

Apoio de C++

Seguem algumas dicas de práticas e de sintaxe em C++. Algumas são orientações para programadores vindos de outras linguagens; outras são dicas para aqueles que quiserem aperfeiçoar suas habilidades com C++.

1. Abertura de arquivos

Um problema muito comum na disciplina é a abertura de arquivos. Há várias maneiras de fazê-lo, mas muitas delas não são portáteis entre OSs. Seguem exemplos dos jeitos apropriados de abrir arquivos.

(DE = diretório de execução do programa)

```
fopen("IDJ.txt");    // abre arquivo que está no DE
fopen("../UnB.txt"); // abre arquivo na pasta acima do DE
fopen("textos/CIC.txt"); // CERTO - arquivo em pasta contida no DE
```

Outras formas podem não funcionar em compiladores ou OSs diferentes. Exemplos incluem uso de ‘\’ em vez de ‘/’ (Windows-only), ou uso de caminhos absolutos (que só existem no seu PC).

Ressalto duas coisas: Primeiro, apesar de eu usar a API do C (fopen, da stdio) como exemplo, essa regra também vale para fstream (C++) e, a menos que a documentação da mesma afirme o contrário, para a abertura de arquivos de qualquer formato, em qualquer biblioteca externa, incluindo a SDL.

Segunda coisa, muito importante: Usuários de IDE devem atentar, nas configurações do projeto, para “Launch Directory” ou “Working Directory”. Essa configuração altera o diretório de execução do programa, o que é interessante para organizar o projeto. Porém, se você não está ciente de para qual pasta está setado, poderá encontrar erros de “arquivo não encontrado” quando buscar recursos, mesmo o código e os arquivos estando corretos.

2. Herança

Não há uma palavra reservada para herança em C++. Para declarar que uma classe é filha de outra, usa-se a seguinte sintaxe:

```
class Parent {
    ...
}
class Child : public Parent {
    ...
}
```

Diferentes modificadores de acesso após os dois-pontos tem diferentes efeitos na herança. Especificamente, afetam a visibilidade de membros da classe mãe dentro da classe filha. Na maioria dos casos, usamos public. Ver detalhes em: <http://www.cplusplus.com/doc/tutorial/inheritance/>.

3. Chamar Construtor da Classe Mãe/De Membros

Não existe também a palavra reservada *super*. Se tiver uma classe que precisa passar argumentos para o construtor da sua superclasse, na definição do construtor, use:

```
Filha::Filha() : Mae("../IDJ.txt") {  
    . . .  
}
```

Em alguns casos, você pode também querer chamar o construtor de um dos membros da classe ao invés de atribuí-lo (isso é obrigatório se não existir o construtor sem argumentos na classe do membro). Adicione todos os construtores que quiser, separados por vírgula.

```
Filha::Filha() : Mae("../IDJ.txt"),  
                bg("img/bg.jpg"),  
                cameraX(0),  
                cameraY(0) {  
  
}
```

4. Inicialização de membros estáticos

Por causa de algumas regras de definição de variáveis em C++, para que uma classe possa ser declarada com um membro static, é necessário declará-la também fora da classe. A inicialização, se necessária, também é feita nesse momento. Em geral, essas declarações vêm no início do arquivo .cpp.

```
#include "MyClass.h"  
  
int MyClass::myStaticVariable = 0;  
  
MyClass::MyClass() {  
    ...  
}
```

5. Destrutor membro

C++ permite que o destrutor seja chamado da seguinte forma:

```
Object* o = new Object();  
o->~Object();
```

Não faça isso. Ao chamar o destrutor como membro, executa-se o clean-up contido em `~Object()`, mas a área de memória ocupada pelo objeto não é liberada.

Essa feature da linguagem existe para escrever programas que usam memory pools, isto é, alocam uma área grande de memória inicialmente e administram alocações posteriores de objetos internamente. Não sendo esse o caso, use sempre a palavra reservada `delete` para evitar leaks.

```
delete o;    // objeto deletado e liberado!
```

6. Métodos virtuais e virtuais puros

A palavra reservada `virtual` em C++ é a forma da linguagem de lidar com polimorfismo. Quer dizer que aquele método pode ser substituído inteiramente (e não só ocultado) por classes filhas. Declarando o método como `virtual`, não é necessário usar `override` como em C#.*

Uma dúvida comum de pessoas que vem de outras linguagens é como fazer métodos não simplesmente virtuais, mas abstratos, visto que a linguagem não possui a palavra reservada `abstract`. Métodos abstratos são chamados em C++ de “virtuais puros”.

Para declarar métodos virtuais puros, se usa a palavra `virtual`, da mesma maneira, mas ao invés de prover um corpo para a função, “atribui-se” zero a ela. Uma tentativa de instanciar uma classe com métodos assim resulta em erro de compilação, e uma classe filha só poderá ser instanciada se fornecer uma implementação própria para todos os métodos virtuais puros da mãe.

```
class MyBaseClass {  
    // Foo é virtual não-pura. A classe mãe define um  
    // comportamento, que pode ser alterado pelas filhas  
    virtual void Foo() { printf("oi"); }  
  
    // Bar é virtual pura. Sua existência impede a instanciação  
    // de MyBaseClass, e classes filhas precisam implementá-las  
    virtual void Bar() = 0; // Virtual pura. Deve ser definida  
                           // na classe filha  
}
```

* O modificador `override` sequer existia em C++ até a revisão C++11. Se quiser saber como usá-lo, ele está descrito na seção 12.

7. Destrutores Virtuais

O destrutor de uma classe também pode ser declarado como virtual, e isso é indicado caso a classe vá servir de classe base para outras.

Para melhor ilustrar a situação, criei um código exemplo:

<http://pastebin.com/r9DFc6d3>

O problema surge quando se tem um objeto de uma classe derivada, o qual é referenciado por um ponteiro para um objeto da classe mãe (faremos isso algumas vezes nos trabalhos da disciplina).

Você pode usar delete nesse ponteiro. A pergunta é: Qual construtor é chamado? O da classe mãe, ou o da derivada? Trata-se de comportamento indefinido. No meu teste, quando o destrutor da mãe não era virtual, o destrutor da filha não foi chamado. Se ele contivesse algum clean-up específico, teríamos leaks.

É claro que a situação de ter um ponteiro assim nem sempre vai acontecer, mas por segurança, sempre que usar cadeias de herança, garanta que o destrutor da classe mãe é virtual.

8. `std::vector` vs `std::list`

Aprendemos, em ED, que quando queremos acesso direto a elementos, usamos arrays, mas quando temos estruturas em que precisamos remover e adicionar elementos a todo momento, listas são muito mais eficientes. Em C++, isso não é sempre verdade.

Quando iteramos sobre um vector, chegar até o ponto de remoção envolve apenas percorrer uma região contígua de memória. Iterar sobre uma lista envolve dereferenciar ponteiros e mais ponteiros para regiões aleatórias de memória. Acessos aleatórios à memória são caros, e por causa disso, a performance de programas que usam lists grandes acaba sendo muito pior que a de programas que usam vectors.

Essa perda de performance fica ainda mais evidente quando entram em questão as otimizações do compilador: Como vector é contíguo, podemos copiá-lo inteiro para a memória cache, diminuindo ainda mais as penalidades de acesso à memória. Juntando isso tudo fica fácil ver por que vector se tornou quase a estrutura de dados padrão de código C++.

No entanto, vale enfatizar que nem sempre list é inapropriada. Vectors tem o problema de alocar mais memória do que o necessário (seu tamanho dobra sempre que é excedido). Isso resulta em desperdício de memória, especialmente quando o

tamanho dos elementos é grande. Para esses casos, `std::list` pode ser melhor.

Vale lembrar também que a implementação da `std::list` não é ruim, ao contrário do que alguns dizem. É uma lista encadeada comum. Mas por mais que queiramos abstrair o funcionamento do programa o máximo possível, garantir que ele está otimizado envolve escolher nossas estruturas de dados pensando também em como elas se comportam em relação ao hardware.

9. `using namespace std;`

Em praticamente todo tutorial de C++, ensina-se a usar...

```
using namespace std;
```

...no começo do código. Isso é uma prática muito ruim, tornada pior porque algumas pessoas acreditam que não fazer isso se trata apenas de uma preferência.

Ao usar o namespace inteiro, você passa a ter todos os nomes do namespace padrão no seu escopo global (são muitos!). Não há impacto algum de performance no seu programa; como namespaces são resolvidos em tempo de compilação, o binário gerado é o mesmo. No máximo, quem fica lenta é sua IDE.

Acontece que a importação de namespaces completos é a fonte de alguns erros particularmente trabalhosos de se encontrar a causa. O problema surge quando você tenta usar um nome de variável ou função presente no namespace usado, ou ainda pior, se uma biblioteca tenta fazê-lo.

Deixo-lhes as seguintes histórias verídicas:

<http://stackoverflow.com/questions/2712076/how-to-use-an-iterator>
<http://stackoverflow.com/questions/13402789/confusion-about-pointers-and-references-in-c>

Esses casos devem convencê-los de que importar o namespace inteiro é uma má idéia.

Mas se você realmente detesta digitar `std::cout` e `std::string` o tempo inteiro, não tema! Há uma opção inofensiva. A diretiva `using` permite que você selecione um membro específico do namespace para importar, like so:

```
using std::cout;  
using std::string;  
using std::endl;  
...
```

Fazer isso pode ocupar algum espaço no código, dependendo de onde e quantos membros forem importados. Daí, sim, escolher entre isso ou usar sempre `std::` é questão de preferência, e existem argumentos contra e a favor.

10. Modificador `const` para métodos

Uma coisa incomum de se ver ensinada em tutoriais de C++, mas que é um bom hábito, é que se pode colocar o modificador `const` em declarações de funções:

```
class MyClass {
    int GetSize() const;
    void SetSize(int s);
}
```

O propósito desse modificador nesta posição é indicar que aquele método não altera nenhuma variável da classe. Um dos benefícios disso é que você poderá chamar o tal método em referências constantes da classe:

```
void Foo (const MyClass& mc) {
    mc.GetSize();           // Aceito! GetSize é somente leitura!
    mc.SetSize(2);          // Erro de compilação!
}
```

Algumas funções dos trabalhos podem usar esse modificador, como `Sprite::GetWidth`. Tente encontrar outras :P

11. Referências constantes

Um pequeno truque para evitar a cópia desnecessária de objetos em argumentos de funções é passar uma referência ao invés de um valor.

```
void Reopen(string file); // Copia o objeto string
void Reopen(string& file); // Passa uma referência
```

A referência traz a desvantagem, no entanto, de permitir alterações na string original, que normalmente é indesejado. Podemos usar `const`:

```
void Reopen(const string& file); // Referência p/ constante
```

Mas nesse caso, não teríamos como chamar métodos que não tenham o modificador `const`. A string da biblioteca padrão, no caso, tem o modificador nos seus métodos de leitura, mas em classes próprias, o programador precisa ter o cuidado de inserir o modificador. Você já sabe como fazê-lo :P

12. Modificadores `override` e `final`

C++11 acrescentou os modificadores `override` e `final` à linguagem. Seu uso nunca é obrigatório, e não se tratam de palavras reservadas. Esses nomes ainda estão livres, se quiser usá-los como variáveis ou função.

Diferente de outras linguagens, ao invés de usar essas palavras junto aos tipos da classe, usamos na mesma posição do `const` descrito acima. No caso de `final`, podemos usá-lo ao lado de declarações de classe também.

Note que os modificadores passam despercebidos porque não são obrigatórios. `final` evita que uma classe tenha filhas, ou que um método sofra `override`. Mas se você não vai fazer ou não se importa com isso, não precisa fazer essa marcação.

```
class MyClass final {  
    ...  
}  
class MyDerivedClass : public MyClass {  
    ...           // Erro de compilação: Derivando classe 'final'  
}
```

Da mesma forma, `override` existe para explicitar ao compilador que a intenção é substituir um método da classe mãe, assim, caso haja algum problema (ex: assinatura errada), um erro de compilação é gerado.

Ressalto que isso é um detalhe meio obscuro da linguagem e que ninguém vai cobrar que vocês saibam. É só para os mais animados/curiosos. Para estes, segue um exemplo comentado que fiz para demonstrar os modificadores.

<http://pastebin.com/1LPv6xuR>

13. Funtores e Funções Lambda

Se você programou alguma coisa mais avançada em C, deve saber o que é um callback. Um ponteiro pra uma função, que você pode mandar pra lá e pra cá e chamar quando der vontade. Bom, enquanto callbacks do C ainda são permitidos em C++, OO tem a sua própria maneira de lidar com eles, os funtores.

Um functor é uma classe, ou uma instância de uma classe, que tem o `operator()` definido. Isso permite que o objeto seja tratado como uma função, like so:

<http://pastebin.com/k0wbEJzA>

Funtores também são interessantes por poderem guardar estado, apesar de isso raramente ser usado.

<http://pastebin.com/L6dW45PP>

Muitas classes da biblioteca padrão usam funtores para especificar comportamentos adicionais, como a `std::sort`, que precisa saber como comparar um par de elementos, e, o que provavelmente te trouxe aqui, a `std::shared_ptr`, que permite especificar como fazer o cleanup de um ponteiro.

Existe, no entanto, uma inconveniência muito grande de se usar funtores: Acabamos definindo uma classe nova toda vez que usamos um, e é raro serem reusáveis. Para resolver isso, o C++11 introduziu uma nova feature à linguagem, a função lambda.

```
auto myLambda = [lista de captura] (argumentos) {  
    corpo da função;  
};
```

Criar uma função lambda é criar um functor, como quisermos, onde quisermos, sem poluir o namespace (já que a classe da lambda criada é “anônima”) e mantendo código relacionado, junto.

A lista de captura é provavelmente o aspecto mais estranho das lambdas. Ela permite que você leve variáveis do escopo de onde a lambda foi criada, para dentro do escopo da lambda, seja fazendo uma cópia (valor) ou levando a própria variável (referência). Algumas possibilidades:

- `[]` - nenhuma variável
- `[&]` - todas as variáveis por referência
- `[=]` - todas as variáveis por valor
- `[var1, &var2]` - `var1` por valor, `var2` por referência

Com isso em mente, segue um exemplo das funções em funcionamento.

<http://pastebin.com/iG3Ny49d>