Programação Orientada por Objetos

Eng. Informática

Instituto Politécnico de Bragança

Escola Superior de Tecnologia e Gestão de Bragança

José Exposto
Paulo Matos
Reis Quarteu
Paulo Gouveia

Bibliografia

- "The C++ Programming Language", Bjarne Stroustrup, Addison-Wesley, 4th Edition, 2013
- C++ Programming: An Objet-Oriented Approach. B. Behrouz A. Forouzan and Richard F. Gilberg, McGraw-Hill Education, 2020
- C++: Guia Moderno de Programação. Henrique Loureiro, FCA – Editora de Informática, 2019
- Fundamentals of C++ Programming. Richard L. Halterman, School of Comp. South. Adv. University US, 2018

  https://www.freetechbooks.com/fundamentals-of-c-programming-t1062.html
- Modeling with UML: Language, Concepts, Methods. Bernhard Rumpe, Springer 2016
- Modelação de Dados em UML – uma abordagem por problemas. Borges, T. Dias e J. Cunha, FCA – Editora de Informática, 2015
- "Programação com Classes em C++ - 2ª Edição", Pedro Gerreiro, FCA Editores de Informática, 2003

Programa

- Definição de Programação Orientada por Objetos:
  - Motivação
  - Conceitos básicos
- Princípios da Programação Orientada por Objetos:
  - Encapsulamento
  - Herança
  - Polimorfismo
- Conceitos de Modelação Orientada por Objetos
  - Diagramas de classes em UML
  - Associações entre classes: simples, agregação e herança
  - Sobreposição e acréscimo de características, Classes Abstratas, Herança Múltipla
- Introdução à linguagem C++
  - Visual Studio
  - Declarações
  - Constantes
  - Tipos de dados
  - Expressões e operadores
  - Controle de fluxo
  - Funções
- Definição de classes em C++:
  - Atributos
  - Construtores. Categorias de construtores
  - Métodos
- Funcionalidades básicas do C++:
  - Arrays e apontadores de objetos. Autorreferência nas classes
  - Membros constantes
  - Sobrecarga de operadores (Overloading)
- Implementação de associações simples e agregação de classes
- Templates de funções e classes
- Bibliotecas standard do C++.
  - Classes string e set.
  - A classe Coleção
- Implementação de associações:
  - Associações e coleções
  - Coleções de cópia e referência
  - Associações 1-N

    - Associações N-N
    - Classes Associativas
- Referências.
    - Definição de referências
    - Passagem de parâmetros e retorno
- Implementação da herança e de hierarquias de classes
    - Acréscimo e substituição de métodos
    - Construtores e herança. Listas de inicialização
    - Tipos de proteção no acesso aos membros
    - Conversão ascendente (Upcast) e descendente (downcast)
    - Polimorfismo e funções virtuais
    - Classes abstratas e funções virtuais puras
- Gestão de memória dinâmica interna a uma classe:
    - Construtor de cópia
    - Destrutor
    - Operador de atribuição
- Agregação com apontadores
- Implementação de coleções híbridas
- Operadores de conversão. Membros estáticos
- Declarações friend
- ~~Herança múltipla~~
    - ~~Ocorrência múltipla da classe base~~
    - ~~Classes base virtuais~~
- Entrada e saída de dados e Manipulação de ficheiros

Avaliação

Época normal

Trabalho prático: 50%

Exame Escrito Final: 50% (mínimo 7.0 valores)

Época de recurso

Exame Escrito Final: 100%

Épocas Especiais

Exame Escrito 100%

Considerações

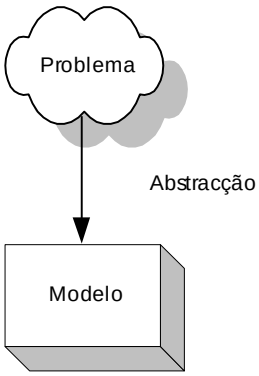
- Programar não se compadece com a simples leitura da bibliografia, é necessário praticar.
- As aulas são uma orientação para expandir o raciocínio lógico, matemático e algorítmico.
- Esta disciplina consome muito tempo aos alunos, grande parte do seu trabalho é feito a praticar nos laboratórios ou em casa.
- Não serão toleradas fraudes nos testes nem nos trabalhos práticos. Suspeitas de trabalhos feitos em cooperação ou contratação terão zero valores.

Programação Orientada por Objetos (POO)

Definição

Método de implementação de software, que tem como objetivo a resolução de **problemas** através da criação de um **modelo abstrato**, ao qual é associado um conjunto de **atributos** e um conjunto de **operações** que alteram esses atributos.

Os modelos são feitos à semelhança das entidades reais (pessoa, carro, caixa automática, calculadora, encomenda, etc.)



Vantagens na utilização da POO

- ✓ Aumento da produtividade;
- ✓ Facilidade de implementação e manutenção;
- ✓ Melhor extensibilidade dos programas;
- ✓ Reutilização do código;
- ✓ Correspondência com as entidades do mundo real.

Objetos reais	Objetos de software
• possuem atributos e comportamento	• São criados à imagem dos reais • Encapsulamento e Tipos de Dados Abstratos
• objetos maiores podem ser formados por outros mais pequenos	• Agregação
• objetos podem herdar caraterística	• Herança e hierarquia
• comunicam entre si	• Envio de mensagens
• são classificados pela sua natureza	• Definidos a partir de um modelo

A POO é uma técnica de controlo de complexidade na modelação de problemas e na sua implementação.

Conceitos básicos

Classe

Descrição genérica de uma entidade com características comuns. Uma classe é definida por:

- Nome
- Atributos
- Métodos

Instância ou objeto

Entidade concreta descrita por uma classe e membro dela. O objeto é uma instanciação de uma classe, no qual é possível executar apenas os métodos definidos na classe e em que os atributos são instanciados com valores concretos. Um objeto é identificado por:

- Identidade
- Estado (dados ou atributos)
- Comportamento (operações ou métodos)

Atributo

Característica de uma classe com especificação do tipo (valores aceitáveis). Na instância o atributo tem um valor concreto.

Método

Operação genérica definida numa classe. Na instância o método opera sobre os valores dos atributos desse objeto. Um método é semelhante a uma função, no entanto, ele está intrinsecamente ligado a um objeto. Um método é definido pelo número e tipo de parâmetros e tipo de retorno (resultado).

Mensagem

Mecanismo de interação com os objetos. As mensagens são enviadas aos objetos indicando o método que devem executar.

Exemplo

Classe: Carro

Atributos: marca: String, modelo: String, cor: String, matricula: String, velocidade: float

Métodos: acelerar(), travar()

Instância de Carro, c1: Ford, Focus, preto, 11-22-AA, 0

Instância de Carro, c2: Renault, Megane, Cinza, 55-55-ZZ, 10

c1.acelerar()

c2.travar();

Princípios da POO

A POO assenta num conjunto muito importante de conceitos que definem a sua funcionalidade:

- Encapsulamento;
- Herança;
- Polimorfismo.

Encapsulamento

Processo pelo qual se definem os objetos individualmente, permitindo a proteção dos dados e limitando o acesso às operações através de uma interface bem definida.

Os dados não devem ser modificados diretamente, mas sim, utilizar uma interface bem definida para os alterar e aceder consistentemente.

Por exemplo, num carro a velocidade não é alterada diretamente, mas sim através de um interface adequado (acelerador e travão).

Por outro lado, para observarmos a velocidade, não é necessário abrir o capot do carro. Utiliza-se uma interface própria, o velocímetro, para a observar.

Herança

Permite a delegação de atributos e operações de umas classes para outras. A Herança facilita a construção de novos objetos utilizando as características de outros já definidos, permite ainda alteração e o acréscimo do estado e o comportamento da classe herdada.

Subclasse (classe derivada) – Classe que obtém as características de outra classe.

Superclasse (classe base) – Classe donde são derivadas as características.

Herança simples – Quando a subclasse possui apenas uma superclasse.

Herança Múltipla – Quando a subclasse possui mais do que uma superclasse.

Polimorfismo

(que assume diferentes (poli) formas (morfismo))

Processo através do qual se torna possível enviar a mesma mensagem (invocar métodos) podendo resultar ações diferentes.

O polimorfismo pode ocorrer em duas formas:

- Sobrecarga (Overloading)
- Acoplamento dinâmico (Dinamic binding)

Na sobrecarga invocam-se métodos com o mesmo nome, mas distinguidos na quantidade e tipo dos parâmetros.

O acoplamento dinâmico manifesta-se na herança e na conversão ascendente, em que uma referência a um objeto pode responder de formas diferentes à invocação do mesmo métodos, consoante o objeto referenciado.

UML

A UML (Unified Modeling Language) é uma linguagem estandardizada para especificar, visualizar, construir e documentar componentes de software.

A UML utiliza fundamentalmente uma notação gráfica permitindo às equipas de projeto comunicar entre si, explorar o desenho do software e validar o desenho arquitetural.

Para mais informação sobre a linguagem:

http://www.agilemodeling.com/essays/umlDiagrams.htm

Para construção dos diagramas, pode ser utilizada a ferramenta *astah* Community

https://members.change-vision.com/members/files/astah_community

Diagramas da UML

Os diagramas da UML 2.0:

- Diagramas estruturais
  - **Diagrama de classes**
  - Diagrama de objetos
  - Diagrama de componentes
  - Diagrama de instalação
  - Diagrama de pacotes
  - Diagrama de estrutura composta
- Diagramas comportamentais
  - Diagrama de casos de uso
  - Diagrama de estados
  - Diagrama de atividades
  - Diagramas de interação
    - Diagrama de sequência
    - Diagrama comunicação
    - Diagrama temporal

Nota importante:

Embora se ilustre e se descreva de forma sucinta vários dos diagramas da UML, no âmbito desta unidade curricular apenas usaremos, na modelação dos problemas, os **Diagramas de Classes**.

Diagrama de casos de uso

Visualizam as relações entre atores (participantes na utilização do software) e os casos de utilização de componentes do software.

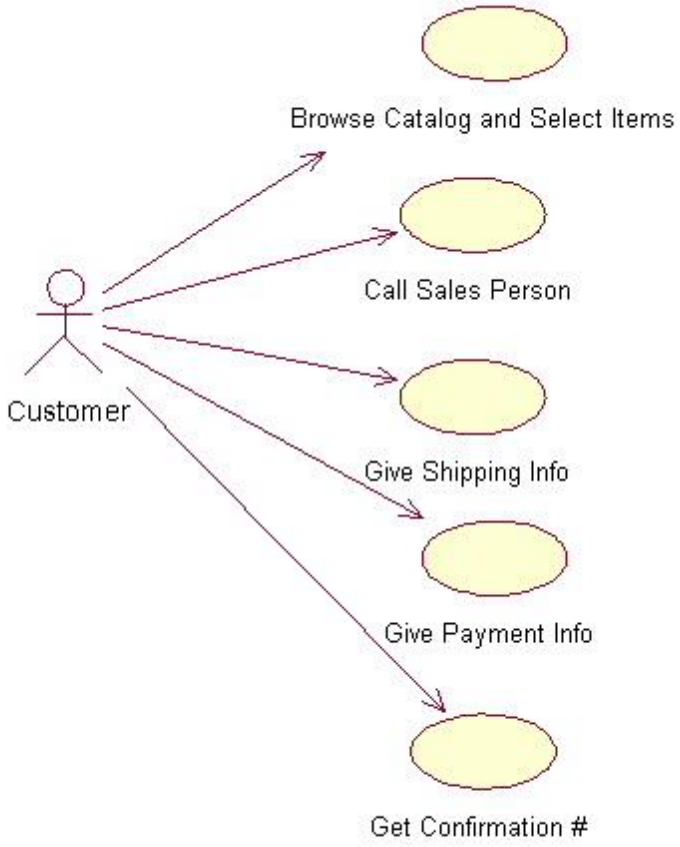


Diagrama de classes

Permitem modelar a estrutura das classes utilizadas para representar o modelo do software.

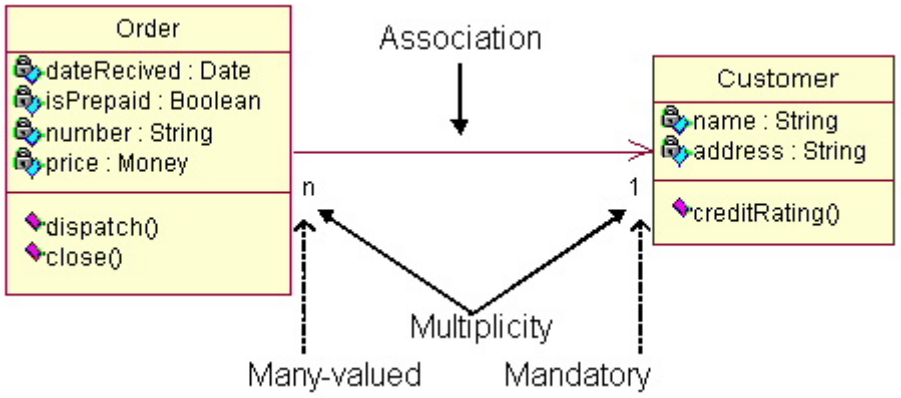


Diagrama de interação

Modelam o comportamento dos casos de utilização descrevendo o modo de interação entre grupos de objetos na concretização de tarefas.

Existem dois tipos de diagramas de interação:

Diagrama de sequência

Visualizam uma sequência temporal de interação entre objetos.

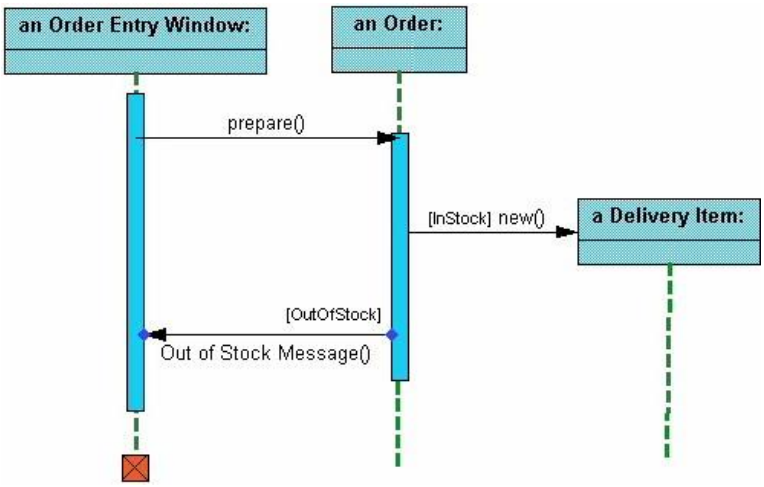


Diagrama de comunicação

Mostram o relacionamento entre objetos e a ordem de envio de mensagens para outros objetos.

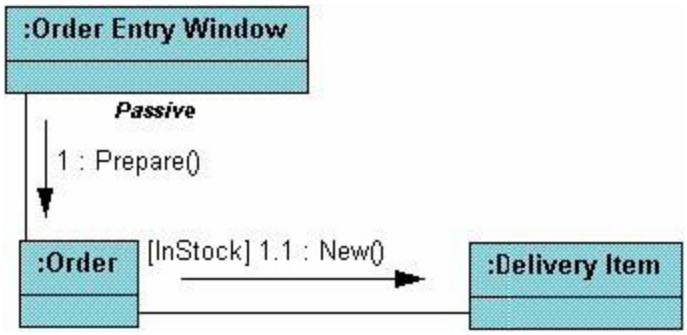


Diagrama de estado

Visualizam a sequência de estados que um objeto percorre durante a sua existência.

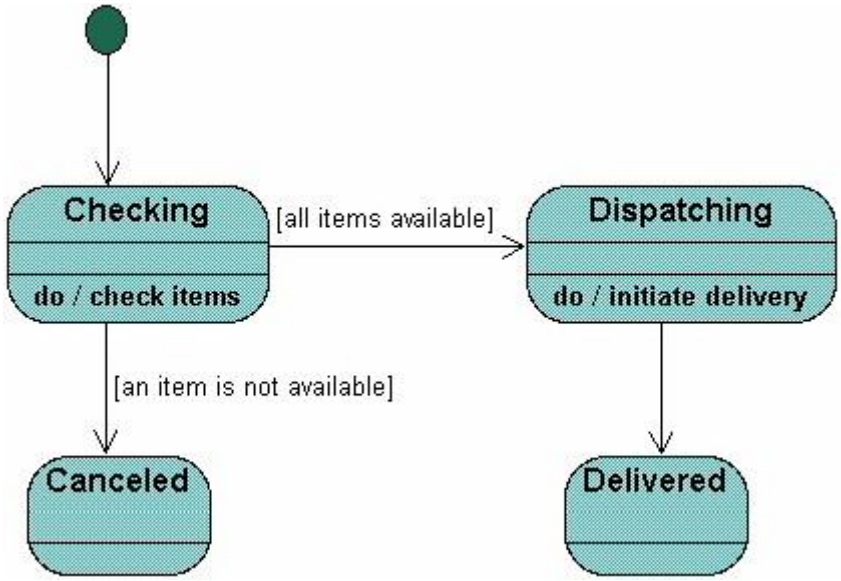


Diagrama de atividades

Descrevem o fluxo comportamental (atividades) do sistema.

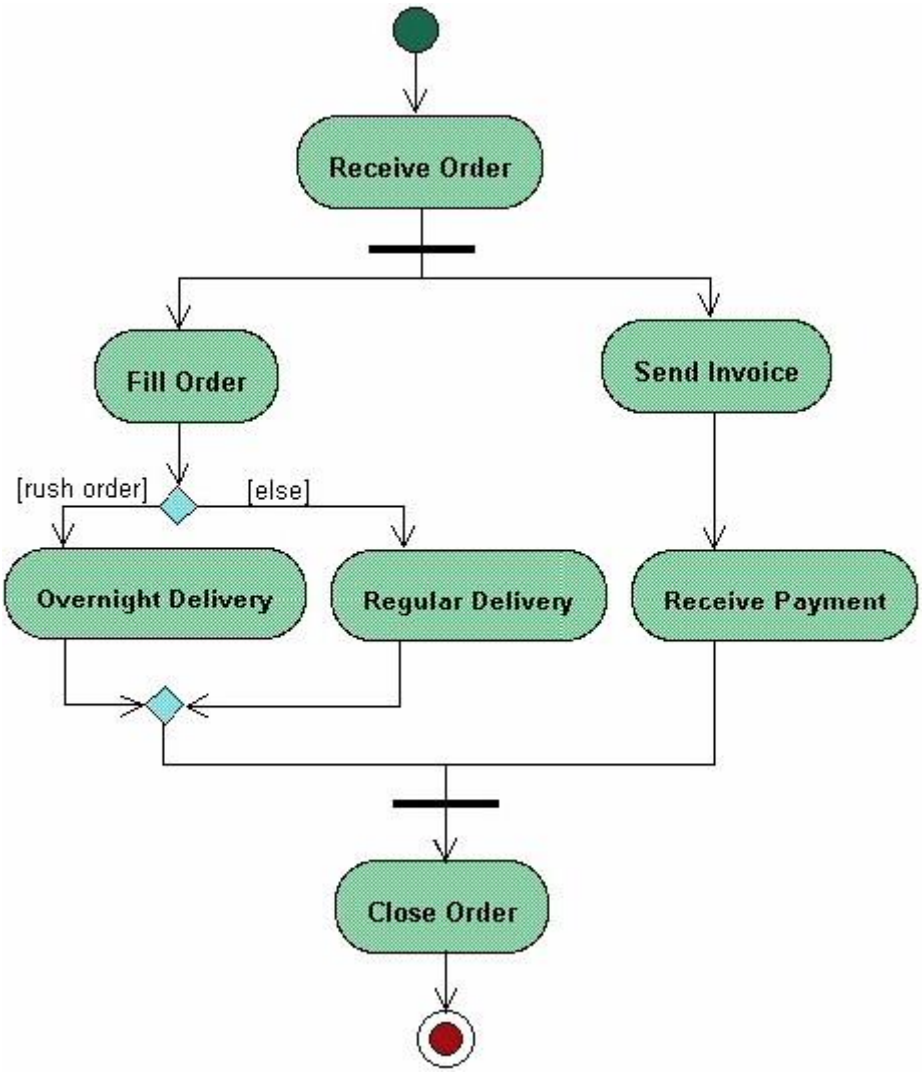


Diagrama físico

Existem dois tipos de diagramas físicos que podem ser combinados:

Diagrama de instalação (deployment)

Mostram o relacionamento entre o software e o hardware.

Diagrama de componentes

Mostram os componentes que fazem parte do sistema de software e o modo como interagem entre si.

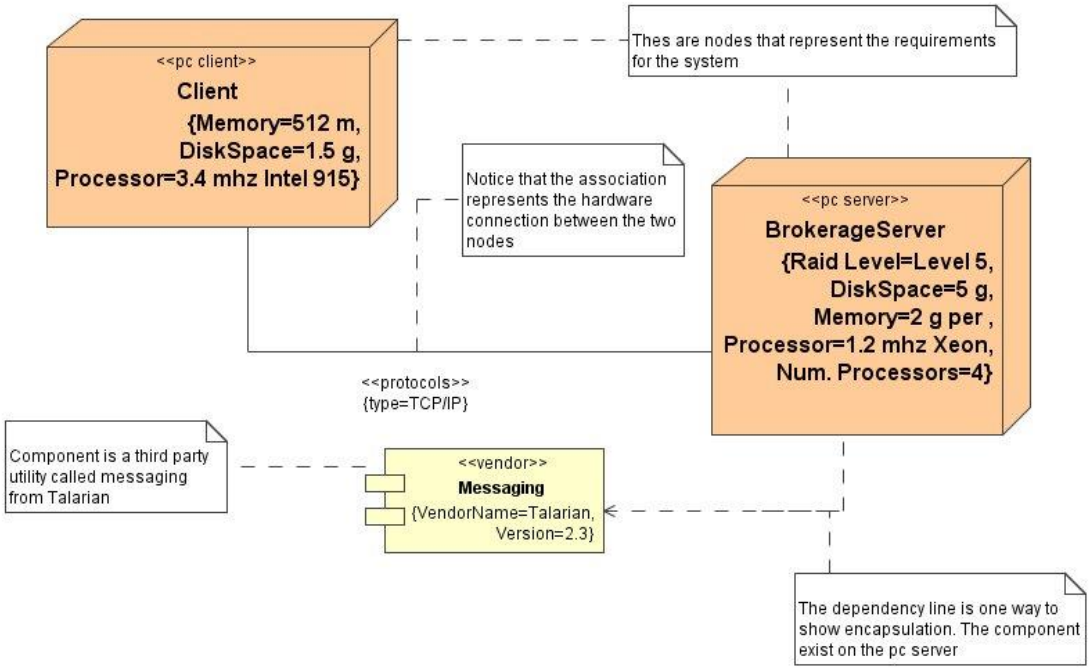
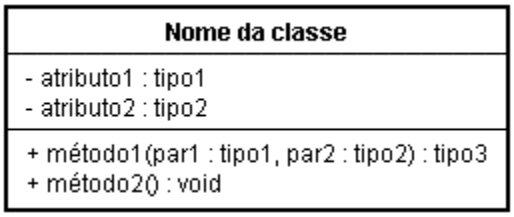
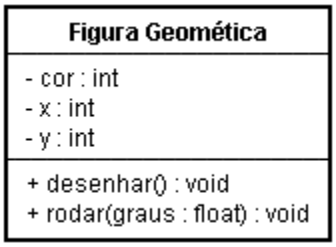


Diagrama de Classes

Os diagramas de classes representam graficamente as classes e as suas associações, respeitantes a um modelo.





Associações entre classes

As associações representam relações entre classes e, consequentemente, entre os objetos de cada uma dessas classes. As associações podem ser:

- Simples.
- Agregação.
- Herança.

Associações simples

As Associações simples estabelecem interações entre classes de forma a permitir a comunicação entre os objetos por elas representados.

Tipicamente, define-se uma associação simples entre duas classes quando:

- Existe troca de mensagens entre elas (invocação de métodos).
- Uma delas cria instâncias da outra.

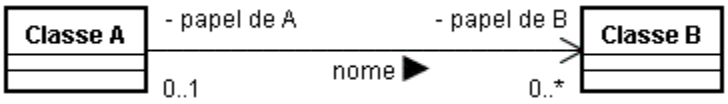
Numa associação é necessário definir:

- O nome. Designação da associação, tipicamente, representada por um verbo.
- A multiplicidade. Quantidade de objetos envolvimentos na associação.
- A navegabilidade. Direção da associação: bidirecional ou unidirecional.

E opcionalmente:

- Os designadores. Ou seja, qual o papel que cada classe desempenha na associação. Tipicamente, necessários em associação entre a mesma classe.

A associação é representada por uma linha reta contínua.



Muitas vezes uma associação pode ser confundida com um atributo. Efetivamente, quase sempre, a associação pode ser um atributo em que o seu tipo é uma outra classe.

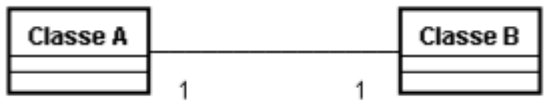
Nestes casos, deve-se sempre desenhar a associação e nunca a declaração do atributo na classe.

Nome da associação

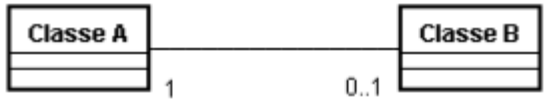
O nome da associação é, tipicamente, representado por um verbo (mas nem sempre), que caracteriza a relação entre as duas classes.

Multiplicidade

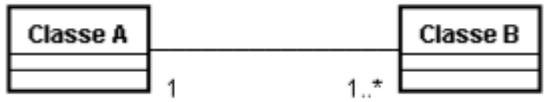
A multiplicidade de uma associação especifica o número de objetos de uma classe que se podem relacionar com um objeto da outra.



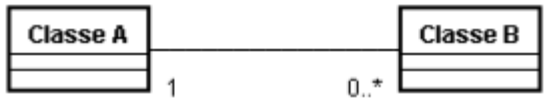
Apenas Um

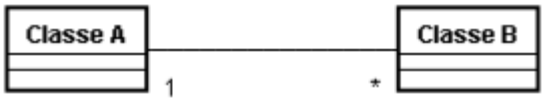


Zero ou Um

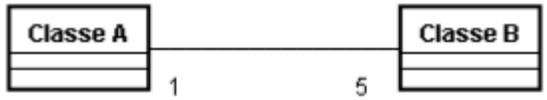


Um ou mais

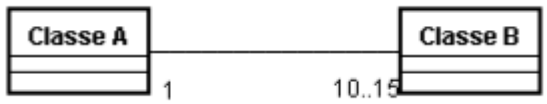




Zero ou mais

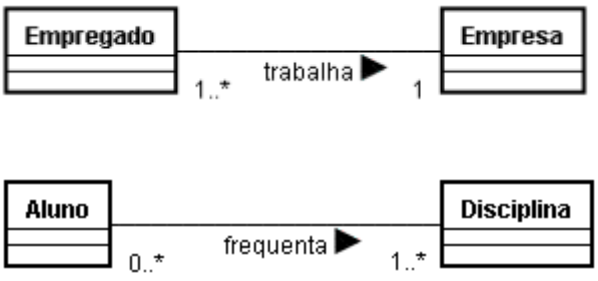


Número fixo



Intervalo

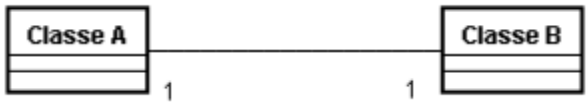
Exemplo:



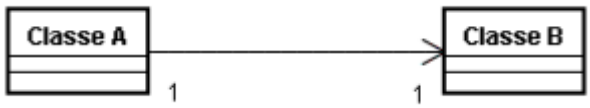
Navegabilidade

Tipicamente uma associação é bidirecional, ou seja, é possível estabelecer a associação nos dois sentidos. Pode também falar-se de visibilidade. Se uma associação é unidirecional apenas existe visibilidade num sentido.

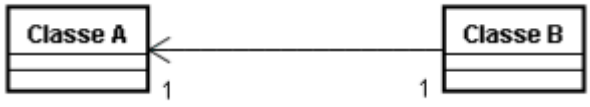
Por vezes, ou por requisitos do problema ou por necessidade de simplificação durante a implementação, pode-se estabelecer a associação apenas num sentido.



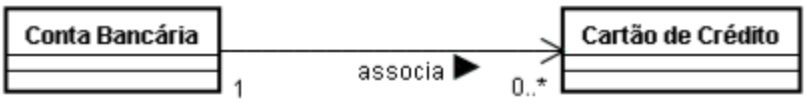
Associação bidirecional



Associação da classe A para B. B não associa A.



Associação da classe B para A. A não associa B.



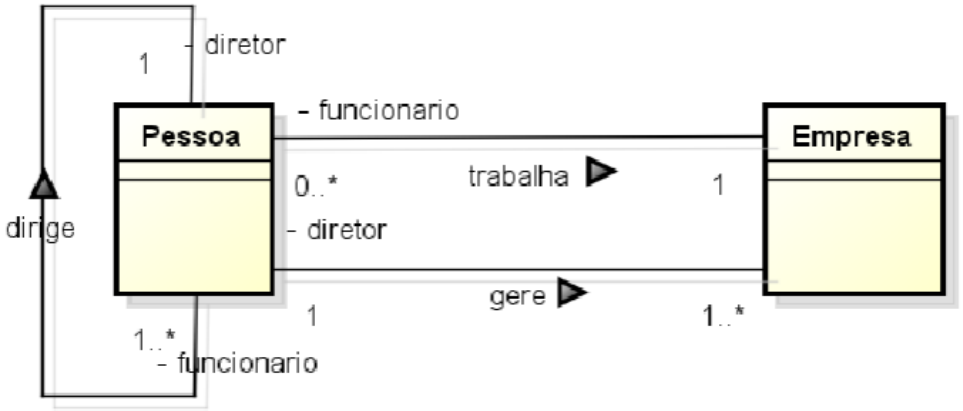
Designadores

Nos extremos de uma associação podem ser também colocados designadores.

O designador de um extremo é um nome que identifica o tipo de papel que um objeto desse extremo desempenha na associação.

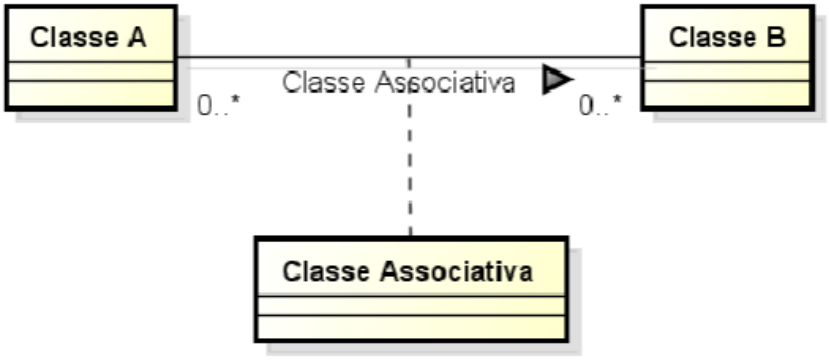
Os designadores são opcionais.

Os designadores são recomendados em associações entre objetos da mesma classe e também quando se torna necessário distinguir associações entre os mesmos pares de classes.



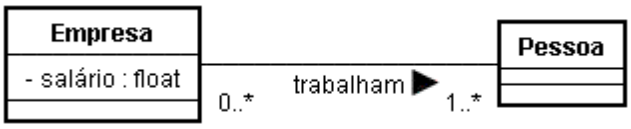
Classes associativas

Algumas vezes as associações podem ser muito complexas. Em situações em que a associação necessita de ser auxiliada por valores ou operações devem ser usadas classes associativas para descrever a associação.

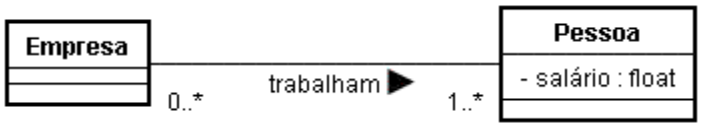


Exemplo:

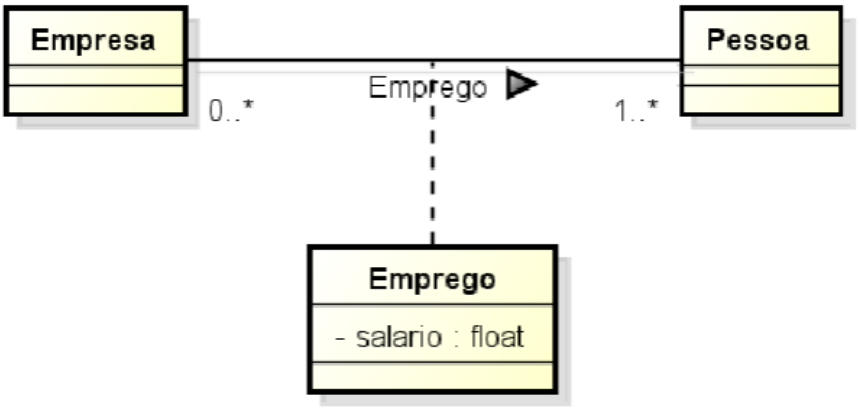
Vamos imaginar que uma pessoa pode trabalhar em várias empresas. Obviamente que numa empresa também trabalham várias pessoas. Como representar o salário de uma pessoa numa empresa.



Desta forma todas as pessoas teriam o mesmo salário.

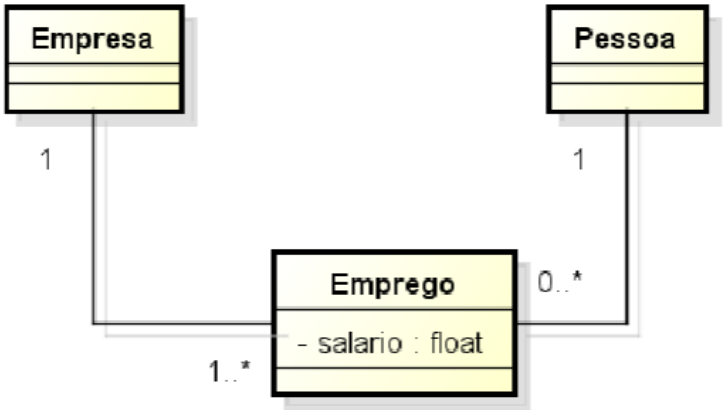


Assim, uma pessoa teria o mesmo salário em todas as empresas que trabalha.



Com a classe associativa, é possível representar os diferentes salários nas diferentes empresas.

Sem utilizar o conceito de classe associativa, podemos representar a mesma solução, desta forma:



Veremos mais tarde, que na implementação optamos por esta visão. Mas a representação por classe associativa é, conceptualmente, mais rica.

Agregação

A agregação é uma relação entre duas classes em que um objeto de uma das classes é parte integrante de um objeto da outra classe, estabelecendo-se uma relação todo-parte.

A agregação pode ser considerada um caso particular de uma associação simples em que o verbo da relação é:

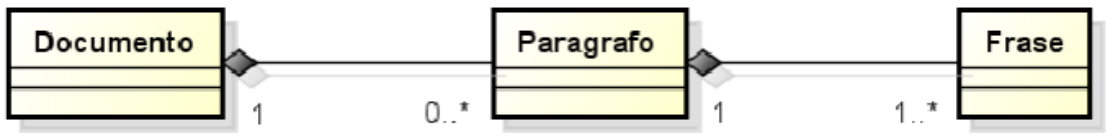
- ter,
- conter,
- pertencer,
- fazer parte de…,
- consistir em…, etc...

O nome da associação na agregação pode assim ser omisso.

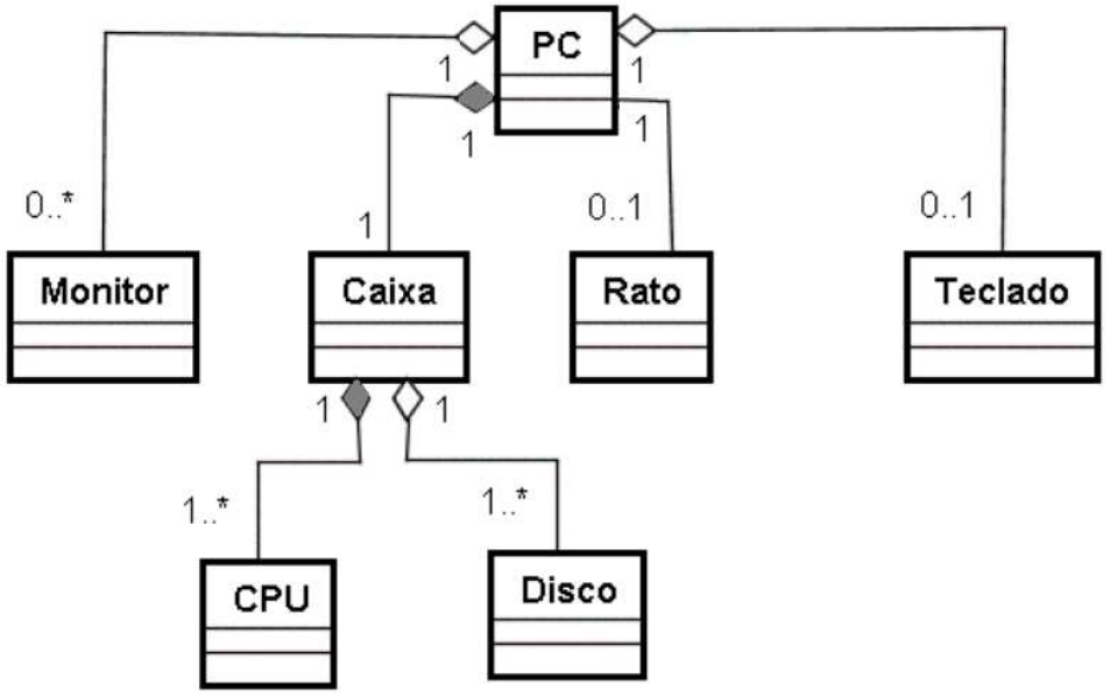
Em caso de dúvida deve usar-se uma associação simples.

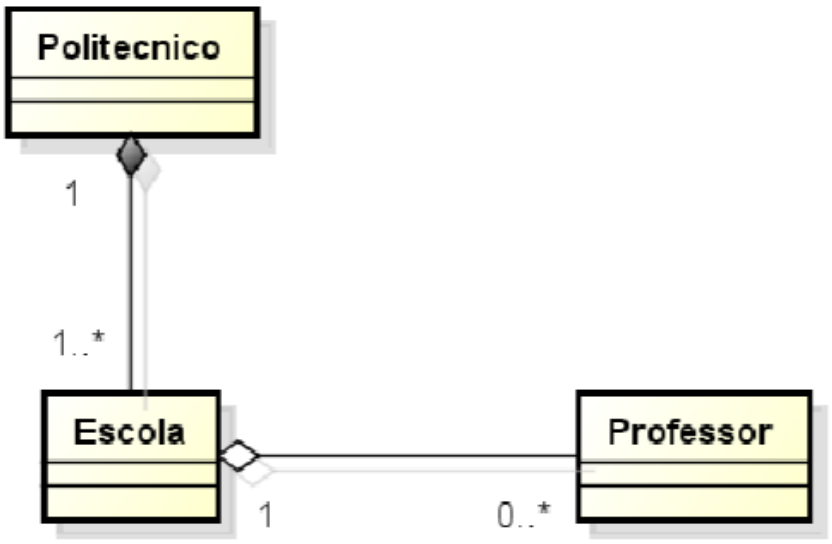
Exemplos:

Um documento de texto é composto por parágrafos, e um parágrafo por frases.



Um PC é formado por vários componentes, e alguns desses componentes são ainda formados por outros componentes.





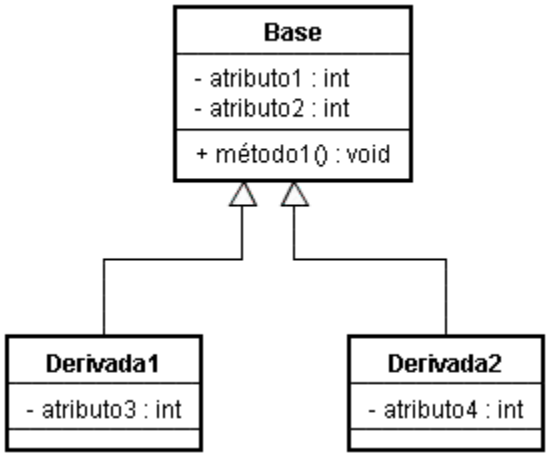
Herança

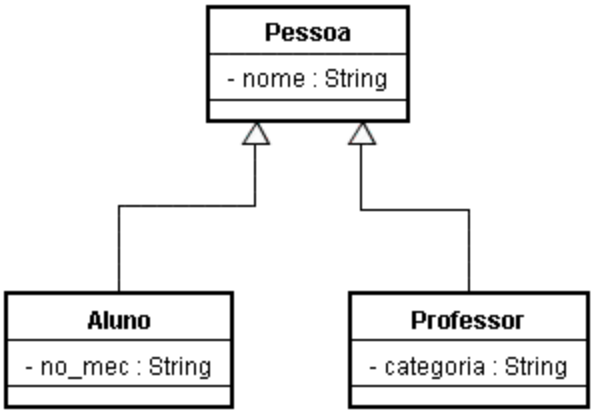
Permite o refinamento de classes, partindo de uma classe mais genérica para uma mais específica.

Os atributos e operações comuns a um grupo de classes são associados a uma classe base.

Cada classe derivada herda todos os aspetos da classe base (atributos e métodos)

A expressão "A é B" deve fazer sentido.





Um Aluno é uma Pessoa

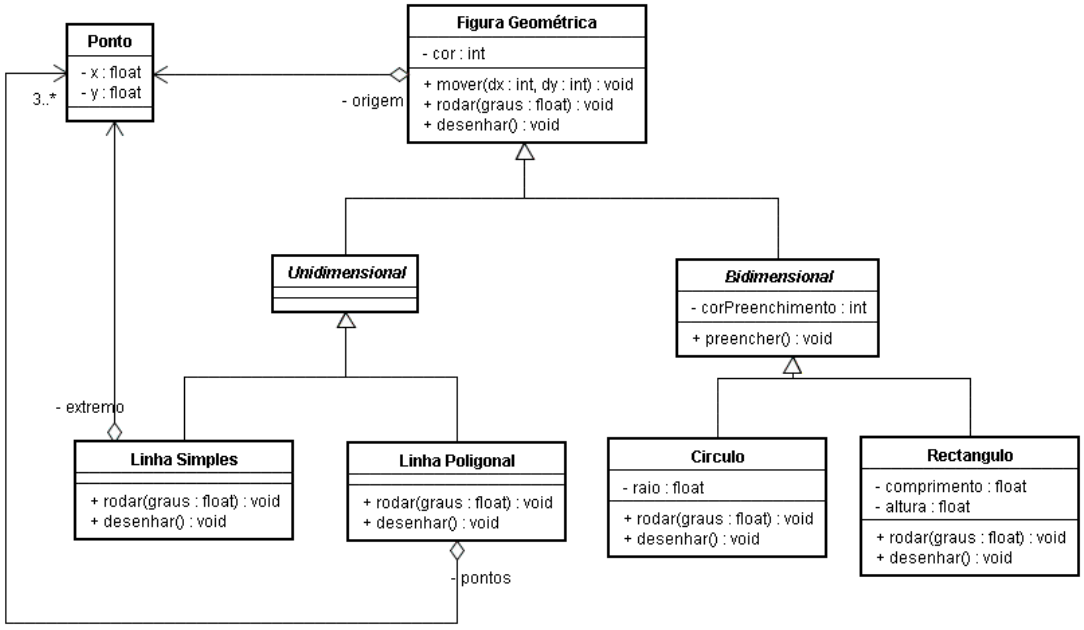
Um Professor é uma Pessoa

Sobreposição e acréscimo de características

Uma classe derivada é normalmente uma especialização da classe base, sobrepondo operações que herda, redefinindo-as, mas mantendo o mesmo nome.

Recorre-se à sobreposição quando é necessário alterar o comportamento herdado.

Podem também ser acrescentadas novas características na classe derivada, adicionando novas operações e atributos.



Classes Abstratas -> não pode ser distanciadas

Uma classe abstrata é uma classe não instanciável, isto é, não é possível criar objetos a partir delas.

Servem para criar a estrutura comum de várias classes derivadas, de modo a forçar a existência dessa estrutura nas classes derivadas.

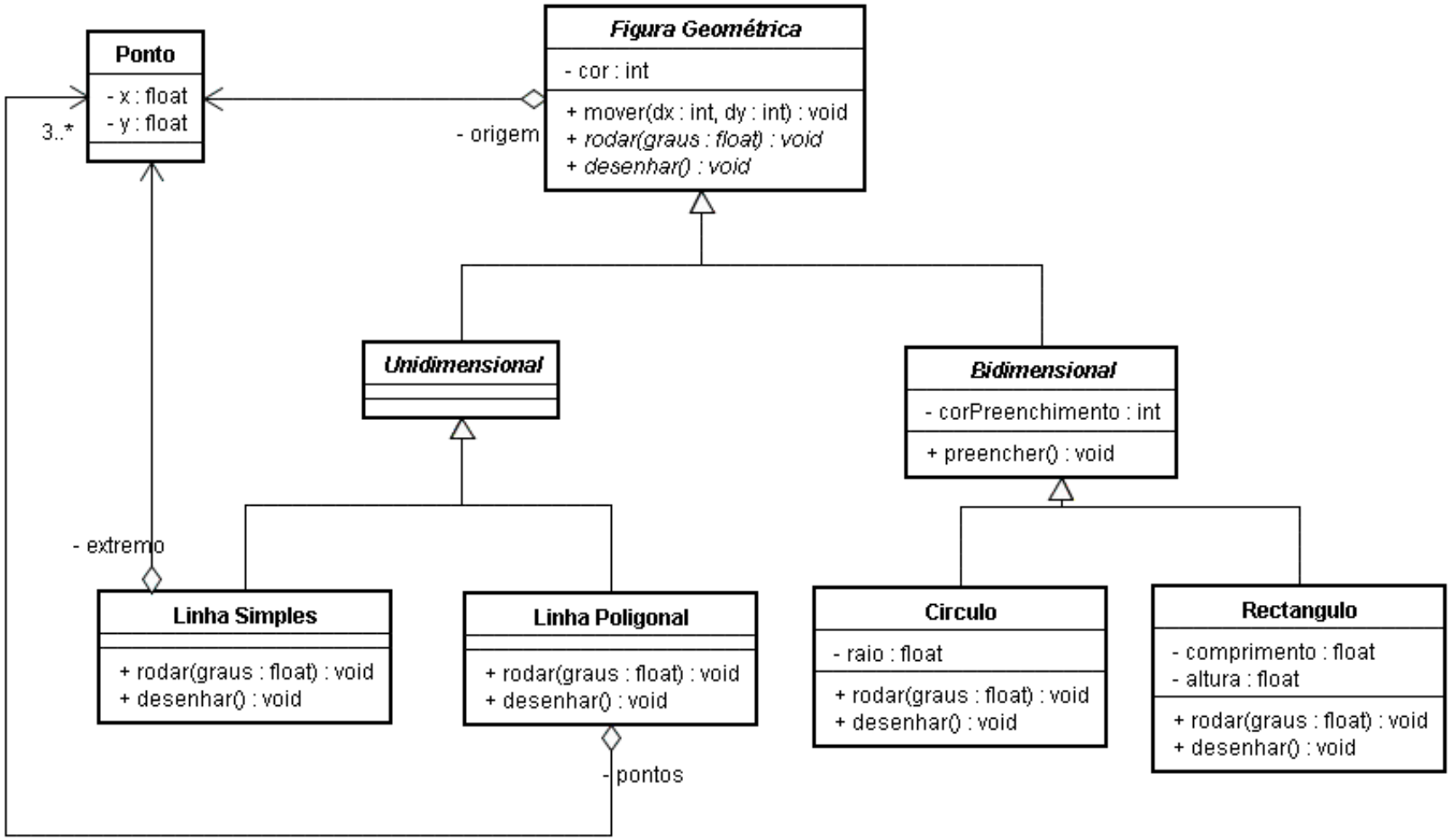
Uma classe abstrata pode definir o protocolo para uma dada operação sem fornecer a sua implementação. Nesse caso estaremos também na presença de uma método abstrato (ou método virtual puro).

se numa classe nos virmos uma classe com o
- método virtual
- virtual puro -> tem método virtual e é inicializa .EX: virtual void f() =0;

Uma classe que contém pelo menos um método abstrato é uma classe abstrata. As suas derivadas serão abstratas até se implementarem todos os métodos abstratos.

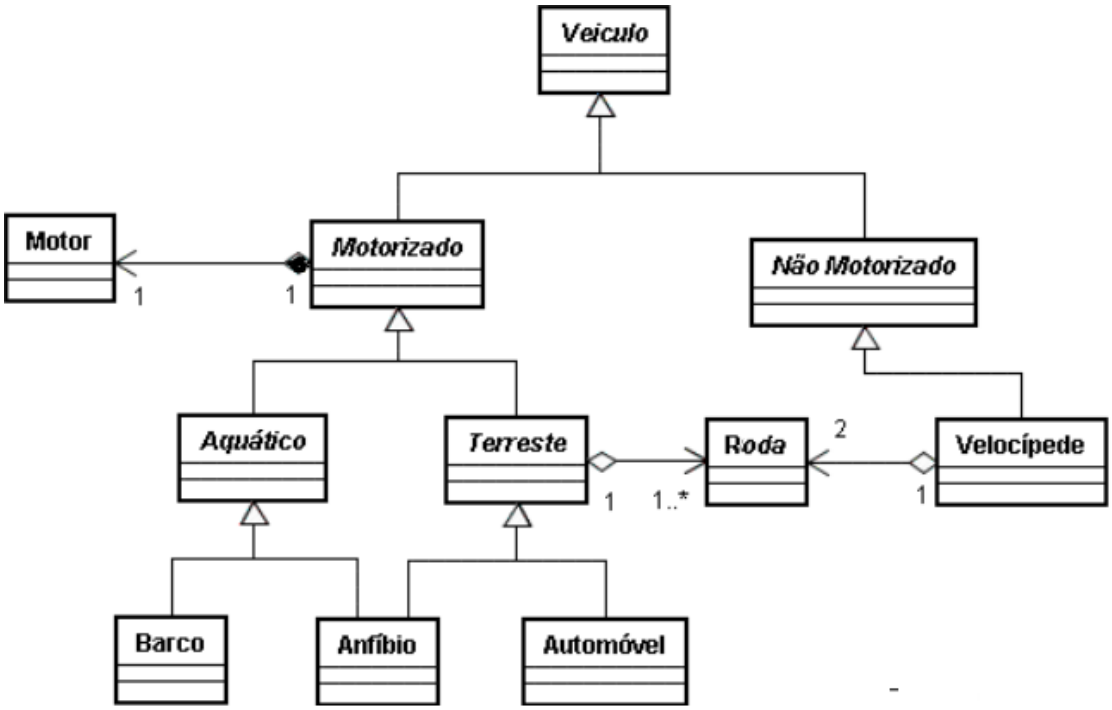
Uma classe ou um método abstrato é representado no diagrama de classe colocando o nome em itálico.

No papel deve marcar, explicitamente, as classes abstratas e os métodos abstratos com o nome "abstrato".


Ponto
- x : float
- y : float
3..*
- origem
Figura Geométrica
- cor : int
+ mover(dx : int, dy : int) : void
+ rodar(graus : float) : void
+ desenhar() : void
Unidimensional
Bidimensional
- corPreenchimento : int
+ preencher() : void
- extremo
Linha Simples
+ rodar(graus : float) : void
+ desenhar() : void
Linha Poligonal
+ rodar(graus : float) : void
+ desenhar() : void
- pontos
Circulo
- raio : float
+ rodar(graus : float) : void
+ desenhar() : void
Rectangulo
- comprimento : float
- altura : float
+ rodar(graus : float) : void
+ desenhar() : void

Herança Múltipla

Herança múltipla permite a uma classe ter mais do que uma classe base e herdar características de todas elas.



Identificação de classes

A própria especificação de um problema refere normalmente objetos concretos, é necessário generalizá-los através de classes.

Algumas vezes é possível classificar os objetos, nesse caso deve usar-se a herança.

Um método simples para identificar classes é sublinhar os substantivos na especificação do problema: "Quando o comboio se aproxima da estação a sua velocidade diminui..."

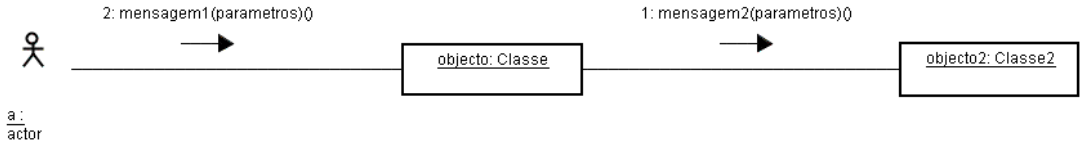
Comboio e estação serão provavelmente classes, mas o mais certo é velocidade ser apenas um atributo de comboio.

Verificar se a potencial classe pode ser caracterizada (estado) e se tem atividade (comportamento).

Deve-se tentar adaptar classes já existentes, por composição ou particularização, de modo a reduzir características.

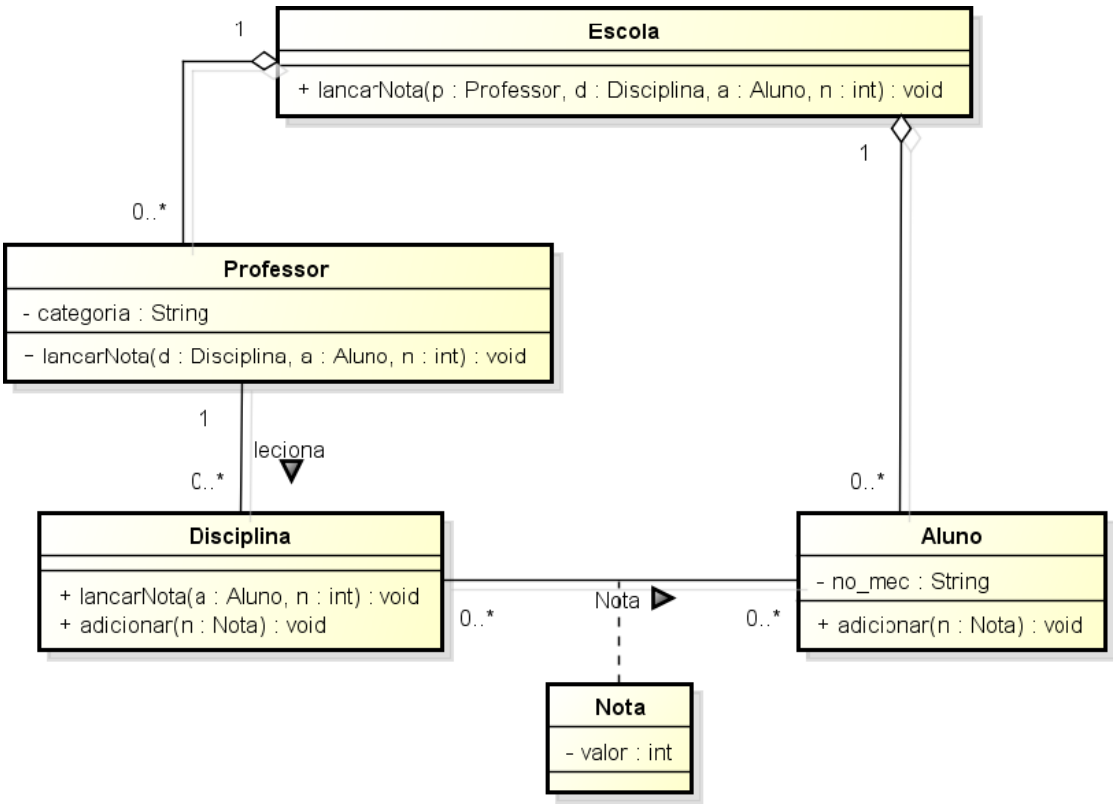
Diagrama de Comunicação

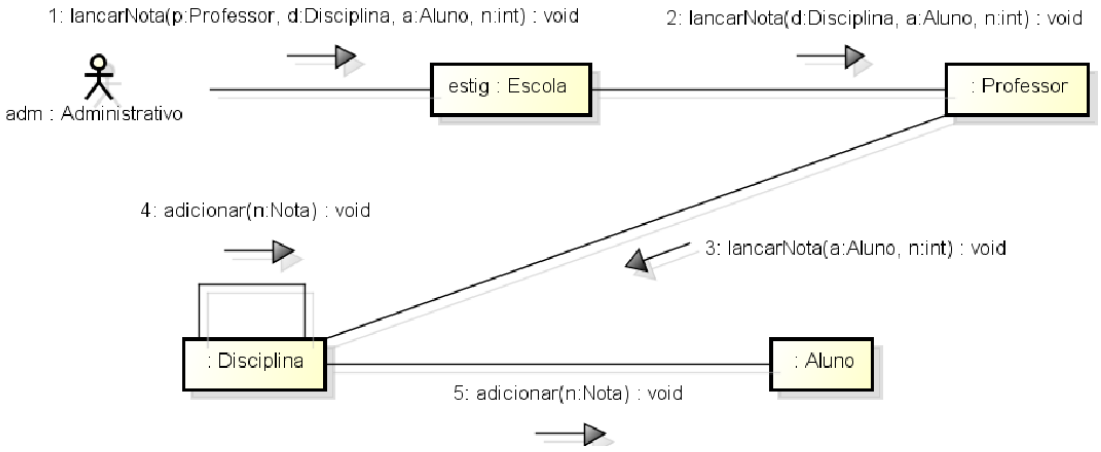
Os diagramas de comunicação revelam a interação entre os objetos das suas classes e as mensagens enviadas entre eles.



Permitem a visualização da sequência de execução das operações num sistema.

Diagramas (exemplo)





Linguagem C++

Introdução à linguagem C++

A linguagem C++ é uma extensão da linguagem C, uma vez que permite toda a eficiência e flexibilidade desta, acrescidas de potencialidades que permitem a programação orientada por objetos.

A linguagem C++ é considerada híbrida, pois permite tirar partido do paradigma de orientação por objeto ou ser tratada como uma linguagem procedimental.

A melhor forma de aprender a linguagem C++ é utilizá-la!

Organização de um programa

Um programa em C++ é constituído por um conjunto de:

Definições, declarações e instruções.

Uma definição fornece a informação que permite ao compilador reservar memória para objetos (ou variáveis) ou gerar o código para as funções.

Uma declaração apresenta nomes de entidades e os seus tipos sem necessariamente definir (alocar) um objeto ou função.

As instruções permitem o acesso a entidades definidas: acesso a variáveis e objetos, invocação de funções e métodos.

Podendo ser organizadas em múltiplos ficheiros.

```
/* O meu primeiro programa */
#include<iostream>

using namespace std;

void main() {
    cout << “Ola mundo” << endl;
}
```

Microsoft Visual Studio Community com Visual C++ 2022

O Visual Studio é uma ferramenta de software categorizado como um Ambiente Integrado de desenvolvimento (do inglês, IDE).

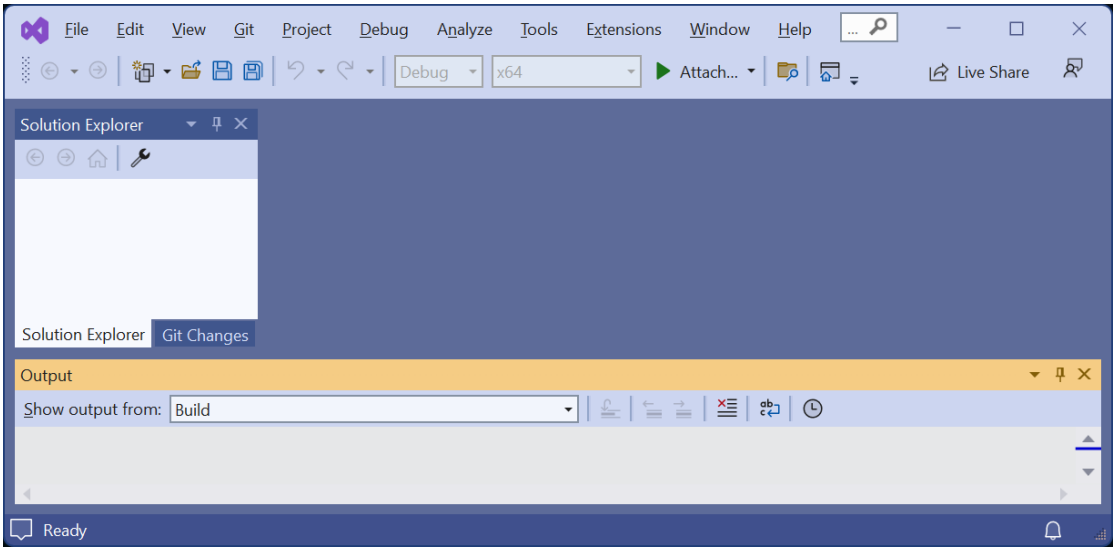
As principais características desta ferramenta são:

- Edição de código
- Iluminação de palavras reservadas
- Acesso imediato à lista de membros dos objetos (CTRL-space)
- Depurador (debugger) integrado

E outras:

- Suporte para diversas linguagens
- Edição de recursos visuais
- Trabalho cooperativo

Soluções e projetos



A organização de itens no Visual Studio é realizada por "soluções".

As soluções permitem a utilização de outros contentores para a gestão eficiente do desenvolvimento de aplicações:

- Soluções:
  - Projetos:
    - Itens:
      - Ficheiros
      - Diretorias

Criação de projetos

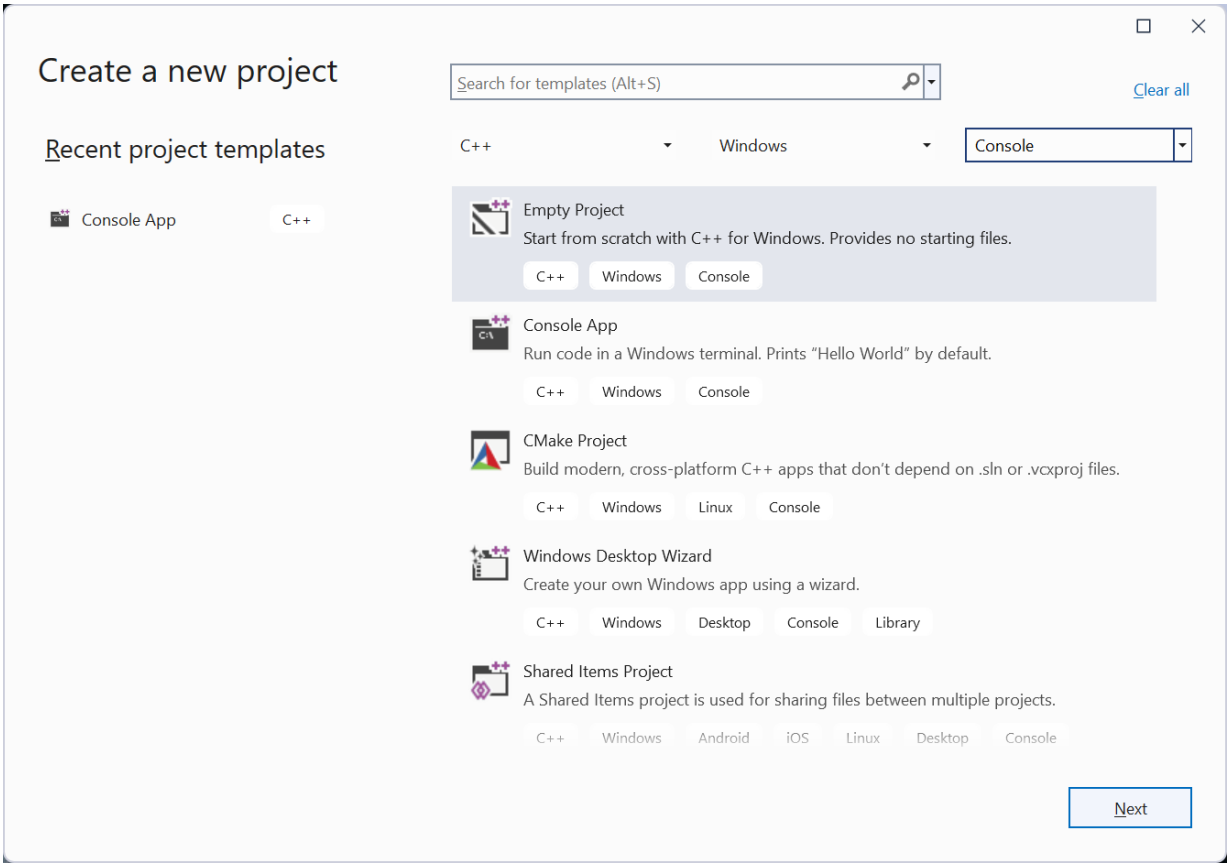
A criação de um projeto novo acarreta a criação de uma solução nova:

File | New | Project…

Podem, no entanto, ser criadas soluções vazias e posteriormente adicionar um ou mais projetos.

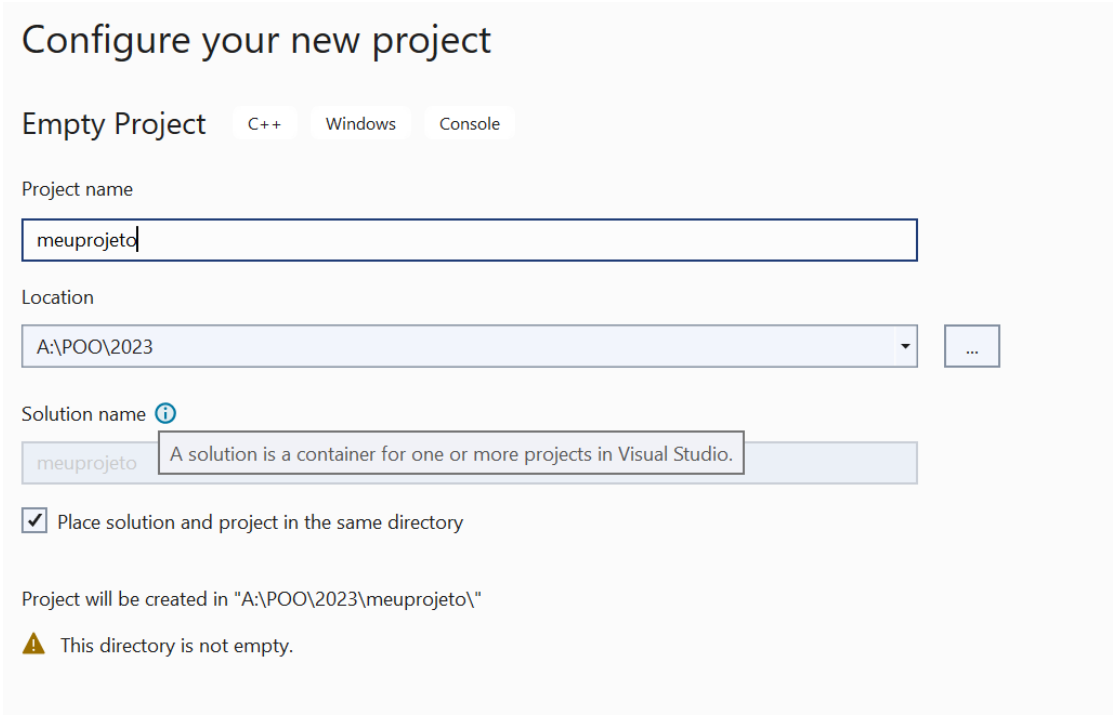
Tipos de projetos e modelos

- C++
  - Windows
    - Console
      - Empty Project

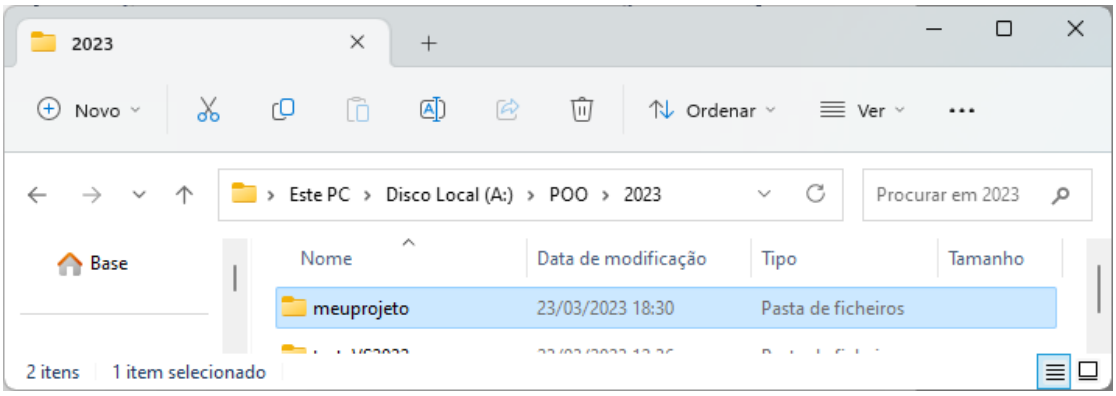


Após selecionar o modelo de projeto, e avançando para o passo seguinte, deverá escolher, numa nova janela de diálogo, o nome do projeto e a sua localização.

- Prevendo que a solução só irá ter um projeto, poderá optar por colocar numa mesma diretoria a solução e o projeto.



O novo projeto ficará numa diretoria com o seu nome.



Abertura e encerramento de soluções

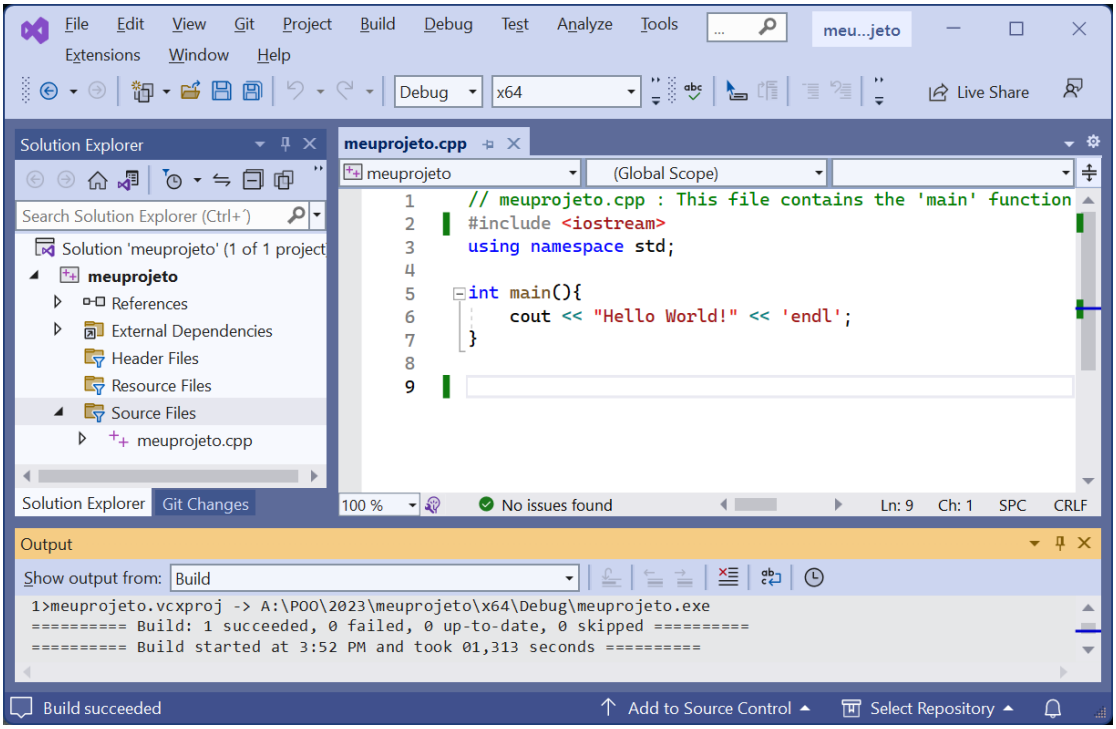
Abertura de uma solução existente: File | Open | Project / Solution…

Deve procurar-se o ficheiro com extensão .sln

Encerramento de uma solução: File | Close Solution

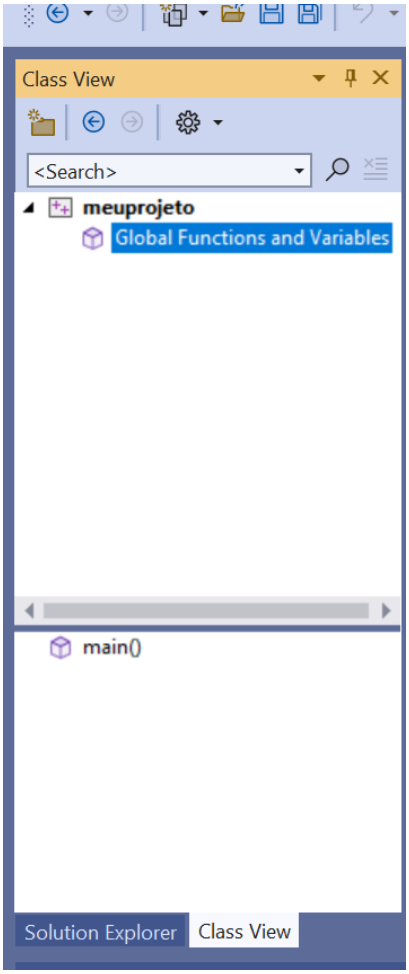
Explorador da solução (Solution Explorer)

Permite visualizar as soluções, os projetos das soluções e os itens atribuídos aos projetos



Vista de classes (View | Class View)

Permite visualizar as entidades lógicas dos projetos (espaços de nomes, classes, métodos, etc.).

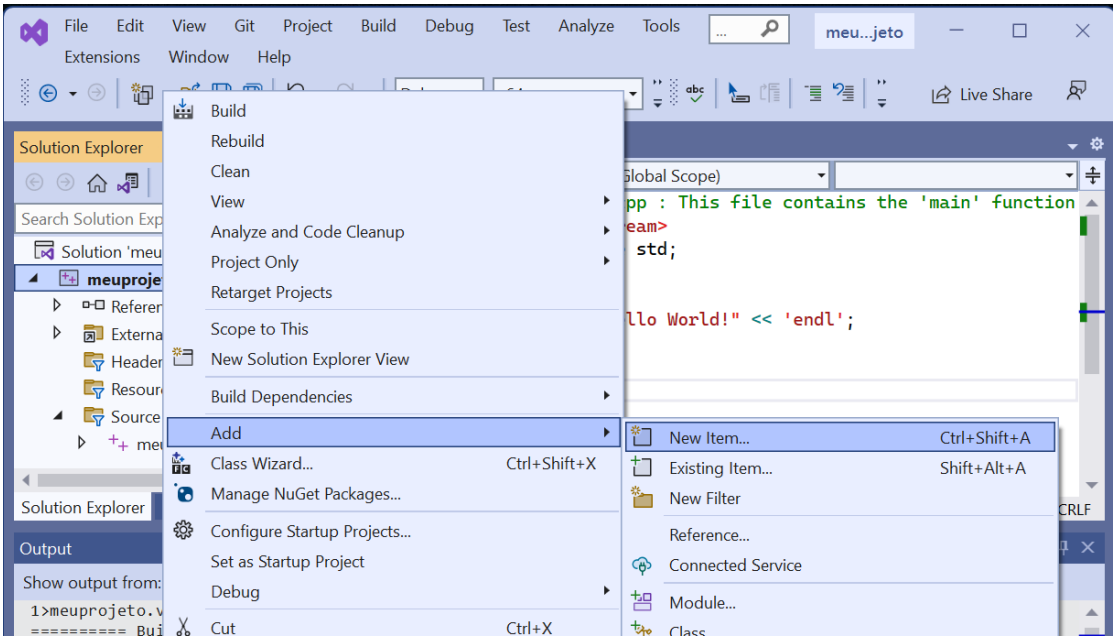


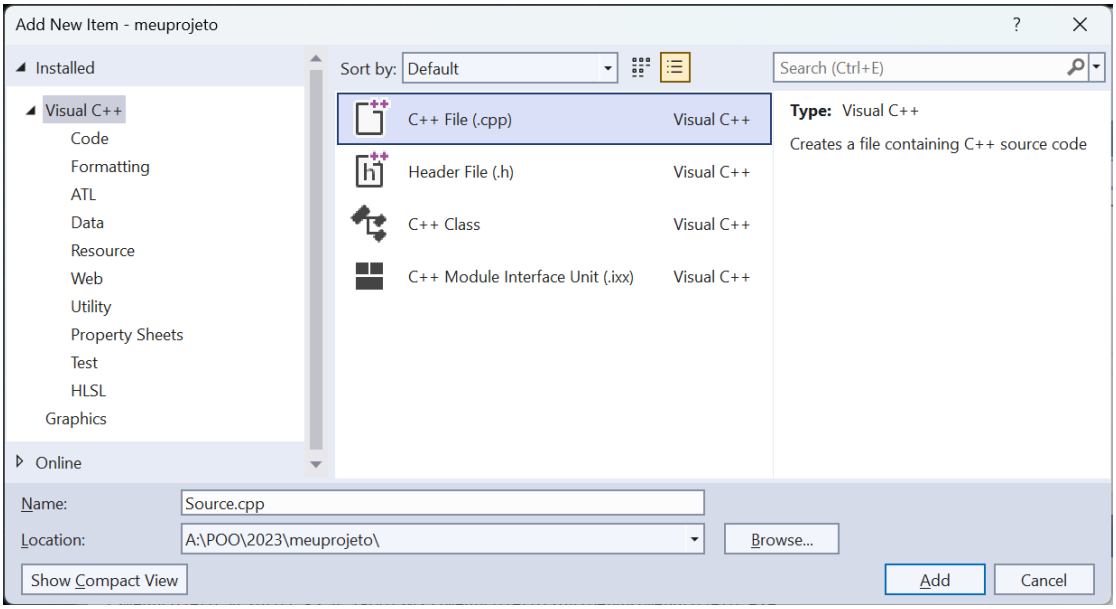
Icon	Description	Icon	Description
	Namespace		Method or Function
	Class		Operator
	Interface		Property
	Structure		Field or Variable
	Union		Event
	Enum		Constant
	TypeDef		Enum Item
	Module		Map Item
	Extension Method		External Declaration
	Delegate		Error
	Exception		Template
	Map		Unknown
	Type Forwarding		

Icon	Description
<No Signal Icon>	Public. Accessible from anywhere in this component and from any component that references it.
	Protected. Accessible from the containing class or type, or those derived from the containing class or type.
	Private. Accessible only in the containing class or type.
	Sealed.
	Friend/Internal. Accessible only from the project.
	Shortcut. A shortcut to the object.

Gestão de itens

Os Itens (ficheiros de código) podem ser adicionados no menu File, no menu Project ou carregando com o botão direito do rato no nome do projeto (dentro do explorador da solução).





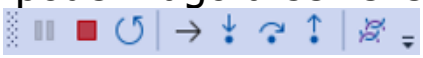
Debugger

O debugger permite analisar a execução do programa, executando-o em passos e observando o conteúdo das variáveis.

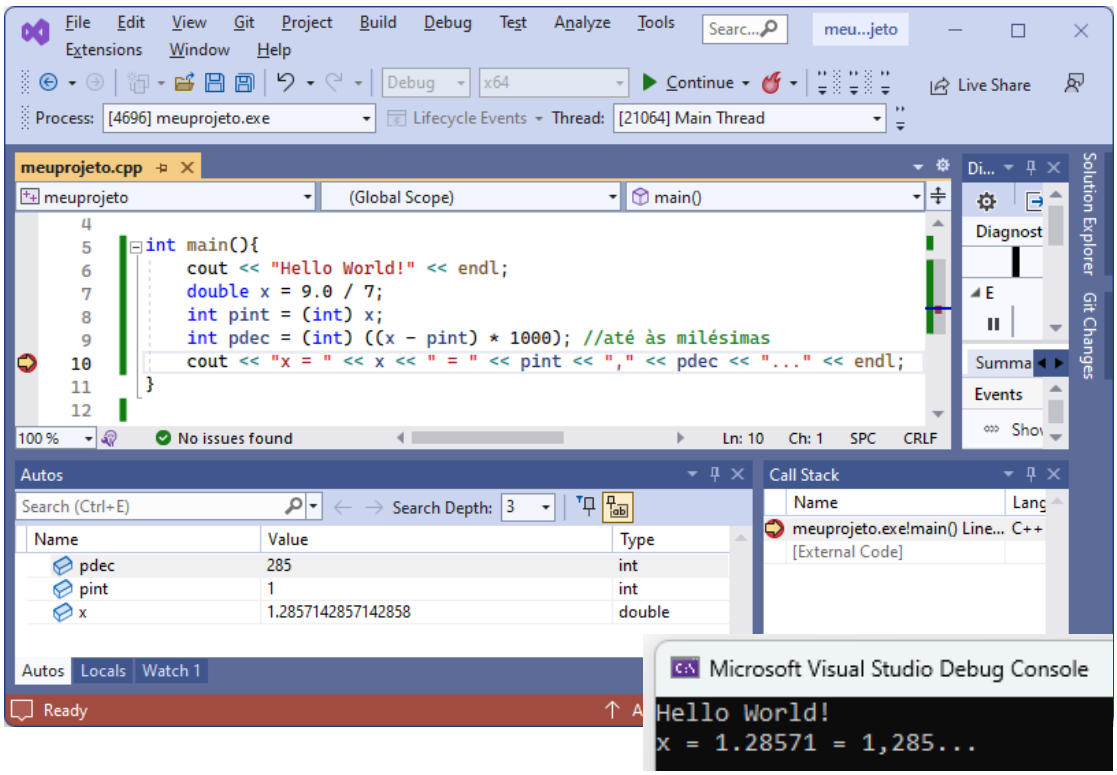
Para que o programa pare numa determinada linha deve-se colocar um ponto de quebra (breakpoint).

O início do programa em modo debug é dado na opção de menu Debug | Start Debugging.

Para exe cutar normalmente deve-se escolher a opção Debug | Start Without Debugging

As instruções podem agora ser executadas passo a passo com a barra: 

As variáveis podem ser observadas na janela "autos" ou "locals".



Principais diferenças entre a linguagem C e C++

- Comentários de linha:

```
/* */
//
```

- O nome de um enumerado é considerado um tipo de dados.

```
enum semana {segunda, terca, quarta,
quinta, sexta, sábado, domingo};
```

não havendo necessidade em defini-lo.

```
typedef enum {segunda, terca, quarta,
quinta, sexta, sábado, domingo} semana;
```

- O nome de uma classe e estrutura é considerado um tipo de dados.

```
struct registo {
    int codigo;
    char* nome;
    unsigned int idade;
}
```

- Declarações dentro de blocos ou após instruções

```
for (int n = 0; n < 10; n++)
```

- Operador de contexto (scope) ::

```
int var;

void func()
{
    int var;
    ::var = 0;
}
```

- Especificador `const` permite bloquear varáveis, oferecendo maior proteção.

```
const int a = 1;
```

- Conversão de tipos explicita.

Para além da conversão (cast) o nome de um tipo predefinido ou definido pelo programador pode ser utilizado como uma função para conversão de tipos.

```
float f = 10;

int i = int(f);
```

- Sobrecarga (overloading) de funções, ou seja, poder ter, no mesmo contexto, funções com o mesmo nome, mas diferenciáveis pelo número ou tipo dos parâmetros.

- Valores de parâmetros por omissão (*default*).
- Passagem de parâmetros por referência

```
void incrementa(int& valor)
{
    valor++;
}
```

- Operadores `new` e `delete`.

```
int* i;
i = (int*) malloc(sizeof(int) * 50)

int* i = new int[50];
```

Declarações e definições

Um programa em C++ é uma sequência de declarações e definições.

A definição de `main` indica o início da execução.

Qualquer identificador antes de ser utilizado, tem que ser declarado. Para isso, é necessário indicar ao compilador o seu tipo.

O tipo de uma entidade indica ao compilador de que entidade se trata.

Definições:

```
char c;
int conta = 1;
string nome = “Vanessa”;
const double pi = 3.14159265358979323385;
int quadrado(int x) { return (x * x); }
```

Declarações:

```
int dobro(int);
extern int erro;
```

Visibilidade ou contexto (*scope*)

O nome resultante de uma declaração apenas é visível desde o ponto que é definido até ao fim do bloco em que é definido.

```
int x;                  // global
void f()
{
    int x;              // local, esconde x
    x = 1;
    {
        int x;          // esconde x
        x = 2;
    }
    x = 3;
}
int* p = &x;                // usa x global
```

Para aceder a uma variável global escondida deve utilizar-se o operador `::`.

```
int x;
Void f2()
{
    int x = 1;
    ::x = 2;
}
```

Tempo de vida

Quando não é especificado o contrário pelo programador, uma entidade é criada quando é definida, e destruída quando fica fora de contexto.

O modificador `static` permite que uma entidade dure até ao fim do programa.

```
int a = 1;

void f()
{
    int b = 1;
    static int c = a;
    cout << " a = " << a++
        << " b = " << b++
        << " c = " << c << "\n";
    c = c + 2;
}

void main()
{
    while (a < 4) f();
}
```

Constantes

As constantes são todos os valores fixos que podem ser colocados ao longo de um programa.

Inteiros

`123`	`int`
`12345671 (ou L)`	`long`
`12345u (U, ul ou UL)`	`unsigned`

Vírgula flutuante

`123.4`	`double`
`1e-2`	`double`
`123.4f`	`float`
`1e-2l`	`long double`

Octal

`037`	`31`

Hexadecimal

`0x1f`	`31`
`0xFUL`	`15`

Caracteres

Os caracteres em **C++** são armazenados internamente como números, a sua interpretação é que os torna caracteres.

```
'x'          31
'0'          48
```

Alguns caracteres são especiais, por não serem acessíveis pelo teclado, e têm que ser escritos através de sequências de escape. Apesar da sua representação poderem ser mais do que 1 caracter, o valor representado é apenas 1.

Tabela de sequências de escape

\a	alert
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash (barra)
\?	?
\'	'
\"	"
\ooo	Número octal (ooo)
\xhh	Número Hexadecimal (hh)

O caracter '\0'

Existe ainda um caracter especial, o '\0', que representa o caracter nulo e corresponde ao inteiro 0. É utilizado para marcar o fim de uma string.

Expressões constantes

É possível definir um identificador para uma expressão constante. A sua utilização implica a substituição literal do valor definido.

```
#define MAXLINE 1000

char line[MAXLINE];
```

Entidades constantes

```
const int a = 5;
const char* s;
```

Constantes String

Ou também designadas apenas por strings são uma cadeia de 0 ou mais caracteres, envolvidos entre aspas

```
“Sou uma string”
```

```
“Tu es o que ? ”
```

Tecnicamente uma string é um array de caracteres, com o caracter '\0' a terminá-la. Por conseguinte, todas as string ocupam mais 1 caracter do o número de caracteres necessários.

Atenção: "x" e 'x' são entidades completamente distintas.

Constantes enumeradas

Uma enumeração é uma lista de valores inteiros constantes, aos quais é atribuído um identificador.

```
enum booleano { FALSO, VERDADEIRO };
```

O identificador `FALSO` vale `0` e `VERDADEIRO` vale `1`.

```
enum meses{ JAN = 1, FEV, MAR,…, NOV, DEZ };
```

Nota: Os elementos da enumeração são efetivamente identificadores, não strings.

Tipos de dados

Tipos fundamentais

```
char
int
float
double
void
```

Modificadores de Tipos

```
signed
unsigned
long
short
```

Tipos existentes em C++

Tipo	Tamanho em bits	Intervalo
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	O mesmo que **char**
int	32	-2.147.483.648 a 2.147.483.647
unsigned int	32	0 a 4.294.967.295
signed int	32	O mesmo que **int**
short int	16	-32768 a 32767
unsigned short int	16	0 a 65.535
signed short int	16	o mesmo que **short int**
long int	32	-2.147.483.648 a 2.147.483.647
signed long int	32	O mesmo que **long int**
unsigned long int	32	0 a 4.294.967.295
float	32	3.402823466e+38
double	64	1.7976931348623158e+308
long double	80	1.189731495357231765e+4932

Apontadores

Para um tipo `T`, `T*` representa o tipo de um apontador para `T`. Assim, variáveis do tipo `T*` podem armazenar endereços para uma variável do tipo `T`.

```
int* pi;
char** cpp;
```

Para se aceder ao endereço de uma variável é utilizado o operador `&`.

```
int a = 5;
int* pi;
pi = &a;

*pi = 10;
```

Arrays

Sendo `T` um tipo, `T[tam]` é o tipo de um array de `tam` elementos de tipo `T`. Os elementos são acedidos de 0 a `tam-1`.

```
float v[3];
int a[2][5];
int a[] = { 1, 2, 3, 4, 5 };
char str[] = ”uma string”;
```

Apontadores e arrays

Os arrays estão estreitamente relacionados com apontadores. O nome de um array pode ser tomado como um apontador que representa o endereço do primeiro elemento do array;

Estruturas

As estruturas permitem agregar num tipo várias entidades de diversos tipos.

```
struct morada {
    char* nome;
    char* rua;
    int numero;
    char* cidade;
    int codigoPostal;
};
```

O acesso aos constituintes de uma estrutura é feito através do operador (.) aplicado às variáveis da estrutura ou (->) quando aplicado a apontadores da estrutura.

```
morada m;

m.nome = “Patricia Marques”;

m.numero = 21;
```

Expressões e instruções

Operadores

Operadores Aritméticos

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da Divisão (modulus)

```
20-2*3 → 14
26 % 5 → 1
```

Operadores Relacionais

>	Maior
>=	Maior ou igual
<	Menor
<=	Menor ou igual
==	Igual a
!=	Diferente

Estes operadores têm prioridade inferior aos aritméticos. Expressões como:

```
i < lim -1      e   i < (lim-1)
```

são equivalentes.

Operadores Lógicos

`&&`	e
`||`	ou
`!`	não

A prioridade de `&&` é maior do que `||`, por sua vez, a prioridade de ambos é inferior aos operadores aritméticos e lógicos.

```
i < lim -1 && c != EOF
```

é equivalente a

```
(i < (lim -1)) && (c != EOF)
```

Por definição o valor de uma expressão relacional ou lógica é `1` se a expressão for verdadeira e `0` se for falsa.

```
10 > 3              → 1
5 != 5 || 7<=10     → 1
```

Conversões de tipo

Conversões implícitas

É necessário ter atenção a utilização de operadores com tipo diferentes. Sempre que um operador tenha operandos de tipos diferentes o tipo mais limitado é convertido para o de maior capacidade.

```
float + int → float
```

Em caso de atribuição para um tipo mais restrito, apesar de não ser ilegal, pode haver perda de informação e um aviso por parte do compilador.

```
int a;
int f=5.2;
a=f;
```

Conversões explicitas

Utiliza-se o casting (moldagem):

```
char c=10;
int x=(int)c; ou
int x=int(c);
```

No caso de objetos, para detetar uma conversão bem sucedida deve utilizar-se, o dynamic_cast.

```
A *a=new A;
B *b=dynamic_cast<B*>(a);
if(b!=0)…
```

Operadores de incremento e decremento

`++`	incrementa
`--`	decrementa

Ambos os operadores podem surgir em duas formas:

`++a`	prefixado
`a++`	posfixado

Em ambas as formas o operando é incrementado, mas na primeira o incremento é feito antes de o operador ser utilizado, na segunda, o operando é utilizado e depois incrementado.

```
n=5;
x=n++;
    x → 5
    n → 6

n=5;
x=++n;
    x → 6
    n → 6
```

Operadores sobre bits

Apenas podem ser utilizados em tipos inteiros: `char, short, int, long`.

`&`	e

|	ou
^	ou exclusivo
<<	deslocamento para a esquerda
>>	deslocamento para a direita
~	complemento de 1

```
n & 0177
```

Coloca a zero todos os bits de n exceto os 7 menos significativos.

```
n=231; (11100111)
n & 0177 → 103 (01100111)
```

Não confundir `&` e `|` com `&&` e `||`.

```
1 & 2 → 0
1 && 2 → 1
```

Operadores de Atribuição e expressões

Uma expressão é um conjunto de operadores e operandos da qual resulta um valor. Os operandos podem ser constantes ou variáveis.

Uma expressão pode ser utilizada em qualquer parte do programa, sendo normalmente atribuída a uma variável, por questões de comodidade e eficiência.

```
variavel = expressão;
```

```
a=50;
```

Expressões como `a=a+5`, podem ser simplificadas, utilizando a notação `a+=5`.

Esta simplificação pode se utilizada em quase todos os operadores binários:

```
+    -    *    /    %    <<   >>   &    ^    |
```

Exemplos:

```
a=10;          →  10
2+3+7*5        →  40
(5/2+8*10)     →  82
a*5-a*2        →  30
a++*7+3        →  73
b=5            →  5

a>0 && a<20    →  1
a && b         →  1
a & b          →  0    (sendo a=10)
a+=9           →  20
```

Expressão condicional

A expressão condicional, permite decidir entre o resultado de duas expressões com base numa condição.

$expr_1$? $expr_2$: $expr_3$

```
    a=5;
    a<0 ? -1 : 1   →  1

    z= (a>b) ? a : b   →  o maior
```

Resumo dos operadores

Tabela de operadores ordenados por precedência.

Símbolo	Descrição	Aplicação
::	Resolução de contexto	nome::membro
-> [] () () sizeof	Seleção Indexação chamada a função construção de objeto dimensão de tipo	apontador->membro apontador[exp] exp(lista_exp) tipo (lista_exp) sizeof(tipo)
++ -- ~ ! - + & * new delete ()	pós ou pré incremento pós ou pré decremento complemento bit a bit negação lógica menos unário mais unário endereço de valor apontado por alocação libertação cast	lvalor++ ou ++lvalor lvalor-- ou --lvalor ~exp !exp -exp +exp &lvalor *exp new tipo delete apontador (tipo)exp
* / %	multiplicação divisão resto da divisão	exp * exp exp / exp exp % exp
+ -	adição subtração	exp + exp exp - exp
<< >>	deslocar para a esq. deslocar para a dir.	lvalor << exp lvalor >> exp
< <= > >=	menor menor ou igual maior maior ou igual	exp < exp exp <= exp exp > exp exp >=exp
== !=	igualdade desigualdade	exp == exp exp != exp
&	AND bit a bit	exp & exp
^	XOR bit a bit	exp ^ exp
|	OR bit a bit	exp | exp
&&	AND lógico	exp && exp
||	OR lógico	exp || exp
?:	operador condicional	exp ? exp : exp
= (operadores de atribuição)	atribuição	lvalor=exp
,	sequência	exp, exp

Controle de Fluxo

As instruções de controle de fluxo permitem especificar por que ordem são executados os cálculos.

Instruções e blocos

Uma expressão quando terminada por um ';' torna-se uma **instrução**:

```
x = 0;
i++;
f();
```

Um **bloco** é um conjunto de instruções agrupadas entre chavetas ({ }), de tal forma que são equivalentes a uma única instrução.

if-else

É utilizado para exprimir execução de instruções com base na avaliação de uma condição.

```
if(expressao)
    instrucao1
else
    instrucao2
```

A parte do `else` é opcional.

A `expressao` é avaliada, se for diferente de 0, `instrucao1` é executado, se for 0, é executado `instrucao2`.

O `else` é sempre associado ao `if` mais próximo.

```
if(n>0)
    if(a>b)
        z=a;
    else
        z=b;
```

Em caso de dúvidas utilizar blocos.

Else-if

Utilizada em caso de decisões múltiplas.

```
if(expressao)
    instrucao
else if(expressao)
    instrucao
    …
else
    instrucao
```

switch

Utilizado em decisões múltiplas em que uma expressão é comparada com um valor constante.

```
switch(expressao){
    case expr-const1: instrucoes1
    case expr-const2: instrucoes2
    …
    default:   instrucoes
}
```

A instrução `break` força a saída para fora do `switch`.

Ciclos - while e for

São instruções que permitem a repetição condicionada de blocos.

```
while(expressao)
    instrucao
```

`instrucao` é executada enquanto `expressao` for verdadeira (não nula).

```
for(expr1; expr2; expr3)
    instrucao
```

que significa: executa `expr1` (inicialização), executa `instrucao`, enquanto `expr2` for verdadeira (não nula), no fim de cada iteração executa `expr3`.

A instrução `break` pode ser usada para interromper incondicionalmente um ciclo.

O ciclo `for` é adequado para situações em que exista uma inicialização e um incremento inerente em cada iteração.

do-while

```
do
    instrucao
```

```
while(expressao);
```

`instrucao` é executada pelo menos uma vez, se a `expressao` for verdadeira é executada de novo.

break e continue

O `break` permite a saída prematura de um ciclo ou bloco.

O `continue` permite que um ciclo passe para a seguinte iteração. Nos ciclos `while` e `do-while`, a condição é executada imediatamente a seguir, enquanto que no ciclo for a instrução de incremento ainda é executada.

Funções

Uma função pode ser definida da seguinte maneira:

```
tipo-de-retorno nome-da-funcao(argumentos)
{
    declaracoes e instrucoes
}
```

Quando o tipo de retorno é omitido, o tipo `int` é assumido.

A interação com uma função é feita através dos seus argumentos, do tipo de retorno e das variáveis globais.

A instrução `return` é utilizada para devolver um valor da função chamada para quem a chamou. Qualquer expressão pode ser devolvida:

```
return expressao;
```

Quando se pretender uma função que não devolva valor, `void` deverá ser utilizado como tipo de retorno. Nesse caso o `return` não deverá ter qualquer parâmetro.

Uma função pode ter mais do que um `return`, desde que em ramos de execução diferentes.

```
int func(char c)
{
    char a='c';
    if(c=='a')
        return 1;
    else
        return 0;
}
```

Sempre que uma função é chamada é feita uma cópia das suas variáveis locais e dos seus argumentos, por essa razão nunca se deve devolver o apontador de uma variável local.

Protótipo de uma função

Uma função para ser utilizada deve estar declarada ou pela sua própria definição ou pelo seu protótipo.

```
double sqrt(double);
```

Passagem de argumentos

Em C++, é possível passar argumentos por valor e por referência.

```
void f(int val, int& ref)
{
    val++;
    ref++;
}

int i = 1, j = 1;
f(i, j);
```

É possível também bloquear parâmetros das funções com o especificador `const`.

```
void f(const float& f)
{
    …
}
```

Sobrecarga de funções

Em C++ é possível definir funções que realizam operações diferentes com o mesmo nome.

Esta vantagem permite que sejam realizadas tarefas idênticas com objetos diferentes, utilizando o mesmo nome.

```
void print(int);

void print(char*);
```

Argumentos por omissão

Numa função é possível omitir os argumentos sequencialmente a partir do último.

```
void print(int num, int base = 10);

print(50);

print(40, 16);
```

Estruturas

Permitem agrupar dados de tipos diferentes.

```
struct Estrutura {
    char c;
    int i;
};


struct Estrutura e;
e.c = 10;
```

Classes em C++

Uma `class` define um novo tipo de dados.

```
class nome_identificador {
private:
    dados e métodos;
    // dados e métodos não podem ser acedidos
diretamente
protected:
    dados e métodos;
    // dados e métodos só podem ser acedidos
por classes derivadas
public:
    dados e métodos;
    // dados e métodos podem ser acedidos
diretamente

};
```

Uma definição de classe é em todo semelhante à definição de uma `struct`. Os dados são definidos como campos de uma `struct` ou variáveis. Os métodos são definidos da mesma forma que funções.

Os dados e os métodos são acedidos e invocados, respetivamente, através do operador . (ponto) ou "`->`" no caso de o objeto ser um apontador.

Os atributos devem ser privados para evitar alterações que tornem os objetos inconsistentes. Devendo criar-se métodos públicos para o seu acesso e modificação (getters e setters). Por exemplo, para acertar um relógio analógico utiliza-se o regulador (método) para que as horas e os minutos (atributos) não fiquem inconsistentes.

Construtores

Os construtores permitem a inicialização automática das instâncias. É permitida a definição de vários construtores de forma a disponibilizar um leque de inicializações possíveis.

O construtor é considerado um método especial, cujo nome é igual ao nome da classe.

Um construtor é invocado automaticamente quando é definido um objeto de uma classe.

Existem 4 categorias de construtores:

- Construtor por omissão;
- Construtor de cópia;
- Construtor de conversão;
- Todos os outros

Construtor por omissão

É um construtor que não possui parâmetros.

O compilador gera este construtor automaticamente, salvo sejam definidos outros pelo utilizador.

Construtor de cópia

O construtor de cópia é aquele que tem exatamente um parâmetro cujo tipo é a própria classe.

É invocado sempre que uma instância precisa de ser copiada durante a execução normal do programa.

O compilador gera um construtor deste tipo automaticamente, em que é feita uma cópia dos valores dos membros.

Este construtor é invocado automaticamente pelo programa em situações como:

- Criação de instâncias a partir de outras da mesma classe;
- Retorno de instâncias por valor em funções;
- Na passagem de parâmetros por valor.

Construtor de conversão

Um construtor de conversão é aquele que tem exatamente um parâmetro e não é um construtor de cópia.

Estes construtores não são gerados automaticamente.

O seu objetivo é converter o parâmetro num objeto da classe em questão.

Sempre que se cria um construtor de conversão está-se, implicitamente, a criar uma conversão do tipo de dados do argumento para o tipo de dados da classe.

É invocado automaticamente:

- Criação de instâncias a partir de outros tipos;
- Retorno de instâncias por valor em funções;
- Na passagem de parâmetros por valor (a atribuição é um caso, como se verá mais à frente).

Arrays e apontadores de objetos

A criação de vetores (arrays) de objetos só é possível se o construtor por defeito estiver definido.

```
class C
{
    int x;
public:
    c() {
        x = 0;
    }
    c(int i) {
        x = i;
    }
};

C vecC[10];
```

É também possível definir apontadores para instâncias. Tal como os apontadores de tipos simples, os apontadores de instâncias podem referenciar objetos já existentes ou serem alocados dinamicamente.

```
C a;
C* p0 = &a;
C* p1 = new C;
C* p2 = new C(5);

C* ap = new C[10];
```

As instâncias alocadas devem ser obrigatoriamente libertadas.

Autorreferência nas classes

No código de definição dos métodos de uma classe, encontra-se sempre disponível um apontador especial que referencia o objeto para onde foi enviada a mensagem do método que está a ser executado.

Este apontador é designado por `this`, e é utilizado em objetos que se auto referenciam ou para retirar ambiguidades.

Quando um método de uma classe é invocado dentro de outro método da mesma classe, o apontador `this` está implícito.

Membros constantes

Quer os atributos, quer os métodos podem ser definidos como constantes.

Os atributos são inicializados na altura da construção do objeto.

Os métodos declarados como tal, é-lhes retirada a capacidade de modificar os atributos do objeto.

1. Um método declarado como constante só pode invocar no seu corpo métodos constantes;
2. Um método não constante só pode ser invocado por objetos não constantes, enquanto que, um método constante pode ser invocado por um objeto constante e não constante.

Membros constantes

Os membros constantes devem ser utilizados sempre que se pretenda efetuar proteção de dados.

```
class Obj {
    const int c;
    int x;
public:
    Obj(int a, int r) : c(r) {
        x = a;
    }
    int getX() const {
        return(x);
    }
    void setX(int a) {
        x = a;
    }
};

const Obj oc(5, 10);
Obj o(10, 20);

o.getX()     // Válido
oc.getX()    // Válido

o.SetX(5)    // Válido

oc.SetX(5)   // não válido
```

Sobrecarga de operadores (Overloading)

A sobrecarga de operadores permite sobrepor a utilização normal dos operadores nas classes criadas pelo utilizador.

```
tipo operator<operador>(argumentos);
```

A grande vantagem é poder utilizar os operadores em instâncias da mesma forma que seriam utilizados nos tipos predefinidos.

Para os operadores ++ e --

```
tipo operator++();
tipo operator--();
```

significa a versão prefixada

```
tipo operator++(int);
tipo operator--(int);
```

significa a versão posfixada

Em C++ os operadores podem ser sobrecarregados quer como métodos quer como funções globais

Tipo de operador	Forma reduz. de inv.	Método	Função Global
Unário prefixado	§ a	a.operator§( )	operator§(a)
Unário sufixado	a §	a.operator§(int)	operator§(a, int)
Binário	a § b	a.operator§(b)	operator§(a, b)
Operador Afectação	a = b	a.operator=(b)	----------

Legenda: § – símbolo do operador; a – objecto receptor; b – operando direito nos op. bin.

Quando um operador se encontra sobrecarregado (definido) quer como método quer como função global, é o método que é executado na invocação na forma reduzida.

Nos operadores binários, o objeto recetor da mensagem é sempre o do lado esquerdo.

O operador atribuição não pode ser sobrecarregado como função global.

Exemplo: (sobrecarga dos operadores "+", "=" e "*")

```
class T{
     int x;
public:
     T(int v){x=v;}
     int get()const{return x;}
     T operator+(const T &outro)const{
          T t(x+outro.x); return t;
     }
     T &operator=(const T &outro){
          x=outro.x; return *this;
     }
};

T operator*(const T &t1, const T &t2){
     T t(t1.get()*t2.get());
     return t;
}

void main(){
     T a(10), b(15), c(0);
     c=a+b; //forma abreviada de invocar o método
                              operator+(T) da classe T
     cout<<c.get()<<endl;
     c=a*b; //forma abreviada de invocar a função
                                global operator*(T, T)
     cout<<c.get()<<endl;
};
```

Se o operador + estivesse também implementado como função global, continuaria a ser invocada a função membro da classe T

equivalente a c=a.operator+(b);
↓ por sua vez equivalente a
c.operator=(a.operator+(b))

equivalente a c=operator*(a,b);
↓ por sua vez equivalente a
c.operator=(operator*(a,b))

Associações simples e Agregação de classes

Em C++, as associações simples são, tipicamente, representadas por apontadores membros, enquanto que a agregação é, tipicamente, representada por instâncias membros, podendo, no entanto, ser também representada por apontadores.

Na associação simples, tipicamente, os objetos associados existem mesmo antes do objeto que associa ser criado, não sendo destruídos quando o objeto que associa é destruído.

Na agregação, tipicamente, os objetos agregados são criados e destruídos quando os objetos agregadores são criados e destruídos, respetivamente.

Associação simples – 1-1

```
class A{
     int x;
public :
     A(){ x = 0; }
};


class B{
     A* a;
public:
     B(){ a = NULL; }

     void setA(A* na){ a = na; }
};


void main(){
     A* a = new A;
     B b;
     b.setA(a);
     delete a;
}
```

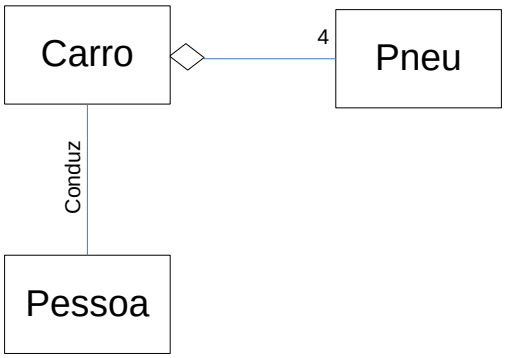
Agregação de classes

Quando se definem objetos como membros de outros objetos, coloca-se a questão de como são criados os objetos agregados.

```
class A {
    int x;
public:
    A()
    {
        x = 0;
    }
};
class B {
    A a;
public:
    B() {}
};

void main() {
    B b;
}
```

```
#include<iostream>
using namespace std;
class Interna{
    int x;
public:
    Interna(int a) {
        x = a;
    }
    void print() {
        cout << x << endl;
    }
};
```

```
class Externa{
    int y;
    Interna x;
    Interna z;
public:
    Externa(int a) : x(20), z(-36), y(a)
    {
    }
    void print() {
        cout << y << endl;
    }
    void print_x() {
        x.print();
    }

    void print_z() {
        z.print();
    }
};



void main()
{
    Externa o(-12);
    Interna i(25);
    o.print_x();
    o.print_z();
    o.print();
    i.print();
}
```




```
#include<iostream>
#include<string.h>

using namespace std;

class Pneu
{
     char marca[50];
     int diametro;
public:
     Pneu(){
          marca[0]='\0';
          diametro=0;
     }
     Pneu(char *m, int d){
          strcpy(marca, m);
          diametro=d;
     }
     void print() const{
          cout << "Pneu: Marca: " << marca << ",
Diametro: " << diametro << endl;
     }
};
```

```
class Pessoa
{
     char nome[50];
public:
     Pessoa(char *n){
          strcpy(nome, n);
     }
     void print() const{
          cout << "Pessoa: " << nome << endl;
     }
};

class Carro
{
     char marca[50];
     char cor[50];
     Pneu pneus[4];
     Pessoa *condutor;
public:
     Carro(char *m, char *c, char *mp, int d){
          strcpy(marca, m);
          strcpy(cor, c);
          for(int i=0;i<4;i++)
               pneus[i]=Pneu(mp, d);
          condutor=0;
     }
     void definirCondutor(Pessoa *p){
          condutor=p;
     }

     void print() const{
          cout << "Carro: Marca: " << marca << ",
Cor: " << cor << endl;
          cout << "Pneus: " << endl;
          for(int i=0;i<4;i++)
               pneus[i].print();
          if(conductor==NULL)
               cout <<"Não existe condutor"<< endl;
          else
               condutor->print();
     }
};
```

```
void main(){
     Carro c("Porsche", "Amarelo","Good Year",17);
     c.print();

     Pessoa p("Felisberto");
     c.definirCondutor(&p);
     c.print();

}
```

Getters e Setters

Os atributos devem ser sempre privados utilizando-se métodos para aceder e modificar os seus valores. Mesmo em atributos independentes, deve definir-se um par de métodos getAtributo e setAtributo.

```
class Pessoa
{
     char nome[50];
public:
     Pessoa(char *n){
          strcpy(nome, n);
     }
     const char *getNome()const{
          return(nome);
     }
     void setNome(char *n){
          strcpy(nome, n);
     }
};
```

Templates (modelos) de funções e classes

Um modelo ou template define uma família de funções ou classes, em que se torna possível especificar um tipo variável ou um valor variável.

Existem duas categorias de parâmetros:

Parâmetros de tipo (typename X ou class X). Os parâmetros de tipo podem ser substituídos por um qualquer tipo simples, combinado ou classe.

Parâmetros de valor (int VAL). Os parâmetros de valor são tipicamente valores do tipo especificado.

Tentemos, com o exemplo que se segue, perceber o verdadeiro interesse duma template de classes

Exemplo de implementação de uma pequena coleção

- *definição de uma classe simples que permita colecionar 10 inteiros*

```
class Lista{
    int array[10];
    int n;
public:
    Lista(){n=0;}
    void inserir(int e){
        if(n<10) array[n++]=e;
    }
    int remover(){
        int r=array[--n];
        return r;
    }
    bool empty(){return n<=0;}
    bool full(){return n==10;}
};
```

Esta Coleção é bastante simples. Mas se a quiséssemos implementar da forma mais adequada, dar-nos-ia bem mais trabalho…

> Mas será que faria sentido investirmos muitas horas de trabalho numa coleção tão limitada: apenas nos permite guardar 10 inteiros…

Solução: criar um **Template** de Classes em vez de uma classe concreta

Template de Classes

Permite definir uma classe de forma genérica, em que o tipo de alguns dos seus dados, e valores de que necessite, constem como parâmetros na sua definição

Exemplo de template de classes: *definição de uma lista genérica, que permite*

- *conter qualquer tipo de objetos*

- *ter a dimensão que se pretenda*

```
Template<class T, int MAX>
class Lista{
    T array[MAX];
    int n;
public:
    Lista(){n=0;}
    void inserir(T e){
        if(n<MAX) array[n++]=e;
    }
    T remover(){
        T r=array[--n];
        return r;
    }
    bool empty(){return n<=0;}
    bool full(){return n==MAX;}
};
```

Este **template de classes** permite instanciar posteriormente várias **classes** para diferentes tipos de objetos e dimensões

- torna-se possível, por exemplo, instanciar os seguintes objetos:

```
Lista<int,50> lint;
Lista<Pessoa,100> lpessoa;
```

Uma classe instanciada a partir de um **template de classes** é designada por **classe template**

`Lista` → é um Template de Classes

`Lista<int,50>` → é uma Classe Template (Instância do template Lista)

`Lista<Pessoa,100>` → é uma Classe Template (Instância do template Lista)

`lint` → é um objeto (Instância da classe `Lista<int,50>`)

`lpessoa` → é um objeto (Instância da classe `Lista<Pessoa,100>`)

Quando instanciámos a seguinte classe template:

```
Lista<int,50>
```

T foi associado ao tipo int, e

MAX ao valor 50

o parâmetro-tipo T tanto pode ser associado a um tipo elementar predefinido, como a uma classe ou a qualquer outro tipo de dados

- Um template pode ter um qualquer nº de parâmetros
- os parâmetros-tipo podem ser identificados indistintamente pelos classificadores typename ou class

Exempos: template <class T, int DIM> ...

template<typename T, int N, float K> ...

Um template de classes ou de funções, só por si, não gera código

- O compilador só gera código para as classes (ou funções) template
- por esse motivo, é habitual colocar os templates em ficheiros .h

```
//Lista.h
template<class T, int MAX>
class Lista{
     T array[MAX];
     int n;
public:
     Lista(){n=0;}
     void inserir(T e){
          if(n<MAX)array[n++]=e;
     }
     T remover(){
          T r=array[--n];
          return r;
     }
     bool empty(){return n<=0;}
     bool full(){return n==MAX;}
};
```

```
//main.cpp


#include "Pessoa.h"
#include "Lista.h"


void main(){
     Lista<int, 50> lint;
     lint.inserir(7);
     Lista<Pessoa, 100> lpessoa;
     Pessoa p("manuel");
     lpessoa.inserir(p);

}
```

Template de Funções

Permite definir uma família de funções

Exemplo de template de funções: *definição de uma família de funções* soma() *que permite somar valores de qualquer tipo*

```
template<class I>
I soma(I a, I b){
    I s=a+b;
    return(s);
}
```

- Uma função gerada a partir de um **template de funções** é designada por **função template**
- geralmente, a chamada de uma função não requer que se explicite o valor dos seus argumentos template

- o compilador consegue inferir o tipo (I) através dos argumentos da função

Exemplo:

```
float x=5.0f, y=1.5f, z;
z=soma(x,y);
```

Exemplo:

```
double x=5.0, y=1.5, z;
z=soma(x,y);
```

Porém, nem sempre isso é possível

- quando não é possível inferir o tipo I através dos argumentos da função, tem que se indicar explicitamente o significado desse tipo

Exemplos:

```
void main(){
    cout << soma(5.0,7.0) << endl;
    cout << soma(5,7) << endl;
    cout << soma<double>(5,7) << endl;

    Complexo x(2,5), y(3,8), Z;
    Z= soma(x,y);

}
```

Membros função de um template de classes

Um método de um template de classes é, implicitamente, um template de funções

- por isso, quando um método de um template de classes é definido fora da sua classe, deve ser explicitamente declarado como template

  Exemplo:

```
template<class T, int MAX>
void Lista<T,MAX>::inserir(T e){
        if(n<MAX)
            array[n++]=e;
}
```

Biblioteca STL (Standard Template Library)

A STL disponibiliza um conjunto de classes e templates com diversas funcionalidades, entre as quais, strings, coleções e iteradores.

Nas coleções podemos encontrar implementações de: vetores, listas, conjuntos, dicionários.

Ver: http://www.cplusplus.com/reference/stl/

Utilização de um conjunto (set)

Definição do template:

```
template<class K>
class set
{
public:
     class iterator{
     };

};
```

Além do template principal, está ainda definida uma classe que permite a iteração dos elementos: a classe **iterator**.

```
class iterator
{
public:
     K operator*() const;
     K* operator->() const;
     iterator operator++(int);
     iterator operator--(int);

};
```

Iteradores

- Um iterador é um objeto que permite ao programador percorrer os elementos de uma coleção, sem precisar de perceber como esses elementos se encontram organizados internamente.
- Do ponto de vista semântico, podemos ver um iterador como sendo um apontador inteligente, que em cada momento aponta para um dos elementos contidos na coleção, dispondo, por isso, de métodos que asseguram duas operações essenciais:
    - o acesso a um elemento particular da coleção (o referenciado);
    - a modificação do seu próprio estado, para que passe a referenciar o elemento seguinte na coleção;
- Deve também existir uma forma de pôr um iterador a apontar para o primeiro elemento, e uma maneira de verificar se o iterador chegou ou não ao fim da coleção.

- O principal objetivo do iterador é então permitir que o utilizador aceda a cada elemento sem precisar de conhecer a estrutura interna de uma coleção.
    - Permite que a coleção armazene os elementos como bem entender e, ainda assim, o utilizador a trate como se fosse uma simples lista ou sequência.

- Os iteradores apresentam uma interface base comum, mas são implementados em função da estrutura interna da coleção que terão que saber percorrer
    - Podemos mesmo afirmar que cada iterador fica especializado em percorrer apenas um tipo particular de coleção.
- Com esta abordagem, torna-se mesmo possível criar vários iteradores para percorrerem em simultâneo uma mesma coleção.

Principais métodos do template set

```
template<class K>
class set
{
public:
     pair<iterator, bool> insert(const K&);
     iterator find(const K&) const;
     size_type erase(const K&);
     void clear();
     size_type size() const;
     bool empty() const;
     iterator begin();
     iterator end();

};
```

A classe `pair<iterator,bool>` tem dois atributos públicos: `first` e `second`, que permitem aceder, respetivamente, a um iterador e a um booleano. O booleano indica se o elemento foi inserido ou não. O iterador refere o elemento contido na coleção (o inserido ou o que já existia).

Como a coleção `set`, ordena os elementos, é necessária a definição de um operador de relação de ordem dos elementos, neste caso, o operador `<`.

Exemplo

```
#include<iostream>
#include<string>
#include<set>

using namespace std;

class Pessoa
{
     string nome;
public:

     Pessoa(string n){
          nome=n;
     }
     string getNome() const{
          return(nome);
     }

     void setNome(string s) {
          nome=s;
     }

     bool operator<(const Pessoa &p) const{
          return(nome<p.nome);
     }
};
```

```
void main()
{
     set<Pessoa> seto;
     set<Pessoa*> setp;

     Pessoa p("Jose");
     Pessoa t("Carlos");

     seto.insert(p);
     pair<set<Pessoa>::iterator, bool>
r=seto.insert(p);
     seto.insert(t);

     Pessoa q("Jose");

     seto.erase(q);

     Pessoa *pp=new Pessoa("Luis");
     setp.insert(pp);

     set<Pessoa>::iterator i;
     for(i=seto.begin();i!=seto.end();i++){

          cout << i->getNome() << endl;
     }

     set<Pessoa*>::iterator j;
     for(j=setp.begin();j!=setp.end();j++){
          Pessoa *n=*j;
          cout << n->getNome() << endl;
     }
}
```

Utilização simplificada

Para simplificar a utilização do set, irá utilizar-se uma classe designada *Coleccao*.

```
template<class K>
class Coleccao: public set<K>
{
public:
     bool insert(const K &c);
     K *find(const K &c);
     int size() const;
     void erase(const K &);
     //void clear();
     //bool empty() const;
     //iterator begin();
     //iterator end();
};

template<class K>
bool Coleccao<K>::insert(const K &c)
{
     pair<set<K>::iterator, bool>
r=set<K>::insert(c);
     return(r.second);
}
```

Second pq é o bool, se fosse a posição (iterator) colocaca "r.i

```
template<class K>
K *Coleccao<K>::find(const K &c)
{
     K *r=0;
     set<K>::iterator i=set<K>::find(c);
     if(i!=set<K>::end())
          r=i.operator ->();
     return(r);
}

template<class K>
int Coleccao<K>::size() const{
     return((int)set<K>::size());
}
```

```
template<class K>
void Coleccao<K>::erase(const K &c ) {
     set<K>::erase(c);

}


void main()
{
     Coleccao<Pessoa> seto;
     Coleccao<Pessoa*> setp;

     Pessoa p("Jose");
     Pessoa t("Carlos");

     seto.insert(p);
     bool r=seto.insert(p);
     seto.insert(t);

     Pessoa q("Jose");

     seto.erase(q);

     Pessoa *pp=new Pessoa("Luis");
     setp.insert(pp);

     Coleccao<Pessoa>::iterator i;
     for(i=seto.begin();i!=seto.end();i++){

          cout << i->getNome() << endl;
     }

     Coleccao<Pessoa*>::iterator j;
     for(j=setp.begin();j!=setp.end();j++){
          Pessoa *n=*j;
          cout << n->getNome() << endl;
     }
}
```

Associações e coleções

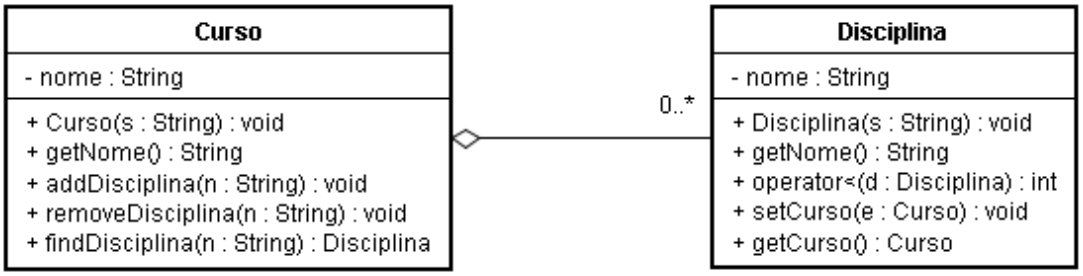
As associações para N podem ser simplificadas utilizando coleções, uma vez que a gestão das operações de inserção, remoção, pesquisa, etc., podem ser centralizadas.

Genericamente, existem duas formas de gerir as instâncias que cada classe associa, e para isso, definem-se dois tipos de coleções:

- **Por cópia.**
    - Neste caso a classe que associa fica responsável pela gestão das instâncias que associa, devendo criar uma cópia na inserção e libertar a instância nas remoções.
    - Esta forma tem, no entanto, uma desvantagem: as alterações das instâncias dentro da coleção não se refletem para as instâncias que foram inseridas (originais) e vice-versa.
    - É mais onerosa em termos de armazenamento. Tipicamente é utilizada para guardar o repositório original de instâncias, uma única vez.

- **Por apontador ou referência.**
    - Neste caso a classe que associa não fica responsável pela gestão das instâncias que associa.
    - Os apontadores originais são diretamente inseridos na coleção, não sendo, por isso, efetuadas cópia das instâncias que apontam.
    - Consequentemente, também não é necessário libertar as instâncias apontadas nas remoções.
    - A entidade que utiliza a classe é que deve ser responsável pela gestão (alocação e libertação) das instâncias.
    - É possível alterar diretamente as instâncias colecionadas, uma vez que existem um apontador para o original.
    - É mais vantajosa em termos de armazenamento.
    - Tipicamente é utilizada para guardar associações que referenciam as instâncias originais.

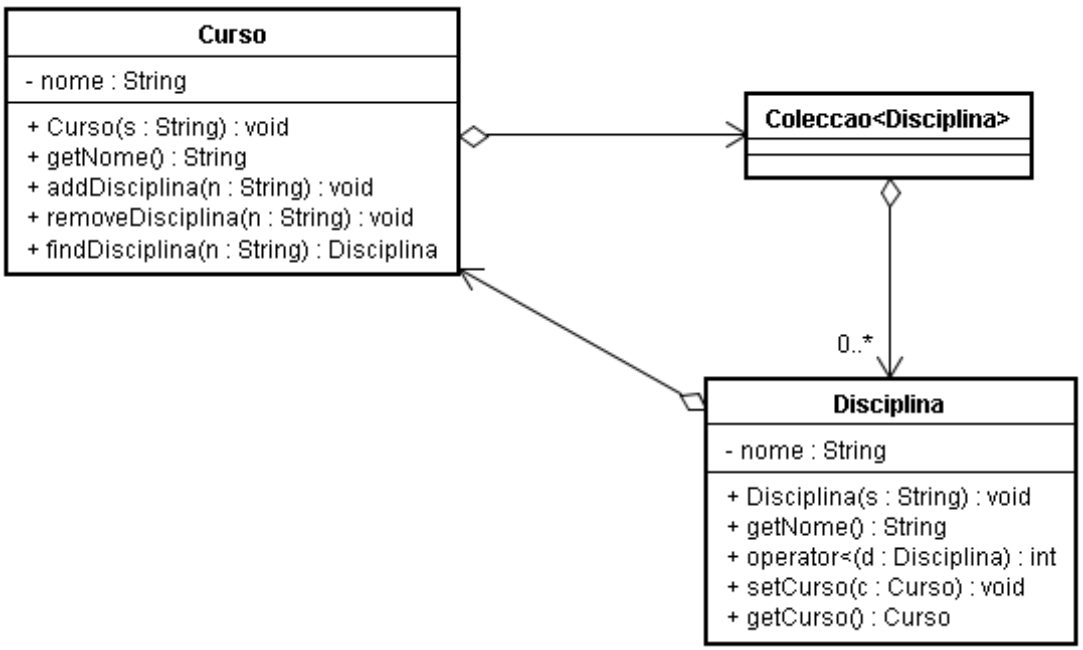
Associações 1-N



Quando existe uma associação com vários associados envolvidos é conveniente recorrer à criação de uma coleção desses elementos.

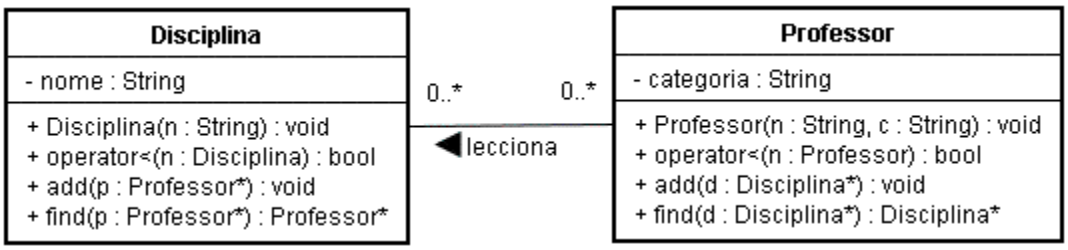
A coleção pode ser implementada por diversas estruturas de dados, no entanto, convém que estejam definidas, pelo menos, as operações: adicionar (add), remover (remove), procurar (find), entre outras que possam ser convenientes.

A associação pode ser vista como:

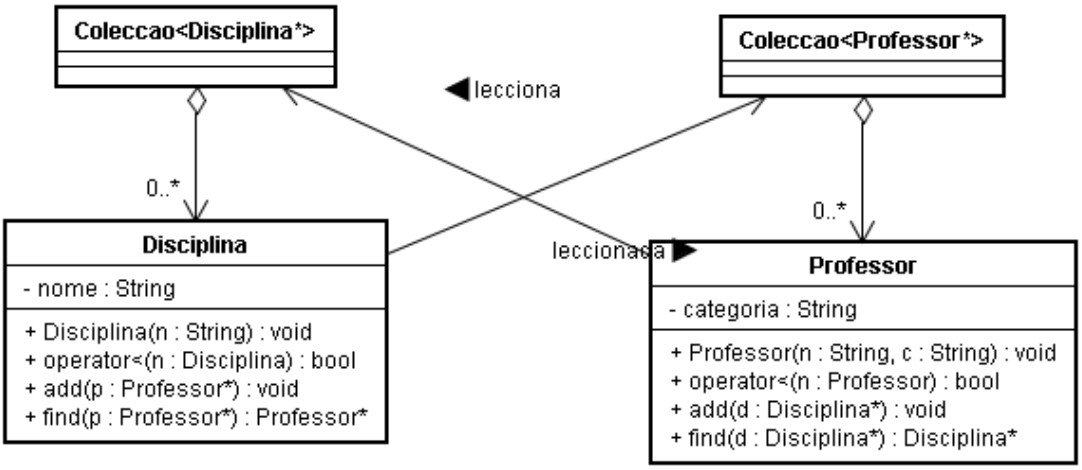


Na realidade esta visão não deve ser representada no diagrama de classes, mas sim a anterior.

Associações N-N

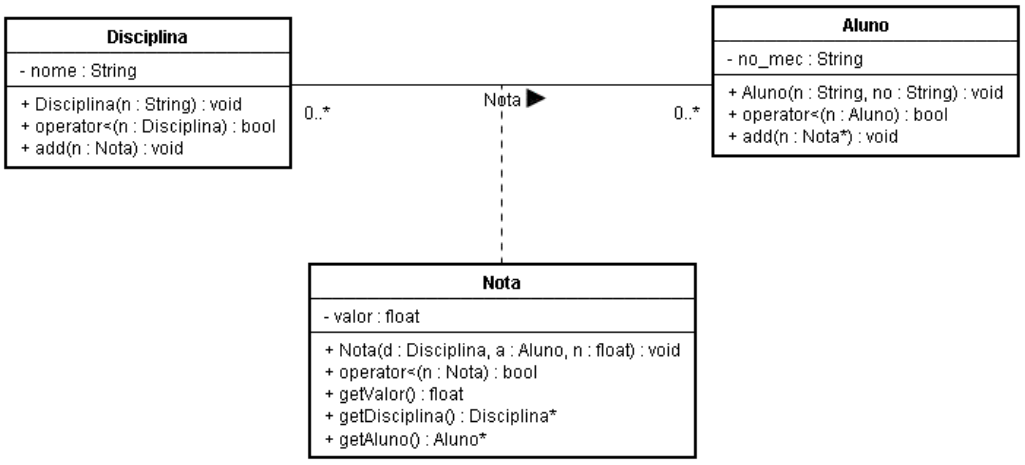


A associação N-N pode ser vista como:

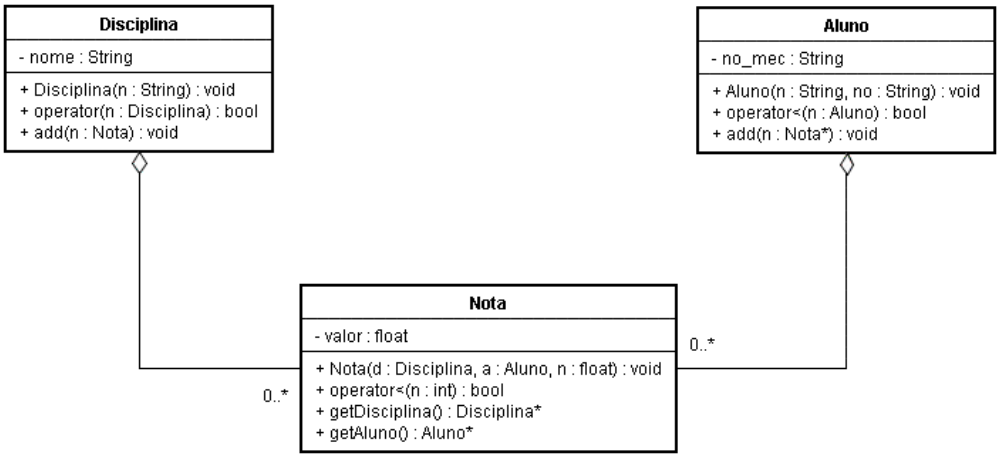


Mas na realidade esta visão não deve ser representada no diagrama de classes, mas sim a anterior.

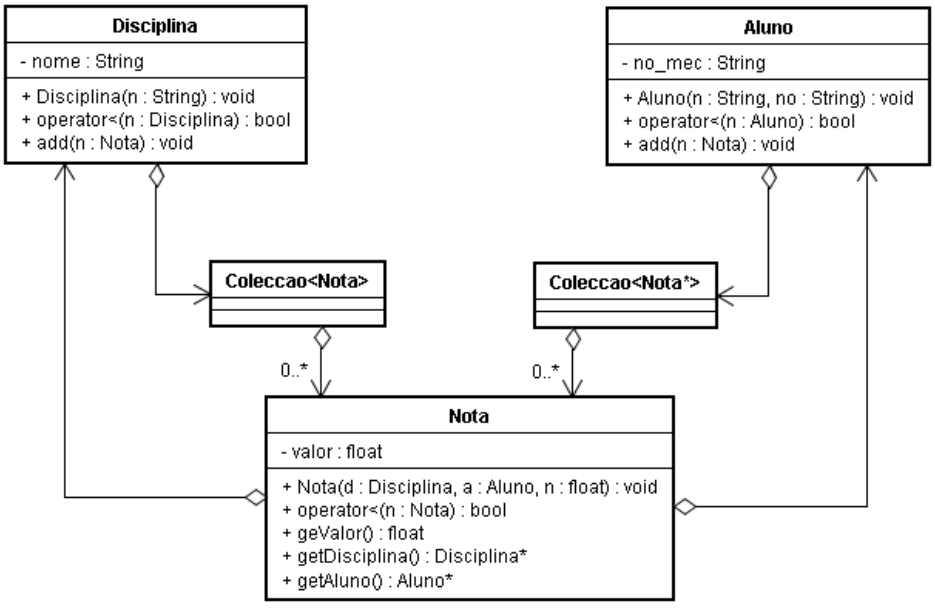
Classes Associativas N-N



Pode ser convertido em:



Com coleções ficaria:



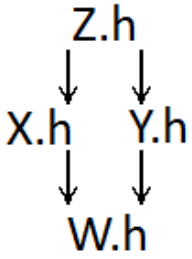
Inclusão de ficheiros de cabeçalho (.h)

Na declaração de classes, por vezes, há a necessidade de ficheiros de cabeçalho (.h) incluírem outros ficheiros de cabeçalho (.h)

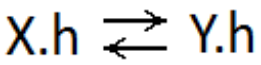
- com isso, a inclusão de ficheiros .h pode tornar-se complexa e dar mesmo origem a conflitos

Exemplos de situações de conflito:

1. X.h e Y.h inclui Z.h, e W.h inclui X.h e Y.h (inclusão múltipla de Z.h)



2. X.h inclui Y.h, e Y.h inclui X.h (inclusão circular)



Algumas regras para a resolução de conflitos:

- Nada de pânico com a quantidade de erros!
- Implementar uma classe por um par ficheiros: um .h e um cpp. Para a classe X, deve definir um X.h e um X.cpp.
    - Implementar os métodos no ficheiro .cpp.
- Escrever #pragma once no inicio dos ficheiro .h
    - Resolve o conflito 1

- Evitar fazer o #include”X.h” num ficheiro Y.h.
    - Tentar sempre que possível fazê-lo no Y.cpp.
- Se num ficheiro Y.h apenas existirem declarações de apontadores de instâncias de uma classe X, deve apenas efetuar-se a declaração em avanço de X:

  class X;

    - Isto resolve o conflito 2
    - Em inclusões circulares X ←→ Y, a que tipicamente correspondem agregações, a classe que agrega deve fazer um include e a classe agregada deve fazer uma declaração em avanço.
- Os seguintes erros de compilação:

  P.ex.: syntax error : missing ';' before '*‘

  P.ex.: missing storage-class or type specifiers

  normalmente, implicam a remoção de um include e a colocação de uma declaração em avanço.

    - Muitas das situações de conflito são resolvidas substituindo um include (que não estará a fazer nada devido ao #pragma once) por uma declaração em avanço.

Referências

As referências são nomes alternativos para objetos.

São obrigatoriamente inicializadas e referenciam o valor contido na variável que inicializou a referência.

Basicamente uma referência é um endereço, simplesmente, torna-se desnecessária a utilização do operador * (valor apontado por).

```
int a;
int &r=a;
r=5;    // efetivamente altera a
&r      // devolve o endereço de a
```

Em parâmetros de funções e métodos:

Neste caso, os parâmetros com referências irão assumir as variáveis que se colocam na chamada da função.

Por serem referências, os parâmetros passados na invocação têm que ser obrigatoriamente variáveis.

```
void troca(int &a, int &b);

troca(x, y); // os parâmetros devem ser
variáveis
```

Referências

No retorno de funções e métodos:

A utilização de referências no retorno de funções é particularmente importante para ser possível a modificação do valor devolvido no exterior da função. Por exemplo, a possibilidade de colocar essa função à esquerda de uma atribuição (l-value).

O valor retornado na função **não** pode ser uma variável local da função nem pode ser constante.

```
#include<iostream>

using namespace std;


class Numero
{
     int a;
public:
     Numero(int x){
          a=x;
     }
     int &altera(){
          return(a);
     }
     void print() const{
          cout << a << endl;
     }
};

void main()
{
     Numero a(10);
     a.print();
     a.altera()=20;
     a.print();
}
```

Herança

O conceito de herança permite relacionar classes de forma hierárquica. Assim, todas as classes descendentes de uma outra classe herdam todas as variáveis e métodos, podendo ser acrescidas de mais variáveis métodos.

Subclasses - Classes descendentes.

Superclasses - Classes ascendentes.

Uma classe pode ser derivada duma outra através da construção:

```
class classe_derivada: public classe_base
{
    ...
};
```

Acréscimo e substituição de métodos

Quando uma classe é herdada é possível acrescentar novos métodos à classe derivada.

Se se definir na classe derivada um método com o mesmo nome da classe base, o método da classe base é sobreposto (overriding).

```
class Ponto
{
protected:
    float x, y;
public:
    Ponto();
    Ponto(float, float);
    void print();
};

class Ponto3D : public Ponto
{
    float z;
public:
    Ponto3D();
    Ponto3D(float, float, float);
    void print();
};

Ponto p(2, 3);
Ponto3D p3d(5, 6, 7);
p.print();
p3d.print();
```

Um método pode fazer uso de um outro método com o mesmo nome da classe base utilizando:

```
Ponto::print();
```

Construtores e herança

Como os construtores são métodos especiais, não é possível utilizar o mesmo mecanismo utilizado para os métodos, i.e., invocação no corpo do método.

Para isso utilizam-se as listas de inicialização. Quando não são utilizadas as listas de inicialização, o construtor da classe derivada invoca sempre o construtor por defeito da classe base.

Com lista de Inicialização

```
Ponto3D::Ponto3D()
{
    z = 0;
}

Ponto3D::Ponto3D(int a, int b, int c) :
Ponto(a, b)
{
    z = c;
}
```

Tipos de proteção no acesso aos membros

Veja-se o seguinte exemplo:

```
class A {
public:
    int a;
protected:
    int b;
private:
    int c;
};

class B : public A {
    void Bfunction();
};

void B::Bfunction() {
    a = 5; // ok, public
    b = 5; // ok, protected é visível nas
classes derivadas
    c = 5; // Ilegal - private não é visível
}


void main() {
    B b;
    b.a = 5;  // ok, public
    b.b = 5;  // Ilegal - protected
    b.c = 5;  // Ilegal - private
}
```

Conversão ascendente (Upcast)

Um apontador ou referência de uma classe derivada pode ser sempre convertido num apontador ou referência de uma classe base.

```
Ponto3D* p3d = new Ponto3D(2, 5, 8);
Ponto* p3d1 = new Ponto3D(5, 10, 15);

Ponto3D p3(2, 3, 4);
Ponto& p = p3;

p3d->print();

p3d1->print();
```

Conversão descendente (Downcast)

Nalgumas situações é necessário fazer conversão contrária, isto é, converter de uma classe base para uma descendente.

```
Ponto* p = new Ponto3D(5, 10, 15);
p->getZ(); // Errado!
```

Neste caso p, não dispõe de acesso a um método getZ(),por exemplo.

```
Ponto3D* p3 = dynamic_cast<Ponto3D*>(p);
p3->getZ();
```

O dynamic_cast assegura que a conversão só é possível se, efetivamente, o objeto a converter é do tipo para o qual se pretende converter. Caso contrário, devolve zero.

A conversão ascendente é particularmente utilizada em parâmetros de funções ou métodos.

```
#include<iostream>
using namespace std;

enum notas {Do, Re, Mi };

class Instrumento{
public:
     void toca(notas n) {
          cout << "Instrumento::toca\n";
     }
};

class Flauta : public Instrumento{
public:
     void toca(notas n) {
          cout << "Flauta::toca\n";
     }
};

class Viola : public Instrumento{
public:
     void toca(notas n) {
          cout << "Viola::toca\n";
     }
};

void afinar(Instrumento& i){ i.toca(Do); }

void main(){
     Viola v;
     afinar(v);
}
```

O mesmo exemplo poderia utilizar apontadores.

Polimorfismo e funções virtuais

O polimorfismo permite que, em tempo de execução, sejam invocados diferentes métodos utilizando a mesma mensagem no mesmo apontador ou referência.

Definindo o método print como uma função virtual, podemos tirar partido do polimorfismo:

```
class Ponto
{
protected:
    float x, y;
public:
    Ponto();
    Ponto(float, float);
    virtual void print();
};
```

é invocado o método do objeto que foi criado e não da referência do objeto que invoca o método.

```
p3d1->print(); // invoca print de Ponto3D
```

Gestão de memória dinâmica interna a uma classe

Quando uma classe efetua internamente gestão de memória dinâmica (alocação e libertação) é necessário tomar algumas precauções, definindo um conjunto de métodos de suporte para o correto funcionamento dessas classes:

- Destrutor
- Operador de atribuição
- Construtor de cópia

```
class Aponta
{
      int a;
      int *ap;
public:
      Aponta(){
            a=0;
            ap=new int;
            *ap=0;
      }
      Aponta(int x, int y){
            a=x;
            ap=new int;
            *ap=y;
      }
      Aponta(const Aponta &c){
            a=c.a;
            ap=new int;
            *ap=*c.ap;
      }
      Aponta &operator=(const Aponta &c){
            if(this!=&c){
                  a=c.a;
                  delete ap;
                  ap=new int;
                  *ap=*c.ap;
            }
            return(*this);
      }
      ~Aponta(){
            delete ap;
      }
};
```

Destrutor

Os destrutores são também métodos especiais que são invocados automaticamente sempre que se torna necessário destruir uma instância de uma classe.

São invocados em situações como:

- Libertação explícita de apontadores para essa instância;
- Destruição normal de instâncias locais a uma função ou bloco ou parâmetros por valor;

Operador de atribuição

É um caso particular da sobrecarga de operadores e deve ser definido sempre que o construtor de cópia também for definido. Permite que uma instância seja atribuída a outra já criada.

```
<classe> &operator=(const <classe>&);
```

Por omissão, na atribuição os dados de um objeto são copiados um por um bit por bit. Nos membros apontadores apenas é copiado o endereço, não sendo copiados os valores apontados. Por isso, é que a definição de operadores de atribuição se manifesta tão importante.

Construtor de cópia

É invocado sempre que há necessidade de efetuar uma cópia de uma instância.

Construtores e Destrutores

Um construtor é invocado sempre que uma instância é definida.

Num contexto local, os destrutores são invocados por ordem inversa da sua construção.

Os construtores das instâncias globais são executados antes do main iniciar.

Os construtores de instâncias são executados pela ordem da sua definição dentro do mesmo ficheiro.

Os destrutores de instâncias globais são executados por ordem inversa da sua construção a seguir à finalização do main.

Destrutores na herança

Os destrutores das classes base são sempre invocados automaticamente pelo compilador pela ordem inversa da sua construção.

Os construtores são executados de acordo com a ordem em que as classes são derivadas, desde a classe base até à última classe derivada.

Os destrutores são executados de acordo com a ordem inversa em que as classes são derivadas, desde a última classe derivada até à classe base.

Polimorfismo e destrutores

Devido ao seu funcionamento, na utilização de polimorfismo deve ser sempre definido o destrutor como virtual para que seja sempre invocado o destrutor da classe correta, ou seja, o da classe alocada e não da classe base.

Exemplo

```
class A
{
    int x;
public :
    A()
    { x = 0; }
};

class B
{
    A* a;
public:
    B()
    { a = new A; }
    B(const B& b)
    { a = new A(*b.a); }

    B& operator = (const B& b)
    {
        if (this != &b)
        {
            delete a;
            a = new A(*b.a);
        }
        return (*this);
    }
    ~B()
    { delete a; }
};
void main()
{
    B b;
    B c(b);
}
```

Coleções híbridas

As coleções híbridas recorrem ao upcast e ao polimorfismo para colecionar, na mesma entidade, instâncias com uma classe base comum.

Como a coleção tem que ser implementada com apontadores, a gestão de memória dos elementos da coleção deve ser da responsabilidade de quem utiliza a coleção e não da coleção.

Para o correto funcionamento de um set da STL com apontadores, foi necessário criar um novo template ColeccaoHibrida, que apenas aceita apontadores como parâmetros.

O operador< da classe base foi definido como virtual.

As sobreposições de métodos nas derivadas podem requerer um dynamic_cast.

AS classes que contêm coleções híbridas têm que definir o construtor de cópia, destrutor e operador de atribuição.

Na cópia é necessário recorrer a um processo de duplicação dinâmica (clonagem).

Classes abstratas

Em muitas situações a classe base representa um conceito abstrato servindo apenas para agregar dados e operações ou definir uma interface padrão que terá que ser definida nas classes derivadas.

Um método que tenha o inicializador: =0 é designado de **função virtual pura**.

Uma classe é **abstrata** quando contém um ou mais funções virtuais puras.

De uma classe abstrata não é possível definir objetos.

```
class Figura{
...
    virtual void Rodar()=0;
    virtual void Desenhar()=0;
...
};
```

Uma função virtual pura mantém-se pura enquanto não é definida, mantendo-se a classe abstrata e, portanto, impossibilitando a definição de objetos.

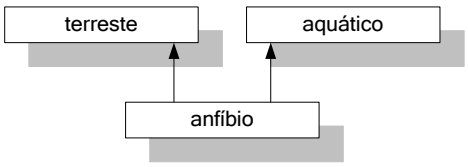
```
class X {
public:
    virtual void f() = 0;
    virtual void g() = 0;
};

X b;  // erro: definição de objeto com classe
abstrata

class Y : public X {
    void f() {

        ;   // sobrepoe X::f
    };
};

Y b;    // erro: Y continua abstrata

class Z : public Y {
    void g(); // sobrepoe X::g
};

Z c;  // ok
```

Só é possível definir objetos quando a classe derivada tiver o interface completamente definido.

Herança Múltipla

Considera-se herança múltipla quando uma classe derivada tem várias classes base diretas.

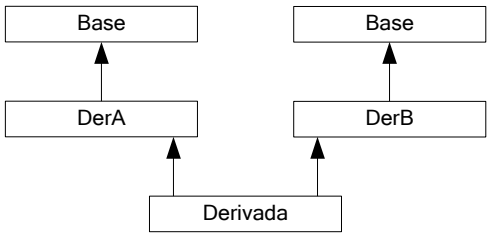
Exemplo de herança múltipla:



Qualquer método das superclasses pode ser invocado tal como na herança simples, sendo também utilizada a resolução de contexto (::) quando existam métodos sobrepostos.

Ocorrência múltipla da classe base

A herança múltipla pode decorrer em situações em que uma classe base aparece mais do que uma vez, trazendo obviamente problemas de conflito em relação à invocação dos métodos.



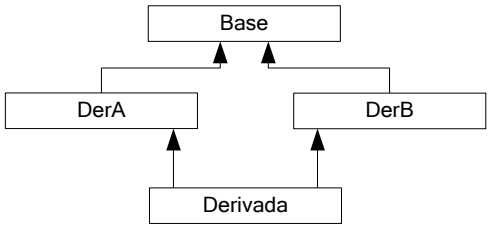
Neste caso se na classe *Derivada* for invocado um método da classe *Base*, o compilador não sabe em qual das classes *Base* deve invocar o método.

Nota importante:

Embora se explique a seguir como ultrapassar a dificuldade anterior, no âmbito desta unidade curricular não se implementarão soluções com recurso a herança múltipla. Por isso, relativamente a este tópico da matéria, será suficiente que o aluno compreenda o alcance do que se disse na página anterior.

Classes base virtuais

Existem situações em que não é conveniente a existência múltipla da classe base. Para tal, define-se a classe base como virtual, ficando a classe Base comum nas classes derivadas.



Exemplo: Ocorrência múltipla da classe base

No caso da herança múltipla com ocorrência múltipla da classe base, a existência de duas classes 'Base' pode originar ambiguidades.

A invocação de métodos pode requerer uma qualificação adicional:

DerA::Base::print(), para referir o método print da classe 'Base', base de 'DerA'.

A ordem de invocação dos construtores é sempre da base para as derivadas, pela ordem que são colocadas na derivação.

```
#include<iostream>

using namespace std;

class Base{
     char nome[10];
public:
     Base(char *n){
          strcpy(nome, n);
     }
     virtual void print(){
          cout << "Base: (" << nome << ")" <<
endl;
     }
};

class DerA: public Base{
public:
     DerA(): Base("DerA"){}
     void print(){
          Base::print();
          cout << "DerA" << endl;
     }
};
```

```
class DerB: public Base{
public:
     DerB(): Base("DerB"){}
     void print(){
          Base::print();
          cout << "DerB" << endl;
     }
};

class Derivada: public DerA, public DerB{
public:
     void print(){
          DerB::Base::print();      // print da
Base de DerB
          DerA::print();
          DerB::print();

          cout << "Derivada" << endl;
     }
};

void main()
{
     Derivada d1; // ok
     d1.print();

//   Base* d=new Derivada; // Errado! Podem ser
duas Base
//   d->print();
}
```

Exemplo: classes base virtuais

Com classes base virtuais, a classe base 'Base' apenas existe uma única vez, pelo que a ordem normal de invocação dos construtores é ligeiramente alterada.

A ordem é a seguinte:

Base – uma vez

DerA – Mas não invoca o construtor de Base

DerB – Mas não invoca o construtor de Base

E Derivada

Quem invoca o construtor de Base é a Derivada, pelo que, as listas de inicialização devem aí ser colocadas.

As listas de inicialização da classe 'Base' nas classes DerA e DerB não têm qualquer efeito.

```
#include<iostream>

using namespace std;

class Base{
     char nome[10];
public:
     Base(char *n){
          strcpy(nome, n);
     }
     virtual void print(){
          cout << "Base: (" << nome << ")" <<
endl;
     }
};

class DerA: public virtual Base{
public:
     DerA(): Base("DerA"){}
```

```
    void print(){
        Base::print();
        cout << "DerA" << endl;
    }
};

class DerB: public virtual Base{
public:
    DerB(): Base("DerB"){}
    void print(){
        Base::print();
        cout << "DerB" << endl;
    }
};

class Derivada: public DerA, public DerB{
public:
    Derivada(): Base("Unica") {}
    void print(){
        //DerB::Base::print();        // não é
necessário DerB::Base, porque a base é só uma
        Base::print();
        DerA::print();
        DerB::print();

        cout << "Derivada" << endl;
    }
};



void main()
{
    Derivada d1; // ok
    d1.print();

    Base* d=new Derivada; // Agora sim, uma
classe Base pode referenciar uma instância de
Derivada
    d->print();
    delete d;
}
```

Operadores de conversão

Permitem a conversão de um objeto para outro tipo.

```
class Real{
    double d;
public:
    Real(double pd) {
        d = pd;
    }
    operator double() const {
        return(d);
    }
};


Real r(10.0);

double d = r;
```

Membros estáticos

Um membro (dados ou funções) definido como estático é partilhado por todos os objetos.

```
#include<iostream>
using namespace std;

class ABCD {
     int ch;
     static int s;
public:
     ABCD() {ch = 0; }
     void setCH(int c) { ch = c;  }
     void setS(int a) {   s = a; }
     void printCH() { cout << ch << endl; }
     void printS() { cout << s << endl; }
     static int getS() { return(s); }
};

int ABCD::s = 0;

void main()
{
     ABCD a, b, c, d;
     b.setCH(5);
     b.setS(10);
     b.printCH();
     b.printS();

     c.setCH(50);
     c.setS(100);
     b.printCH();
     b.printS();
}
```

Declarações `friend`

As declarações `friend` são feitas na definição de classes e permitem que a entidade definida como tal, tenha acesso a todos os elementos privados dessa classe.

São normalmente funções ou classes externas à classes em questão.

```
int pode_entrar(Temfriends t)
{
    t.a = 10;
}

class Temfriends
{
    int a;
public:
    Temfriends();
    friend int pode_entrar(Temfriends);
    ...
}
```

Entrada/Saída da dados em C++ (iostream)

Saída Formatada

```
cout << "Numero: " << 10 << endl;

double d=3.1415927;
```

Define o comprimento do campo da saída subsequente.

```
cout.width(10);
```

Define a precisão do campo da saída subsequente.

```
cout.precision(2);
cout << d << "\n";
```

Sobrecarga do operador <<

```
ostream &operator<<(ostream &s, const <classe>
&var);
```

Sendo definido como friend na classe.

Entrada Formatada

```
int a;
cout << "Introduza numero: ";
cin >> a;
```

Sobrecarga do operador >>

```
istream &operator>>(istream &s, <classe> &var);
```

Sendo definido como friend na classe.

Entrada/Saída de dados em C++ (iostream)

Entrada e saída não formatada

Envia um caracter para uma saída.

```
ostream& ostream::put(char ch);
```

Recolhe um caracter de uma entrada.

```
istream& istream::get(char &ch);
```

Devolve o caracter para a entrada.

```
istream& istream::putback(char ch);
```

Recolhe uma linha para line, com tamanho máximo *size*-1 até encontrar o caracter *terminator*. Este não é colocado em *line*.

```
istream& istream::getline(char line[], int
size, char terminator='\n');
```

Input/Output em C++ (fstream)

Acesso a ficheiros

O acesso a ficheiros é feito, definindo instâncias de ifstream e ofstream, respetivamente para ficheiros de entrada e saída.

Os operadores sobre streams standard aplicam-se às streams de ficheiros.

Deve-se incluir o namespace <fstream>.

```
ifstream f("c:\\dados.txt"); // entrada

ofstream f("c:\\dados.txt");  // saída

fstream f("c:\\dados.bin", ios::out |
ios::binary)   // saída em modo binário
```

Modos

ios::app	Acresenta	ios::binary	Modo binário
ios::ate	Posiciona no fim do ficheiro	ios::trunc	Apaga o conteúdo
ios::in	Entrada	ios::nocreate	Falha se não existe
ios::out	Saída	ios::noreplace	Se existe, falha, excepto se ios::app ou ios::ate

Também é possível abrir um ficheiro definindo a instância do canal por omissão e utilizar o método open.

```
ofstream f;
f.open("abc.txt");
```

Métodos

```
void open(const char *filename,
ios_base::openmode _Mode);

write(unsigned char *p, int n);

read(unsigned char *p, int n);

void close();
```

Exemplo

```
#include<iostream>
#include<fstream>


using namespace std;

void main()
{
    char buf[256];
    ifstream f("io.cpp");
    if(!f){
        cout << "Não foi possivel abrir
o ficheiro";
        return;
    }
    while(!f.eof()){
        f.getline(buf, 255);
        cout << buf << endl;
    }
    f.close();

}
```