Estruturas de Dados - T.332

Capítulo 6

Introdução: C++

Classes em C++

- Classes em C++ são extensões do sistema de tipos:
  - Cada tipo ***class*** representa um conjunto único de objetos em uma estrutura, de operações (métodos) e conversões para manipular, criar e destruir estes objetos.
- Especialidades:
  - Em C++ estruturas (**`struct`**) e uniões (**`union`**) são tipos especiais de classes com certas restrições.
  - Em C++ Classes não são objetos, sendo meramente construções sintáticas.
  - Embora uma classe precise ter necessariamente uma superclasse, podem haver classes derivadas de outra.
  - Herança múltipla é permitida.

Sintaxe de declaração de uma Classe em C++:

```
chave-de-classe <informação-de-tipo> nome-de-classe
   <:lista-base>
   { <lista-de-membros> };
```

- **chave-de-classe** é uma dentre **class, struct, union**
- **nome-de-classe** é um identificador único dentro de um escopo. O nome da classe.
- A **{ <lista-de-membros> }** é a declaração dos membros propriamente ditos, que definem a estrutura e o comportamento da classe que está sendo definida:
  - Membros podem ser membros-de-objeto ou funções-membro.
  - Membro-de-objeto é a nomenclatura em C++ para designar variáveis de instância e de classe.
  - Função-membro é o nome dado em C++ para os métodos, tanto de classe como de instância deste objeto.

Sintaxe de declaração de uma Classe em C++:

- Lista de membros-de-classe é uma seqüência de :
  - declaração de dados de qualquer tipo (inclusive de outras classes, enumerações, campos de bits, vetores, ponteiros, etc),
  - declaração e/ou definição de funções.
  - Todos com especificadores opcionais de classe de armazenamento (somente **static**).

```
class Exemplo
   {
       int x;
       int multiplicado ( int y ) {
              return ( x * y );
              }
};
Exemplo umEXemplo, *ptrExemplo, muitosExemplos[10];
```

Visibilidade de declaração

- Membros após a chave `public:` são visíveis ao mundo exterior.
  - Variáveis de instância podem ser diretamente manipuladas.
  - Métodos podem ser chamados por ~métodos de outros objetos.
- O resto é privado, isto é:
  - Só pode ser acessado de dentro do próprio objeto.

```
class Exemplo
  {
      int x;
  public:
      int multiplicado ( int y ) {
             return ( x * y );
             }
};
Exemplo x;
int z;
z = x.multiplicado( 4 );
```


Acesso a membros

- Você pode acessar membros de instâncias de uma classe (objetos-de-classe) usando a mesma sintaxe que você já usou para estruturas:

```
class Exemplo
  {
   public:
      int x;
      int multiplicado ( int y )
             { return ( x * y ); }
};
Exemplo x, *xPtr;
int z;
x.x = 3;
xPtr = &x;                          // xPtr aponta para x...
z = x.multiplicado( 4 );            // z é 12
z = xPtr->multiplicado( 7 );        // z é 21
```

Comparação com Smalltalk

- Em Smalltalk você *manda* uma mensagem a um objeto, o resultado é atribuído a uma variável:

```
variável := objeto mensagem: param1 chave: param2.
```

- Em C++ o "objeto" é referenciado pela variável que o contém ou por um ponteiro para ele, sendo usada a sintaxe de acesso a campos do "C" para referenciar o membro:

```
variável = objeto.funcMembro(param1,param2);
variável = ptrProObjeto->funcMembro(param1,param2);
```

- Em Smalltalk, variáveis de instância são sempre privadas, você precisa de métodos de acesso.
- Em C++, membros podem ser públicos e acessados através da sintaxe de acesso a campos do "C".

INE - UFSC - Disciplina Estruturas de Dados - Prof. Dr. Aldo von Wangenheim Transparência 7

Procedimento organizacional em C++

- Concepção:
  - OOP: Concebemos um Objeto através da descrição da estrutura, i.é, de quais dados deve "conter" e do comportamento que queremos que tenha (Top-Down)
    - Ex.: Uma Fila FIFO deve poder receber e devolver elementos, deve conter os elementos em uma ordem, etc.
- Programação:
  - Organizamos um programa em **Módulos**, cada qual contendo um ou mais Objetos relacionados
  - Um módulo contém a declaração de classes (.h) em um arquivo e em outro as funções relativas a um objeto (.C).
  - Todas as funções-membro devem estar declaradas no arquivo .h (dentro da classe !!!)

INE - UFSC - Disciplina Estruturas de Dados - Prof. Dr. Aldo von Wangenheim Transparência 8

Organização Geral: C++

■ Definição da Classe: Exemplo.h

```
class Exemplo {
public:
  int x;
  int multiplicado(int y);
};
```

■ Comportamento do Objeto: Exemplo.C

```
#include "Exemplo.h"
int Exemplo::multiplicado(int y)
  {
    return ( x * y );
  }
```

O arquivo .h, como **define um tipo**, deve obrigatóriamente ser incluído em todos os arquivos que fizerem uso de instâncias deste objeto, da mesma forma que fazíamos nos TADs.

O operador **::** é chamado de **modificador de acesso de escopo**. Ele diz que, embora `multiplicado` esteja sendo definida fora da definição de classe de *Exemplo*, é ainda assim uma função-membro (método) de *Exemplo*, complementando a referência feita no arquivo.h

```
tipo-retorno nome-classe::método(params)
  { definição-função }
```


Razões para a divisão em .h e .C

- O arquivo .h eu incluo em todos os outros módulos que devem "conhecer" o tipo (classe) *Exemplo*.
- O arquivo .C eu não incluo em lugar nenhum.
  - As chamadas a funções-membro (métodos) de instâncias de *Exemplo* que forem acontecendo, serão resolvidas pelo linkeditor de C++ em tempo final de compilação.
  - Basta para isso que eu inclua o arquivo *Exemplo.C* na lista dos arquivos a serem compilados.
- Eu poderia declarar as funções-membro diretamente dentro da declaração de *Exemplo*.
  Isto limitaria porém o escopo onde eu o poderia usar, pois não poderia incluir *Exemplo* completo em mais de um módulo, pois implicaria em declará-lo mais de uma vez.

Autoreferência

- Todo objeto deve ser capaz, em um método, de se referenciar a si mesmo, de maneira que o método possa manipular o objeto como um todo.
  - Tradicionalmente um objeto que recebe uma mensagem se referencia a si mesmo através da palavra reservada ***self*** (Exemplo: Smalltalk)
- Em C++ existe para esse fim a palavra-chave ***this***.
  - Métodos não-estáticos operam sobre o objeto com o qual foram chamados.
  - Se x é um objeto da classe X, xptr um ponteiro para x e f() um método de X, as chamadas de função x.f() e xptr->f() operam em x.
  - ***this*** é um ponteiro para x passado como argumento escondido para f(), caso ela seja não-estática.

Autoreferência

```
class Exemplo
   {
    public:
        int x;
        int multiplicado ( int y )
                { return ( x * y ); }
};
```

é o mesmo que:

```
class Exemplo
   {
    public:
        int x;
        int multiplicado ( int y )
                { return ( (this->x) * y ); }
};
```

Variáveis de Instância e de Classe

- Toda variável de um objeto em C++ é a princípio uma variável de instância e todo método é a princípio um método de instância.
- Como C++ não encara classes como objetos, não existem variáveis e métodos de classe diretamente. Eles podem ser simulados através da classe de armazenamento estática:
  - Uma variável (membro-de-objeto) declarado como `static` existe uma vez só, independente de a classe já ter instâncias ou não.
  - Um método (função-membro) declarado com a classe de armazenamento `static` é linkado externamente e age como um método de classe.

Membros Estáticos

- Membros estáticos simulam variáveis de classe:
  - Com membros não-estáticos (declarados "normalmente"), existe uma cópia deste membro para cada instância da classe. Cada instância, ao ser criada, aloca memória para este membro.
  - Para membros estáticos, só existe uma cópia e ela pode ser acessada sem que seja feita referência a alguma instância particular da classe, mesmo até se ainda não foi criada nehuma instância dessa classe.
  - O uso principal é o de representar dados comuns a todas as instâncias de uma classe.
  - O acesso é realizado através do operador de escopo **::**. Membros estáticos de uma classe globais podem ser inicializados.

Membros Estáticos

```
class Exemplo
   {
    public:
        static int x;
        int multiplicado ( int y )
                { return ( x * y ); }
};

int Exemplo::x = 3;         // Inicialização, mesmo não
                            // existindo nenhum Exemplo
Exemplo y;
int z;
z = y.multiplicado( 4 );    // z é 12
```


Funções-Membro Estáticas

- Funções-Membro (Métodos) estáticas simulam métodos de classe:
  - Para funções-membro estáticas, não é passada uma referência à instância com a qual foi chamada, ela não faz referência a alguma instância particular da classe.
  - Pode ser usada mesmo se ainda não foi criada nehuma instância dessa classe.
  - O acesso pode ser realizado através do operador de escopo **::**.
  - Pode-se usar também uma variável representando uma instância ou um ponteiro, essas expressões não serão porém avaliadas.

Funções-Membro Estáticas

```
class Exemplo
   {
    public:
       int x;
       static int multiplicado ( int y )
              { return ( 3 * y ); }
                                    // se x é um membro não-
                                    // estático, não posso
                                    // referenciá-lo.
};

int z;
z = Exemplo::multiplicado( 4 );   // z é 12
```


New e *Delete*

- O operador new seguido do nome de uma classe devolve um ponteiro para uma nova instância dessa classe, chamando o construtor dela:

```
Exemplo* ExPtr;                // ponteiro p/Exemplo
ExPtr = new Exemplo;           // Nova instancia de Exemplo
```

- Memória é liberada através de delete :

```
delete ExPtr;
```

Construtores e Destrutores

- Há várias funções-membro que indicam como uma instância é criada e como é destruída**:**
  - **Construtores**: Semelhante ao malloc(), alocam memória e também realizam outras coisas que queiramos, como dar valores iniciais a variáveis de instância.
    Exemplo: Uma fila FIFO deve inicializar os seus ponteiros *inicio* e *fim* para NULL.
  - **Destrutores**: Semelhante ao free(), liberam memória e podem fazer qualquer outra coisa, como liberar a memória de outros objetos que tenham sido alocados como parte do objeto que está sendo destruído.
    Exemplo: uma lista que está sendo destruída deve, antes de mais nada, destruir todos os seus nodos, um a um.

Construtores e Destrutores

- Aspectos gerais**:**
  - Eles não podem ter um valor de retorno, nem mesmo void e você não pode obter seus endereços.
  - Não podem ser herdados, embora uma classe derivada possa chamar os construtores e destrutores de sua superclasse.
  - Construtores podem ter argumentos-default ou usar listas de inicialização de membros.
  - Construtores não podem ser chamados da mesma forma como funções normais são chamadas. Destrutores sim, desde **::**.
  - Construtores são chamados implicitamente sempre que uma nova instância de uma classe é criada.
  - Destrutores são chamados implicitamente quando um objeto é destruído.
  - Se você não os declara, o compilador cria versões-default.

Construtores

- Distingüem-se por possuírem o mesmo nome da classe a que pertencem.
- Quando um objeto da classe é criado ou está sendo copiado, o seu construtor é chamado implicitamente.
- Construtores para variáveis globais são chamados antes da função main() ser invocada.
- Exemplo:

```
class OutroExemplo {
public:
     int i;
     X () { i = 0; }; // construtor para X, inicializa i
};
OutroExemplo x;       // o construtor é chamado
main () { .....
}
```


Construtores

- Sobrecarga de funções:
  - Em C++ você pode definir várias funções com o mesmo nome e parâmetros diferentes. Dependendo da forma da lista de parâmetros, uma ou outra é chamada:

```
class Exemplo2 {
       int    parteInteira;
       float  parteReal;
  public:
       X (int i) { parteInteira = i; }; // inicializa i
       X (float j) {parteReal = j; };   // inicializa j
  };
Exemplo2 x(2);          // o 1° construtor é chamado
Exemplo2 y(2.0);        // o 2° construtor é chamado
main () {
}
```

Destrutores

- O destrutor de uma classe é chamado para liberar (free()) membros (variáveis de instância) de uma classe antes do objeto ser destruído.
- O destrutor é uma função cujo nome é o nome da classe precedido de um til (~).
- Um destrutor não pode aceitar qualquer tipo de parâmetro nem possui qualquer tipo de valor de retorno, nem mesmo void.
- Se não for declarado para uma classe, o compilador gera um.

```
class Exemplo2 {
      int    parteInteira;
      float  parteReal;
  public:
      ~X () {}; // destrutor para instâncias de Exemplo2
  };
```


Destrutores: mais detalhes

- Se uma variável que representa um objeto cai fora de seu escopo o destrutor para aquele objeto é chamado
  - Exemplo: uma variável local de uma função contém um objeto e a execução da função acabou.
- Se um objeto foi criado explicitamente com ***new*** e só existe um ponteiro apontando para ele, ele não é destruído automaticamente quando não há mais ponteiros apontando para ele.
  - Como ocorreria em Smalltalk, Java ou LISP e outras linguagens com coletor de lixo.
  - Nesse caso ele deve ser destruído explicitamente com ***delete***, que chama o destrutor.

Curiosidades:

```
char Buffer[sizeof(Exemplo)];  //Vetor de bytes do tamanho
                               //de uma instância de Exemplo
Exemplo* Ptr;
Ptr = new(&Buffer) Exemplo;    // New pode ter um endereço
                               // como parâmetro...
....
....

Ptr->Exemplo::~Exemplo();      // Chamada explícita ao
                               // destrutor
```


Exercício (hoje)

- Reimplemente a sua fila FIFO utilizando os recursos de classes de C++.
- Crie, além das funções de manipulação de fila vistas em aula, um destrutor para uma fila que permita que qualquer fila seja destruída a qualquer momento. Lembre-se que se a fila possuir nodos, eles devem ser destruídos um a um, antes de a fila ser destruída.
- Inclua um ponto de menu: destruir fila.
- Lembre-se que a função criaFila() deixa de existir e passa a ser substituída pelo construtor dessa fila.