

Outros Tópicos

Argumentos para a Função main

Quando executamos a função principal, `main`, podemos também passar argumentos do ambiente externo para `main`. Para tal, essa função é usualmente declarada na forma

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

O primeiro parâmetro define uma variável local, `argc`, que é do tipo inteiro. Essa variável conterá o número de “tokens”, ou seja, *strings*, que estiverem presentes na linha de chamada da função `main`. Dependendo do sistema operacional, a maneira de construir a linha de chamada pode variar bastante. De qualquer forma, vamos imaginar que o programa principal foi compilado e agora temos um arquivo executável de nome `prog`. Então, uma ativação para execução de `prog` na forma

```
prog abc -de3 hjy
```

conterá 4 “tokens”, a saber: `prog`, `abc`, `-de3` e `h jy`. Assim, nesse caso, a variável `argc` será inicializada com o valor 4 dentro da função `main`. Note que o nome do programa que executa é também contado como um dos “tokens” da linha de ativação.

A variável `argc` indica o número de argumentos. O nome de cada argumento é passado para a função `main` através da variável `argv`. Essa variável é do tipo vetor de apontadores para `char`, ou seja é um vetor de *strings*. Naturalmente, a *string* na posição apontada por `argv[0]` é o nome do primeiro “token”, a *string* na posição `argv[1]` nos dá o nome do segundo “token” e assim por diante. Observe que, desta forma, `argv[0]` nos dará o nome do programa que foi chamado na linha de ativação.

Exemplo

Considere o seguinte simples trecho de código:

```
int main(int argc, char *argv[ ]) {  
    int i;  
    printf("Os argumentos passados foram: \n");  
    for( i=0; i<argc; i++) {  
        printf("Argumento %d = %s\n",i,argv[i]);  
    }  
    return 0;  
}
```

O comando `for` varre o vetor `argv[]` da posição 0 até a posição correspondente ao último argumento que foi passado, i.e. até a posição `argv-1`. O comando `printf` imprime

a *string* correspondente a cada um dos argumentos que foram passados na linha de ativação. Com a chamada acima, esse código imprimiria

```
prog
abc
-de3
hjt
```

Passando Comandos para o Sistema Operacional

Ao executarmos um programa em C, é possível repassar comandos para o sistema operacional, capturar os resultados e analisá-los. É claro que os comandos que podem ser repassados e as formas de retorno da execução desses comandos é extremamente dependente do sistema operacional onde o programa C executa.

A forma de repassar um comando para o sistema operacional é através da função `system` em C. Essa função recebe como argumento uma *string* que simplesmente é repassada intacta ao sistema operacional para execução. Por exemplo, a linha de C:

```
system("dir *.c /b > meuarq")
```

solicita que o sistema operacional execute o comando `dir *.c /b > meuarq`. Num ambiente DOS/WINDOWS, esse comando lista todos os nomes de arquivo que terminam em `.c` no diretório corrente, jogando o resultado da listagem no arquivo `meuarq`, também no diretório corrente. O modificador `/b` simplesmente suprime todos os detalhes da listagem, exceto pelo nome dos arquivos.

Outras funções de C ativam comandos específicos do sistema operacional. Como exemplo:

- `remove("nome_arq")`: essa função solicita que o sistema operacional remova o arquivo `nome_arq`, no diretório corrente. Observe que a string de argumento pode conter nomes de absolutos de arquivos, i.e. de arquivos em outros diretórios que não o diretório corrente. Basta indicar o diretório desejado usando o símbolo `\` (no sistema DOS/WINDOWS). Por exemplo, a string `"..\dir2\toto"` indicaria o arquivo de nome `toto`, alcançável a partir do diretório corrente subindo-se para o diretório pai e descendo pelo diretório irmão `dir2`.
- `rename("arq_velho", "arq_novo")`: renomeia o arquivo de nome `arq_velho` para `arq_novo`.
- `tmpnam(NULL)`: retorna uma *string* contendo o nome de um arquivo que não existe no diretório local. Existe um limite no número de chamadas a esta função com garantias de não repetição. Esse número depende de cada ambiente de execução.

Exemplo

Considere o trecho de código:

```
int main(void) {
    char *tmp;
    char com[100], linha[50];
```

```

int i=0;
FILE *com_sai;

tmp=tmpnam(NULL);
printf("Arquivo temporario eh <%s>\n",tmp);

sprintf(com,"dir *.c /b > %s",tmp);
system(com);

com_sai=fopen(tmp,"r");
while( fscanf(com_sai,"%s",linha) != EOF) {
    fprintf(stdout,"Arquivo numero %1d eh %s\n",++i,linha);
}

printf("Ha %d arquivos com terminacao .c.\n",i);
fclose(com_sai);
remove(tmp);

return 0;
}

```

O primeiro comando atribui à variável `tmp` o nome de um novo arquivo, sem conflitar com nenhum outro nome no diretório corrente. Em seguida, esse nome é impresso. O comando `sprintf` se comporta exatamente como o comando `fprintf`, exceto que, ao invés de colocar o resultado em um arquivo externo, coloca o resultado da impressão em uma *string*, que deve vir designada como seu primeiro argumento. No caso, a *string* para retorno é o vetor `cmd`. Assim, o resultado do comando `sprintf` é escrever no vetor `cmd` a *string* `"dir *.c /b > toto"`, assumindo que `toto` foi o nome único retornado pela função `tmpnam`.

Em seguida, a função `system` repassa para o sistema operacional o comando `dir *.c /b > toto`. Após executar o comando, o arquivo externo `toto` conterá o nome de todos os arquivos do diretório corrente que terminam com o sufixo `.c`. No caso, o diretório corrente é o diretório a partir do qual o programa foi ativado. Note como é importante a unicidade do nome `toto`. Se não fosse assim, um dos arquivos no diretório corrente poderia ter seu conteúdo destruído e substituído pelo resultado do comando `dir *.c /b > . . .`.

O próximo comando abre para leitura o arquivo `toto`. O comando `while` vai ler o conteúdo do arquivo `toto`, uma *string* de cada vez. Nesse caso, lerá um nome de arquivo que termina em `.c` por vez. Cada leitura guarda a *string* lida na variável `linha`. Em seguida, no corpo do `while`, a *string* apontada pela variável `linha` é impressa no arquivo de saída `stdout`. Ou seja, os nomes de arquivos no diretório corrente que terminam com `.c` são impressos no arquivo de saída `stdout`, um por linha. O comando `while` termina quando chega ao final do arquivo `toto`.

O restante do código imprime o número de arquivos com sufixo `.c` que foram encontrados no diretório corrente, fecha o arquivo `toto` e, finalmente, remove o arquivo `toto`.

Medindo a Passagem do Tempo

Podemos medir a passagem do tempo ao executarmos programas em C. De um modo geral, essas medidas de tempo são aproximações um tanto grosseiras. Mas podem ser úteis quando queremos ter uma idéia do tempo decorrido entre dois pontos do programa, durante

sua execução. Vamos ver as funções mais simples para esse fim. Detalhes maiores e outras funções mais sofisticadas poderão ser estudadas consultando seu manual de C.

A função `time(NULL)` retorna uma aproximação para o tempo real, de relógio, nesse instante. O retorno é do tipo “`time_t`”. Não é importante obter detalhes desse tipo agora. Consulte seu manual de C, ou examine o arquivo “`time.h`” da sua particular implementação de C. A função `difftime(time_t t1, time_t t2)` retorna a diferença, convertida para segundos, entre os instantes `t1` e `t2` (ambos variáveis de tipo `time_t`).

Assim, se num primeiro instante executamos um comando na forma

```
t1=time(NULL);
```

e, logo mais adiante num segundo instante, executamos outro comando na forma

```
t2=time(NULL);
```

poderemos então executar a função `difftime` para obter a quantidade de segundos (aproximadamente) decorridos entre esses instantes de execução da função `time`.

Uma outra forma de medição de tempo é através da função `clock()`. Essa função retorna o um objeto de tipo “`clock_t`”, cujos detalhes também não são importantes agora. A implementação do tipo `clock_t` é muito dependente da arquitetura da máquina e da implementação específica do compilador C que estamos usando. Em geral, podemos fazer um *cast* para inteiro de valores desse tipo.

Uma execução da função `clock()` retorna o valor de um contador interno da CPU do computador. Em geral, esse contador avança várias vezes por segundo. Ou seja, sua medida é mais precisa que o valor em segundos de um relógio de tempo real. Assim, à semelhança do que foi feito com a função `time`, podemos obter diretamente a diferença entre duas medições desse contador interno (após convertidas para `int`). Se soubermos quantos pontos desse contador interno há por segundo, poderíamos transformar essa diferença em segundos. Essa informação está contida no nome definido `CLOCKS_PER_SEC`, que obtemos ao incluir o arquivo `time.h`.

Exemplo

Considere o código abaixo.

```
#define NUMINTOPS 1000000000
#define NUMFLOATOPS 1000000000

int main(void) {
    register int i;
    long tinicio;
    long relógio_ini;
    long int cpu_ini, cpu_fim;
    long op1, op2, res;
    double f1, f2, fres;
    double tiq_seg;

    relógio_ini=time(NULL);
    tinicio=clock();
    tiq_seg=(double)CLOCKS_PER_SEC;
    srand(time(NULL));
```

```

printf(" %d multiplic inteiras: ",NUMINTOPS);
op1=rand( ); op2=rand( );
cpu_ini=clock();
for (i=1;i<=NUMINTOPS;i++) res=op1*op2;
cpu_fim=clock();
printf(" %g segundos\n",(cpu_fim - cpu_ini)/tiq_seg);

f1=(double)op1;f2=(double)op2;

printf(" %d multiplic flutuantes: ",NUMFLOATOPS);
cpu_ini=clock();
for (i=1;i<=NUMFLOATOPS;i++) fres=f1*f2;
cpu_fim=clock();
printf(" %g segundos\n",(cpu_fim - cpu_ini)/tiq_seg);
printf(" Tempo cpu total: %g segundos\n",(clock()-tinicio)/tiq_seg);

printf(" Tempo relógio total: %.3fsegundos\n", difftime(time(NULL),relogio_ini));

return 0;
}

```

Os primeiros dois comandos capturam nas variáveis `relogio_ini` e `tinicio` os valores do tempo real e do contador interno, respectivamente, quando o programa inicia sua execução. Em seguida convertemos o valor de `CLOCKS_PER_SEC` para `double` e inicializamos a semente do gerador aleatório com o valor corrente do tempo real, em segundos. Note que, desta forma, a semente será inicializada com valores diferentes cada vez que o programa executar.

O próximo bloco de código imprime o número de operações com inteiros que iremos testar e obtém dois operandos aleatórios, armazenados nas variáveis `op1` e `op2`. Logo em seguida, o valor do contador interno nesse primeiro instante é armazenado na variável `cpu_ini`.

O comando `for` simplesmente realiza o número especificado de operações com inteiros. Logo após o comando `for`, o novo valor do contador interno é capturado na variável `cpu_fim`. Em seguida o número de segundos decorridos entre as duas últimas execuções da função `clock()` é impresso, já convertido para segundos. Esse seria, aproximadamente, o número de segundos que o computador levou para executar o número especificado de operações de multiplicação de inteiros. Assim, podemos ter uma idéia da velocidade da CPU para executar uma operação de multiplicação de inteiros.

O próximo bloco de código repete esse esquema, agora realizando um certo número de operações de ponto flutuante.

Por fim, o código imprime o tempo decorrido, em segundos, desde o seu início até esse ponto final. Primeiro, usa a função `clock()`. Note como a segunda chamada de `clock()` é realizada diretamente no cálculo da expressão que será impressa. A segunda tentativa usa a função `time(NULL)` e a função `difftime`, também diretamente na expressão contida na chamada da função `printf`.

Variáveis de Ambiente

Alguns sistemas operacionais permitem a definição de variáveis de ambiente. Uma variável de ambiente é um nome simbólico ao qual podemos atribuir um valor, geralmente uma *string*. Quando um programa C executa podemos ter várias dessas variáveis já definidas e valoradas. Durante a execução do programa, os nomes das variáveis de ambiente e seus valores podem ser consultados. Para tal, os parâmetros da função `main` deve ser na forma

```
. . . main( int argc, char * argv[], char * amb[]) { . . .
```

Observe o terceiro parâmetro, um vetor de *strings*. Da mesma forma que a variável `argv`, a nova variável `amb` conterá um apontador para cada variável de ambiente que estiver definida quando da execução do programa. Cada *string* apontada por `amb[i]` será da forma

`nome=valor`

onde `nome` é a designação (i.e. o nome) da variável de ambiente e `valor` é o seu correspondente valor. Os nomes e valores de variáveis de ambiente variam muito de sistema operacional para sistema operacional.

Exemplo

Considere o código simples:

```
int main(int argc, char *argv[], char *amb[]) {
    int i=0;
    char env[250], *p;

    while ( amb[i]!=NULL ) {
        if ((i+1)%10 == 0) { system("PAUSE"); }
        printf("Ambiente %d: %s\n",i,amb[i++]);
    }
    printf("-----\nDiretorios no caminho de busca:\n");
    strcpy(env,getenv("PATH"));
    printf(" \"%s\"\n",strtok(env,";"));
    while( (p=strtok(NULL, ";"))!=NULL ) { printf(" \"%s\"\n",p); }
    putchar('\n');

    return 0;
}
```

O comando `while` percorre o vetor `amb[i]`, processando cada uma das *strings* que encontra. Simplesmente, o valor de cada *string* é impressa, sendo que fazemos uma pausa a cada dez *strings* que são encontradas.

Em seguida, uma das variáveis de ambiente mais comuns, `PATH`, é analisada em mais detalhes. Primeiro, a função `getenv` é executada para extrair o valor dessa variável, se estiver definida, copiando esse valor para a *string* `env`, seu primeiro argumento. Note que para usar a função `getenv` não precisaríamos declarar o terceiro argumento de `main`; essa função realiza seu trabalho independentemente dessa declaração. Para analisar o valor da variável `PATH`, usamos a função `strtok` (i.e. "*string to token*"). Essa função separa uma *string* em *tokens*. Cada *token* é delimitado por um separador, indicado no segundo argumento da primeira chamada da função `strtok`. Usualmente, o valor da variável `PATH` é da forma

```
path1;path2;path3; ... ;pathn
```

ou seja, são várias *substrings* separadas pela *string* ";", formada por um único caractere. Assim, a primeira chamada da função `strtok` sobre um valor como esse, tendo ";" como segundo argumento, extrai e retorna a primeira *substring*, `path1`. Chamadas subsequentes à função `strtok` devem conter `NULL` como primeiro argumento, mantendo ";" como segundo argumento. Essas novas chamadas vão extraíndo, em seqüência, as *substrings* separadas por ";", isto é, vão retornando as *substrings* `path2`, `path3`, e assim por diante.

O comando `while` perfaz essa tarefa, armazenando cada *substring* extraída na variável `p` e imprimindo-a em seguida. Quando a extração resultar em `NULL`, terminamos de processar o valor da variável `PATH` e o comando `while` termina.