Aula 02:

Introdução ao C++ Parte II

ECOP13A - Programação Orientada a Objetos
Prof. André Bernardi
andrebernardi@unifei.edu.br

Universidade Federal de Itajubá

Como sempre, levaremos a programação muito a sério nesse semestre!

Mas o que faz um programa melhor que os outros?

1. Um bom programa é fácil tanto para escrever quanto para ler;
2. É também elegante, utiliza um padrão consistente de indentação e escrita;
3. É menos propenso a erros, devido ao conhecimento da linguagem e organização do programador;
4. O programa é mais sustentável e fácil de manter;
5. Executa rápido...
6. ... Consumindo a menor quantidade possível de recursos.

E os comentários?

```
// comentário de uma linha
/*
   comentário multilinhas
/*
```

Comentários continuam importantes e possuem a mesma sintaxe do C. Lembre-se de que o compilador não interpreta o conteúdo de seus comentários.

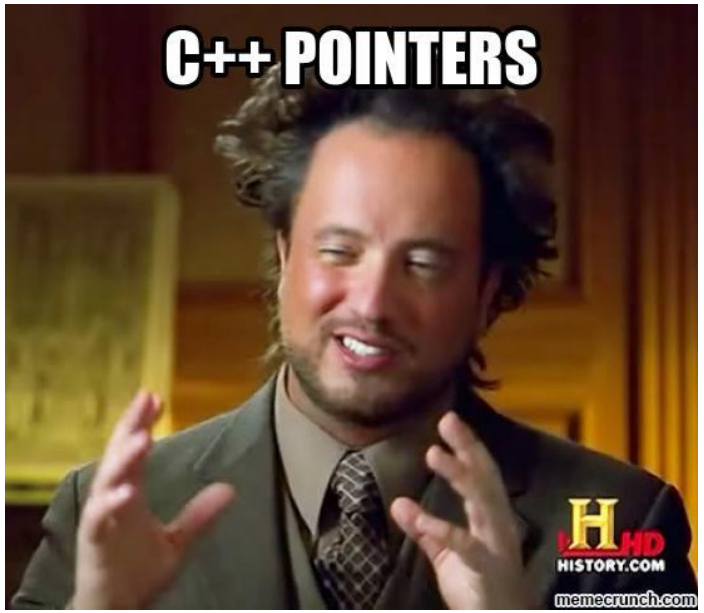
Veja algumas dicas:

- Mantenha-os concisos;
- Não escreva comentários que relatam o que pode ser obviamente visto no código. (ex.: a = b + c; // a recebe soma de b e c)
- Relate a sua intenção nos comentários, esclarecendo o propósito de uma função ou classe, o algoritmo utilizado para solução de um problema, um comentário explicando alguma parte do código que seja mais complicada.

Lembre-se também de que comentário ruim pode ser pior do que um não-comentário.

Obviamente, sempre podemos nos referir a um objeto pelo seu nome, como variáveis comuns. No entanto, em C++, todos os objetos possuem um endereço específico na memória, e podemos acessá-lo também se soubermos esse endereço e seu tipo.

As funcionalidades da linguagem utilizadas para manter e utilizar endereços são os **ponteiros** e as **referências** (essa é nova).

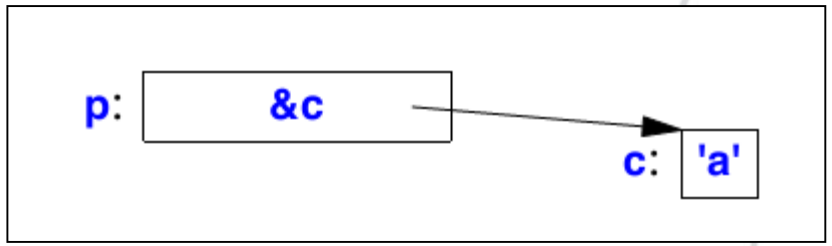


Ponteiros

Para um tipo T qualquer, T* é o tipo “ponteiro para T”.
Isso quer dizer que uma variável do tipo T* pode guardar o endereço de (&) um objeto do tipo T. Por exemplo:

```
char c = ‘a’;
char* p = &c;  // p recebe o endereço de c.        & é o operador “endereço de”
```

Graficamente, temos:



Como já sabemos, a operação fundamental de um ponteiro é a de-referência, que é o que acontece quando queremos acessar o objeto para o qual o ponteiro aponta. Essa operação também pode ser chamada de “***indireção***”, e seu operador é o *.

```
char c2 = *p;      // o caractere c2 recebe o valor do objeto para o qual p aponta (‘a’)
```

Nós tentaremos fazer com que um ponteiro sempre aponte para um determinado objeto de interesse, de maneira que a de-referência seja válida.

Quando um ponteiro não tem um objeto específico para o qual apontar, precisamos representar a noção de "**nenhum objeto disponível**" (por exemplo, para o fim de uma lista encadeada), e damos ao ponteiro o valor nullptr (o ponteiro "nulo"). Em códigos mais antigos, utilizava-se 0 ou NULL, mas nullptr elimina qualquer confusão entre inteiros (0, NULL) e ponteiros (nullptr).

Há apenas um nullptr, compartilhado por todos os tipos de ponteiros. Veja:

```
double* pd = nullptr;
Link<Record>* lst = nullptr;
int c = nullptr;                    // ERRO: nullptr não é um inteiro.
```

É sempre importante que verifiquemos se o ponteiro está apontando para um objeto válido antes de tentar acessar seu conteúdo.

```cpp
#include <iostream>
#include <cstdlib> // veja que bibliotecas de C também estão aqui

using namespace std;

#define SIZE 100

int count_x(int* v, int key)
{
    if(v == nullptr) return 0; // verifica se ponteiro não está vazio
    int count = 0;

    for(int i = 0; i < SIZE; i++)
    {
        if(v[i] == key) ++count;
    }
    return count;
}

int main()
{
   int *vect = nullptr;

    // cria vetor na free store
    vect = new int[SIZE];

    for(int i = 0; i < SIZE; i++)
        vect[i] = rand()%10;

    cout << "Ocorrencias de 1: " << count_x(vect, 1) << "\n";
    cout << "Ocorrencias de 5: " << count_x(vect, 5) << "\n";
    cout << "Ocorrencias de 10: " << count_x(vect, 10) << "\n";

    // deleta objetos criados com new
    delete[] vect;
}
```

```
Ocorrencias de 1: 14
Ocorrencias de 5: 7
Ocorrencias de 10: 0
```

Repare que podemos, em C++, alocar um espaço de memória no que chamamos de "*free store*", assim como fazíamos em C através de malloc (e free).

Em C++ utilizamos o comando new para alocar memória e delete para liberá-la.

Lembre-se de que um objeto alocado na *free store* é de responsabilidade do programador e não será desalocado por simples regras de escopo ou *garbage collection*.

Referência

Em uma declaração, o uso do prefixo **&** significa “referência para”.
Uma referência é bastante parecida com um ponteiro, exceto que:

- É possível acessar uma referência com a mesma sintaxe de um objeto comum, ou seja, não é necessário utilizar o operador de desreferência * para acessar seu valor.
- Uma referência **sempre se refere ao objeto ao qual ela foi inicializada**.
- Não existe **“referência nula”**, e podemos **sempre** assumir que uma referência está se referindo a um objeto existente.

Uma referência funciona como um apelido para um objeto, é implementada para manter em si o endereço de um objeto e não oferece nenhuma perda de desempenho com relação ao ponteiro. Sua maior utilização é na passagem de parâmetros e valores de retorno em funções e operadores sobrecarregados.

Exemplo:

```
int var;
int& r {var};       // r e var agora se referem ao mesmo objeto na memória
int& v;             // erro de compilação: falta inicialização
```

Considerações sobre Funções
– Passagem de Parâmetros

Em C++, podemos passar parâmetros para uma função por **valor** ou **referência**, como fazíamos em C.

A passagem de valores por referência possibilita que um parâmetro tenha seu valor modificado dentro da função, além de gerar um ganho de desempenho, pois evita que sejam feitas cópias de valores na memória. Quanto maior o objeto copiado, maior o *overhead*.

No entanto, utilizávamos o conceito de ponteiros (*) para passagem por referência, agora podemos utilizar o próprio conceito de **referência (&)**. Veja:

```cpp
#include <iostream>

using namespace std;

// passagem simples, por valor (copia de valores)
void duplicate_value(int a, int b, int c)
{
    a = a*2;
    b = b*2;
    c = c*2;
}

// passagem dos endereços das variáveis (sintaxe complexa)
void duplicate_pointer(int* a, int* b, int* c)
{
    *a = *a*2;
    *b = *b*2;
    *c = *c*2;
}

// passagem dos endereços das variáveis (sintaxe simples)
void duplicate_ref(int& a, int& b, int& c)
{
    a = a*2;
    b = b*2;
    c = c*2;
}

int main()
{
    int a{2}, b{10}, c{50};
    duplicate_value(a, b, c);
    cout << "(valor) a, b, c = " << a << ", " << b << ", " << c << "\n";
    duplicate_pointer(&a, &b, &c);
    cout << "(pointer) a, b, c = " << a << ", " << b << ", " << c << "\n";
    duplicate_ref(a, b, c); // chamada simples
    cout << "(ref) a, b, c = " << a << ", " << b << ", " << c << "\n";
}
```

```
(valor) a, b, c = 2, 10, 50
(pointer) a, b, c = 4, 20, 100
(ref) a, b, c = 8, 40, 200
```

Passagem por valor não modifica os valores das variáveis, pois elas nada mais são do que **cópias** das variáveis passadas como parâmetro.

O desempenho da passagem por referência é melhor, pois não é necessário realizar a cópia de valores. No entanto, é preciso tomar cuidado caso não se deseje modificar os seus valores iniciais.

Como resolvemos isso?

Considerações sobre Funções - const

A palavra reservada **const** do C++ sempre funciona como uma promessa (e uma garantia) de que seu valor inicial nunca será modificado no programa. Devemos sempre preferir este tipo de declaração para incluir constantes em nossos programas. Veja um exemplo:

```
const double pi = 3.14156;
```

Caso tentemos modificar seu valor durante a execução, teremos um erro do tipo:

```
funcoes.cpp:32:5: error: assignment of read-only variable 'pi'
  pi = 1;
     ^
```

Desta maneira, caso queiramos passar variáveis para uma função por referência para melhorar o desempenho, mas queremos ter certeza de que não serão modificadas lá dentro, podemos utilizar a palavra const na declaração da função:

```
// passagem por referência: não modifica valores
void write_on_screen(const string& name, const string& last)
{
    cout << "Name: " << name << " " << last << "\n";
}
```

Considerações sobre Funções: *inlining*

Como sabemos, chamadas de funções podem causar um certo impacto de desempenho, devido à cópia de valores, empilhamento de parâmetros na stack, etc.

Muitas vezes, portanto, pode ser mais eficiente simplesmente inserir o código da função onde ela é chamada, ao invés de executar o processo formal de chamada de função.

Quando temos funções bastante simples (apenas a instrução *return*, por exemplo), podemos requisitar ao compilador que o seu código seja substituído no lugar da chamada. Fazemos isso através da palavra reservada inline. Assim, não perdemos em organização do programa, mas ganhamos em performance. Veja um exemplo:

```cpp
#include <iostream>

using namespace std;

inline double square(double d)
{
    return d*d;
}

int main()
{
    cout << "Entre com um numero real: ";

    double real;
    cin >> real;

    cout << real << "^2 = " << square(real) << "\n";
    cout << real << "^2 = " << real*real << "\n"; // equivalente
}
```

```
Entre com um numero real: 12.5
12.5^2 = 156.25
12.5^2 = 156.25
```

Alguns compiladores são capazes de realizar algumas operações de **inline** ainda mais impressionantes:

```cpp
inline int fatorial(int n)
{
    return (n<2 ? 1: n*fatorial(n-1));
}
```

Não custa tentar.

Considerações sobre Funções: Valores *default*

Em C++, as funções podem ter valores padrão para seus argumentos, de maneira que, caso o usuário não os passe, ela prossiga com este valor. Veja o exemplo:

```
double potencia(double base, double expoente = 2)
{
    double ans = 1;
    for(int i = 0; i < expoente; i++)
        ans *= base;
    return ans;
}
```

A função “potência” pode ser chamada de duas maneiras:

`potencia(real, 3)` Para real = 10, resposta = 1000

`potencia(real)` Para real = 10, resposta = 100 (base = 2)

Somente os últimos parâmetros podem ter valores default (ou **todos**)

Operador de qualificação de escopo unário ::

É possível declarar variáveis locais e globais com o mesmo nome. Isso faz com que a variável global seja "oculta" pela variável local no escopo local.

C++ fornece o operador de resolução de escopo unário (::) para acessar uma **variável global** quando uma variável **local** com o mesmo nome está no escopo.

O operador de resolução de escopo unário não pode ser usado para acessar uma variável local com o mesmo nome em um bloco externo.

Uma variável global pode ser acessada diretamente sem o operador de resolução de escopo unário se o nome da variável global não for o mesmo que o de uma variável local no escopo.

Operador de qualificação de escopo unário ::

O programa a seguir (figura 15.9 do livro do Deitel) demonstra o operador de resolução de escopo unário com variáveis globais e locais com o mesmo nome (linhas 6 e 10, respectivamente).

Para enfatizar que as versões local e global da variável **`number`** são distintas, o programa declara uma variável do tipo **`int`** e a outra **`double`**.

Operador de qualificação de escopo ::

```
1  // Fig. 15.9: fig15_09.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  int number = 7;                  // global variable named number
7
8  int main()
9  {
10    double number = 10.5;    // local variable named number
11
12    // display values of local and global variables
13    cout << "Local double value of number = " << number
14       << "\nGlobal int value of number = " << ::number << endl;
15 }
```

Fig. 15.9 | Using the unary scope resolution operator.

```
Local double value of number = 10.5
Global int value of number = 7
```

Sobrecarga de funções

C++ permite que várias funções com o **mesmo nome** sejam definidas, desde que essas funções tenham diferentes conjuntos de parâmetros (pelo menos no que diz respeito aos tipos de parâmetros ou ao número de parâmetros ou à ordem dos tipos de parâmetros, Assinaturas diferentes).

Essa capacidade é chamada de **sobrecarga de função**.

Quando uma função sobrecarregada é chamada, o compilador C++ seleciona a função adequada examinando o número, os tipos e a ordem dos argumentos na chamada.

Sobrecarga de funções

A sobrecarga de função é comumente usada para criar várias funções com o mesmo nome que realizam tarefas semelhantes, mas em dados de tipos diferentes.

Por exemplo, muitas funções na biblioteca matemática são sobrecarregadas para diferentes tipos de dados numéricos.

Sobrecarga de funções

```
1   // Fig. 15.10: fig15_10.cpp
2   // Overloaded square functions.
3   #include <iostream>
4   using namespace std;
5
6   // function square for int values
7   int square( int x )
8   {
9       cout << "square of integer " << x << " is ";
10      return x * x;
11  }
12
13  // function square for double values
14  double square( double y )
15  {
16      cout << "square of double " << y << " is ";
17      return y * y;
18  }
19
```

```
20  int main()
21  {
22      cout << square( 7 ); // calls int version
23      cout << endl;
24      cout << square( 7.5 ); // calls double version
25      cout << endl;
26  }
```

Fig. 15.10 | Overloaded square functions.

Sobrecarga de funções

As funções sobrecarregadas são diferenciadas por suas **assinaturas** — uma combinação do nome de uma função e seus tipos de parâmetros em ordem (mas não seu tipo de retorno).

O compilador codifica cada identificador de função com um número e os tipos de seus parâmetros (às vezes chamados de *name mangling* ou *name decoration*) para habilitar a vinculação segura de tipo.

Isso garante que a função sobrecarregada adequada seja chamada e que os tipos de argumento estejam em conformidade com os tipos de parâmetro.

Sobrecarga de funções

A Figura 15.11 mostra os *mangled names* de funções produzidos em linguagem assembly pelo GNU C++.

Cada *mangled names* (exceto main) começa com dois sublinhados (__) seguidos pela letra Z, um número e o nome da função.

O número que segue Z especifica quantos caracteres há no nome da função. Por exemplo, a função ***square*** tem 6 caracteres em seu nome, então seu nome mutilado é prefixado com __Z6.

Sobrecarga de funções

```
1  // Fig. 15.11: fig15_11.cpp
2  // Name mangling to enable type-safe linkage.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  }
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 }
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 }
22
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27    return 0;
28 }
29
30 int main()
31 {
32    return 0; // indicates successful termination
33 }
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
_main
```

Fig. 15.11 | Name mangling to enable type-safe linkage.

Sobrecarga de funções

O compilador usa apenas as **listas de parâmetros** para distinguir entre funções de mesmo nome.

Funções sobrecarregadas não precisam ter o mesmo número de parâmetros.

Os programadores devem ter cuidado ao sobrecarregar funções com parâmetros padrão, porque isso pode causar ambiguidade.

Criar funções sobrecarregadas com listas de parâmetros idênticas e tipos de retorno diferentes é um erro de compilação.

Tipos definidos pelo usuário

Chamamos os tipos que podem ser construídos a partir dos tipos fundamentais de **"tipos nativos"**: int, float, char, double, etc.

Estes tipos são muito úteis, mas também são de baixíssimo nível: foram projetados para refletir as capacidades do hardware do computador diretamente e de maneira muito eficiente. Muitas vezes, a representação de um problema somente através destes tipos se torna muito complicada ou antinatural.

O C++ nos apresenta com um conjunto de mecanismos de abstração (classes, especialmente), que permitem aos programadores expandir o universo de trabalho, através da criação de tipos novos, com suas próprias características, comportamentos e operações.

Ex: Carro, Registro, Coordenada, Mapa, Personagem, Circuito, etc.

A utilização de "tipos definidos pelo usuário" leva a uma programação de mais alto nível, onde os objetos se referem mais diretamente a entidades do mundo real.

O primeiro passo nessa direção são as **structs**.

structs

As estruturas são tipos de dados construídos usando-se elementos de outros tipos (podem até incluir outras structs). Sua vantagem é que, enquanto arrays podem conter elementos de um único tipo, structs podem conter elementos de tipos variados, que compõem um novo tipo mais complexo em sua construção. Por isso, são chamados muitas vezes de “heterogêneas”. Veja o exemplo:

```
#include <iostream>

using namespace std;

struct Tempo {
    int hora;
    int minuto;
    int segundo;
};

int main()
{
    Tempo t {8, 10, 52};
    cout << "Hora: " << t.hora << ":" << t.minuto;
    cout << ":" << t.segundo << "\n";
}
```

Repare que o estilo é idêntico ao de C, mas agora podemos declarar uma variável do tipo Tempo sem a palavra *struct* e inicializar seus valores através do inicializador universal.

structs

Alguns pontos podem ser destacados:

- A palavra reservada **struct** inicia sua definição;
- O identificador Tempo dá nome à estrutura e passa a ser o nome do tipo criado;
- As variáveis declaradas no corpo da **struct** são chamadas de membros;
  - Podem ser de qualquer tipo de dados já criado.
- A definição da **struct** deve sempre terminar com **ponto-e-vírgula**.
  - O mesmo acontecerá com as classes, como veremos mais adiante.
- **Observação**: Uma **struct** não pode conter uma instância de si mesma, mas pode conter uma auto-referência, ou seja, um ponteiro para ela mesma. Veja um exemplo bastante conhecido de vocês:

```
struct TreeNode{
    int value;
    TreeNode *left; // Não são instâncias,
    TreeNode *right; // são referências!
};
```

structs

- Os membros de dados da estrutura podem ser acessados de duas maneiras, dependendo se a referência está sendo realizada a partir de um objeto (operador ponto) ou de um ponteiro para objeto (operador seta):

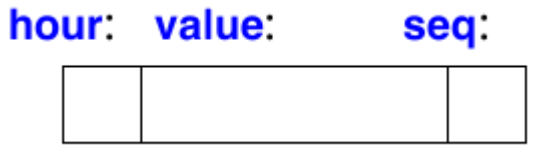
```
Tempo temp {0,30,5};
Tempo *temptr = &temp;

temp.hora = 10;
cout << "Membro hora: " << temptr->hora << "\n";
```

- As structs podem ser passadas como parâmetros para funções normalmente, assim como podem servir de retorno. Veja um exemplo:

```
// Imprime hora sem modificar valor
void print_time(const Tempo& t)
{
    cout << t.hora << ":" << t.minuto;
}
```

structs

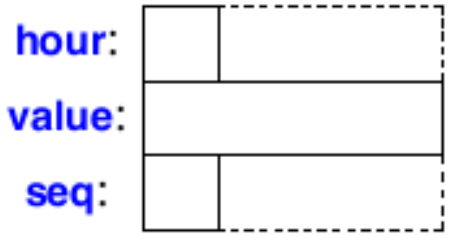
Com relação à memória, você provavelmente pensaria que uma estrutura do tipo *Readout* seria alocada da seguinte maneira:



Não é verdade

```
struct Readout {
    char hour;      // [0:23]
    int value;
    char seq;       // sequence mark ['a':'z']
};
```

Os membros são alocados na memória na ordem da declaração, no entanto, o tamanho da struct pode não ser exatamente igual à soma dos tamanhos de seus membros. Inteiros são, com bastante frequência, alocados em limites de palavras de bytes, o que deixaria “buracos” na estrutura. Da seguinte maneira:

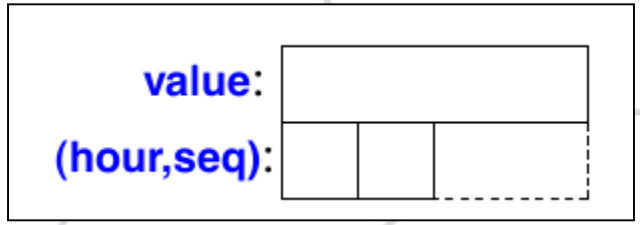


Assim, o tamanho da struct passa a ser 12, ao invés do que pensamos, que seria 6.

structs

Isso é parcialmente resolvido alocando-se objetos por ordem decrescente de tamanho. Veja:

```
struct Readout {
        int value;
        char hour;      // [0:23]
        char seq;       // sequence mark ['a':'z']
};
```



Repare que a estrutura continua tendo tamanho 8, mas otimizamos sua disposição.

A dica é: **Sempre coloque os membros em ordem decrescente de tamanho**. Normalmente, isso gera uma economia de espaço de memória.

structs

Alguns problemas relacionados à criação de dados com structs em estilo C.

- Repare como os dados estão desprotegidos. A inicialização pode ser realizada de qualquer maneira, o programador pode acessar cada um dos membros de dados isoladamente, modificar seus valores. **Tudo é público**.
  - Não há uma **interface clara**, que defina os comportamentos do tipo Tempo rigidamente, a partir de regras.
- Comparações e operações como impressão não podem ser realizados como unidade. É necessário realizar a operação para cada um dos membros também.
- Veremos que existe uma série de mecanismos de abstração (como herança, polimorfismo) que não podem ser realizados a partir de **structs** e podem ser fundamentais para nosso sistema.

C++ resolve muitos desses problemas através das... (*wait for it...*)

Introdução à Tecnologia de Objetos

Orientação a objetos, “uma maneira natural de pensar sobre o mundo e escrever programas de computador”.

Conceitos de programação orientada a objetos.

- Atributos e comportamentos
- Design orientado a objetos e herança
- Encapsulamento e ocultação de informações
- Polimorfismo

Conceitos de Programação Orientada a Objetos

Para onde quer que você olhe no mundo real, você vê objetos — pessoas, animais, plantas, carros, aviões, edifícios, computadores e assim por diante.

Os humanos pensam em termos de objetos.

Telefones, casas, semáforos, fornos de micro-ondas e bebedouros são apenas mais alguns objetos que vemos ao nosso redor todos os dias.

Atributos e comportamentos

Os objetos têm coisas em comum:

Todos eles têm **atributos** (por exemplo, tamanho, forma, cor e peso) e todos eles exibem **comportamentos** (por exemplo, uma bola rola, quica, infla e desinfla; um bebê chora, dorme, engatinha, anda e pisca; um carro acelera, freia e vira; uma toalha absorve água).

Os humanos aprendem sobre objetos existentes estudando seus **atributos** e observando seus **comportamentos**.

Objetos diferentes podem ter atributos semelhantes e podem exibir comportamentos semelhantes. Comparações podem ser feitas, por exemplo, entre bebês e adultos e entre humanos e chimpanzés.

Design orientado a objetos e herança

O design orientado a objetos (OOD) modela o software em termos semelhantes aos que as pessoas usam para descrever objetos do mundo real.

Ele aproveita os relacionamentos de classe, onde objetos de uma determinada classe, como uma classe de veículos, têm as mesmas características — carros, caminhões, pequenos carrinhos vermelhos e patins têm muito em comum.

OOD tira proveito de relacionamentos de **herança**, onde **novas classes** de objetos são derivadas absorvendo características de classes existentes e adicionando características únicas próprias.

Design orientado a objetos e herança

Um objeto da classe "conversível" certamente tem as características da classe mais geral "automóvel", mas mais especificamente, o teto sobe e desce.

O design orientado a objetos fornece uma maneira natural e intuitiva de visualizar o processo de design de software — ou seja, modelar objetos por seus atributos, comportamentos e inter-relacionamentos, assim como descrevemos objetos do mundo real.

Design orientado a objetos e herança

OOD também modela a **comunicação** entre objetos. Assim como as pessoas enviam **mensagens** umas às outras (por exemplo, um sargento ordena que um soldado fique em posição de sentido), os objetos também se comunicam por meio de mensagens.

Um objeto de conta bancária pode receber uma mensagem para diminuir seu saldo em uma determinada quantia porque o cliente sacou essa quantia de dinheiro.

Encapsulamento e ocultação de informações

OOD encapsula atributos e operações (comportamentos) em objetos — os atributos e operações de um objeto estão intimamente ligados.

Os objetos têm a propriedade de ocultar informações.

Isso significa que os objetos podem saber como se comunicar uns com os outros por meio de interfaces bem definidas, mas normalmente não têm **permissão** para saber como outros objetos são implementados — os detalhes da implementação ficam ocultos dentro dos próprios objetos.

Encapsulamento e ocultação de informações

Podemos dirigir um carro de forma eficaz, por exemplo, sem saber os detalhes de como motores, transmissões, freios e demais sistemas funcionam internamente — desde que saibamos como usar o pedal do acelerador, o pedal do freio, o volante e assim por diante.

A ocultação de informações, é crucial para uma boa engenharia de software.

Programação Orientada a Objetos

A programação na linguagem **C++** é chamada de programação orientada a objetos (**POO**), e permite que você implemente um design orientado a objetos como um sistema de software funcional.

Linguagens como C, por outro lado, são procedurais, então a programação tende a ser orientada a ações.

Em C, a unidade de programação é a **função**.

Em C++, a unidade de programação é a "**classe**" da qual os objetos são eventualmente instanciados (um termo POO para "criado").

As classes C++ contêm **funções** que implementam operações e **dados** que implementam atributos.

Programação Orientada a Objetos

Os programadores C concentram-se em escrever funções.

Os programadores agrupam ações que realizam alguma tarefa comum em funções e agrupam funções para formar programas.

Os dados são certamente importantes em C, mas a visão é que os dados existem principalmente em suporte às ações que as funções realizam.

Os verbos em uma especificação de sistema ajudam a determinar o conjunto de funções que trabalharão juntas para implementar o sistema.

Classes,
Membros de Dados e Funções de Membro

Os programadores C++ concentram-se em criar seus próprios tipos definidos pelo usuário, chamados **classes**.

Cada classe contém **dados**, bem como o conjunto de **funções** que manipulam esses dados e fornecem serviços aos clientes (ou seja, outras classes ou funções que usam a classe).

Os componentes de dados de uma classe são chamados **membros de dados (atributos)**. Por exemplo, uma classe de conta bancária pode incluir um número de conta e um saldo.

Classes,
Membros de Dados e Funções de Membro

Os componentes de função de uma classe são chamados **funções de membro** (usualmente chamados **métodos** em outras linguagens de programação orientadas a objetos).

Por exemplo, uma classe de conta bancária pode incluir funções de membro para fazer um depósito (aumentando o saldo), fazer um saque (diminuindo o saldo) e consultar qual é o saldo atual.

Você usa tipos embutidos (e outros tipos definidos pelo usuário) como os "blocos de construção" para construir novos tipos definidos pelo usuário (classes).

Classes,
Membros de Dados e Funções de Membro

Os **substantivos** em uma especificação de sistema ajudam o programador C++ a determinar o conjunto de **classes** a partir das quais os objetos são criados que trabalham juntos para implementar o sistema.

As classes são para os objetos o que as plantas são para as casas — uma classe é um "projeto" para construir um objeto da classe.

Assim como podemos construir muitas casas a partir de uma planta, podemos instanciar (criar) muitos objetos de uma classe. Você não pode cozinhar refeições na cozinha de uma planta; você pode cozinhar refeições na cozinha de uma casa. Você não pode dormir no quarto de uma planta; você pode dormir no quarto de uma casa.

Classes,
Membros de Dados e Funções de Membro

As classes podem ter relacionamentos com outras classes. Em um design orientado a objetos de um banco, a classe "caixa de banco" se relaciona com outras classes, como a classe "cliente", a classe "gaveta de dinheiro", a classe "cofre" e assim por diante.

Esses relacionamentos são chamados de associações.

Empacotar software como classes torna possível que futuros sistemas de software reutilizem as classes.

De fato, com a tecnologia de objetos, você pode construir grande parte do novo software que precisará combinando classes existentes, assim como os fabricantes de automóveis combinam peças intercambiáveis.

Classes,
Membros de Dados e Funções de Membro

Cada nova classe que você cria pode se tornar um valioso ativo de software que você e outros podem **reutilizar** para acelerar e melhorar a qualidade dos esforços futuros de desenvolvimento de software

A reutilização de classes existentes ao construir novas classes e programas economiza tempo, dinheiro e esforço.

A reutilização também ajuda a construir sistemas mais confiáveis e eficazes, porque as classes existentes geralmente passaram por testes extensivos, depuração e ajuste de desempenho.


CLASSES!

Classes

As classes (class) em C++ são uma evolução natural da noção de struct de C. Elas habilitam o programador a modelar objetos que possuem atributos (membros de dados) e comportamentos, operações ou métodos (representados como funções membro).

Em geral, queremos sempre manter a implementação inacessível ao usuário de nosso tipo, provendo-lhe apenas com uma **interface pública** contendo as operações que podem ser realizadas no nosso tipo. O padrão é que todos os dados sejam privados (**private**) e operações sejam públicas (**public**). Veja o exemplo:

```
class Tempo {

    // membros públicos (podem ser acessados de fora da classe)
    // proveem mecanismos para realização de qualquer operação
    // interessante para nosso tipo
    public:
        Tempo();
        void setTempo(int, int, int);
        void imprime();

    // membros privados (não podem ser acessados fora da classe)
    private:
        int hora, minuto, segundo;
};
```

Definição da classe (declaração de membros)

```
// Construtor da classe Tempo
Tempo::Tempo() {hora = minuto = segundo = 0; }

// Veja o operador de escopo ::, definindo o método setTempo de Tempo.
void Tempo::setTempo(int h, int m, int s = 0)
{
    hora = h;
    minuto = m;
    segundo = s;
}

void Tempo::imprime()
{
    cout << hora << ":" << minuto << ":" << segundo;
}

int main()
{
    Tempo t;

    // acesso a membros através de operador . ou ->
    t.setTempo(10,20);
    t.imprime();
}
```

Implementação e Uso da classe

Exercício

No exemplo anterior, acrescente uma nova função membro (método) para cálculo da quantidade de segundos passados da meia noite. Ela deve retornar um valor do tipo inteiro.

A seguir, imprima a quantidade de segundos de um objeto do tipo Tempo na função main().

Referências

- https://cplusplus.com/reference/

- C++ How to program – Deitel e Deitel.

- Notas de aula da disciplina Programação Orientada a Objetos, Prof. André Bernardi, Prof. João Paulo Reus Rodrigues Leite.