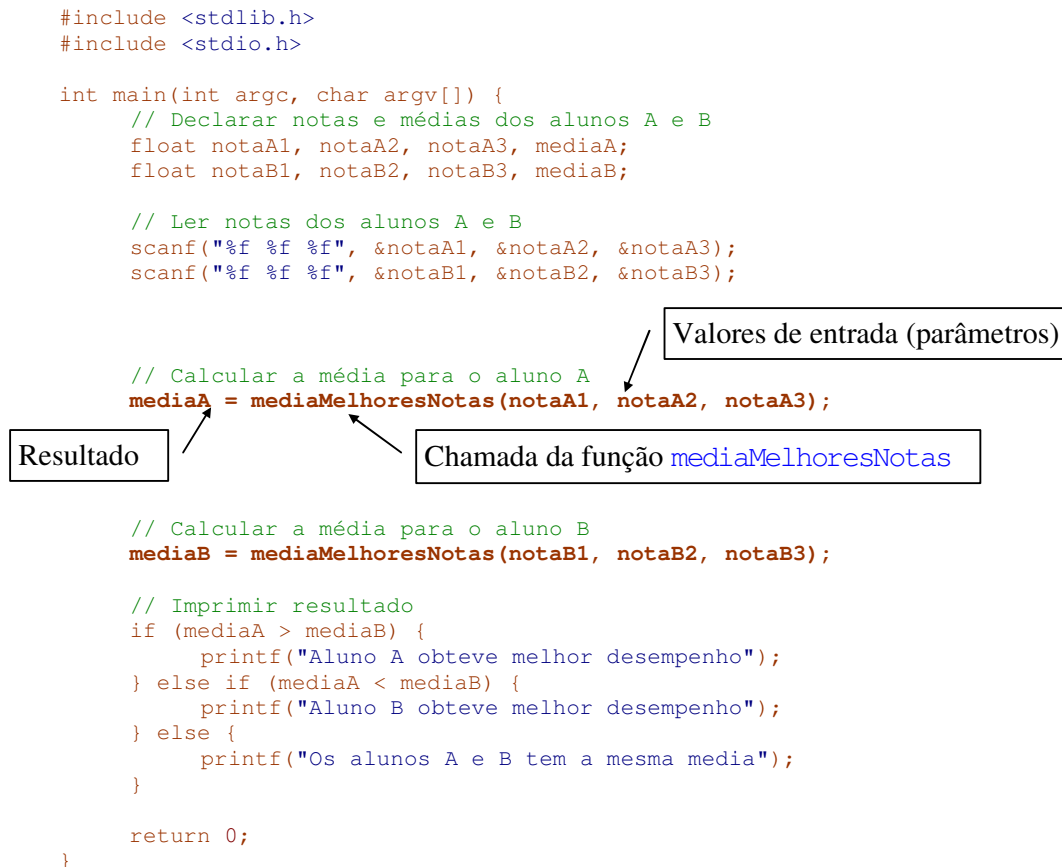


Consulte: Funcoes\Chamada01\Chamada01.vcproj

Assim que a execução chega na linha contendo a função `sqrt`, o programa principal é interrompido. O computador passa a executar o código da função `sqrt`, usando o valor atual armazenado na variável `v`. Após algum processamento, esta função retorna a raiz quadrada do valor de `v` como resultado. A execução do programa continua com a atribuição do valor retornado para a variável `d`.

Exemplo

Vamos supor que nosso programa dispõe da função `mediaMelhoresNotas`. Para chamar a função no programa, informamos o seu nome e entre parênteses listamos os três valores que desejamos que ela utilize durante os cálculos. A função calcula e retorna um resultado, que pode ser atribuído à uma variável ou utilizado em uma expressão aritmética.



Consulte: Funcoes\Notas02\Notas02.vcproj

Observe em particular as duas linhas:

```

mediaA = mediaMelhoresNotas(notaA1, notaA2, notaA3);
mediaB = mediaMelhoresNotas(notaB1, notaB2, notaB3);

```

Todo o esforço de programação está resumido em elaborar uma única vez um algoritmo e codificá-lo numa função denominada `mediaMelhoresNotas`. O programa em si ficou bem simples.

Definição de função

Agora, vamos aprender como definir funções em C. Como exemplo, utilizaremos a função `mediaMelhoresNotas`, que calcula a média das duas melhores notas entre três notas dadas. O exemplo será construído aos poucos, conforme aprendemos os conceitos envolvidos na criação de funções. Enquanto isso, vários trechos da definição da função serão omitidos e representados por três pontos (...).

Uma função é definida da seguinte forma:

```
tipo nome(parâmetros) {  
    declarações de variáveis  
    ...  
    instruções;  
    ...  
    return valor;  
}
```

Nome

Tal como para variáveis, as funções recebem um **nome** (também chamado de identificador). O nome é utilizado para identificar a função no momento da chamada. Na escolha do nome da função valem as mesmas regras que aquelas já vistas para se escolher nomes de variáveis: o nome é formado por letras maiúsculas, letras minúsculas, dígitos numéricos ou o símbolo sublinhado (_), e sempre começando com uma letra.

Programadores de C costumam escolher identificadores formados por substantivos e o símbolo de sublinhado (_) entre as palavras. Os programadores de C++ utilizam um verbo como identificador.

Exemplo:

```
... mediaMelhoresNotas(...) {  
    ...  
    // Corpo da função  
    ...  
}
```

Nome da função

Instruções que são executadas pela função

Parâmetros

A maioria das funções recebe pelo menos um valor como entrada. Estes valores são os parâmetros. Na definição da função, um **parâmetro** é uma variável especial, que existe somente dentro da função. Tal como qualquer outra variável, a variável associada a cada parâmetro também deve ter seu tipo declarado. Os parâmetros (e seus respectivos tipos) são informados em uma lista separada por vírgulas e entre parênteses, logo após o nome da função.

Exemplo:

```
... mediaMelhoresNotas(  
    float nota1, float nota2, float nota3) {  
    ...  
    // Corpo da função  
    ...  
}
```

Lista de três parâmetros

No exemplo, indicamos que a função vai usar três variáveis de tipo `float`. No corpo da função, essas variáveis serão denominadas `nota1`, `nota2` e `nota3`. Tudo se passa como se as três variáveis fossem declaradas *localmente*, logo no início do corpo da

função. A diferença para as outras declarações que ocorrem no corpo da função está no fato de que as variáveis passadas como parâmetros serão *automaticamente inicializadas*, com aqueles valores que forem passados pelo código que chamar a função. Essa inicialização ocorre *antes* do início da execução do código da função

A lista de parâmetros pode estar na mesma linha que o nome da função, ou pode estar dividida em várias linhas. Escolha a opção de melhor legibilidade. O importante é delimitar a lista de parâmetros com parênteses.

Parâmetros na chamada de função

Quando chamamos a função, precisamos informar os valores de cada um dos parâmetros, na mesma ordem que aparecem na lista de parâmetros. Os valores precisam ter tipo compatível com o tipo do respectivo parâmetro.

Como exemplo, se a definição da função fosse:

```
... potencia(float base, float expoente) {  
    ...  
    // Corpo da função  
    ...  
}
```

Poderíamos chamar a função como:

```
float resultado;  
  
resultado = potencia(2.0f, 5.0f);
```

Neste caso, os valores `2.0f` e `5.0f` são atribuídos aos parâmetros na mesma ordem em que foram declarados na definição da função, ou seja, o parâmetro `base` receberá o valor `2.0f`, e o parâmetro `expoente` o valor `5.0f`. Assim, quando a função começar a executar, a variável local `base` estará armazenando o valor `2.0f`, e a variável local `expoente` conterá o valor `5.0f`. O compilador se encarrega de gerar código para que essa transferência de valores realmente ocorra na memória do computador.

O parâmetro conterá sempre *uma cópia* do valor usado na chamada da função. Se, no corpo da própria função, esta atribuir um novo valor a um de seus parâmetros, então esta atribuição *não estará refletida* no código chamador. Ou seja, se a função é chamada tendo um parâmetro `x` que recebe como valor o conteúdo de uma variável `t` do programa chamador e, se quando a função executa, é atribuído um outro valor à variável `x`, então o valor da variável `t` não terá se alterado após o término da execução da função e o retorno da execução ao programa chamador.

Funções sem parâmetros

Existem casos especiais de funções que não têm parâmetros. Nestas situações, a lista de parâmetros é vazia. Mas, mesmo assim, deve-se escrever os parênteses, tanto na definição, quanto na chamada da função.

```
... funcaoSemParametros() {  
    ...  
    // Corpo da função  
    ...  
}
```

Lista de zero parâmetros

Corpo e variáveis locais

O **corpo da função** contém todas as instruções da linguagem C, em sequência, que implementam o algoritmo da função. Cada função pode ser entendida como um programa independente, escrito em C.

No início do corpo da função, é comum declarar **variáveis locais**. Elas são conhecidas somente dentro do corpo da função. Em qualquer outro ponto do programa, que não seja dentro do corpo desta função, estas variáveis não são conhecidas e será impossível acessar o valor das mesmas. Como os parâmetros já são entendidos como declarados localmente, não faz sentido declarar no corpo da função outras variáveis com nome idênticos aos nomes dos parâmetros.

É importante entender o que se passa na memória do computador quando este executa funções com variáveis locais ou parâmetros. As declarações, inclusive de parâmetros, são apenas indicativos de quanta memória será necessária para armazenar valores das respectivas variáveis. Cada vez que uma função entra em execução novamente, novos endereços de memória são alocados para as variáveis locais, possivelmente diferentes daqueles que foram alocados para essas mesmas variáveis quando da última execução da função. E o programa não tem nenhum controle sobre em que posição de memória uma certa variável estará alocada, em determinada ativação. Ao terminar a execução da função, esses endereços de memória são liberados para uso por outros programas.

Daí porque o programador não consegue recuperar valores atribuídos às variáveis quando da última execução da função. Isso é o que se passa usualmente na ativação, execução e término de funções, com algumas exceções específicas que veremos mais adiante.

Exemplo:

```
... mediaMelhoresNotas(  
    float nota1, float nota2, float nota3) {  
    float media;  
    if ((nota1 <= nota2) && (nota1 <= nota3)) {  
        media = (nota2 + nota3) / 2.0f;  
    } else if ((nota2 <= nota1) && (nota2 <= nota3)) {  
        media = (nota1 + nota3) / 2.0f;  
    } else {  
        media = (nota1 + nota2) / 2.0f;  
    }  
    ...  
}
```

Primeiro, o corpo da função declara uma variável (chamada `media` e de tipo `float`), que armazena a média calculada dentro da função. Além disso, já existem outras três variáveis locais e especiais `nota1`, `nota2` e `nota3` (todas do tipo `float`) que vão ser inicializadas com valores passados como parâmetros na chamada da função.

Valor de Retorno

As funções (algoritmos) produzem um resultado. O comando `return` termina a execução da função e define o valor de retorno. Na chamada da função, o código chamador pode atribuir este valor retornado a uma variável ou usá-lo em uma expressão.

No exemplo da chamada da função `mediaMelhoresNotas` o resultado (valor de retorno da função) era armazenado nas variáveis `mediaA` e `mediaB`:

```
mediaA = mediaMelhoresNotas(notaA1, notaA2, notaA3);  
mediaB = mediaMelhoresNotas(notaB1, notaB2, notaB3);
```

Na especificação da função, precisamos indicar o tipo do valor de retorno, antes do nome da função.

Exemplo:

Tipo do valor de retorno

```
float mediaMelhoresNotas(  
    float nota1, float nota2, float nota3) {  
  
    float media;  
    if ((nota1 <= nota2) && (nota1 <= nota3)) {  
        media = (nota2 + nota3) / 2.0f;  
    } else if ((nota2 <= nota1) && (nota2 <= nota3)) {  
        media = (nota1 + nota3) / 2.0f;  
    } else {  
        media = (nota1 + nota2) / 2.0f;  
    }  
  
    return media;  
}
```

Término da função, indicando qual valor deve ser retornado.

O comando **return** não necessariamente precisa ser a última instrução da função. Ao invés de salvar o resultado na variável **media** para retornar o valor desta variável somente no final da função, poderíamos terminar a execução da função assim que o resultado for calculado.

Exemplo:

```
float mediaMelhoresNotas(  
    float nota1, float nota2, float nota3) {  
  
    if ((nota1 <= nota2) && (nota1 <= nota3)) {  
        return (nota2 + nota3) / 2.0f;  
    } else if ((nota2 <= nota1) && (nota2 <= nota3)) {  
        return (nota1 + nota3) / 2.0f;  
    } else {  
        return (nota1 + nota2) / 2.0f;  
    }  
}
```

Funções sem valor de retorno

Em casos especiais, a função apenas executa o código, mas não gera resultado que possa ser retornado. Nesses casos, o tipo da função será **void**. O comando **return** passa a ser opcional e pode ser chamado sem valor de retorno, apenas para terminar a função. Estas funções costumam gerar algum outro efeito, como imprimir um resultado na tela ou escrever algum dado em um arquivo.

Exemplo:

```
void comparaNotas(  
    float notaA, float notaB) {  
  
    if (mediaA > mediaB) {  
        printf("Aluno A obteve melhor desempenho");  
    } else if (mediaA < mediaB) {  
        printf("Aluno B obteve melhor desempenho");  
    } else {  
        printf("Os alunos A e B tem a mesma media");  
    }  
}
```

A rigor, funções que não retornam um valor deveriam ser chamadas de procedimentos, mas a linguagem C preferiu não distinguir estes casos.

Observação: A função ou sempre gera um resultado (e, portanto, declara um tipo para o valor de retorno) ou será de tipo `void` (nunca retorna valor). Não existe uma função que retorna um valor somente sob determinada condição! Se sua função deveria retornar um valor somente em alguns casos, então a lógica do seu programa está confusa.

Exemplo Completo

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

float mediaMelhoresNotas(float nota1, float nota2, float nota3) {
    float media;
    if ((nota1 <= nota2) && (nota1 <= nota3)) {
        media = (nota2 + nota3) / 2.0f;
    } else if ((nota2 <= nota1) && (nota2 <= nota3)) {
        media = (nota1 + nota3) / 2.0f;
    } else {
        media = (nota1 + nota2) / 2.0f;
    }
    return media;
}

void comparaNotas(float notaA, float notaB) {
    if (notaA > notaB) {
        printf("Aluno A obteve melhor desempenho");
    } else if (notaA < notaB) {
        printf("Aluno B obteve melhor desempenho");
    } else {
        printf("Os alunos A e B tem a mesma media");
    }
}

int main(int argc, char *argv[]) {
    // Declarar notas e médias dos alunos A e B
    float notaA1, notaA2, notaA3, mediaA;
    float notaB1, notaB2, notaB3, mediaB;

    // Ler notas dos alunos A e B
    scanf("%f %f %f", &notaA1, &notaA2, &notaA3);
    scanf("%f %f %f", &notaB1, &notaB2, &notaB3);

    // Calcular a média para o aluno A
    mediaA = mediaMelhoresNotas(notaA1, notaA2, notaA3);

    // Calcular a média para o aluno B
    mediaB = mediaMelhoresNotas(notaB1, notaB2, notaB3);

    // Imprimir resultado
    comparaNotas(mediaA, mediaB);

    return 0;
}
```

Consulte: Funcoes\Notas02\Notas02.vcproj

Note como o programa principal ficou bastante simples! Ele simplesmente delega as operações para as funções específicas.

Funções Importantes

Função main

Todos os programas precisam conter uma função especial, denominada `main`. Ela é executada automaticamente ao iniciar o programa.

Um bom programa apresenta uma função `main` curta. Ela apenas delega as tarefas para outras funções que implementam operações específicas. Isto deixa o programa claro.

Escrita e leitura

Os comandos `printf` e `scanf`, utilizados para escrever e ler dados, são implementados como funções. Note a semelhança do uso de `printf` com uma chamada de função:

```
printf("Resultado: %d", valor);
scanf("%d %d", &linhas, &colunas);
```

Nestes dois exemplos, `"Resultado %d"` e `valor` são os dois parâmetros para a função `printf`. Já para `scanf`, os parâmetros são `"%d %d"`, `&linhas`, e `&colunas`. As funções `scanf` e `printf` (e as respectivas variantes para escrever ou ler de arquivo, `fprintf` e `fscanf`) são casos especiais de funções, pois permitem um número variável de parâmetros.

Manipulação de arquivos

Toda a manipulação de arquivos é realizada por comandos que também são funções: `fopen`, `fclose`, `fscanf`, `fprintf`, `rewind`, `fflush`, `fputc`, `fputs`, `fgetc`, `fgets`, `ftell`, `fseek`, entre outras. Veremos detalhes mais adiante.

Funções matemáticas

Ao adicionar a diretiva `#include <math.h>` no começo do programa, são disponibilizadas novas funções matemáticas específicas, tais como `sqrt` (raiz quadrada), `sqr` e `pow` (potência), `sin`, `cos`, `tan` e muitas outras. Para conhecer a lista completa das funções matemáticas, consulte a documentação do compilador C.

Vetores de caracteres

Ao adicionar a diretiva `#include <string.h>` no começo do programa, são disponibilizadas funções para manipular vetores de caracteres, como por exemplo: `strcmp` (comparar dois vetores de caracteres), `strcpy` (copiar o conteúdo de um vetor de caracteres para outro), `strlen` (obter o tamanho do texto armazenado em um vetor de caracteres). Para conhecer a lista completa das funções de manipulação de vetores de caracteres, consulte a documentação do compilador C.

Gerenciamento de memória dinâmica

Aprenderemos neste curso as seguintes funções para gerenciar memória dinâmica: `malloc`, `realloc` e `free`.

Exemplos de funções

Matemática

Funções em C são uma forma natural para implementar funções matemáticas. Por exemplo, podemos criar a função potência (de expoentes inteiros não negativos) utilizando multiplicações:

```
float potencia(float base, int expoente) {
    float resultado = 1.0f;
    int contador;
    for (contador = 0; contador < expoente; contador++) {
        resultado = resultado * base;
    }
    return resultado;
}
```

A primeira linha declara o valor de retorno, o nome e os argumentos da função:

```
float potencia(float base, int expoente)
```

Em outras palavras, o resultado (o valor de retorno) será interpretado como um tipo *float*. A função é identificada pelo nome *potencia*. Ela tem dois argumentos, o primeiro (*base*) é a base da potência e tem tipo *float*. O segundo (*expoente*) é um número inteiro não negativo que indica quantas vezes devemos multiplicar a base por si mesma. Ao chamar a função, o código chamador precisa informar estes dois números.

O corpo da função começa declarando uma variável auxiliar para calcular o resultado. É comum que funções utilizem variáveis auxiliares com este propósito.

Em seguida, um **for** multiplica o resultado pela base tantas vezes quanto indicado pelo argumento *expoente*. Note que essa versão simples do **for** não calcula corretamente a potência se *expoente* for um valor negativo.

No fim, o corpo da função executa o comando **return**, que faz a função terminar e a execução voltar para o código chamador, além de retornar o valor da variável resultado.

Um uso simples desta função é calcular a potência de vários números.

```
int main(int argc, char *argv[]) {
    float base, resultado;
    int expoente;

    printf("Base: ");
    scanf("%f",&base);
    printf("Expoente: ");
    scanf("%d", &expoente);

    resultado = potencia(base, expoente);

    printf("Resultado: %f\n", resultado);

    return 0;
}
```

Um exemplo do uso dessa função seria em chamadas repetitivas (com diferentes valores para o expoente) em uma mesma expressão. Como, por exemplo, para avaliar o valor de um polinômio. Se o polinômio fosse $x^3 - 2x^2 + x - 3$, cada chamada da função na forma *potencia(x, i)* calcularia o valor de um termo x^i .

```
int main(int argc, char *argv[]) {
    float x, p;

    printf("Valor de x: ");
    scanf("%f",&x);
    p = potencia(x, 3) - 2 * potencia(x, 2) + potencia(x, 1) - 3;
    printf("Polinomio p = %f\n", p);

    return 0;
}
```

Este algoritmo não é eficiente, embora esteja correto. Note que as potências de *x* são recalculadas cada vez que a função *potencia* é chamada. Seria possível alterar o algoritmo para evitar essa ineficiência?

Outras funções

Nem todas as funções precisam realizar cálculos matemáticos. Existem casos quando desejamos apenas criar uma função para evitar repetir uma mesma lógica de execução. Suponha que necessitamos solicitar um número através de uma mensagem, verificar o número digitado e repetir a solicitação até que um número válido seja digitado. Se este conjunto de operações ocorrer várias vezes no programa para solicitar diversos números, então será interessante criar uma função para esta tarefa.

```
int le_numero(int minimo, int maximo) {
    int leitura;

    scanf("%d", &leitura);
    while ((leitura < minimo) || (leitura > maximo)) {
        printf("Valor fora do intervalo permitido (%d a %d)\n",
minimo, maximo);
        printf("Digite novamente: ");
        scanf("%d", &leitura);
    }
    return leitura;
}
```

Para chamar a função, poderíamos escrever:

```
int main(int argc, char *argv[]) {
    int valor;

    printf("Pense em um numero de 1 a 100: ");
    valor = le_numero(1, 100);

    printf("Voce pensou em: %d\n", valor);

    return 0;
}
```

Consulte: Funcoes\Entrada01\Entrada01.vcproj

Com valor de retorno, mas sem parâmetros

Existem funções que não possuem parâmetros e mesmo assim retornam um valor. Neste caso, elas precisam obter alguma entrada de uma outra fonte que não sejam os argumentos. Lembre que a função implementa um algoritmo e todos os algoritmos precisam de uma entrada, caso contrário sempre vão gerar o mesmo valor.

Funções para organização de código

Um outro caso de funções sem parâmetros ocorre quando desejamos organizar um programa grande. Nestas situações, dividimos o programa em grandes blocos, cada qual responsável por uma operação. O programa principal (função `main`) é encarregado apenas de chamar a função que contém o bloco desejado.

Suponha um programa com várias funcionalidades: calcular raízes de polinômio, calcular o determinante de uma matriz e derivar funções. No início, o usuário escolhe qual opção deseja executar. Uma abordagem ruim seria:

```
int main(int argc, char *argv[]) {
    char opcao;
    printf("Que operacao deseja realizar?\n");
    printf("  P - raiz de polinomio\n");
    printf("  M - determinante de matriz\n");
    printf("  D - derivar funcoes\n");
    scanf("%c", &opcao);
    if (opcao == "P") {
        // Código para calcular raízes de polinômios
        // ...
    } else if (opcao == "M") {
        // Código para calcular determinantes de matrizes
        // ...
    }
}
```

```

        } else if (opcao == "D") {
            // Código para calcular derivadas de funções
            // ...
        }
    }
}

```

Acontece que cada uma das três operações exige um grande número de linhas de código e o programa ficará muito extenso e difícil de entender. Além disso, é confuso localizar onde começa e termina o código de uma opção.

Seria muito mais interessante dividir o programa em vários blocos, cada qual implementado através de uma função:

```

void resolver_polinomio() {
    // Código para calcular raízes de polinômios
    // ...
}

void calcular_determinante() {
    // Código para calcular determinantes de matrizes
    // ...
}

void derivar_funcao() {
    // Código para calcular derivadas de funções
    // ...
}

```

Cada função não recebe parâmetros nem gera um valor de retorno pois, na verdade, são programas totalmente independentes.

Nenhuma função deveria ter mais de 200 linhas de código. Se este for o caso, verifique se não é possível reorganizar o código.

O programa principal (função `main`) será um mero delegador de execução. Dependendo da opção fornecida pelo usuário, ele chama a respectiva função.

```

int main(int argc, char *argv[]) {
    char opcao;
    printf("Que operacao deseja realizar?\n");
    printf("  P - raiz de polinomio\n");
    printf("  M - determinante de matriz\n");
    printf("  D - derivar funcoes\n");
    scanf("%c", &opcao);
    if (opcao == "P") {
        resolver_polinomio();
    } else if (opcao == "M") {
        calcular_determinante();
    } else if (opcao == "D") {
        derivar_funcao();
    }
}

```

Tempo de vida de variáveis

Cada função pode declarar suas próprias variáveis. Como vimos, essas variáveis só serão visíveis quando a respectiva função executar.

Além disso, C permite declarar variáveis fora do corpo de todas as funções. Essas variáveis estariam sempre disponíveis, durante a execução de qualquer função.

Mas o que ocorre quando variáveis são declaradas com o mesmo nome, mas em funções diferentes? Uma função consegue acessar variáveis de outra função? Para obter as respostas, é importante entender as regras de **tempo de vida de variáveis**.

Tempo de vida

As regras para se determinar o tempo de vida de uma variável controlam por quanto tempo o conteúdo de uma variável permanece armazenado na memória. Existem variáveis que armazenam um valor até o final da execução do programa. Outras mantêm o valor apenas enquanto o computador está executando uma determinada função.

O tempo de vida da variável está diretamente associado com as regras que associam espaço de memória a ela. Quando uma variável passa a existir (inicia seu tempo de vida), o computador reserva determinado espaço na memória para armazenar o valor da variável. Enquanto a variável estiver viva, este espaço de memória estará disponível para leitura e escrita, ou seja, para recuperar e atribuir o valor da variável. Em certo momento, quando a variável deixar de existir, i.e., termina seu tempo de vida, este espaço de memória é liberado e o conteúdo da variável é perdido para sempre.

O tempo de vida de uma variável está diretamente ligado ao seu ponto de declaração no código fonte. Isso dá origem a **variáveis globais** e **variáveis locais**.

Variáveis globais

A **variável global** é aquela declarada fora de qualquer função, tipicamente no início do arquivo do código fonte. Ela permanece viva durante toda a execução do programa (sem restrição no tempo de vida). Antes de iniciar a execução do programa, uma variável global recebe um espaço na memória, o qual vai reter até o final da execução do programa, ou seja, a variável armazena valores até o programa terminar.

No exemplo abaixo, a variável `v` é global, pois ela é declarada fora do corpo de todas as funções. Tanto a função `f1` como `f2` podem acessar e modificar valor de `v`.

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

int v = 0;

void f1() {
    v++;
    printf("f1: v = %d\n", v);
}

void f2() {
    v+=2;
    printf("f2: v = %d\n", v);
}

int main(int argc, char *argv[]) {
    f1();
    f2();
    f1();
}
```

Consulte: Funcoes\Visibilidade01\Visibilidade01.vcproj

O resultado da execução será:

```
f1: v = 1
f2: v = 3
f1: v = 4
```

Como a variável `v` é global, cada chamada (de `f1` ou de `f2`) modifica o valor de `v`, conforme indica a saída do programa.

Variáveis locais

A **variável local** (também chamada de *automática*) é declarada dentro do corpo de uma função, ou como parâmetro. Ela vive enquanto esta função estiver executando (tempo de vida restrito à função). Após a execução de um comando **return**, que finaliza a função, ela deixa de existir, o espaço de memória associado à variável é desmobilizado, o valor lá armazenado é perdido. Portanto, o tempo de vida se restringe ao período de execução da função.

Cada vez que a função é executada, memória é novamente alocada para a variável, possivelmente em *posições diferentes* na memória do computador a cada execução da função. Durante *esta* execução da função, as variáveis locais retêm suas associações de memória e seus valores podem ser alterados e consultados. Toda vez que uma função termina sua execução, o espaço de memória associado à variáveis locais é demobilizado e os valores lá armazenados são perdidos. A variável local *não consegue* lembrar o valor que tinha na execução anterior da mesma função.

No exemplo abaixo, a variável **v** é local para cada uma das funções **f1** e **f2**. Note que, apesar de ter o mesmo nome, cada vez que **f1** ou **f2** iniciam, a respectiva variável **v** é estabelecida em uma posição de memória (possivelmente diferentes, não importa!) e, portanto, se comportam como variáveis diferentes. Além disso, por serem locais, a variável **v** declarada em **f1** é desalocada da memória antes do início de **f2** e, portanto, **f2** não pode acessar seu conteúdo. Quando **f2** inicia, o computador vai alocar memória para a variável **v** de **f2**, possivelmente em outra posição diferente da memória. Quando **f2** executa, essa nova posição é usada para guardar seus valores até que esta execução de **f2** termine. Nesse ponto a memória alocada para essa variável local é liberada. O ciclo de alocação e liberação de memória se repete quando **f1** voltar a executar novamente. Repare que as variáveis de **f1** e de **f2**, e mesmo de diferentes execuções de **f1** ou de **f2** estarão localizadas em posições diferentes da memória, sem possibilidade de comunicação entre si. Assim, não há ambiguidade quando **f1** ou **f2** acessa sua respectiva variável **v**, em uma particular execução.

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

void f1() {
    int v = 0;
    v++;
    printf("f1: v = %d\n", v);
}

void f2() {
    int v = 0;
    v+=2;
    printf("f2: v = %d\n", v);
}

int main(int argc, char argv[]) {
    f1();
    f2();
    f1();
}
```

Consulte: Funcoes\Visibilidade02\Visibilidade02.vcproj

O resultado da execução será:

f1: v = 1

```
f2: v = 2
f1: v = 1
```

Observe que `f1` modificou a variável `v`, tal como fez `f2`. No entanto, cada uma das funções modificou sua própria variável `v` e as alterações não foram propagadas de `f1` para `f2`. Além disso, a variável `v` perde seu valor no final da execução da função e a segunda chamada de `f1` não conhece mais o valor da primeira chamada.

Sobreposição nomes de variáveis globais e variáveis locais

Em algumas situações pode ocorrer que duas variáveis *de mesmo nome* estejam vivas, num dado instante. Então, se usarmos o nome dessa variável em uma expressão que valor vamos obter, isto é, o nome comum referencia qual das variáveis? Nesta situação, o compilador adota a seguinte política: a variável *declarada mais recentemente* é aquela que está acessível nesse instante, ou seja, essa variável esconde as outras variáveis de mesmo nome que, por sua vez, ficam inacessíveis. É o caso quando uma função declara uma variável local com um mesmo nome que uma outra variável global já declarada. Nesse caso, a variável local tem prioridade sobre a variável global.

No próximo exemplo, enquanto a função `f` executa, qualquer referência ao nome da variável `v` é local, isto é, acessa a posição de memória associada à cópia local da variável `v`. Já na função `g`, toda referência ao nome de variável `v` vai acessar a posição de memória associada à declaração global de `v`, fora das funções.

```
int v = 0;

int f(void) {
    int v;
    ...
    v = 2;
    printf("f: v = %d\n", v);
}

int g(void) {
    v = 3;
    printf("g: v = %d\n", v);
}

int main(int argc, char *argv[]) {
    printf("main: v = %d\n", v);
    f();
    printf("main: v = %d\n", v);
    g();
    printf("main: v = %d\n", v);
}
```

O resultado da execução será:

```
main: v = 0
f: v = 2
main: v = 0

g: v = 3
main: v = 3
```

Logo no início da execução o programa imprime o valor zero para `v`. É o valor que inicializa a variável na declaração global. Quando a função `f` executa, o valor impresso é 2, correspondendo ao valor atribuído à `v` no corpo da função `f`. Porém, quando termina a execução de `f` e se retorna ao programa principal `main`, o valor

agora impresso é o mesmo zero que foi impresso antes da chamada e execução da função `f`. Isto porque `f` declara uma variável `v` de mesmo nome que aquela da declaração global. O computador aloca uma outra posição de memória para essa declaração de `v` no corpo de `f`, e usa essa nova posição de memória nas atribuições à `f` durante a execução de `f`. Quando a execução de `f` termina, essa nova posição é desmobilizada e, agora, uma referência à variável `v` vai acessar a posição original, alocada quando da declaração global. Como, durante a execução de `f` esta posição ficou intocada, a segunda impressão em `main`, reproduz o valor que a variável global `v` detinha logo antes de `f` iniciar sua execução, ou seja, imprime o valor zero.

Já a função `g` não declara uma variável local de nome `v`. Então, a atribuição à `v` em `g` altera a mesma posição de memória que foi associada à variável global `v`. Logo, o valor atribuído à `v` em `g`, foi reproduzido na terceira impressão que ocorre em `main`, mostrando que a atribuição de `g` foi mantida e se perpetuou após o término da execução de `g`. Na verdade, a atribuição à `v` em `g` e em `main`, estão acessando a *mesma* posição de memória.

Arquivos de Cabeçalho ("headers")

Já sabemos que podemos incluir arquivos especiais em um programa usando a diretiva de compilação `#include`. Normalmente, usamos essa diretiva escrevendo o nome do arquivo a incluir entre os símbolos `<` e `>`, indicando que este é um arquivo tipo "header file" padrão do sistema. Nesses casos, o compilador sabe onde encontrar esse arquivo nos diretórios do sistema operacional. O efeito, como já vimos, seria que o compilador vai ler o texto do arquivo incluído no ponto de invocação de diretiva `#include`, ou seja, tudo se passa como se o texto do arquivo incluído fosse parte do texto do nosso programa, aparecendo exatamente no ponto de invocação da diretiva.

A linguagem C também permite com que o programador construa seus próprios arquivos de cabeçalho e os inclua no seu programa. Usualmente, o nome desses arquivos terminam com a extensão `.h`, um indicativo de que é um "header file". Um arquivo desse tipo pode conter qualquer código C mas, normalmente, contém definições comuns a várias rotinas. Assim, para ter essas definições presentes em qualquer ponto de outro arquivo que contenha código fonte C, bastará invocar o nome do arquivo de inclusão no ponto desejado do código fonte, usando para isso a diretiva `#include`.

No caso de arquivos de inclusão definidos pelo programador, na diretiva `#include` o nome do arquivo deve estar entre aspas duplas. E devemos também indicar em que diretório o arquivo de inclusão se encontra, uma vez que o compilador não saberá localizá-lo. Se o nome do diretório não for indicado, o compilador vai procurar o arquivo de inclusão no diretório corrente.

No exemplo seguinte, criamos um arquivo de inclusão chamado `consts.h`, que contém outras inclusões e algumas definições.

```
// includes padrao do sistema
#include <stdio.h>

// definicao de algumas constantes
#define BRANCO ' ' // char para o espaco em branco
#define FIM_DE_LINHA '\n' // char para o return
#define TAB '\t' // char para o tab
```

Como pode ser visto, o primeiro `#include` já é tradicional, e faz com que o conteúdo do arquivo `stdio.h` seja expandido no ponto da sua invocação. Tudo se passa como se o texto do arquivo `consts.h` fosse aumentado, incluindo todo o texto do arquivo `stdio.h` naquele ponto. Da forma da invocação, fica claro que esse último arquivo faz parte do sistema. As linhas restantes no arquivo `consts.h` contêm a definição de três nomes, `BRANCO`, `FIM_DE_LINHA` e `TAB`. O primeiro corresponde ao caracter "espaço em branco", o segundo ao caracter de "nova linha" e o terceiro ao caracter de tabulação.

O programa a seguir inclui o texto do arquivo `consts.h`.

```
#include "consts.h" // meu arquivo de includes
char c; // variavel global
// funcao para pular brancos
char limpabrancos(void) { /* assume lo. char ja carregado em c */
    . . . .
}
// funcao para imprimir a palavra
void impal(void) { /* assume lo. char ja em c */
    . . . .
}
// rotina principal
int main(void) {
    . . .
}
```

Logo na primeira linha o arquivo `consts.h` é incluído. Teríamos um efeito totalmente equivalente se tivéssemos transcrito todo o conteúdo do arquivo naquele ponto. Note como a diretiva `#include` presente no arquivo `consts.h` é também incluída no texto do programa, trazendo para o código do programa todo o conteúdo do arquivo `stdio.h`.

Na segunda linha, a variável global `c` é declarada.

O restante do código apresenta duas funções e a rotina principal. A primeira função está transcrita abaixo.

```
char limpabrancos(void) { /* assume lo. char ja carregado em c */
    while ( (c!=FIM_DE_LINHA) && ((c==TAB) || (c==BRANCO)) ) c=getchar(
);
    return c;
}
```

Essa função não usa parâmetros e retorna um valor tipo `char`. O efeito dessa função deve ser retornar o primeiro caracter da entrada antes do "fim de linha", e que também não seja uma pontuação. Note que, ao chamarmos essa função, assumimos que a variável `c` já contém o primeiro caracter da entrada (veja os comentários na definição da função). Assim, se o primeiro caracter não for `FIM_DE_LINHA`, mas for `BRANCO` ou `TAB`, o corpo do `while` é executado e o programa vai ignorar todos os caracteres `BRANCO` ou `TAB`, até que encontre um `FIM_DE_LINHA`, ou um caracter que não seja nem `BRANCO` nem `TAB`. Caso contrário, se o primeiro caracter for `FIM_DE_LINHA`, ou não for `BRANCO` nem `TAB`, o `while` não é executado e esse primeiro caracter é retornado imediatamente.

O comando `while` é formado apenas pelo comando

```
c=getchar( );
```

e faz todo o trabalho. A rotina `getchar()`, também invocada sem parâmetros, retorna o próximo caracter da entrada. Esse valor é atribuído à variável `c`. Em seguida, o fluxo de controle retorna para o teste do `while`. A expressão será

verdadeira sempre que a variável `c` não contiver o caracter de "fim de linha", mas contiver ou o caracter de tabulação ou o caracter de "espaço em branco". Ou seja, se ainda não esgotamos a linha de entrada e o último caracter lido for um "espaço em branco" ou uma tabulação, vamos pegar o próximo caracter da entrada. O efeito líquido, então, será ler os próximos caracteres de entrada até encontrar um que não seja nem o "espaço em branco" nem uma tabulação (assumindo que não encontramos ainda o "fim de linha"). Quando esse caracter for encontrado, o `while` termina e a rotina retorna o valor desse caracter. Note que esse algoritmo só funciona corretamente se o primeiro caracter a ser impresso *já está armazenado em `c` antes da rotina `limpabrancos` iniciar*. Para isso, optamos por definir `c` como uma variável global. Mesmo porque a próxima rotina também vai utilizar a variável `c`.

A segunda função aparece abaixo.

```
void imppal(void) { /* assume lo. char ja em c */
    while ( (c != FIM_DE_LINHA) && (c != BRANCO) && (c != TAB) ) {
        putchar(c);
        c=getchar( );
    }
    putchar(FIM_DE_LINHA);
    return;
}
```

O efeito da função é imprimir uma palavra que é lida da entrada. A palavra é formada por todos os caracteres que estão na entrada, até que se encontre um "espaço em branco", uma tabulação ou o "fim de linha". A expressão que controla o `while` continua verdadeira enquanto essa condição não acontece. A cada passo, o último caracter lido da entrada é colocado na saída, pela chamada `putchar(c)`, e o próximo caracter é lido e atribuído à variável `c`. Note que essa rotina faz todo o seu trabalho sem precisar de parâmetros de entrada, e mesmo sem precisar retornar nada ao programa que a chama.

A rotina principal poderia ser codificada como segue.

```
int main(void) {
    printf("Entre com uma frase: ");
    c=getchar();
    do {
        if ( limpabrancos() == FIM_DE_LINHA ) break;
        imppal( );
    } while (1);

    system("PAUSE");
    return 0;
}
```

O código é simples. Inicialmente, o primeiro caracter é armazenado na variável `c`. Em seguida, entramos em um laço onde a função `limpabrancos` é acionada para desconsiderarmos os próximos "espaços em branco" e tabulações. A função `limpabrancos` retorna o próximo caracter que não é um "espaço em branco" ou uma tabulação. Esse caracter é então testado para ver se é um "fim de linha". Se já encontramos o "fim de linha", o programa termina. Caso contrário, encontramos o início de uma palavra a ser impressa. Nesse caso, a função `imppal` é acionada para imprimir essa palavra, e o ciclo se repete. Note que o `do...while` repete eternamente, pois a condição `1` no `while` será sempre verdadeira, obviamente. A única forma de sair do laço é através do comando `break`.

Consulte: Funcoes\Pegapa\PegaPal.vcproj

Pragmas de Funções

Quando um programa usa funções, podemos escrever o código das funções antes do código da rotina principal. Um exemplo típico de trecho de código seria

```
#include <stdio.h>
// funcao para calcular o MDC; entrada sao os numeros; retorna o MDC
int mdc(int i, int j) {
    while (i != j) {
        if (i>j) i -=j;
        else j -=i;
    }
    return i;
}
// rotina principal
int main(void) {
    int n, m; /* inteiros dados */
    printf("Entre com dois inteiros positivos: ");
    scanf("%d %d", &n, &m);
    while ( (n<=0) || (m<=0) ) {
        printf("Os dois valores devem ser inteiros positivos. Tente de novo: ");
        scanf("%d %d", &n, &m);
    }
    printf("O mdc entre %ld e %ld vale %ld\n", n, m, mdc(m, n));
    system("PAUSE");
    return 0;
}
```

Ocorre que, com frequência, o programa ficaria mais bem organizado se pudéssemos escrever o código da rotina principal *antes* do código da função auxiliar. Assim, quem examinasse o código logo se depararia com o programa principal, podendo se inteirar do que o programa faria. Em seguida listaríamos os códigos das rotinas auxiliares.

O problema com esta abordagem é que o compilador ao analisar o código da rotina principal, vai se deparar com chamadas das rotinas auxiliares e, nestes instantes, o compilador não saberia reconhecer os nomes das rotinas auxiliares, pois estas *não teriam sido declaradas* até então. Nesse caso, o compilador anunciaria que os nomes das rotinas seriam identificadores desconhecidos (ou não declarados) e anunciaria um erro de compilação! Note que os nomes de rotinas de entrada e saída, tais como `printf` ou `scanf`, não se enquadram nesse problema justamente porque o arquivo `stdio.h`, que deve ter sido incluído logo no início do programa, contém a descrição dessas rotinas.

Para resolver esse dilema, a linguagem C permite com que simplesmente declaremos as funções logo de início, incluindo seus tipos de retorno e parâmetros, porém *sem listar o código das rotinas propriamente ditos*. Estes códigos podem ser escritos mais adiante, em pontos que o programador julgar mais apropriado. Assim, logo de início, o compilador já teria cadastrado os nomes das rotinas auxiliares, junto com seus tipos de retorno e parâmetros. Ao analisar o código da rotina principal encontrando uma linha com uma chamada de uma das rotinas auxiliares, o compilador poderia então determinar o identificador do nome da rotina é conhecido e poderia verificar se os parâmetros de chamada e seus tipos de retorno estão sendo usados de forma coerente pelo programador. No caso anterior, o código poderia ser colocado na forma

```

/* calcular o maximo divisor comum */
#include <stdio.h>
int mdc(int n, int m); // um pragma: so a definicao da funcao e seus
                        // parametros

int main(void) {
    int n, m; /* inteiros dados */
    printf("Entre com dois inteiros positivos: ");
    scanf("%d %d", &n, &m);
    while ( (n<=0) || (m<=0) ) {
        printf("Os dois valores devem ser inteiros positivos. Tente de
novo: ");
        scanf("%d %d", &n, &m);
    }
    // chamada da rotina mdc
    printf("O mdc entre %ld e %ld vale %ld\n", n, m, mdc(m, n));
    system("PAUSE");
    return 0;
}

// corpo completo da rotina mdc
int mdc(int i, int j) {
    while (i != j) {
        if (i>j) i -=j;
        else j -=i;
    }
    return i;
}

```

Na terceira linha temos apenas a declaração

```
int mdc(int n, int m);
```

Essa declaração apenas anuncia que o identificador `mdc` corresponde a uma função que recebe dois parâmetros de tipo `int` e retorna um valor também de tipo `int`. A partir desse ponto, o compilador saberá reconhecer toda ocorrência de `mdc` como uma chamada a uma tal função. Note que quando encontra uma dessas chamadas, o compilador não precisa conhecer em detalhe o código da rotina; precisa apenas saber que nesse ponto deve gerar um desvio para algum lugar da memória onde estará esse código.

Essas declarações de funções são também conhecidas como *pragmas*.

Note que o nome da rotina se torna conhecido logo após o compilador passar pelo ponto onde está o seu pragma. Em qualquer ponto mais adiante no código, o programador está livre para invocar a rotina pelo seu nome, desde que o faça de forma coerente com os parâmetros que precisam ser passados e desde que utilize o valor de retorno também de forma coerente com o seu tipo declarado no pragma. É o que é feito na rotina principal, quando esta invoca a rotina `mdc`.

Também, dada impede que o programador crie um arquivo tipo cabeçalho ("header file") onde pode reunir todas os pragmas de funções. Esse arquivo pode ser incluído em qualquer ponto para tornar as declarações dessas rotinas conhecidas. Desta forma, o programador pode melhor organizar seu código.