5. FUNÇÕES

Utilizando-se funções (subrotinas) consegue-se dividir grandes tarefas de computação em tarefas menores. Também é possível que programadores trabalhem em cima de funções já desenvolvidas, ao invés de partir do nada. É interessante comentar que em C e C++ todas as funções possuem valores de retorno, apesar de que estes valores podem ser "nada" (*void*).

A declaração de uma função fornece as informações necessárias para chamar a função: ela fornece o nome da função; mostra o número e tipo de valores que devem ser passados para a função como argumentos; e descreve o valor (se existe) que a função retorna. Uma declaração de função é também chamada de protótipo de função [3].

Sendo assim, um protótipo de função, resumidamente, é uma maneira de dizer ao compilador que a função existe e como ela deveria ser chamada. Com o protótipo de função sempre se usa um modelo quando se estiver declarando e definindo uma função. Quando a função é chamada, o compilador usa o modelo para assegurar que os argumentos apropriados são passados e que o valor de retorno é tratado corretamente. Se o programador errar quando chamar a função, o erro é detectado em tempo de compilação [1].

Alguns exemplos de protótipos de funções são [8]:

```
char* strcpy(chat* to, const char* from);
void exit(int);
double sqrt(double);
```

A definição de uma função fornece a mesma informação que a sua declaração, mas também fornece os comandos que são executados quando a função é chamada. Uma função pode ser declarada muitas vezes (em arquivos fonte e *header* diferentes, por exemplo), mas ela pode ser definida apenas uma vez. Declarações para a mesma função devem concordar umas com as outras e com a definição. Também é importante salientar que a definição ou declaração deve proceder o primeiro uso da função no arquivo fonte [3].

As definições de funções, basicamente, têm a forma:

<tipo_retorno> <nome_função> (<lista_declarações_argumentos>) { <corpo_função> }

onde, *<tipo_retorno>* e *<lista_declarações_argumentos>* podem ser *void*, e *<corpo_função>*, normalmente, é uma instrução composta. Um exemplo simples de uma definição de função completa é [2]:

```
int max (int, int, int);            // declaração
int max (int a, int b, int c) {     // definição
        int m = (a > b) ? a : b; // "?" é um operador lógico, ":" é equivalente a "senão"
        return (m > c) ? m : c;
}
```

Uma função é chamada escrevendo-se o seu nome seguido por parênteses que "cercam" quaisquer valores que devem ser passados para a função. Os parênteses devem ser preservados mesmo que "sejam vazios", pois eles servem como operadores responsáveis pela chamada da função. Um comando que chama uma função deve ser seguido de ponto e vírgula [3].

Quando uma função é chamada, o armazenamento é determinado a partir dos seus argumentos formais e cada argumento formal é inicializado pelo argumento atual correspondente. A semântica da passagem de argumentos é idêntica à semântica de inicialização. Em particular, o tipo de um argumento atual é verificado de acordo com o tipo do argumento formal correspondente, e todas conversões de tipo padrão e definida pelo usuário são realizadas [8]. Logo abaixo é exemplificada a chamada da função definida no exemplo anterior:

```
int m = max (10, 14, 11);
```

Um valor deve ser retornado de uma função que não é declarada como *void*, da mesma maneira que um valor não pode ser retornado de uma função *void*. O valor de retorno é especificado pelo comando *return*. Por exemplo:

```
int f1() { }                    //erro: nenhum valor é retornado
void f2() { }                   // correto
int f3() { return 1; }          // correto
void f4() { return 1; }         // erro: valor é retornado em uma função void
int f5() { return; }            //erro: falta valor de retorno
void f6() { return;  }          // correto
```

É interessante comentar que pode haver mais de um comando *return* em uma função, como ilustra o próximo exemplo, que apresenta uma função recursiva (função que chama a si mesma).

```
int fat (int n) {
        if (n>1) return n*fac2(n-1);
        return 1;
}
```

Freqüentemente é uma boa idéia dar a diferentes funções nomes diferentes, mas quando algumas funções conceitualmente desempenham a mesma tarefa em objetos de tipos diferentes, pode ser mais conveniente colocar o mesmo nome nestas funções. O uso do mesmo nome para operações em tipos diferentes é chamado de *overloading* ou sobrecarga. Esta técnica é usada para operações básicas no C++. Por exemplo, há somente um nome para adição, +, mas pode-se usar valores inteiros, *float*, etc. Esta idéia é facilmente estendida para funções definidas pelo programador [8]:

```
void print (int);               // imprime um inteiro
void print (const char *);      // imprime um string
```

Torna-se importante comentar que existem vários tipos de funções em C++, descritos a seguir:

- **Função Membro**: uma função declarada como um membro de uma classe (sem "*friend*") é denominada função membro, e sua chamada é feita usando-se a sintaxe de membro de classe. É considerada como estando dentro do escopo de sua classe [2].
- **Função *Friend***: é uma função que não é um membro da classe, mas tem permissão para usar os nomes de membros privados e protegidos da classe. O nome de uma *friend* não está no escopo da classe, e a *friend* não é chamada com os operadores de acesso a membro, a menos que ela seja um membro de outra classe [2]. Este tipo de função é detalhadamente descrito na seção 11.
- **Função *Inline***: fazer uma chamada de função consome tempo, pois o processamento deve alcançar uma região de definição do código, o compilador salva o endereço na pilha, variáveis locais devem ser alocadas, etc. Para eliminar este desperdício de tempo utiliza-se o especificador *inline* na definição da função. A palavra-chave *inline* instrui o compilador a substituir a chamada de uma função por seu código no instante em que ela é invocada. Assim o código é inserido na linha de chamada, evitando *overheads* de chamadas a funções e, potencialmente, aumentando o tamanho do código. Em pequenas funções, pode-se ganhar tempo de processamento ou mesmo espaço (desde que não utilizadas intensamente). Quando a função é muito complicada ou longa, o compilador pode ignorar a especificação *inline* e tomá-la como uma função comum. Este tipo de função contém qualquer instrução e executa qualquer ação da mesma maneira que outras funções, tendo como diferença o fato de não possui um *prototype* e ser normalmente declarada antes da *main* [9]. Para criar uma função deste tipo é só digitar *inline* na frente da declaração normal, como mostra o exemplo a seguir:

```
# include <iostream>
# define PI 3.14159265
inline float Cone (float radius, float height) { return ( (PI * (radius * radius) * height) / 30 ); }
main () {
        float radius, height, volume;
        cout << "Digite raio de base do cone: ";
        cin >> radius;
        cout << "Digite altura do cone: ";
        cin >> height;
        volume = Cone (radius, height);
        cout << "Volume do Cone = " << volume;
}
```

- **Função Virtual**: como já visto na seção 2 (polimorfismo), a palavra reservada *virtual* permite que classes derivadas forneçam versões diferentes de uma função da superclasse. Uma vez que uma função é declarada como virtual, ela pode ser redefinida em qualquer subclasse.
- **Função Membro *Static***: membro de dados ou função de uma classe pode ser declarado como *static* na declaração de classe. Neste caso, só existe uma cópia de um membro de dados estático, compartilhada por todos os objetos da classe em um programa. As funções membro *static* não podem referenciar diretamente dados não estáticos e funções não estáticas declaradas em suas classes, pois não possui um ponteiro *this*. Uma função *static* pode ser chamada usando objeto de sua classe ou usando o nome da classe e o operador de determinação de escopo (::) [2, 10].